Online Appendix to:
Formal Approaches to Secure Compilation A survey of fully abstract compilation and related work

MARCO PATRIGNANI, Stanford University, USA and CISPA, Germany
AMAL AHMED, Northeastern University, USA
DAVE CLARKE, Uppsala University, Sweden

A ADDITIONAL PROGRAM EQUIVALENCES FOR EXPRESSING SECURITY PROPERTIES

This section presents a probabilistic variant of contextual equivalence (Appendix A.1) Then it formalises alternatives to contextual equivalence, which have been devised since direct proofs of contextual equivalence are usually intractable. Specifically, this section discusses bisimilarity (Appendix A.2), trace semantics (Appendix A.3), and logical relations (Appendix A.4).

A.1 Probabilistic Contextual Equivalence

Contextual equivalence is not enough for languages with a notion of randomisation or cryptography, as the universal quantification over contexts models a too powerful attacker. If a strong cryptographic function is used to encrypt data, then most contexts will not be able to decrypt that data and retrieve the contents. So, by considering all contexts, there is one that by definition has the right key to decrypt the data. That context can observe some behavioural difference that we know can be done only by breaking the cryptographic function. As the used cryptographic functions are generally assumed to be unbreakable, this model violates a key assumption of the system. Additionally, when keys are computable (e.g., they could be bitstrings), the context can just exhaust the search space and try all keys. However, in practice, very large keys are used and these contexts would run for too long to calculate the right key. As these contexts also violate a key assumption of the system, often contexts are limited to be *polynomial* in the size of the key, so they cannot exhaust the key search space [50, 51].

To filter out contexts that violate these key assumptions, probabilistic contextual equivalence can be used. With probabilistic contextual equivalence, two programs are equivalent if they are equivalent up to a certain probability (i.e., in *the majority* of contexts) for polynomial contexts.[1]

In the example above, contexts that have ways to break the cryptographic functions must not qualify as the majority of contexts, so they can be safely discarded when considering program equivalence.

To model the semantics of randomisation, oracles can be made part of the semantics [8]; they provide infinite streams of random values for the randomisation function to use.

Examples A.1 and A.2 describe the usage of oracles. Example A.3 discusses how guessing (either random numbers or cryptographic keys) affects the definition of contextual equivalence.

[1] We state this together with the assumption on polynomial contexts, though in cases when secrets are not computable, this condition can be dropped [9, 61].

0360-0300/2018/02-ART125 $15.00
https://doi.org/10.1145/3280984

Example A.1 (Obvious Equivalences). With an oracle, obvious equivalences must be respected. For example, a program P_r that returns a random value must be equivalent to itself.

```
public getRandom(){   //P_r
  return rand.next();
}
```

In this case, for any oracle O, there exists an oracle (the same oracle O) that makes the behaviour of two copies of P_r coincide.

P_r is also equivalent to the following snippet P_{rr}, which generates two random numbers and only returns the second.

```
public getRandom(){   //P_rr
  rand.next();
  return rand.next();
}
```

The oracle that must be used with P_{rr} is one that has all elements of O interleaved with other elements.

Example A.2 (Obvious Inequivalences). Consider a program that returns two random numbers and a program that returns the same random number twice.

```
public getRandom(){
  var x = rand.next();
  return new Pair(x, rand.next());
}
```

```
public getRandom(){
  var x = rand.next();
  return new Pair(x, x);
}
```

For any oracle used with the right-hand side snippet, the oracle that contains the same value twice will make the left-hand side snippet equivalent to it. However, there is no oracle that will make the opposite true: given an oracle for the left-hand side snippet, there is no way to build one that will make the right-hand side one behave the same. In fact, an oracle that provides two different values will make the left-hand side return a pair with two different values, and this cannot be done by the right-hand side snippet.

We can thus conclude that if any two components are equivalent, then for any oracle used with the first one there must exist one for the second such that they exhibit the same behaviour *and vice versa.*

Example A.3 (Guessing). Consider two programs that store a random number and then receive input from the external code via variable guess. If that input matches the random number, then one returns 0 and the other returns 1; otherwise, they both return 2.

```
public getRandom(){
  var x = rand.next();
  var guess = callback();
  if (guess == x)
    return 0;
  return 2;
}
```

```
public getRandom(){
  var x = rand.next();
  var guess = callback();
  if (guess == x)
    return 1;
  return 2;
}
```

If the random function is strong enough, then the context has very little chance of telling these programs apart. As already said however, the contexts considered in the definition of contextual equivalence are universally quantified. Thus, there is also a context that differentiates between the two programs by guessing the number, so the definition of probabilistic contextual equivalence needs to incorporate probability concepts.

The probabilistic notion of contextual equivalence states that two programs are equivalent if, when the context is polynomial, they behave the same to a certain probability. Denote the probability of a certain event with $\mathbb{P}(\cdot)$ and let σ range over the [0, 1] interval. We indicate a context being polynomial given the size s of intended secrets as $\vdash \mathbb{C} : \text{poly}\ s$.

Definition A.4 (Contextual Preorder).

$$P_1 \sqsubseteq_\sigma P_2 \triangleq \mathbb{P}(\forall \mathbb{C}. \vdash \mathbb{C} : \text{poly}s,\ O_1, \exists O_2. \mathbb{C}[P_1, O_1]\Uparrow \iff \mathbb{C}[P_2, O_2]\Uparrow\ ) > \sigma$$

Probabilistic contextual equivalence can be defined as following.

Definition A.5 (Probabilistic Contextual Equivalence [9]). $P_1 {\simeq_{\text{ctx}\sigma}} P_2 \triangleq P_1 \sqsubseteq_\sigma P_2$ and $P_2 \sqsubseteq_\sigma P_1$.

A.2 Bisimilarity

Bisimilarity has been frequently employed in the field of process algebra to state when two processes exhibit the same behaviour [113]. It has also been frequently used to prove compilation schemes for process calculi secure [5–7, 13, 27, 43, 74]. Intuitively, two processes have the same behaviour when they perform the same "*actions*" and become new processes that continue to have the same behaviour.

The notion of bisimilarity (Definition A.8) relies on the concept of a labelled transition system (LTS), which is used to provide a model of the process.

Definition A.6 (LTS). A labelled transition system is a triple $(S, \Lambda, \to)$, where S is a set of states, Λ is a set of labels, and $\to \subseteq S \times \Lambda \times S$ is a ternary relation of labelled transitions.

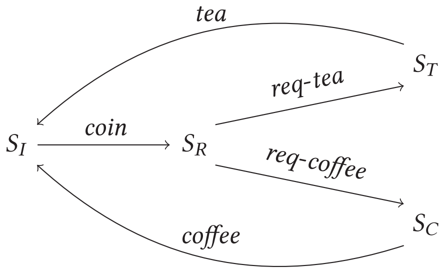
A transition between two states S_1 and $S_2 \in S$ on a label $\lambda \in \Lambda$ is indicated as $S_1 \xrightarrow{\lambda} S_2$. Labels represent what an entity external to S can observe from the states of S, as S performs computations; labels often concern inputs and outputs, as presented in the following example.

Example A.7 (LTS [113]). Consider the LTS of a vending machine that produces tea or coffee after receiving coins, after the appropriate request is made. It is formalised as

$$(\{S_I, S_R, S_T, S_C\}, \{\mathit{coin}, \mathit{req\text{-}tea}, \mathit{req\text{-}coffee}, \mathit{tea}, \mathit{coffee}\},$$

$$\{S_I \xrightarrow{coin} S_R,\ S_R \xrightarrow{req\text{-}tea} S_T,\ S_R \xrightarrow{req\text{-}coffee} S_C,\ S_T \xrightarrow{tea} S_I,\ S_C \xrightarrow{coffee} S_I\}),$$

and it is depicted below.



S_I models the state of a vending machine waiting for input, *coin* expresses the user input, and S_R models the state in which the machine waits for the type of product to deliver. Based on the two different inputs from S_R, the machine can reach two states: S_T and S_C, the states where the machine produces tea and coffee, respectively. Then, both S_T and S_C transition back to S_I, labelled with the output it provides to the user: *tea* or *coffee*.

Often, labels are also equipped with decorations that indicate the direction of the action: ! is an observable produced from the program, ? is an observable received by it. The aforementioned transitions can thus be decorated as follows $S_I \xrightarrow{coin?} S_R$, $S_R \xrightarrow{req\text{-}coffee?} S_C$ and $S_T \xrightarrow{tea!} S_I$.

Definition A.8 (Bisimilarity). Given a LTS $(S, \Lambda, \rightarrow)$, a relation $\mathcal{R} \subseteq S \times S$ is a bisimulation if, for any pair $(S_1, S_2) \in \mathcal{R}$, for all $\lambda \in \Lambda$, the following holds:

(1) for all S_1' such that $S_1 \xrightarrow{\lambda} S_1'$, there exists S_2' such that $S_2 \xrightarrow{\lambda} S_2'$ and $(S_1', S_2') \in \mathcal{R}$;
(2) for all S_2' such that $S_2 \xrightarrow{\lambda} S_2'$, there exists S_1' such that $S_1 \xrightarrow{\lambda} S_1'$ and $(S_1', S_2') \in \mathcal{R}$.

Bisimilarity is the union of all bisimulations.

Two components P and Q are thus bisimilar if there exists a bisimulation relation $\mathcal{R}$ such that $P \mathrel{\mathcal{R}} Q$.

A more permissive variant of bisimilarity that is often used for program equivalence is weak bisimilarity. Its definition is the same as that of bisimilarity except that $\Rightarrow \lambda$ is used in place of $\xrightarrow{\lambda}$. Relation $\Rightarrow \lambda$ abstracts away from transitions that model internal computations (often called "silent" transitions and defined as a τ). Formally $\Rightarrow \lambda$ is defined as $\xrightarrow{\tau}^{*}\xrightarrow{\lambda}\xrightarrow{\tau}^{*}$. Thus, programs `x = 0 ; x += 1` and `x = 1` are weakly bisimilar (if `x` is not observable) even though they perform a different number of internal steps; while they are not bisimilar according to Definition A.8.

When working with bisimulation in place of contextual equivalence for secure compilation, labels model what the external code (i.e., the context in contextual equivalence) can observe about a program. The external code is modelled as a black box that triggers transitions. So, bisimilarity abstracts from the behaviour of the attacker but it captures the reaction of the component to certain actions of the attacker. This abstraction is the great advantage of bisimulation over contextual equivalence. Thus, it is crucial, when replacing contextual equivalence with bisimulation, that all possible attacker behaviour is captured by the labels, to have a precise characterisation of what the attacker can do. Finally, the bisimulation proof technique is more amenable to proofs than the unstructured induction over all possible reductions that is offered by contextual equivalence.

Example A.9 (Security Properties via Bisimilarity: Memory Size). Consider the code snippets of Figure 3, the following LTSs describe their behaviour in a language with unbounded memory size (Figure 5). States ext and cb indicate that control is outside the snippets. State $S0$ indicates the beginning of the code snippets, states Oi indicate the state where the ith object is allocated, and states S indicate the end of the code snippets. States are given a subscript l or r to indicate whether they describe the behaviour of the left-hand side or of the right-hand side snippet.

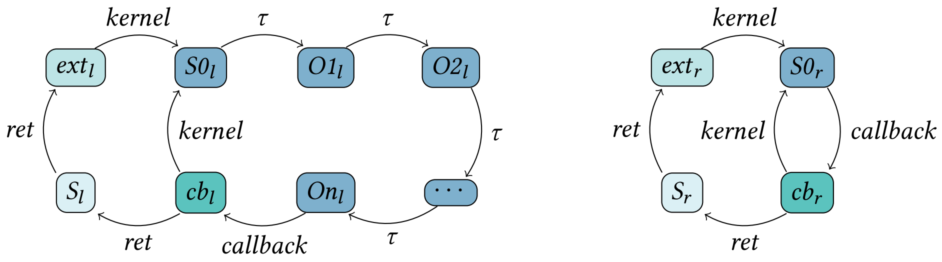


Fig. 5. LTSs describing weakly bisimilar snippets of Figure 3. Nodes coloured with the same colour are bisimilar.

To indicate that the code snippets are weakly bisimilar, the following bisimulation relation can be built:

$$\mathcal{R}_{mem} = \{(ext_l, ext_r), (S0_l, S0_r), (O1_l, S0_r), (O2_l, S0_r), \ldots, (On_l, S0_r), (cb_l, cb_r), (S_l, S_r)\}.$$

Relation $\mathcal{R}_{mem}$ is a weak bisimulation as it satisfies Definition A.8 when relation $\Rightarrow \lambda$ is used. In fact, for all pairs in $\mathcal{R}_{mem}$, the first element can perform a possibly-empty sequence of τs followed by an action λ ending up in a state. The second element can also perform a possibly-empty sequence of τs followed by the same action λ and end up in a state that is related to the previous one in $\mathcal{R}_{mem}$. And vice versa.

A.3 Trace Equivalence

Like bisimilarity, trace equivalence has been used successfully to replace contextual equivalence in secure compilation work [61, 64, 99, 102].

Given the LTS of a component P, the behaviour of P can be described with sequences of labels that can be generated according to the LTS. These sequences of labels, denoted with $\overline{\lambda}$, are called *traces*. The trace semantics of a program is the set of of all traces that can be associated with a component, based on its labelled transition system.

Definition A.10 (Trace Semantics). The trace semantics of a component P is indicated as $\mathsf{Traces}(P)$. $\mathsf{Traces}(P) = \{\overline{\lambda} \mid \exists P'.P \Rightarrow \overline{\lambda}P'\}$.

Two programs are trace equivalent, denoted with $P_1 \overset{\mathsf{T}}{=} P_2$, if their traces coincide.

Definition A.11 (Trace Equivalence). $P_1 \overset{\mathsf{T}}{=} P_2 \triangleq \mathsf{Traces}(P_1) = \mathsf{Traces}(P_2)$.

The same considerations made for bisimulation apply also here: since labels capture attacker actions, it is crucial that all that the attackers can observe is captured in the labels. This is captured by a proof stating that trace equivalence is as precise as contextual equivalence [62, 100].

Trace equivalence also comes with a somewhat simpler proof technique than the one of contextual equivalence. Traces are often defined inductively, so they are particularly amenable as a proof technique as they enable a neat, structural argument for proofs adopting them.

Before seeing an example of trace semantics at work for defining security properties (Example A.13), Example A.12 informally presents a trace semantics for a Java-like program. The example is reminiscent of the work of Reference [62]. The interested reader is referred to work on fully abstract trace semantics for a more in-depth discussion of the subject [62, 70, 100, 137].

Example A.12 (Trace Semantics for a Java-Like Program). Consider the class of the following snippet to be the component whose trace semantics we are interested in. The context this component interacts with is the `Log` class, which we assume to just implement function `addLogEntry()`. The other code does not need to be specified, as contexts are for contextual equivalence; the trace semantics needs to capture only its power in terms of the actions it can perform.

A trace semantics for such a Java-like language needs to capture how classes interact with each other, i.e., method calls and returns (assume all fields are private for simplicity). The labels below indicate when they are generated. We use a convention of decorating actions that are performed by the component with a "!", while ?-decorated actions are performed by the context.

log calling method `deposit` on line 6	`call acc.deposit( n )?`
executing line 8	`ret n!`
log calling method `logUsage` on line 10	`call acc.logUsage( u )?`
executing line 11	`call log.addLogEntry( u )!`
log returning to line 12	`ret unit?`
executing line 12	`ret unit!`

```java
import Log;

public class Account{
  private int balance = 0;

  public int deposit( int amount ) {
    this.balance += amount;
    return balance;
  }
  public void logUsage( String user ) {
    log.addLogEntry( user );
    return unit;
  }
}
public Account acc;
```

Listing 7. Example of Java source code.

A trace will therefore be a concatenation of those labels in a way that the semantics allows, e.g.,:

`call acc.deposit( n )`? `ret n`!

`call acc.deposit( n )`? `ret n`! `call acc.deposit(m)`? `ret n+m`!

`call acc.logUsage( u )`? `log.addLogEntry( u )`! `ret unit`? `ret unit`!

So `call deposit( n )`? can be followed by `ret n`! but not from `call addLogEntry( u )`!.

Example A.13 (Security Properties via Traces: Integrity). Consider the code snippets of Figure 2 and a trace semantics describing the behaviour of the code snippets based on labels for calls and returns as in Example A.12. If the behaviour of those code snippets can be described by (a concatenation of) the following trace, for any possible value v, then the programs are equivalent:

`call proxy(callback())`? `call callback()`! `ret` v? `ret 0`!

However, if there exists a trace that belongs to the trace semantics of a program but not to the trace semantics of the other, then they are not equivalent. For example, the following trace can be a valid trace of the left-hand side snippet but not of the right-hand side one.

`call proxy(callback())`? `call callback()`! `ret` v? `ret 1`!

Consider a language with pointers, which can be used to modify the contents of the stack (e.g., C), this trace is a valid description of the behaviour of the left-hand side snippet. This trace highlights a difference in the behaviour of the two snippets. As those snippets modelled integrity concerns, this trace captures a violation of the integrity property of the `secret` variable.

A.4 Logical Relations

Logical relations are a proof technique that has been used to prove when programs have nontrivial properties such as normalisation (of the simply-typed λ-calculus) [126], noninterference [59], equivalence of modules [19, 87] and compiler correctness [24, 60, 106]. Additionally, it has frequently been used to prove compiler security [17, 18, 26, 37, 94, 129].

Logical relations can be set up to prove contextual equivalence of components written in the same language. Once such a logical relation is defined, it must be proved sound (and perhaps also complete) with respect to contextual equivalence. But logical relations can also be used to formally express a desired notion of equivalence between components from two different languages—we refer to these as *cross-language* logical relations. Note that the peculiar thing about cross-language logical relation is that we cannot prove them sound with respect to any notion of contextual equivalence, since there is no such notion that spans the two different languages. Below, to illustrate how logical relations are defined, we present a cross-language logical relation between the source and (highly idealized) target language of a compiler. Some of the aforementioned papers on secure compilation use cross-language logical relations, but others use relations between terms of the same language or between terms of multi-languages; we discuss these details later in the article.

Example A.14 (Cross-Language Logical Relations). Consider the simply-typed λ*–calculus* with Booleans as source language. The types of the language conform to the following grammar $\tau ::= \texttt{Bool} \mid \tau \to \tau$, the syntax of values and expressions is omitted for the sake of brevity. Reduction steps are indicated with $\twoheadrightarrow$, their reflexive, transitive closure is denoted with $\twoheadrightarrow^*$; the capture-avoiding substitution of a value $\mathbf{v}$ for variable $\mathbf{x}$ in $\mathbf{t}$ is denoted as $\mathbf{t}[\mathbf{v}/\mathbf{x}]$.

Consider an untyped λ-calculus with Booleans as the target language.

The logical relation that states whether two terms from the two languages are contextually equivalent is defined as follows:

$$
\begin{aligned}
\mathcal{V}[\![\texttt{Bool}]\!] &\stackrel{\text{def}}{=} \{(k, \mathbf{v}, \mathsf{v}) \mid (\mathbf{v} \equiv \texttt{true} \text{ and } \mathsf{v} \equiv \texttt{true}) \text{ or } (\mathbf{v} \equiv \texttt{false} \text{ and } \mathsf{v} \equiv \texttt{false})\},\\
\mathcal{V}[\![\tau' \to \tau]\!] &\stackrel{\text{def}}{=} \left\{ (k, \mathbf{v}, \mathsf{v}) \;\middle|\; \begin{array}{l} (\mathbf{v}, \mathsf{v}) \in \texttt{oftype}(\tau' \to \tau) \\ \text{and } \exists \mathbf{t}, \mathsf{t}. \mathbf{v} \equiv \lambda \mathbf{x} : \tau'.\, \mathbf{t} \text{ and } \mathsf{v} \equiv \lambda \mathsf{x}.\, \mathsf{t} \\ \text{and } \forall k' \sqsupset_{\triangleright} k, (k', \mathbf{v}', \mathsf{v}') \in \mathcal{V}[\![\tau']\!].\, (k', \mathbf{t}[\mathbf{v}'/\mathbf{x}], \mathsf{t}[\mathsf{v}'/\mathsf{x}]) \in \mathcal{E}[\![\tau]\!] \end{array} \right\},\\
\mathcal{K}[\![\tau]\!] &\stackrel{\text{def}}{=} \{(k, \mathbb{C}_{\mathsf{S}}, \mathbb{C}_{\mathsf{T}}) \mid \forall k' \leq k, (k', \mathbb{C}_{\mathsf{S}}[\mathbf{v}], \mathbb{C}_{\mathsf{T}}[\mathsf{v}]) \in \mathcal{OBS}\},\\
\mathcal{E}[\![\tau]\!] &\stackrel{\text{def}}{=} \{(k, \mathbf{t}, \mathsf{t}) \mid \forall (k, \mathbb{C}_{\mathsf{S}}, \mathbb{C}_{\mathsf{T}}) \in \mathcal{K}[\![\tau]\!], (k, \mathbb{C}_{\mathsf{S}}[\mathbf{t}], \mathbb{C}_{\mathsf{T}}[\mathsf{t}]) \in \mathcal{OBS}\},\\
\mathcal{G}[\![\emptyset]\!] &\stackrel{\text{def}}{=} \{(k, \emptyset, \emptyset)\},\\
\mathcal{G}[\![\Gamma, (\mathbf{x} : \tau)]\!] &\stackrel{\text{def}}{=} \{(k, \gamma[\mathbf{v}/\mathbf{x}], \gamma[\mathsf{v}/\mathsf{x}]) \mid (k, \gamma, \gamma) \in \mathcal{G}[\![\Gamma]\!] \text{ and } (k, \mathbf{v}, \mathsf{v}) \in \mathcal{V}[\![\tau]\!]\},\\
\mathcal{OBS} &\stackrel{\text{def}}{=} \{(k, \mathbf{t}, \mathsf{t}) \mid (\mathbf{t}, \mathsf{t} \text{ terminate}) \vee (\exists \mathbf{t}', \mathsf{t}'. \mathbf{t} \twoheadrightarrow^{\mathsf{k}} \mathbf{t}', \mathsf{t} \twoheadrightarrow^{\mathsf{k}} \mathsf{t}')\}.
\end{aligned}
$$

This logical relation is indexed by a natural number k that, in essence, represents the number of remaining steps for which the terms are related—after k steps, there is no guarantee that the terms are related. This influences the notion of observation ($\mathcal{OBS}$), which says that either the two terms converge or they are still running after k steps have elapsed. Step-indexed logical relations are an instance of Kripke logical relations (as noted by Reference [15]). Here, the world is composed solely of the step index, which describes the number of steps still available in the current world. For languages with more advanced features, such as dynamically allocated mutable memory, the logical relation is indexed by worlds that keep track of the remaining number of steps as well as sophisticated relational invariants on the memory locations of the two programs being related.

Two Booleans are related when they are the "same" Boolean. Two functions are related when given related argument, they evaluate to related terms. Also, two functions are checked to be in the right syntactic form by function $\texttt{oftype}(\tau' \to \tau)$. Two contexts are related when supplied related values, they are observationally equivalent terms. Two expressions are related when we can embed them in related contexts and this results in observationally equivalent terms. Finally, two substitutions are related when they replace related variables (i.e., with the same name) with related values.

With this relation, we can define when two open terms are related as follows:

$$
\begin{aligned}
\Gamma \vdash \mathbf{t} \propto \mathsf{t} : \tau &\stackrel{\text{def}}{=} \forall k, \Gamma \vdash \mathbf{t} \propto_k \mathsf{t} : \tau,\\
\Gamma \vdash \mathbf{t} \propto_k \mathsf{t} : \tau &\stackrel{\text{def}}{=} \forall j \leq k, \forall (j, \gamma, \gamma) \in \mathcal{G}[\![\Gamma]\!], (j, \mathbf{t}\gamma, \mathsf{t}\gamma) \in \mathcal{E}[\![\tau]\!].
\end{aligned}
$$

B PROOF TECHNIQUES FOR EQUIVALENCE-PRESERVING COMPILATION

Most of the secure compilation results that are discussed in Section 5 establish secure compilation by proving fully abstract compilation, so we now focus on proof techniques to prove the more difficult part of compiler full abstraction, namely equivalence preservation. Existing techniques all rely on the general concept of *back-translation of a target context* (Appendix B.1). There are two broad categories of back-translation techniques: The first is based on partial evaluation of the target context (Appendix B.2), while the second is based on an embedding of the target context into the source language (Appendix B.3).

B.1 Back-translation, Informally

As stated in Section 4.3.1, preservation of contextual equivalence is stated as follows:

$$\mathit{Preservation} = \forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}, \mathbf{P}_1 \simeq_{\mathbf{ctx}} \mathbf{P}_2 \Rightarrow [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{\mathsf{ctx}} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}}.$$

For the sake of simplicity, this statement is re-stated in contrapositive form, as it turns the universal quantifiers hidden inside the definition of contextual equivalence into existentials:

$$\mathit{Preservation}\ (\mathit{contrapositive}) = \forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}, [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \not\simeq_{\mathsf{ctx}} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}} \Rightarrow \mathbf{P}_1 \not\simeq_{\mathbf{ctx}} \mathbf{P}_2.$$

To prove this statement, it is sufficient to consider two source programs whose compiled counterparts are different and build a source context that differentiates those source programs.

The following example gives an informal intuition of what it means to generate the differentiating source-level context.

Example B.1 (Informal Context Back Translation). Consider these two STLC terms:

$$\mathbf{t_l} \overset{\text{def}}{=} \lambda \mathbf{x} : \mathbf{Bool}.\mathtt{false}, \qquad \mathbf{t_r} \overset{\text{def}}{=} \lambda \mathbf{x} : \mathbf{Bool}.\mathtt{true}.$$

Assume they are compiled to an untyped λ-calculus simply by erasing their types as follows:

$$[\![\mathbf{t_l}]\!]^{\mathsf{S}}_{\mathsf{T}} \overset{\text{def}}{=} \lambda\mathsf{x}.\mathtt{false}, \qquad [\![\mathbf{t_r}]\!]^{\mathsf{S}}_{\mathsf{T}} \overset{\text{def}}{=} \lambda\mathsf{x}.\mathtt{true}.$$

The compiled programs are contextually inequivalent; the following context in fact terminates when it interacts with $[\![\mathbf{t_l}]\!]^{\mathsf{S}}_{\mathsf{T}}$ and it diverges when it interacts with $[\![\mathbf{t_r}]\!]^{\mathsf{S}}_{\mathsf{T}}$:

$$\mathbb{C}_{\mathsf{T}} \overset{\text{def}}{=} \mathsf{if}\ ([\cdot]\mathtt{true})\ \mathsf{then\ unit\ else}\ \Omega$$

$$\begin{array}{ll} \mathbb{C}_{\mathsf{T}}[[\![\mathbf{t_l}]\!]^{\mathsf{S}}_{\mathsf{T}}] & \qquad \mathbb{C}_{\mathsf{T}}[[\![\mathbf{t_r}]\!]^{\mathsf{S}}_{\mathsf{T}}] \\ \rightarrow \mathsf{if}\ (\mathtt{false})\ \mathsf{then\ unit\ else}\ \Omega & \qquad \rightarrow \mathsf{if}\ (\mathtt{true})\ \mathsf{then\ unit\ else}\ \Omega \\ \rightarrow \Omega \rightarrow^{*} \Omega \cdots & \qquad \rightarrow \mathsf{unit} \end{array}$$

In this case, we can generate a source-level context that also witnesses the difference in behaviour between $\mathbf{t_l}$ and $\mathbf{t_r}$ as follows:

$$\mathbb{C}_{\mathbf{S}} \overset{\text{def}}{=} \mathbf{if}\ ([\cdot]\mathtt{true})\ \mathbf{then\ unit\ else}\ \Omega.$$

To prove preservation of contextual equivalence, we need to prove that such a back-translated context $\mathbb{C}_{\mathbf{S}}$ exists *for any* $\mathbb{C}_{\mathsf{T}}$ that can differentiate $[\![\mathbf{t_l}]\!]^{\mathsf{S}}_{\mathsf{T}}$ and $[\![\mathbf{t_r}]\!]^{\mathsf{S}}_{\mathsf{T}}$.

B.2 Back-translation by Partial Evaluation

The first technique to back-translate target contexts is partial evaluation. This technique rests on the idea that only the interaction between the context and the compiled terms are of interest, and it is precisely those interactions that the back-translated context must capture. Whatever reductions the target context was doing internally does not matter.

Two ways exist to make sure that these context-internal reductions do not matter: considering only normal-form contexts (Appendix B.2.1) and relying on target-level trace semantics (Appendix B.2.2).

B.2.1 Type-directed back-translation by partial evaluation. Ahmed and Blume [18] and, later, Bowman and Ahmed [26] both make use of a type-directed back-translation by partial evaluation, where a key idea is that only subterms of translation type need to be back-translated. The technique requires a complex well-foundedness argument as the back-translation is not inductively defined.

Back-translation by partial evaluation relies on the fact that in some languages, if a term has translation type, then any uses of non-translation type in a subterm are inessential and can therefore be eliminated by partial evaluation. However, this is not a realistic assumption for non-terminating languages or languages with information hiding. For instance, a diverging program may do arbitrary computation using non-translation type that cannot be eliminated by partial evaluation. Furthermore, this technique alone will not work when both languages have state or existential types, since programs of target type can use values of non-translation type in their closure or as the existential witness. For this reason, back translation by partial evaluation has only been applied to purely functional, normalizing languages. However, it has the benefit of being fairly systematic when it is applicable.

B.2.2 Back-translation via Traces. This kind of back-translation can be done when target-level trace equivalence is used to replace target-level contextual equivalence, and the preservation statement is expressed as follows:

$$\mathit{Preservation}\ (\mathit{contrapositive}) = \forall \mathbf{P_1}, \mathbf{P_2} \in \mathbf{S}, [\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}} \not\overset{\mathsf{T}}{=} [\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}} \Rightarrow \mathbf{P_1} \not\simeq_{\mathsf{ctx}} \mathbf{P_2}.$$

Since $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}} \not\overset{\mathsf{T}}{=} [\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$, we know that the traces describing the behaviour of $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}}$ is different from the traces of $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$. So there is a single trace $\overline{\lambda}_{\mathit{diff}}$ that is in the trace semantics of $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}}$ and not in the trace semantics of $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$. By definition, $\overline{\lambda}_{\mathit{diff}} \equiv \overline{\lambda}\lambda'!$ and there exist a trace in the traces of $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$ that has the form $\overline{\lambda}\lambda''!$ with $\lambda'! \neq \lambda''!$. Trace $\overline{\lambda}$ is called the *common prefix* (note that it can be as small as a single action) that accounts for possibly equivalent behaviour of $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}}$ and $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$. Single actions $\lambda'!$ and $\lambda''!$ are called *differentiating actions*.

The back-translation (often called "algorithm" in the literature [14, 62, 64, 98, 99, 101, 102]) must produce a context $\mathbb{C}_{\mathsf{S}}$ that performs all interactions in the common prefix and then reacts to the differentiating actions in two different ways. More precisely, the context $\mathbb{C}_{\mathsf{S}}$ must preform all ?-decorated actions in $\overline{\lambda}$, since the !-decorated ones are done by $\mathbf{P_1}$ or $\mathbf{P_2}$.

B.3 Back-translation by Embedding

The second technique to back-translate target contexts is to embed them into source ones. The kind of embedding required depends, intuitively, on the gap in expressive power between the source and target languages of the translation. Back-translation is easiest when the source and target are syntactically identical and gets harder as the target language contains features not directly expressible in the source.

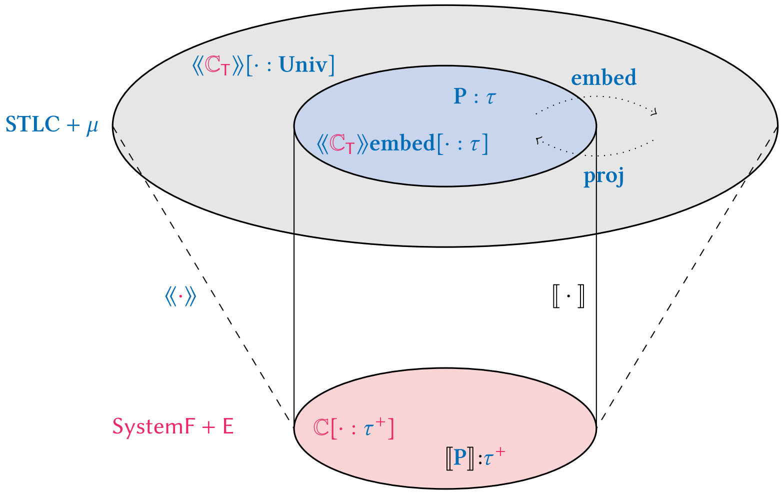

$\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle[\cdot : \mathbf{Univ}]$
$\mathsf{STLC} + \mu$
$\mathbf{P} : \tau$
embed
$\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle\mathbf{embed}[\cdot : \tau]$
proj
$\langle\!\langle \cdot \rangle\!\rangle$
$[\![\cdot]\!]$
$\mathsf{SystemF} + \mathsf{E}$
$\mathbb{C}[\cdot : \tau^+]$
$[\![\mathbf{P}]\!]{:}\tau^+$

Fig. 6. Diagrammatic representation of the compiler ($[\![\cdot]\!]$) and of the precise backtranslation ($\langle\!\langle \cdot \rangle\!\rangle$) of New et al. [94].

B.3.1 Precise Embedding. In the work of Ahmed and Blume [18] and Fournet, Swamy, Chen, Dagand, Strub, and Livshits [52], the target language is the same as the source. Hence, the back-translation can be done using boundaries—called "wrappers"—encoded in the same language. These wrappers are witness to a type isomorphism and consequently the translation is fully abstract. New et al. [94] characterize this form of back-translation as *precise*, because the embedding of the target language is into isomorphic types.

B.3.2 Over-approximating Embedding. New et al. [94] prove full abstraction of closure conversion from the simply typed lambda calculus with recursive types to a target language with type abstraction and a modal type system to track exceptions. These languages are, respectively, called $\mathsf{STLC} + \mu$ and $\mathsf{SystemF} + \mathsf{E}$ in the explanatory Figure 6. Since they have recursive types in the source, they back-translate target contexts $\mathbb{C}_\mathsf{T}$ (denoted as $\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle$) to a universal type $\mathbf{Univ}$ and show that the boundaries between types τ and $\mathbf{Univ}$ are retractions. These boundaries are two functions: $\mathbf{proj} : \mathbf{Univ} \rightarrow \tau$, which projects from the universal type into normal types, and $\mathbf{embed}{:}\tau \rightarrow \mathbf{Univ}$, which embeds from a normal type into the universal one, such that the following holds:

$$\mathbf{proj}\ \mathbf{embed}\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle \simeq_{\mathrm{ctx}} \langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle.$$

Their back-translation is over-approximating in that it embeds the target language into the source at types that include many more behaviours (the grey area in Figure 6), but due to the boundaries the code is only run on "good" values, i.e., those that represent source values.

B.3.3 Under-approximating Embedding. Devriese et al. [37] present a translation from the simply typed lambda calculus with recursive functions to the untyped lambda calculus. To prove that the translation is fully abstract, they must back-translate the untyped language to a simply typed language *without* recursive types, so they cannot construct a universal type like New et al. [94]. However, they can construct arbitrarily large approximations to the universal type, and a family of increasingly precise approximations to it. They show that since for any particular program and context observing termination only takes a finite number of steps, they can find a large enough approximation (based on the number of steps) to show that equivalence is preserved. We say their back-translation is *under-approximating*, since it embeds the target language at types that include only a subset of the behaviours of the target.

Such a technique is generally useful when the source types are less expressive than the target by resorting to additional information present in the formal tools (e.g., steps). For example, when compiling System F to an untyped lambda calculus, the back-translation type needs to account for type variables and their instantiation. This turns the universal type into a universal type operator, which is expressible in System-ω but not in System-F. To overcome these kinds of issues, under-approximating back-translation can be used.[2]

Independently from the work of Devriese et al. [37], the idea of using an approximate back-translation was also mentioned recently by Schmidt-Schauß et al. [115]. In this work, the authors present a framework for reasoning about fully abstract compilation and related notions using families of translations, i.e., an approximate back-translation. They apply the idea to show that a simply-typed lambda calculus without fix but with stuck terms can be embedded into a simply-typed lambda calculus with fix. Although not very detailed, their proof seems simpler than that of Devriese et al. [37]. This is partly because the proof addresses a simpler problem, but the idea of approximate back-translation also seems simpler to use for a language embedding. This suggests that proofs based on under-approximate back-translation can be simplified by factoring the proof into two separate passes:

(1) embedding STLC into STLC$^\mu$ (i.e., with recursive types) and using an under-approximate embedding;
(2) compiling STLC$^\mu$ with recursive types into the untyped lambda calculus using an over-approximating embedding.

To provide further clarification of this proof technique, we provide a picture from the work of Devriese et al. [37].

$$\begin{tikzcd}
 \arrow[dddd, Rightarrow, "\text{approx. compiler security}"] & & & \mathbf{t_1} \simeq_{\mathrm{ctx}} \mathbf{t_2} & & \\
& & \langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n}\left[\mathbf{t_1}\right] \Downarrow_{\_} \arrow[rr, Rightarrow, "(2)"'] & & \langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n}\left[\mathbf{t_2}\right] \Downarrow_{\_} \arrow[dd, Rightarrow, "(3)"'] & \\
& \begin{array}{l} \langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n} \gtrsim_n \mathbb{C}_\mathsf{T} \\ \mathbf{t_1} \gtrsim_{\_} \llbracket \mathbf{t_1} \rrbracket \end{array} & & & & \begin{array}{l} \langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n} \lesssim_{\_} \mathbb{C}_\mathsf{T} \\ \mathbf{t_2} \lesssim_{\_} \llbracket \mathbf{t_2} \rrbracket \end{array} \\
& & \mathbb{C}_\mathsf{T}\left[\llbracket \mathbf{t_1} \rrbracket\right] \Downarrow_\mathsf{n} \arrow[uu, Rightarrow, "(1)"'] \arrow[rr, Rightarrow, "?"] & & \mathbb{C}_\mathsf{T}\left[\llbracket \mathbf{t_2} \rrbracket\right] \Downarrow_{\_} & \\
 & & & \llbracket \mathbf{t_1} \rrbracket \overset{?}{\simeq}_{\mathrm{ctx}} \llbracket \mathbf{t_2} \rrbracket & &
\end{tikzcd}$$

This picture proves the hard part of compiler full abstraction by dividing it into three substeps. The approximation relation is indicated with $\lesssim$ and $\gtrsim$ to express that a term (or context) $\mathbf{t}$ terminates whenever t does ($\mathbf{t} \gtrsim \mathsf{t}$) and vice versa ($\mathbf{t} \lesssim \mathsf{t}$). The approximation is equipped with a subscript indicating how many steps it is known to hold for.

The proof states that given a target context $\mathbb{C}_\mathsf{T}$, we can construct its back-translation $\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n}$ that approximates $\mathbb{C}_\mathsf{T}$ for n steps. This, together with the knowledge that $\mathbf{t} \gtrsim \llbracket \mathbf{t} \rrbracket$, lets us deduce implication (1). Implication (2) follows directly from the source terms being contextually equivalent.

The second condition on the back translated context approximation $\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n}$ is that it is *conservative*, to deduce implication (3). Conservativeness means that the source-level context may diverge in situations where the original did not, but not vice versa, as expressed by $\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n} \lesssim_{\_} \mathbb{C}_\mathsf{T}$. This implies that if $\langle\!\langle \mathbb{C}_\mathsf{T} \rangle\!\rangle_\mathbf{n}[\mathbf{t}]$ terminates in any number of steps, then so must $\mathbb{C}_\mathsf{T}[\llbracket \mathbf{t} \rrbracket]$.

[2] Personal communication with Devriese, Patrignani, and Piessens, who are devising a fully abstract compiler from System F to lambda seal [108].