Full Abstraction for Compiling From a Total to a Partial Language[*]

Hyeyoung Shin
Northeastern University
hyeyoungshinw@gmail.com

1 Background and Motivation

Programmers reason about a program under guarantees made by the the programming language. For example, Coq is a *total language* (all programs terminate), so a Coq programmer could write

$$\lambda x : \mathsf{unit} \to \tau.\, x\ ();\mathsf{true} \tag{1}$$

$$\lambda x : \mathsf{unit} \to \tau.\, \mathsf{true} \tag{2}$$

and assume that these two programs are (extensionally) equivalent, because there is no input f of type $\mathsf{unit} \to \tau$ that distinguishes them. Consider $[\cdot]\ f$ as a "context" whose hole $[\cdot]$ can be filled (or linked) with the two functions above. When no context can observe a difference between two program components, e_1 and e_2, we say e_1 and e_2 are *contextually equivalent*. For the rest of this abstract, equivalence means contextual equivalence.

However, Coq programs compile down to **OCaml**, a language which is *not* total and makes no termination guarantees. An **OCaml** context $[\cdot]\ \mathbf{diverge}_\tau$ [1] can break the equivalence on which the programmer's reasoning may rely. When such reasoning guarantee is broken by the target context, programming errors are inevitable, and such errors can cause serious breaches in a program's security.

We seek a compiler that respects programmers' reasoning in the source language. Language-based security research has emphasized the importance of the compilers that are not only correct but also equivalence preserving. We call such compilers fully abstract. More formally, a compiler is *fully abstract* if it has the following properties:
equivalence preservation: if two source program components are equivalent, their corresponding translation in the target are equivalent;
equivalence reflection: if two translated target program components are equivalent, their corresponding source program components are equivalent.

A fully abstract compilation is secure because the source and translated programs are indistinguishable by any observation in the target language where linking with additional adversarial contexts is possible. Therefore, it empowers programmers to reason about their programs in the source language only (which is the way it should be!).

Previous work on secure compilation includes full abstraction of typed closure conversion between the same source and target languages (both total) [2], CPS transformation from simply typed λ-calculus (STLC) to System F (both total) [3], closure conversion from STLC to polymorphic λ-calculus (both partial) [6]. However, we are not aware of any prior results about fully abstract compilation *from a total source language to a partial (i.e., nontotal) target language*, despite the fact that such compilation occurs in practice (e.g., Coq-to-**OCaml** compilation).

In this paper, we show the first fully abstract compiler from a total source language to a partial target language. The source language we consider here is STLC with unit and boolean. The target is STLC with unit, boolean, recursive type, and modal types. The translation is the identity on terms and the identity with pure effect annotations on types, which is trivially type-preserving.

2 Main Idea

Our proof of full abstraction is simpler than the standard proofs that use proof by contradiction via "back-translation". A back-translation is a translation from target to source. The argument goes: suppose the two contextually equivalent source terms e_1 and e_2 translate to two non-contextually equivalent target terms $\mathbf{e_1}$ and $\mathbf{e_2}$. If the target context that distinguishes $\mathbf{e_1}$ and $\mathbf{e_2}$ can be back-translated to a source context that distinguishes e_1 and e_2, then it is a contradiction [2].

However, coming up with an appropriate back-translation is often nontrivial. In our case, we would have to come up with a way to backtranslate a nonterminating target context to source in which nonterminating computation is not expressible.

Instead, we found a novel way to avoid back-translation by using the logical equivalence between $\beta\eta$-equality and contextual equivalence [7] (*i.e.* contextual equivalence is included in $\beta\eta$-equality and vice versa.) Using this result, it suffices to show that our compiler preserves $\beta\eta$-equality. Unfortunately, the idea of using $\beta\eta$-equality instead of contextual equivalence does not generalize to languages with recursive and higher-order types. Therefore, for our target language we define a logical relation that is sound and complete with respect to contextual equivalence and use it as a way of testing target level equivalence. In future work, we hope to show that the identity compiler preserves and reflects contextual equivalence from System F to System F with

[*]We use colors to distinguish source (blue) and target (red) languages.
[1]$\mathbf{diverge}_\tau = \lambda\ \_ :\ \mathbf{unit}.\mathrm{fix}_\tau(\lambda x : \tau.x)$

recursive type, perhaps by applying the back-translation method.

3 Languages

The source language is STLC. The target language is STLC plus recursive type and modal types.

$$\begin{array}{llcl} \textit{Type} & \phi & ::= & \tau \mid \sigma \\ \textit{Value Type} & \tau & ::= & \mathbf{unit} \mid \mathbf{bool} \mid \tau \rightarrow \sigma \mid \alpha \mid \mu\alpha.\tau \\ \textit{Computation Type} & \sigma & ::= & \mathrm{E}^{\delta}\tau \\ \textit{Effect} & \delta & ::= & \circ \mid \bullet \end{array}$$

In the target's modal type system, terms are divided into values, with value types τ, and computations, with computation types $\mathrm{E}^{\{\circ \vee \bullet\}}\tau$. Computation types are value types with effect annotations. In our case, pure effects $\circ$ represents termination and impure effects $\bullet$ possible nontermination. Modal types play a crucial role in disallowing a compiled source program to link with impure target contexts. However, they also add extra work in proofs because for each value type τ there are two cases $\mathrm{E}^{\circ}\tau$ and $\mathrm{E}^{\bullet}\tau$ to prove.

We handle this by noticing that the behavior of impure computation includes that of pure computation. In consequence, a pure type $\mathrm{E}^{\circ}\tau$ is a subtype of the impure type $\mathrm{E}^{\bullet}\tau$ and a property we prove about a pure type implies that about the impure type. Also, following typing rules facilitate crossing values over to computations and pure computations over to impure computations.

$$\frac{\Gamma \vdash_{\mathrm{v}} \mathbf{v} : \tau}{\Gamma \vdash_{\mathrm{e}} \mathbf{v} : \mathrm{E}^{\circ}\tau} \qquad \frac{\Gamma \vdash_{\mathrm{e}} \mathbf{e} : \mathrm{E}^{\circ}\tau}{\Gamma \vdash_{\mathrm{e}} \mathbf{e} : \mathrm{E}^{\bullet}\tau}$$

Translation$(+, \div)$ Our compiler is the identity on terms and the identity with pure type annotations on types. The term translation, $+$, maps a source term e to the corresponding target term $\mathbf{e} = e^{+}$. Here is the function translation:

$$\frac{\Gamma, x : \tau \vdash e : \tau \rightsquigarrow \Gamma^{+}, \mathbf{x} : \tau^{+} \vdash e^{+} \ : \ \tau'^{\div}}{\Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau' \rightsquigarrow \Gamma^{+} \vdash \lambda(\mathbf{x} : \tau^{+}).\, e^{+} : \ (\tau \to \tau')^{\div}}$$

where the translated function is typed under the translated target environment via the twofold type-preserving tranlsation $+$ and $\div$. First, $+$ maps the function type $\tau \to \tau'$ to the corresponding value type $\tau^{+} \rightarrow \mathrm{E}^{\circ}\tau'^{+}$ in the target,

$$(\tau \to \tau')^{+} = \tau^{+} \rightarrow \mathrm{E}^{\circ}\tau'^{+},$$

and then $\div$ turns the value type to a pure computation type:

$$(\tau^{+} \rightarrow \mathrm{E}^{\circ}\tau'^{+})^{\div} = \mathrm{E}^{\circ}(\tau^{+} \rightarrow \mathrm{E}^{\circ}\tau'^{+}).$$

For example, (1) and (2) will be translated to

$$\lambda(\mathbf{x} : \mathbf{unit} \to \mathrm{E}^{\circ}\tau^{+}).\, \mathbf{x}\ ();\mathbf{true}$$

$$\lambda(\mathbf{x} : \mathbf{unit} \to \mathrm{E}^{\circ}\tau^{+}).\, \mathbf{true}$$

Note that the translated functions take arguments that have translation type which is always pure, ruling out linking with the adversary argument $\mathbf{diverge}_{\tau}$ that inevitably has the impure type $\mathbf{unit} \rightarrow \mathrm{E}^{\bullet}\tau$.

4 Proof Sketch

The major part of a fully abstract translation proof is showing that the translation preserves the source's contextual equivalence. However, proving this directly is difficult because we have to reason about *arbitrary* contexts. Our novel idea is to use $\beta\eta$-equality instead. This way we do not need to use proof by contradiction in which back-translating target contexts is required.

Definition 4.1 ($\lambda^{1,\mathrm{b},\to}$ $\beta\eta$-equality**).**
$\Gamma \vdash e_1 \approx_{\beta\eta} e_2 : \tau$ is the smallest equivalence relation on well-typed terms with congruence closure of $\beta\eta$-rules.

Theorem 4.2 ($\approxeq^{\mathrm{ctx}} \Leftrightarrow \approx_{\beta\eta}$**).**
$\Gamma \vdash e_1 \approxeq^{\mathrm{ctx}} e_2 : \tau$ *if and only if* $\Gamma \vdash e_1 \approx_{\beta\eta} e_2 : \tau$.

Similarly, to avoid proving about the equivalence between target terms with *arbitrary* contexts, we use a logical relation that is defined inductively on the structure of types, and is thus easier to work with. We prove that two target terms are logically related if and only if they are contextually equivalent. With this set up, we can prove equivalence preservation (downarrow on the left) indirectly (downarrow on the right) as shown in Figure 1.

$$\begin{tikzcd} (\lambda^{1,\mathrm{b},\to}, \approxeq^{\mathrm{ctx}}) \arrow[r, Leftrightarrow] \arrow[d, Rightarrow] & (\lambda^{1,\mathrm{b},\to}, \approx_{\beta\eta}) \arrow[d, Rightarrow] \\ (\lambda^{\mathbf{1},\mathbf{b},\to,\mu,\circ,\bullet}, \cong^{\mathrm{ctx}}) \arrow[r, Leftrightarrow] & (\lambda^{\mathbf{1},\mathbf{b},\to,\mu,\circ,\bullet}, \simeq_{\mathbf{lr}}) \end{tikzcd}$$

Figure 1. Equivalence diagram.

Theorem 4.3 ($\cong^{\mathrm{ctx}} \Leftrightarrow \simeq_{\mathbf{lr}}$**).**
$\mathbf{\Gamma} \vdash \mathbf{e_1} \cong^{\mathrm{ctx}} \mathbf{e_2} : \sigma$ *if and only if* $\mathbf{\Gamma} \vdash \mathbf{e_1} \simeq_{\mathbf{lr}} \mathbf{e_2} : \sigma$.

Proof. By the soundness and completeness with respect to $\lambda^{\mathbf{1},\mathbf{b},\to,\mu,\circ,\bullet}$ contextual equivalence. □

Theorem 4.4 (Full Abstraction)**.** *The translation preserves and reflects contextual equivalence.*

Proof. By lemma 4.5 and 4.6. □

Lemma 4.5 (Equivalence Preservation)**.**
If $\Gamma \vdash e_1 \approxeq^{\mathrm{ctx}} e_2 : \tau$, *then* $\Gamma^{+} \vdash e_1^{+} \cong^{\mathrm{ctx}} e_2^{+} : \tau^{\div}$.

Proof. By showing $\approx_{\beta\eta}$ implies $\simeq_{\mathbf{lr}}$. □

Proving equivalence reflection is a result of compiler correctness (or semantics preservation [5]). Intuitively, a source term and its translation are semantically equivalent since our compiler is the identity.

Lemma 4.6 (Equivalence Reflection)**.**
If $\Gamma^{+} \vdash e_1^{+} \cong^{\mathrm{ctx}} e_2^{+} : \tau^{\div}$, *then* $\Gamma \vdash e_1 \approxeq^{\mathrm{ctx}} e_2 : \tau$.

Proof. By compiler correctness. □

The complete proofs and logical relations are available in the tech report [2].

[2]https://github.com/hyeyoungshin/popl19src

5 Acknowledgements

This work is guided by Amal Ahmed and Daniel Patterson. I'd like to also thank Max S. New for contributing to the main idea, Gabriel Scherer for references, and William J. Bowman and Michael Ballantyne for their feedbacks on the writing.

References

[1] M. Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, ICALP '98, pages 868–883, Berlin, Heidelberg, 1998. Springer-Verlag. ISBN 978-3-540-68681-1. doi: https://doi.org/10.1007/BFb0055109.

[2] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 157–168, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411227. URL http://doi.acm.org/10.1145/1411204.1411227.

[3] A. Ahmed and M. Blume. An equivalence-preserving cps translation via multi-language semantics. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2011, Sept. 2011.

[4] A. Kennedy. Securing the .net programming model. *Theoretical Computer Science*, 364(3):311–317, 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.08.014. URL https://doi.org/10.1016/j.tcs.2006.08.014.

[5] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111042. URL http://doi.acm.org/10.1145/1111037.1111042.

[6] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 103–116, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951941. URL http://doi.acm.org/10.1145/2951913.2951941.

[7] R. Statman. λ-definable functionals and $\beta\eta$ conversion. *Arch. Math. Logik Grundlag.*, 23(1-2):21–26, 1983. ISSN 0003-9268. doi: 10.1007/BF02023009. URL https://doi-org.colorado.idm.oclc.org/10.1007/BF02023009.