MODULAR, FULLY-ABSTRACT COMPILATION
BY APPROXIMATE BACK-TRANSLATION [*]

DOMINIQUE DEVRIESE [a], MARCO PATRIGNANI [b,†], FRANK PIESSENS [a],
AND STEVEN KEUCHEL [c]

[a] imec-DistriNet, KU Leuven, Belgium
   e-mail address: first.last@cs.kuleuven.be

[b] MPI-SWS, Saarbrücken, Germany
   e-mail address: first.last@mpi-sws.org

[c] UGent, Belgium
   e-mail address: first.last@ugent.be

ABSTRACT. A compiler is *fully-abstract* if the compilation from source language programs to target language programs reflects and preserves behavioural equivalence. Such compilers have important security benefits, as they limit the power of an attacker interacting with the program in the target language to that of an attacker interacting with the program in the source language. Proving compiler full-abstraction is, however, rather complicated. A common proof technique is based on the *back-translation* of target-level program contexts to behaviourally-equivalent source-level contexts. However, constructing such a back-translation is problematic when the source language is not strong enough to embed an encoding of the target language. For instance, when compiling from a simply-typed λ-calculus (λ^τ) to an untyped λ-calculus (λ^u), the lack of recursive types in λ^τ prevents such a back-translation.

We propose a general and elegant solution for this problem. The key insight is that it suffices to construct an *approximate* back-translation. The approximation is only accurate up to a certain number of steps and conservative beyond that, in the sense that the context generated by the back-translation may diverge when the original would not, but not vice versa. Based on this insight, we describe a general technique for proving compiler full-abstraction and demonstrate it on a compiler from λ^τ to λ^u. The proof uses asymmetric cross-language logical relations and makes innovative use of step-indexing to express the relation between a context and its approximate back-translation. The proof extends easily to common compiler patterns such as modular compilation and, to the best of our knowledge, it is the first compiler full abstraction proof to have been fully mechanised in Coq. We believe this proof technique can scale to challenging settings and enable simpler, more scalable proofs of compiler full-abstraction.

We typeset source and target language terms in **blue** resp. pink; we recommend to view/print this paper in colour for maximum clarity.

1. Introduction

A compiler is *fully-abstract* if the compilation from source language programs to target language programs preserves and reflects behavioural equivalence [Abadi, 1999, Gorla and Nestman, 2014]. Such compilers have important security benefits. It is often realistic to assume that attackers can interact with a program in the target language, and depending on the target language this can enable attacks such as improper stack manipulation, breaking control flow guarantees, reading from or writing to private memory of other components, inspecting or modifying the implementation of a function etc. [Abadi, 1999, Kennedy, 2006, Patrignani et al., 2015, Abadi and Plotkin, 2012, Fournet et al., 2013, Agten et al., 2012]. A fully-abstract compiler is sufficiently defensive to rule out such attacks: the power of an attacker interacting with the program in the target language is limited to attacks that could also be performed by an attacker interacting with the program in the source language.

Formally, we model a compiler as a function $[\![\cdot]\!]$ that maps source language terms $\mathbf{t}$ to target language terms $[\![\mathbf{t}]\!]$. Elements of the source language are typeset in a **blue**, **bold** font, while elements of the target language are typeset in a pink, sans-serif font. Roughly, the compiler is fully-abstract, if for any two source language terms $\mathbf{t_1}$ and $\mathbf{t_2}$, we have that they are behaviourally equivalent ($\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$) if and only if their compiled counterparts are behaviourally equivalent ($[\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$) [Abadi, 1999]. The notion of behavioural equivalence used here is the canonical notion of contextual equivalence: two terms are equivalent if they behave the same when plugged into any valid context. Specifically, we take contextual equivalence to be equi-termination: $t \simeq_{ctx} t' \stackrel{\mathsf{def}}{=} \forall \mathfrak{C}, \mathfrak{C}[t] \Downarrow \iff \mathfrak{C}[t'] \Downarrow$. The universal quantification over contexts $\mathfrak{C}$ ensures that the results produced by t and t' are the same [Plotkin, 1977, Curien, 2007].

The full-abstraction property can be split into two parts: the right-to-left implication and the left-to-right implication, which we call (contextual) equivalence *reflection* and *preservation* respectively.

Equivalence reflection. ($\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} \Leftarrow [\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$) requires that if the compiler produces equivalent target programs, then the source programs must have been equivalent. In other words, non-equivalent source programs must be compiled to non-equivalent target programs. Intuitively, this property captures an aspect of compiler correctness: if programs with different source language behaviour become equivalent after compilation, the compiler must have incorrectly compiled at least one of them (this is also called adequacy of the translation by Schmidt-Schauß et al. [2015]).

We build on cross-language logical relations: a technique that has been proposed for proving compiler correctness [Hur and Dreyer, 2011, Benton and Hur, 2009, 2010]. The general idea of this approach is depicted in Fig. 1 (purposely ignoring language-specific things such as the types of the terms involved). The proof starts from the knowledge that $[\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$ and sets out to prove that $\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$. That is, for an arbitrary valid context $\mathfrak{C}$, it shows that $\mathfrak{C}[\mathbf{t_1}] \Downarrow$ if and only if $\mathfrak{C}[\mathbf{t_2}] \Downarrow$. By symmetry, it suffices to show that $\mathfrak{C}[\mathbf{t_1}] \Downarrow \Rightarrow \mathfrak{C}[\mathbf{t_2}] \Downarrow$.

The idea of the approach is to define a cross-language logical relation $\mathbf{t} \approx t$ that expresses when a compiled term t behaves as a target-level version of source-level term $\mathbf{t}$. This logical

$$\begin{array}{c}
\mathbf{t_1} \simeq^{?}_{ctx} \mathbf{t_2} \\
\mathfrak{C}[\mathbf{t_1}] \Downarrow \quad \stackrel{?}{\Rightarrow} \quad \mathfrak{C}[\mathbf{t_2}] \Downarrow \\
\mathfrak{C} \approx [\![\mathfrak{C}]\!] \qquad \qquad \qquad \qquad \mathfrak{C} \approx [\![\mathfrak{C}]\!] \\
\mathbf{t_1} \approx [\![\mathbf{t_1}]\!] \qquad \Big\| (1) \quad (3) \Big\uparrow \qquad \mathbf{t_2} \approx [\![\mathbf{t_2}]\!] \\
[\![\mathfrak{C}]\!]\big[[\![\mathbf{t_1}]\!]\big] \Downarrow \stackrel{(2)}{\Rightarrow} [\![\mathfrak{C}]\!]\big[[\![\mathbf{t_2}]\!]\big] \Downarrow \\
[\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]
\end{array}$$

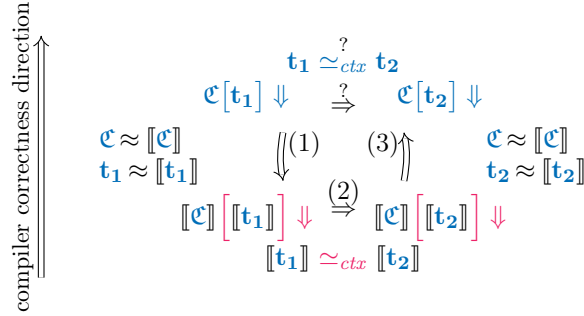(vertical label at left: *compiler correctness direction*)

Figure 1: Proving one half of full-abstraction: compiler correctness. Only one direction of this half is presented ($\Rightarrow$), the other one follows by symmetry.

relation is not compiler-specific: it should be understood as a specification of a target-level calling convention rather than precise representation choices for a specific compiler. If we can then prove that any term is logically related to its compilation ($\mathbf{t} \approx [\![\mathbf{t}]\!]$), and that the same result holds for contexts ($\mathfrak{C} \approx [\![\mathfrak{C}]\!]$), then equivalence reflection follows.[1] Starting from $\mathbf{t_1} \approx [\![\mathbf{t_1}]\!]$ and $\mathbf{t_2} \approx [\![\mathbf{t_2}]\!]$ and $\mathfrak{C} \approx [\![\mathfrak{C}]\!]$, the proof uses the inherent compositionality of logical relations to know $\mathfrak{C}[\mathbf{t_1}] \approx [\![\mathfrak{C}]\!]\big[[\![\mathbf{t_1}]\!]\big]$ and the same for $\mathbf{t_2}$. If the logical relations are constructed adequately, then related terms necessarily equi-terminate. Thus, $\mathfrak{C}[\mathbf{t_1}] \Downarrow$ iff $[\![\mathfrak{C}]\!][\![\mathbf{t_1}]\!]\Downarrow$ and similarly for $\mathbf{t_2}$. In particular, this yields the implications (1) and (3) in Fig. 1. Since implication (2) follows directly from the hypothesis of (contextual) equivalence for $[\![\mathbf{t_1}]\!]$ and $[\![\mathbf{t_2}]\!]$, the proof for equivalence reflection is finished.

This direction of compiler full-abstraction is often called *compiler correctness*, as a compiler that is correct trivially has this property. However, it is important to note that full abstraction alone does not yield correctness as it only talks about equivalence classes and not about respecting a cross-language relation that encodes the meaning of compiled code. For example, a fully-abstract compiler can swap how `true` and `false` are compiled (and how compiled boolean operators use booleans) and still be fully abstract. However, this intuitively violates correctness, that intuitively tells that `true` is compiled to `true`. This is why often, fully-abstract compilers are also proven to be correct (as we also do), to both get one half of full abstraction and be sure to respect the intended meaning of compiled programs.

Equivalence preservation. ($\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} \Rightarrow [\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$) requires that equivalent programs remain equivalent after compilation. This means that no matter what target-level manipulations are done on compiled programs, the programs must behave equivalently if the source programs were equivalent. This precludes all sorts of target-level attacks that break source-level guarantees.

If the source language is strong enough, it is possible to apply a strategy analogous to proving equivalence reflection for proving preservation, as depicted in Fig. 2.[2] Given an arbitrary target-level context $\mathfrak{C}$, we need to prove that $\mathfrak{C}[[\![\mathbf{t_1}]\!]]\Downarrow$ implies $\mathfrak{C}[[\![\mathbf{t_2}]\!]]\Downarrow$. In

---

[1] As contexts $\mathfrak{C}$ are also programs, they can be compiled with the same compiler for terms.

[2] Actually, both Figs. 2 and 3 are simplifications. Perceptive readers may notice that the proof depicted here would falsely imply equivalence preservation for *any* correct compiler. We correct the simplifications in Section 5.6.
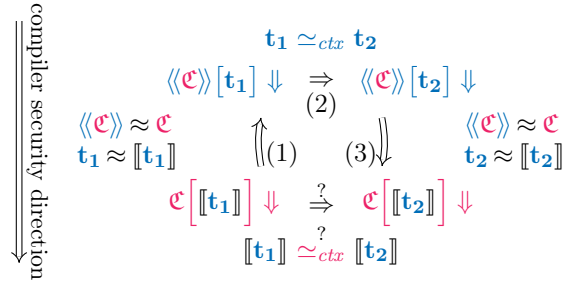
$$\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$$

$$\langle\!\langle \mathfrak{C} \rangle\!\rangle [\mathbf{t_1}] \Downarrow \quad \underset{(2)}{\Rightarrow} \quad \langle\!\langle \mathfrak{C} \rangle\!\rangle [\mathbf{t_2}] \Downarrow$$

$$\langle\!\langle \mathfrak{C} \rangle\!\rangle \approx \mathfrak{C} \qquad\qquad\qquad\qquad \langle\!\langle \mathfrak{C} \rangle\!\rangle \approx \mathfrak{C}$$
$$\mathbf{t_1} \approx [\![\mathbf{t_1}]\!] \qquad (1) \qquad (3) \qquad \mathbf{t_2} \approx [\![\mathbf{t_2}]\!]$$

$$\mathfrak{C}\big[[\![\mathbf{t_1}]\!]\big] \Downarrow \quad \overset{?}{\Rightarrow} \quad \mathfrak{C}\big[[\![\mathbf{t_2}]\!]\big] \Downarrow$$

$$[\![\mathbf{t_1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{t_2}]\!]$$

(left margin, vertical) compiler security direction

Figure 2: Proving the other half of full-abstraction: compiler security.

a sufficiently-powerful source language, we can construct a *back-translation* $\langle\!\langle \mathfrak{C} \rangle\!\rangle$ for any target-level context $\mathfrak{C}$. Using the same logical relation as above, it then suffices to prove that $\langle\!\langle \mathfrak{C} \rangle\!\rangle$ is a valid source-level context and that $\langle\!\langle \mathfrak{C} \rangle\!\rangle \approx \mathfrak{C}$ for any valid context $\mathfrak{C}$. Together with $\mathbf{t_1} \approx [\![\mathbf{t_1}]\!]$, and similarly for $\mathbf{t_2}$, compositionality and adequacy of the logical relation then yield implications (1) and (3) in the Figure. The remaining implication (2) follows from the assumed (contextual) equivalence of $\mathbf{t_1}$ and $\mathbf{t_2}$.

Constructing a back-translation of contexts is not easy, but it can be done if the source language is sufficiently expressive. Consider, for example, a compiler that translates terms from a simply-typed λ-calculus *with* recursive types ($\lambda^{\tau;\mu}$) to an untyped λ-calculus (λ^{u}). Constructing a back-translation of target-level contexts can be done based on a $\lambda^{\tau;\mu}$ type that can represent arbitrary λ^{u} values. Particularly, we can encode the unitype of λ^{u} values in a type UVal as follows:

$$\mathrm{UVal} \overset{\mathsf{def}}{=} \mu\alpha. \, \mathcal{B} \uplus (\alpha \times \alpha) \uplus (\alpha \uplus \alpha) \uplus (\alpha \rightarrow \alpha)$$

given that λ^{u} has base values of type $\mathcal{B}$, pairs, coproducts and lambdas. In other words, all λ^{u} values can be represented as $\lambda^{\tau;\mu}$ values of type UVal. We can then construct a back-translation of λ^{u} contexts to $\lambda^{\tau;\mu}$ contexts such that the latter work with values in UVal wherever the original λ^{u} contexts work with arbitrary λ^{u} values.

Contributions of this paper. If the types of the source language are not powerful enough to embed an encoding of target terms, is it possible to have a fully-abstract compiler between those languages? In this paper we answer positively to this question and develop a general technique for proving this. We instantiate this proof technique and develop a fully-abstract compiler from a simply-typed λ-calculus *without* recursive types (λ^{τ}) to an untyped λ-calculus (λ^{u}). With such a source language, we cannot construct a type like UVal to represent the values that a λ^{u} context works with. Fortunately, we can solve this problem by observing that a fully accurate back-translation is sufficient for the proof but in fact not necessary. An *approximate* back-translation is enough for the full-abstraction proof to work, without sacrificing the overall simplicity and elegance of the proof technique. The basic idea is depicted in Fig. 3. The differences from Fig. 2 are the use of asymmetric logical relations $\lesssim$ and $\gtrsim$ (also known as logical approximations) to express (roughly) that a term (or context) $\mathbf{t}$ terminates whenever t does ($\mathbf{t} \gtrsim \mathsf{t}$) and vice versa ($\mathbf{t} \lesssim \mathsf{t}$) and the addition of subscripts n where logical approximations hold only up to a limited number of steps n. Note that n in the figure is defined as the number of steps in the evaluation $\mathfrak{C}[[\![\mathbf{t_1}]\!]]\Downarrow_{\mathsf{n}}$ and that we write _ for an unknown number of steps.

$$\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$$

$$\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}}[\mathbf{t_1}] \Downarrow_{\_} \quad \Rightarrow \quad \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}}[\mathbf{t_2}] \Downarrow_{\_}$$
$$(2)$$

$$\begin{array}{ccc} \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}} \gtrsim_{\mathsf{n}} \mathfrak{C} & & \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}} \lesssim_{\_} \mathfrak{C} \\ \mathbf{t_1} \gtrsim_{\_} [\![\mathbf{t_1}]\!] & \Big\Uparrow (1) \qquad (3) \Big\Downarrow & \mathbf{t_2} \lesssim_{\_} [\![\mathbf{t_2}]\!] \end{array}$$

$$\mathfrak{C}\Big[[\![\mathbf{t_1}]\!]\Big] \Downarrow_{\mathsf{n}} \quad \overset{?}{\Rightarrow} \quad \mathfrak{C}\Big[[\![\mathbf{t_2}]\!]\Big] \Downarrow_{\_}$$

$$[\![\mathbf{t_1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{t_2}]\!]$$

approx. compiler security
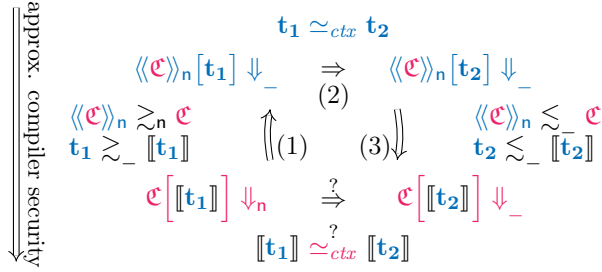
Figure 3: Proving equivalence preservation using an *n-approximate* back-translation. An _ subscript indicates *any* number of steps.

The proof starts, again, from an arbitrary target-level context $\mathfrak{C}$ and the knowledge that $\mathfrak{C}[[\![\mathbf{t_1}]\!]] \Downarrow_{\mathsf{n}}$. We then construct a λ^{τ} context $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}}$ that satisfies two conditions. First, it approximates $\mathfrak{C}$ *up to n steps*: $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}} \gtrsim_{\mathsf{n}} \mathfrak{C}$. This means that if $\mathfrak{C}[\mathbf{t}]$ terminates in less than n steps then $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}}[\mathbf{t}]$ will also terminate for a term $\mathbf{t}$ related to $\mathbf{t}$. This, together with the knowledge that $\mathbf{t} \gtrsim [\![\mathbf{t}]\!]$, allows us to deduce implication (1) in the figure. As before, implication (2) follows directly from the (contextual) equivalence of $\mathbf{t_1}$ and $\mathbf{t_2}$.

Then we use a second condition on the n-approximation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}}$, namely that it is *conservative*, to deduce implication (3). Intuitively, the source-level context produced by the n-approximation may diverge in situations where the original did not, but not vice versa. Intuitively, the divergence will occur when the precision n of the approximate back-translation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathbf{n}}$ is not sufficient for the context to accurately simulate the behaviour of $\mathfrak{C}$. This is expressed by the logical approximation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}} \lesssim \mathfrak{C}$ which implies that if $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}}[\mathbf{t}]$ terminates (in any number of steps), then so must $\mathfrak{C}[\mathbf{t}]$. This allows us to deduce implication (3).

The advantage of this approximate back-translation approach is that it can be easier to construct a conservative approximate back-translation than a full one. For example, considering λ^{τ} without recursive types, we can construct a family of λ^{τ} types $\mathrm{UVal}_{\mathsf{n}}$, indexed by non-negative numbers n:

$$\mathrm{UVal}_0 \overset{\mathsf{def}}{=} \mathtt{Unit}$$

$$\mathrm{UVal}_{\mathsf{n}+1} \overset{\mathsf{def}}{=} \begin{array}{l} \mathtt{Unit} \uplus \mathcal{B} \uplus (\mathrm{UVal}_{\mathsf{n}} \times \mathrm{UVal}_{\mathsf{n}}) \uplus \\ (\mathrm{UVal}_{\mathsf{n}} \uplus \mathrm{UVal}_{\mathsf{n}}) \uplus (\mathrm{UVal}_{\mathsf{n}} \to \mathrm{UVal}_{\mathsf{n}}). \end{array}$$

Without giving full details here, $\mathrm{UVal}_{\mathbf{n}}$ is an n-level unfolding of UVal with additional unit values at every level to represent failed approximations. This approximate version of UVal is enough to construct a conservative n-approximate back-translation of an untyped program context, and as such, it allows us to circumvent the lack of expressiveness of λ^{τ} without recursive types.

In order to make this approximate back-translation approach work, we need a way to formalise the relation between an untyped context and its approximate back-translation. However, it turns out that existing well-known techniques from the field of logical relations are almost directly applicable. Asymmetric logical relations (like $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\mathsf{n}} \lesssim \mathfrak{C}$ above) are a well-established technique. More interestingly, the approximateness of the relation can very naturally be expressed using step-indexed logical relations. Despite this naturality, it appears that this use of step-indexing is novel. The technique is normally used as a way to construct

well-founded logical relations and one is not actually interested in terms being related only up to a limited number of steps.

An earlier version of this paper, published at POPL 2016, introduced the technique of approximate back-translations, and applied it to prove full abstraction for a whole-program compiler from λ^τ to λ^u [Devriese et al., 2016]. This journal version extends the conference version in two ways. First, we extend the full abstraction proof to a *modular* compiler from λ^τ to λ^u; i.e., one that operates on open programs (or, components) and links them together after compilation. This is how most modern compilers operate for efficiency reasons, so this furthers our belief that this proof technique scales to real-world compilers. Moreover, the work required to extend the proof to a modular compiler is relatively small, so this provides evidence of the broader applicability of our proof technique. Finally, proving modular full abstraction yields that the compiler ensures component-based compartmentalisation, which provides more fine-grained security guarantees than plain full abstraction.

Second, the original proof has been completely mechanised in Coq, providing additional assurance about the correctness of our results. Additionally, this highlights that the reasoning principle behind our proof technique is amenable to mechanisation. To the best of our knowledge, this is the first fully mechanised proof of compiler full-abstraction.

To summarise, the contributions of this work are:

- a new and general proof technique for proving compiler modular full-abstraction using asymmetric, cross-language logical relations and targeting untyped languages;
- a fully mechanised instantiation of that proof technique showing full abstraction of a modular compiler from a simply-typed λ-calculus without recursive types to the untyped λ-calculus;
- a novel application of step-indexed logical relations for expressing approximateness of a back-translation.

This paper is structured as follows. First, we formalise the source and target languages λ^τ and λ^u (Section 2). Second, we present the cross-language logical relations that we use to express the relation between λ^τ terms and their compilations as well as between λ^u contexts and their back-translation (Section 3). We define the compiler in Section 4. It applies type erasure and dynamic type wrappers that enforce the requirements and guarantees of λ^τ types during execution. We then present the approximate back-translation (Section 5) which we use to prove compiler full-abstraction (Section 6). Then we present how to scale the proof technique to modular compilers (Section 7). Finally, we discuss the mechanisation of the proofs (Section 8). We then offer some discussion (Section 9), compare with related work (Section 10) and conclude (Section 11).

2. Source and Target Languages

The source language λ^τ is presented in Fig. 4. It is a strict, simply-typed λ-calculus with Unit, Bool, lambdas, product and sum types and a fix operator providing general recursion. The figure presents the syntax of terms $\mathbf{t}$, values $\mathbf{v}$, types τ, typing contexts $\mathbf{\Gamma}$ and evaluation contexts $\mathbb{E}$. We indicate the reduction relation with $\hookrightarrow$; we define that a term $\mathbf{t}$ terminates ($\mathbf{t} \Downarrow$) if it reduces in a finite number of steps to a value: $\mathbf{t} \Downarrow \overset{\mathsf{def}}{=} \exists \mathbf{n}, \mathbf{v}.\mathbf{t} \hookrightarrow^\mathbf{n} \mathbf{v}$. Apart from the type and evaluation rules for $\mathrm{fix}_{\tau_1 \to \tau_2}$, the typing rules and evaluation rules are standard. The evaluation rules use evaluation contexts to impose a strict evaluation order. The type and evaluation rule for $\mathrm{fix}_{\tau_1 \to \tau_2}$ are somewhat special compared to a more standard definition (see

$$\mathbf{t} ::= \mathtt{unit} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda \mathbf{x} : \tau.\ \mathbf{t} \mid \mathbf{x} \mid \mathbf{t}\ \mathbf{t} \mid \mathbf{t.1} \mid \mathbf{t.2} \mid \langle \mathbf{t}, \mathbf{t} \rangle \mid \mathrm{inl}\ \mathbf{t} \mid \mathrm{inr}\ \mathbf{t}$$
$$\mid \mathrm{case}\ \mathbf{t}\ \mathrm{of}\ \mathrm{inl}\ \mathbf{x_1} \mapsto \mathbf{t} \mid \mathrm{inr}\ \mathbf{x_2} \mapsto \mathbf{t} \mid \mathbf{t}; \mathbf{t} \mid \ \mathrm{if}\ \mathbf{t}\ \mathrm{then}\ \mathbf{t}\ \mathrm{else}\ \mathbf{t} \mid \mathrm{fix}_{\tau \to \tau}\ \mathbf{t}$$
$$\mathbf{v} ::= \mathtt{unit} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda \mathbf{x} : \tau.\ \mathbf{t} \mid \langle \mathbf{v}, \mathbf{v} \rangle \mid \mathrm{inl}\ \mathbf{v} \mid \mathrm{inr}\ \mathbf{v}$$
$$\tau ::= \mathtt{Unit} \mid \mathtt{Bool} \mid \tau \to \tau \mid \tau \times \tau \mid \tau \uplus \tau$$
$$\mathbf{\Gamma} ::= \emptyset \mid \mathbf{\Gamma}, \mathbf{x} : \tau$$
$$\mathbb{E} ::= [\cdot] \mid \mathbb{E}\ \mathbf{t} \mid \mathbf{v}\ \mathbb{E} \mid \mathbb{E}.\mathbf{1} \mid \mathbb{E}.\mathbf{2} \mid \langle \mathbb{E}, \mathbf{t} \rangle \mid \langle \mathbf{v}, \mathbb{E} \rangle \mid \mathrm{inl}\ \mathbb{E} \mid \mathrm{inr}\ \mathbb{E} \mid \mathrm{case}\ \mathbb{E}\ \mathrm{of}\ \mathrm{inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \mid \mathrm{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2}$$
$$\mid \mathbb{E}; \mathbf{t} \mid \mathrm{if}\ \mathbb{E}\ \mathrm{then}\ \mathbf{t}\ \mathrm{else}\ \mathbf{t} \mid \mathrm{fix}_{\tau \to \tau}\ \mathbb{E}$$

$$\frac{}{\mathbf{\Gamma} \vdash \mathtt{unit} : \mathtt{Unit}} \qquad \frac{}{\mathbf{\Gamma} \vdash \mathtt{true} : \mathtt{Bool}} \qquad \frac{(\mathbf{x} : \tau) \in \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash \mathbf{x} : \tau}$$

$$\frac{\mathbf{\Gamma}, (\mathbf{x} : \tau) \vdash \mathbf{t} : \tau'}{\mathbf{\Gamma} \vdash \lambda \mathbf{x} : \tau.\ \mathbf{t} : \tau \to \tau'} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{t_1} : \tau_1 \quad \mathbf{\Gamma} \vdash \mathbf{t_2} : \tau_2}{\mathbf{\Gamma} \vdash \langle \mathbf{t_1}, \mathbf{t_2} \rangle : \tau_1 \times \tau_2} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau_1 \times \tau_2}{\mathbf{\Gamma} \vdash \mathbf{t.1} : \tau_1}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau' \to \tau \quad \mathbf{\Gamma} \vdash \mathbf{t'} : \tau'}{\mathbf{\Gamma} \vdash \mathbf{t}\ \mathbf{t'} : \tau} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau}{\mathbf{\Gamma} \vdash \mathrm{inl}\ \mathbf{t} : \tau \uplus \tau'}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{t} : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)}{\mathbf{\Gamma} \vdash \mathrm{fix}_{\tau_1 \to \tau_2}\ \mathbf{t} : \tau_1 \to \tau_2} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau_1 \uplus \tau_2 \quad \mathbf{\Gamma}, (\mathbf{x_1} : \tau_1) \vdash \mathbf{t_1} : \tau \quad \mathbf{\Gamma}, (\mathbf{x_2} : \tau_2) \vdash \mathbf{t_2} : \tau}{\mathbf{\Gamma} \vdash \mathrm{case}\ \mathbf{t}\ \mathrm{of}\ \mathrm{inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \mid \mathrm{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2} : \tau}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{t} : \mathtt{Bool} \quad \mathbf{\Gamma} \vdash \mathbf{t_1} : \tau \quad \mathbf{\Gamma} \vdash \mathbf{t_2} : \tau}{\mathbf{\Gamma} \vdash \mathrm{if}\ \mathbf{t}\ \mathrm{then}\ \mathbf{t_1}\ \mathrm{else}\ \mathbf{t_2} : \tau} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{t_1} : \mathtt{Unit} \quad \mathbf{\Gamma} \vdash \mathbf{t_2} : \tau}{\mathbf{\Gamma} \vdash \mathbf{t_1}; \mathbf{t_2} : \tau}$$

$$\frac{\mathbf{t} \hookrightarrow \mathbf{t'}}{\mathbb{E}[\mathbf{t}] \hookrightarrow \mathbb{E}[\mathbf{t'}]} \qquad \frac{}{(\lambda \mathbf{x} : \tau.\ \mathbf{t})\ \mathbf{v} \hookrightarrow \mathbf{t}[\mathbf{v}/\mathbf{x}]} \qquad \frac{}{\langle \mathbf{v_1}, \mathbf{v_2} \rangle.\mathbf{1} \hookrightarrow \mathbf{v_1}}$$

$$\frac{}{\mathrm{case}\ \mathrm{inl}\ \mathbf{v}\ \mathrm{of}\ \left| \begin{matrix} \mathrm{inl}\ x_1 \mapsto t_1 \\ \mathrm{inr}\ x_2 \mapsto t_2 \end{matrix} \right. \hookrightarrow \mathbf{t_1}[\mathbf{v}/\mathbf{x_1}]} \qquad \frac{\mathbf{v} \equiv \mathtt{true} \Rightarrow \mathbf{t'} \equiv \mathbf{t_1} \quad \mathbf{v} \equiv \mathtt{false} \Rightarrow \mathbf{t'} \equiv \mathbf{t_2}}{\mathrm{if}\ \mathbf{v}\ \mathrm{then}\ \mathbf{t_1}\ \mathrm{else}\ \mathbf{t_2} \hookrightarrow \mathbf{t'}}$$

$$\frac{}{\mathtt{unit}; \mathbf{t} \hookrightarrow \mathbf{t}} \qquad \frac{}{\begin{matrix} \mathrm{fix}_{\tau_1 \to \tau_2}\ (\lambda \mathbf{x} : \tau_1 \to \tau_2.\ \mathbf{t}) \hookrightarrow \\ \mathbf{t}[(\lambda\ \mathbf{y} : \tau_1.\ \mathrm{fix}_{\tau_1 \to \tau_2}\ (\lambda \mathbf{x} : \tau_1 \to \tau_2.\ \mathbf{t})\ \mathbf{y})/\mathbf{x}] \end{matrix}}$$

Figure 4: Syntax, static and dynamic semantics of the source language λ^τ (selection of). We denote syntactic equivalence as $\equiv$.

e.g. Pierce [2002]): the operator is restricted to function types and an additional η-expansion occurs during evaluation. This is because we have chosen to make fix model the Z fixed-point combinator (also known as the call-by-value Y combinator) [Pierce, 2002, §5] rather than the Y combinator. The reason revolves around the compiler devised in this paper. The target language of that compiler is a *strict* untyped lambda calculus, where Y does not work but Z does and using Z in λ^τ as well keeps the compiler simpler. Working with the more standard

$$t ::= \mathtt{unit} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda x.\, t \mid x \mid t\, t \mid t.1 \mid t.2 \mid \langle t, t \rangle \mid \mathtt{inl}\, t \mid \mathtt{inr}\, t$$
$$\mid \mathtt{case}\, t\, \mathtt{of}\, \mathtt{inl}\, x \mapsto t \mid \mathtt{inr}\, x \mapsto t \mid t; t \mid \mathtt{if}\, t\, \mathtt{then}\, t\, \mathtt{else}\, t \mid \mathtt{wrong}$$
$$v ::= \mathtt{unit} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda x.\, t \mid \langle v, v \rangle \mid \mathtt{inl}\, v \mid \mathtt{inr}\, v$$
$$\Gamma ::= \emptyset \mid \Gamma, x$$
$$\mathbb{E} ::= [\cdot] \mid \mathbb{E}\, t \mid v\, \mathbb{E} \mid \mathbb{E}.1 \mid \mathbb{E}.2 \mid \langle \mathbb{E}, t \rangle \mid \langle v, \mathbb{E} \rangle \mid \mathtt{inl}\, \mathbb{E} \mid \mathtt{inr}\, \mathbb{E}$$
$$\mid \mathtt{case}\, \mathbb{E}\, \mathtt{of}\, \mathtt{inl}\, x_1 \mapsto t_1 \mid \mathtt{inr}\, x_2 \mapsto t_2 \mid \mathbb{E}; t \mid \mathtt{if}\, \mathbb{E}\, \mathtt{then}\, t\, \mathtt{else}\, t$$

$$\frac{t \hookrightarrow t'}{\mathbb{E}[t] \hookrightarrow \mathbb{E}[t']} \qquad \frac{\mathbb{E} \neq [\cdot]}{\mathbb{E}[\mathtt{wrong}] \hookrightarrow \mathtt{wrong}} \qquad \frac{}{(\lambda x.\, t)\, v \hookrightarrow t[v/x]} \qquad \frac{}{\langle v_1, v_2 \rangle.1 \hookrightarrow v_1}$$

$$\frac{\begin{array}{c} v \equiv \mathtt{unit} \Rightarrow t' \equiv t \\ v \not\equiv \mathtt{unit} \Rightarrow t' \equiv \mathtt{wrong} \end{array}}{v; t \hookrightarrow t'} \qquad \frac{}{\mathtt{case}\, \mathtt{inl}\, v\, \mathtt{of} \left| \begin{array}{l} \mathtt{inl}\, x_1 \mapsto t_1 \\ \mathtt{inr}\, x_2 \mapsto t_2 \end{array} \right. \hookrightarrow t_1[v/x_1]}$$

$$\frac{\begin{array}{c} v \equiv \mathtt{true} \Rightarrow t' \equiv t_1 \quad v \equiv \mathtt{false} \Rightarrow t' \equiv t_2 \\ (v \not\equiv \mathtt{true} \wedge v \not\equiv \mathtt{false}) \Rightarrow t' \equiv \mathtt{wrong} \end{array}}{\mathtt{if}\, v\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2 \hookrightarrow t'}$$

Figure 5: Syntax and dynamic semantics of the target language λ^{u} (selection of).

Y fixpoint combinator in λ^{τ} is probably possible but would require the compiler to use an encoding that would be pervasive but irrelevant to the subject of this paper.

λ^{τ} program contexts $\mathfrak{C}$ are λ^{τ} terms that contain exactly one hole $[\cdot]$ in place of a subterm. We also omit the typing judgement for program contexts $\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau$, defined by inductive rules close to those for terms in Fig. 4. The judgement guarantees that substituting a well-typed term $\Gamma' \vdash t : \tau'$ in a well-typed context $\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau$ produces a well-typed term $\Gamma \vdash \mathfrak{C}[t] : \tau$.

Figure 5 presents the syntax, well-scopedness and evaluation rules for the target language λ^{u}: a standard untyped λ-calculus. The calculus has unit, booleans, lambdas, product and sum values, and produces a kind of unrecoverable exception in case of type errors (e.g. projecting from a non-pair value, case splitting on a non-sum value etc.). Such an unrecoverable exception is represented in a standard way (see, e.g., [Pierce, 2002, §14.1]) as a non-value term wrong with a special reduction rule. We omit unsurprising well-scopedness rules. The reduction relation $\hookrightarrow$ and termination for terms $t \Downarrow$ are defined analogously to λ^{τ}. The evaluation rules again use evaluation contexts to impose a strict evaluation order. Note that the termination judgement $t \Downarrow$ requires termination with a value, i.e. not wrong. Again, we omit the well-scopedness judgement for contexts $\vdash \mathfrak{C} : \Gamma' \to \Gamma$, which is inductively defined and guarantees that substituting a well-scoped term $\Gamma' \vdash t$ for the hole produces a well-scoped result term $\Gamma \vdash \mathfrak{C}[t]$.

The interested reader can find all proofs in the companion tech report [Devriese et al., 2017].

$$\underline{\mathsf{W}} ::= (k) \text{ with } k \in \mathbb{N} \qquad\qquad\qquad \rhd (k+1) \stackrel{\mathsf{def}}{=} (k)$$

$$\mathsf{lev}(\underline{\mathsf{W}}) \stackrel{\mathsf{def}}{=} \underline{\mathsf{W}}.k \qquad\qquad\qquad (k) \sqsupseteq (k') \stackrel{\mathsf{def}}{=} k \le k'$$

$$\rhd (0) \stackrel{\mathsf{def}}{=} (0) \qquad\qquad\qquad (k) \sqsupset (k') \stackrel{\mathsf{def}}{=} k < k'$$

$$\mathsf{O}(\underline{\mathsf{W}})_{\lesssim} \stackrel{\mathsf{def}}{=} \left\{ (\mathbf{t}, \mathsf{t}) \ \middle| \ \exists k \le \mathsf{lev}(\underline{\mathsf{W}}), \mathbf{v}. \, \mathbf{t} \hookrightarrow^{\mathbf{k}} \mathbf{v} \Rightarrow \exists k', \mathsf{v}. \, \mathsf{t} \hookrightarrow^{k'} \mathsf{v}) \right\}$$

$$\mathsf{O}(\underline{\mathsf{W}})_{\gtrsim} \stackrel{\mathsf{def}}{=} \left\{ (\mathbf{t}, \mathsf{t}) \ \middle| \ \exists k \le \mathsf{lev}(\underline{\mathsf{W}}), \mathsf{v}. \, \mathsf{t} \hookrightarrow^{k} \mathsf{v} \Rightarrow \exists k', \mathbf{v}. \, \mathbf{t} \hookrightarrow^{\mathbf{k'}} \mathbf{v}) \right\}$$

Figure 6: Logical relations: Worlds.

3. Logical Relations

This section presents the Kripke, step-indexed logical relations that we use to prove compiler full-abstraction. First, this section describes the specifications of the world used by the logical relation (Fig. 6). Then, it defines the logical relations (Fig. 7) and finally it proves standard properties that the relations enjoy. Part of the logical relation is postponed until Section 5.2, where we define the back-translation infrastructure that this part depends on. The goal of this section is to provide an understanding of what it means for two terms to be related; this will be needed for understanding properties of the compiler in the following sections.

The cross-language logical relations used in this paper are roughly based on one by Hur and Dreyer [2011]. Essentially, we instantiate their language-generic logical relations to λ^τ and λ^{u} and simplify them by removing complexities deriving from the System F type system, public/private transitions, references and garbage collection.

Since we do not deal with mutable references, we use a very simple notion of worlds, consisting just of a step-index k that can be accessed with the $\mathsf{lev}(\cdot)$ function (Fig. 6). We define a $\rhd$ modality and a future world relation $\sqsupseteq$, expressing that future worlds allow less reduction steps to be taken. We define two different observation relations $\mathsf{O}(\underline{\mathsf{W}})_{\lesssim}$ and $\mathsf{O}(\underline{\mathsf{W}})_{\gtrsim}$. The former defines that a λ^τ term $\mathbf{t}$ approximates a λ^{u} term t if termination of the first in less than $\mathsf{lev}(\underline{\mathsf{W}})$ steps implies termination of the second (in an unknown number of steps). The latter requires the reverse. All of our logical relations will be defined in terms of either $\mathsf{O}(\underline{\mathsf{W}})_{\lesssim}$ or $\mathsf{O}(\underline{\mathsf{W}})_{\gtrsim}$. For definitions and lemmas or theorems that apply for both instantiations, we use the symbol $\square$ as a metavariable that can be instantiated to either $\lesssim$ and $\gtrsim$.

Figure 7 contains the definition of the logical relations. The first thing to note is that our logical relations are not indexed by λ^τ types τ, but by *pseudo-types* $\hat{\tau}$. The syntax for these pseudo-types contains all the constructs of λ^τ types, plus an additional token type $\mathtt{EmulDV_{n;p}}$, indexed by a non-negative number n and a value $p ::= \mathtt{precise} \mid \mathtt{imprecise}$. This token type is not a λ^τ type; it is needed because of the approximate back-translation. When necessary, we use a function $\mathtt{repEmul}()$ for converting a pseudo-type to a λ^τ type. The function replaces all occurrences of $\mathtt{EmulDV_{n;p}}$ with a concrete λ^τ type. We postpone the definitions and explanations of $\mathtt{EmulDV_{n;p}}$ and of $\mathcal{V}[\![\mathtt{EmulDV_{n;p}}]\!]_{\square}$ to Section 5.2, after we have given some more information about the back-translation. We will sometimes silently use normal types where pseudo-types are expected, which makes sense since the latter are a superset of the former.

Pseudo-types $\hat{\tau}$, pseudo-contexts $\hat{\mathbf{\Gamma}}$, $\mathtt{oftype}(\,\cdot\,)$ and $\mathtt{repEmul}(\cdot)$.

$$\hat{\tau} ::= \mathtt{Bool} \mid \mathtt{Unit} \mid \hat{\tau} \times \hat{\tau} \mid \hat{\tau} \uplus \hat{\tau} \mid \hat{\tau} \rightarrow \hat{\tau} \mid \mathtt{EmulDV_{n;p}}$$

$$\hat{\mathbf{\Gamma}} ::= \emptyset \mid \hat{\mathbf{\Gamma}}, \mathbf{x} : \hat{\tau}$$

$$\mathtt{repEmul}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \cdots \text{ (to be defined later, in Fig. 13)}$$

$$\mathbf{oftype}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \{\mathbf{v} \mid \emptyset \vdash \mathbf{v} : \mathtt{repEmul}(\hat{\tau})\}$$

$$\mathsf{oftype}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \left\{ \mathsf{v} \left| \begin{array}{ll} \mathsf{v} = \mathtt{unit} & \text{if } \hat{\tau} = \mathtt{Unit} \\ \mathsf{v} = \mathtt{true} \text{ or } \mathsf{v} = \mathtt{false} & \text{if } \hat{\tau} = \mathtt{Bool} \\ \exists \mathsf{t}.\, \mathsf{v} = \lambda \mathsf{x}.\, \mathsf{t} & \text{if } \exists \hat{\tau_1}, \hat{\tau_2}.\, \hat{\tau} = \hat{\tau_1} \rightarrow \hat{\tau_2} \\ \exists \mathsf{v_1} \in \mathsf{oftype}(\hat{\tau_1}), \mathsf{v_2} \in \mathsf{oftype}(\hat{\tau_2}).\, \mathsf{v} = \langle \mathsf{v_1}, \mathsf{v_2} \rangle & \text{if } \exists \hat{\tau_1}, \hat{\tau_2}.\, \hat{\tau} = \hat{\tau_1} \times \hat{\tau_2} \\ \exists \mathsf{v_1} \in \mathsf{oftype}(\hat{\tau_1}).\, \mathsf{v} = \mathrm{inl}\ \mathsf{v_1} & \\ \quad \text{or } \exists \mathsf{v_2} \in \mathsf{oftype}(\hat{\tau_2}).\, \mathsf{v} = \mathrm{inr}\ \mathsf{v_2} & \text{if } \exists \hat{\tau_1}, \hat{\tau_2}.\, \hat{\tau} = \hat{\tau_1} \uplus \hat{\tau_2} \end{array} \right. \right\}$$

$$\mathtt{oftype}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \{(\mathbf{v}, \mathsf{v}) \mid \mathbf{v} \in \mathbf{oftype}(\hat{\tau}) \wedge \mathsf{v} \in \mathsf{oftype}(\hat{\tau})\}$$

Logical relations for values ($\mathcal{V}[\![\cdot]\!]$), contexts ($\mathcal{K}[\![\cdot]\!]$), terms ($\mathcal{E}[\![\cdot]\!]$) and substitutions ($\mathcal{G}[\![\cdot]\!]$).

$$\rhd R \stackrel{\mathsf{def}}{=} \{(\underline{W}, \mathbf{v}, \mathsf{v}) \mid \mathsf{lev}(\underline{W}) > 0 \Rightarrow (\rhd \underline{W}, \mathbf{v}, \mathsf{v}) \in R\}$$

$$\mathcal{V}[\![\mathtt{Unit}]\!]_\square \stackrel{\mathsf{def}}{=} \{(\underline{W}, \mathbf{v}, \mathsf{v}) \mid \mathbf{v} = \mathtt{unit} \text{ and } \mathsf{v} = \mathtt{unit}\}$$

$$\mathcal{V}[\![\mathtt{Bool}]\!]_\square \stackrel{\mathsf{def}}{=} \{(\underline{W}, \mathbf{v}, \mathsf{v}) \mid \exists v \in \{\mathtt{true}, \mathtt{false}\}.\, \mathbf{v} = v \text{ and } \mathsf{v} = v\}$$

$$\mathcal{V}[\![\hat{\tau'} \rightarrow \hat{\tau}]\!]_\square \stackrel{\mathsf{def}}{=} \left\{ (\underline{W}, \mathbf{v}, \mathsf{v}) \left| \begin{array}{l} (\mathbf{v}, \mathsf{v}) \in \mathtt{oftype}(\hat{\tau'} \rightarrow \hat{\tau}) \text{ and} \\ \exists \mathbf{t}, \mathsf{t}.\, \mathbf{v} = \lambda \mathbf{x} : \mathtt{repEmul}(\hat{\tau'}).\, \mathbf{t} \text{ and } \mathsf{v} = \lambda \mathsf{x}.\, \mathsf{t} \text{ and} \\ \forall \underline{W}' \sqsupseteq \underline{W}, (\underline{W}', \mathbf{v}', \mathsf{v}') \in \mathcal{V}[\![\hat{\tau'}]\!]_\square.\, (\underline{W}', \mathbf{t}[\mathbf{v}'/\mathbf{x}], \mathsf{t}[\mathsf{v}'/\mathsf{x}]) \in \mathcal{E}[\![\hat{\tau}]\!]_\square \end{array} \right. \right\}$$

$$\mathcal{V}[\![\hat{\tau_1} \times \hat{\tau_2}]\!]_\square \stackrel{\mathsf{def}}{=} \left\{ (\underline{W}, \mathbf{v}, \mathsf{v}) \left| \begin{array}{l} (\mathbf{v}, \mathsf{v}) \in \mathtt{oftype}(\hat{\tau_1} \times \hat{\tau_2}) \text{ and} \\ \exists \mathbf{v_1}, \mathbf{v_2}, \mathsf{v_1}, \mathsf{v_2}.\, \mathbf{v} = \langle \mathbf{v_1}, \mathbf{v_2} \rangle \text{ and } \mathsf{v} = \langle \mathsf{v_1}, \mathsf{v_2} \rangle \text{ and} \\ (\underline{W}, \mathbf{v_1}, \mathsf{v_1}) \in \rhd \mathcal{V}[\![\hat{\tau_1}]\!]_\square \text{ and } (\underline{W}, \mathbf{v_2}, \mathsf{v_2}) \in \rhd \mathcal{V}[\![\hat{\tau_2}]\!]_\square \end{array} \right. \right\}$$

$$\mathcal{V}[\![\hat{\tau_1} \uplus \hat{\tau_2}]\!]_\square \stackrel{\mathsf{def}}{=} \left\{ (\underline{W}, \mathbf{v}, \mathsf{v}) \left| \begin{array}{l} (\mathbf{v}, \mathsf{v}) \in \mathtt{oftype}(\hat{\tau_1} \uplus \hat{\tau_2}) \text{ and either} \\ \exists \mathbf{v}', \mathsf{v}'.\, (\underline{W}, \mathbf{v}', \mathsf{v}') \in \rhd \mathcal{V}[\![\hat{\tau_1}]\!]_\square \text{ and } \mathbf{v} = \mathrm{inl}\ \mathbf{v}' \text{ and } \mathsf{v} = \mathrm{inl}\ \mathsf{v}' \text{ or} \\ \exists \mathbf{v}', \mathsf{v}'.\, (\underline{W}, \mathbf{v}', \mathsf{v}') \in \rhd \mathcal{V}[\![\hat{\tau_2}]\!]_\square \text{ and } \mathbf{v} = \mathrm{inr}\ \mathbf{v}' \text{ and } \mathsf{v} = \mathrm{inr}\ \mathsf{v}' \end{array} \right. \right\}$$

$$\mathcal{V}[\![\mathtt{EmulDV_{n;p}}]\!]_\square \stackrel{\mathsf{def}}{=} \cdots \text{ (to be defined later, in Fig. 12)}$$

$$\mathcal{K}[\![\hat{\tau}]\!]_\square \stackrel{\mathsf{def}}{=} \{(\underline{W}, \mathbb{E}, \mathbb{E}) \mid \forall \underline{W}' \sqsupseteq \underline{W}, (\underline{W}', \mathbf{v}, \mathsf{v}) \in \mathcal{V}[\![\hat{\tau}]\!]_\square.\, (\mathbb{E}[\mathbf{v}], \mathbb{E}[\mathsf{v}]) \in \mathsf{O}(\underline{W}')_\square\}$$

$$\mathcal{E}[\![\hat{\tau}]\!]_\square \stackrel{\mathsf{def}}{=} \{(\underline{W}, \mathbf{t}, \mathsf{t}) \mid \forall (\underline{W}, \mathbb{E}, \mathbb{E}) \in \mathcal{K}[\![\hat{\tau}]\!]_\square.\, (\mathbb{E}[\mathbf{t}], \mathbb{E}[\mathsf{t}]) \in \mathsf{O}(\underline{W})_\square\}$$

$$\mathcal{G}[\![\emptyset]\!]_\square \stackrel{\mathsf{def}}{=} \{(\underline{W}, \emptyset, \emptyset)\}$$

$$\mathcal{G}[\![\hat{\mathbf{\Gamma}}, (\mathbf{x} : \hat{\tau})]\!]_\square \stackrel{\mathsf{def}}{=} \{(\underline{W}, \gamma[\mathbf{x} \mapsto \mathbf{v}], \gamma[\mathsf{x} \mapsto \mathsf{v}]) \mid (\underline{W}, \gamma, \gamma) \in \mathcal{G}[\![\hat{\mathbf{\Gamma}}]\!]_\square \text{ and } (\underline{W}, \mathbf{v}, \mathsf{v}) \in \mathcal{V}[\![\hat{\tau}]\!]_\square\}$$

Figure 7: Logical relations (partial, the missing definition can be found in Figs. 12 and 13).

Logical relations for open terms and program contexts.

$$\hat{\Gamma} \vdash \mathbf{t} \,\square_n\, \mathsf{t} : \hat{\tau} \stackrel{\mathsf{def}}{=} \begin{cases} \mathtt{repEmul}(\hat{\Gamma}) \vdash \mathbf{t} : \mathtt{repEmul}(\hat{\tau}) \text{ and } \mathtt{dom}(\hat{\Gamma}) \vdash \mathsf{t} \text{ and} \\ \quad \forall \underline{\mathsf{W}}.\, \mathsf{lev}(\underline{\mathsf{W}}) \le n \Rightarrow \forall (\underline{\mathsf{W}}, \gamma, \gamma) \in \mathcal{G}[\![\hat{\Gamma}]\!]_\square.\, (\underline{\mathsf{W}}, \mathbf{t}\gamma, \mathsf{t}\gamma) \in \mathcal{E}[\![\hat{\tau}]\!]_\square \end{cases}$$

$$\hat{\Gamma} \vdash \mathbf{t} \,\square\, \mathsf{t} : \hat{\tau} \stackrel{\mathsf{def}}{=} \hat{\Gamma} \vdash \mathbf{t} \,\square_n\, \mathsf{t} : \hat{\tau} \text{ for all } n$$

$$\vdash \mathfrak{C} \,\square_n\, \mathsf{C} : \hat{\Gamma}', \hat{\tau}' \to \hat{\Gamma}, \hat{\tau} \stackrel{\mathsf{def}}{=} \begin{cases} \vdash \mathfrak{C} : \mathtt{repEmul}(\hat{\Gamma}'), \mathtt{repEmul}(\hat{\tau}') \to \mathtt{repEmul}(\hat{\Gamma}), \mathtt{repEmul}(\hat{\tau}) \\ \qquad\qquad \text{and } \vdash \mathsf{C} : \mathtt{dom}(\hat{\Gamma}') \to \mathtt{dom}(\hat{\Gamma}) \\ \quad \text{and for all } \mathbf{t}, \mathsf{t}.\ \text{if } \hat{\Gamma}' \vdash \mathbf{t} \,\square_n\, \mathsf{t} : \hat{\tau}', \text{ then } \hat{\Gamma} \vdash \mathfrak{C}[\mathbf{t}] \,\square_n\, \mathsf{C}[\mathsf{t}] : \hat{\tau} \end{cases}$$

Figure 8: Definition of logically related terms and contexts.

The value relation $\mathcal{V}[\![\hat{\tau}]\!]_\square$ is defined by induction on the pseudo-type. Most definitions are quite standard. All cases require related terms to be in the `oftype` relation, which requires well-typedness of the λ^τ term and an appropriate shape for the λ^u value. `Unit` and `Bool` values are related in any world iff they are the same base value. Pair values are related if both are pairs and the corresponding components are related in strictly future worlds at the appropriate pseudo-type. Similarly, sum values are related if they are both of either the form inl $\cdots$ or inr $\cdots$ and if the contained values are related in strictly future worlds at the appropriate pseudo-type. Finally, function values are related if they have the right type, if both are lambdas and if substituting related values in the body yields related terms in any strictly future world.

The relation on values, evaluation contexts and terms are defined mutually recursively, using a technique known as biorthogonality (see, e.g., Benton and Hur [2009]). So, evaluation contexts are related in a world if plugging in related values in any future world yields related observations. Similarly, terms are related if plugging the terms in related evaluation contexts yields related observations. Relation $\mathcal{G}[\![\Gamma]\!]_\square$ relates substitutions instantiating a context Γ, which simply requires that substitutions for all variables in the context are related at their types.

Figure 8 contains the definition of logically-related open and closed terms as well as contexts. For open terms, we define a logical relation $\hat{\Gamma} \vdash \mathbf{t} \,\square_n\, \mathsf{t} : \hat{\tau}$. This relation expresses that an open λ^τ term $\mathbf{t}$ is related up to n steps to an open λ^u term t at pseudo-type $\hat{\tau}$ in pseudo-context $\hat{\Gamma}$ if the first is well-typed, the second is well-scoped and if closing $\mathbf{t}$ and t with substitutions related at pseudo-context $\hat{\Gamma}$ produces terms related at pseudo-type $\hat{\tau}$, in any world $\underline{\mathsf{W}}$ such that $\mathsf{lev}(\underline{\mathsf{W}}) \le n$. If $\hat{\Gamma} \vdash \mathbf{t} \,\square_n\, \mathsf{t} : \hat{\tau}$ for any n, then we write $\hat{\Gamma} \vdash \mathbf{t} \,\square\, \mathsf{t} : \hat{\tau}$. Finally, we define a logical relation for program contexts $\vdash \mathfrak{C} \,\square\, \mathsf{C} : \hat{\Gamma}', \hat{\tau}' \to \hat{\Gamma}, \hat{\tau}$ which requires that substituting terms related at the appropriate pseudo-type produces terms related at the appropriate pseudo-type.

It is interesting to note that the simple type system of our source calculus does not actually present a technical need for the use of step-indexing. Because there are no recursive types or general references, it is a simple enough system that we can give well-founded logical relations without any step-indexing. However, as mentioned before, we use step-indexing for a different reason than other work: not for constructing a well-founded logical relation, but for stating that two terms are related *only up to a certain number of steps*. More details follow in Section 5.

$$\mathit{fix} \stackrel{\mathsf{def}}{=} \lambda f.\,(\lambda x.\,f\,(\lambda y.\,x\,x\,y))\,(\lambda x.\,f\,(\lambda y.\,x\,x\,y))$$

$$\mathsf{erase}(\mathsf{unit}) \stackrel{\mathsf{def}}{=} \mathsf{unit} \qquad\qquad \mathsf{erase}(\langle \mathbf{t_1}, \mathbf{t_2}\rangle) \stackrel{\mathsf{def}}{=} \langle \mathsf{erase}(\mathbf{t_1}), \mathsf{erase}(\mathbf{t_2})\rangle$$

$$\mathsf{erase}(\mathsf{false}) \stackrel{\mathsf{def}}{=} \mathsf{false} \qquad\qquad \mathsf{erase}(\mathbf{t_1}; \mathbf{t_2}) \stackrel{\mathsf{def}}{=} \mathsf{erase}(\mathbf{t_1}); \mathsf{erase}(\mathbf{t_2})$$

$$\mathsf{erase}(\mathbf{x}) \stackrel{\mathsf{def}}{=} \mathsf{x} \qquad\qquad \mathsf{erase}(\mathbf{t_1}\,\mathbf{t_2}) \stackrel{\mathsf{def}}{=} \mathsf{erase}(\mathbf{t_1})\,\mathsf{erase}(\mathbf{t_2})$$

$$\mathsf{erase}(\mathbf{t.1}) \stackrel{\mathsf{def}}{=} \mathsf{erase}(\mathbf{t}).1$$

$$\mathsf{erase}(\mathbf{t.2}) \stackrel{\mathsf{def}}{=} \mathsf{erase}(\mathbf{t}).2 \qquad\qquad \mathsf{erase}(\mathsf{true}) \stackrel{\mathsf{def}}{=} \mathsf{true}$$

$$\mathsf{erase}(\mathsf{inl}\ \mathbf{t}) \stackrel{\mathsf{def}}{=} \mathsf{inl}\ \mathsf{erase}(\mathbf{t}) \qquad \mathsf{erase}(\lambda \mathbf{x}:\tau.\ \mathbf{t}) \stackrel{\mathsf{def}}{=} \lambda \mathsf{x}.\ \mathsf{erase}(\mathbf{t})$$

$$\mathsf{erase}(\mathsf{inr}\ \mathbf{t}) \stackrel{\mathsf{def}}{=} \mathsf{inr}\ \mathsf{erase}(\mathbf{t}) \qquad \mathsf{erase}(\mathsf{fix}_{\tau_1 \to \tau_2}\ \mathbf{t}) \stackrel{\mathsf{def}}{=} \mathit{fix}\ \mathsf{erase}(\mathbf{t})$$

$$\mathsf{erase}(\mathsf{if}\ \mathbf{t}\ \mathsf{then}\ \mathbf{t_1}\ \mathsf{else}\ \mathbf{t_2}) \stackrel{\mathsf{def}}{=} \mathsf{if}\ \mathsf{erase}(\mathbf{t})\ \mathsf{then}\ \mathsf{erase}(\mathbf{t_1})\ \mathsf{else}\ \mathsf{erase}(\mathbf{t_2})$$

$$\mathsf{erase}(\mathsf{case}\ \mathbf{t}\ \mathsf{of}\ \mathsf{inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \mid \mathsf{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2}) \stackrel{\mathsf{def}}{=} \mathsf{case}\ \mathsf{erase}(\mathbf{t})\ \mathsf{of}\ \left|\begin{array}{l} \mathsf{inl}\ \mathsf{x_1} \mapsto \mathsf{erase}(\mathbf{t_1}) \\ \mathsf{inr}\ \mathsf{x_2} \mapsto \mathsf{erase}(\mathbf{t_2}) \end{array}\right.$$

Figure 9: Type erasure: the first pass of the compiler.

These logical relations are constructed so that termination of one implies termination of the other, according to the direction of the approximation ($\lesssim$ or $\gtrsim$, Lemma 3.1).

Lemma 3.1 (Adequacy for $\lesssim$ and $\gtrsim$).
- *If $\emptyset \vdash \mathbf{t} \lesssim_\mathsf{n} \mathsf{t} : \tau$ and $\mathsf{t} \hookrightarrow^\mathbf{m} \mathsf{v}$ with $n \geq m$, then $\mathsf{t} \Downarrow$.*
- *If $\emptyset \vdash \mathbf{t} \gtrsim_\mathsf{n} \mathsf{t} : \tau$ and $\mathsf{t} \hookrightarrow^\mathbf{m} \mathsf{v}$ with $n \geq m$, then $\mathsf{t} \Downarrow$.*

4. The Compiler

This section presents our compiler from λ^τ to λ^u. The compiler proceeds in two passes: type erasure (Fig. 9) and dynamic typechecking wrappers (Fig. 10).

The erasure function is called **erase**; it converts all λ^τ constructs to the corresponding λ^u constructs. $\mathsf{fix}_{\tau_1 \to \tau_2}$ is erased to a λ^u definition of the Z combinator fix.

The **erase** function can be considered as a compiler, but it is only a correct compiler, not a fully-abstract one, as explained in Example 4.1.

Example 4.1 (Erasure is correct but not secure [Patrignani et al., 2015, Fournet et al., 2013]). Consider the following, contextually equivalent λ^τ functions of type $\mathtt{Unit} \to \mathtt{Unit}$:

$$\lambda \mathbf{x} : \mathtt{Unit}.\,\mathbf{x} \qquad \simeq_{ctx} \qquad \lambda \mathbf{x} : \mathtt{Unit}.\,\mathsf{unit}$$

The **erase** function will map these to the following λ^u functions:

$$\lambda \mathsf{x}.\,\mathsf{x} \qquad \not\simeq_{ctx} \qquad \lambda \mathsf{x}.\,\mathsf{unit}$$

The results of **erase** are *not* contextually equivalent, essentially because applying them to a non-unit value like true will produce true for the left lambda and unit for the right lambda.

$$\text{protect}_{\text{Unit}} \overset{\text{def}}{=} \lambda x.\, x \qquad\qquad \text{protect}_{\text{Bool}} \overset{\text{def}}{=} \lambda x.\, x$$

$$\text{protect}_{\tau_1 \times \tau_2} \overset{\text{def}}{=} \lambda y.\, \langle \text{protect}_{\tau_1}\ y.1, \text{protect}_{\tau_2}\ y.2 \rangle$$

$$\text{protect}_{\tau_1 \uplus \tau_2} \overset{\text{def}}{=} \lambda y.\, \text{case } y \text{ of } \left| \begin{array}{l} \text{inl } x \mapsto \text{inl } (\text{protect}_{\tau_1}\ x) \\ \text{inr } x \mapsto \text{inr } (\text{protect}_{\tau_2}\ x) \end{array} \right.$$

$$\text{protect}_{\tau_1 \to \tau_2} \overset{\text{def}}{=} \lambda y.\, \lambda x. \text{protect}_{\tau_2}\ (y\ (\text{confine}_{\tau_1}\ x))$$

$$\text{confine}_{\text{Unit}} \overset{\text{def}}{=} \lambda y.\, (y; \text{unit})$$

$$\text{confine}_{\text{Bool}} \overset{\text{def}}{=} \lambda y.\, \text{if } y \text{ then } \text{true} \text{ else } \text{false}$$

$$\text{confine}_{\tau_1 \times \tau_2} \overset{\text{def}}{=} \lambda y.\, \langle \text{confine}_{\tau_1}\ y.1, \text{confine}_{\tau_2}\ y.2 \rangle$$

$$\text{confine}_{\tau_1 \uplus \tau_2} \overset{\text{def}}{=} \lambda y.\, \text{case } y \text{ of } \left| \begin{array}{l} \text{inl } x \mapsto \text{inl } (\text{confine}_{\tau_1}\ x) \\ \text{inr } x \mapsto \text{inr } (\text{confine}_{\tau_2}\ x) \end{array} \right.$$

$$\text{confine}_{\tau_1 \to \tau_2} \overset{\text{def}}{=} \lambda y.\, \lambda x.\, \text{confine}_{\tau_2}\ (y\ (\text{protect}_{\tau_1}\ x))$$

Figure 10: Dynamic type checking wrappers: the second pass of the compiler.

In this example, contextual equivalence is not preserved because the original functions are only defined for Unit values, but their compilations can be applied to other values too.

The following lemma states that every λ^τ term is related to its erased term at its type.

Lemma 4.2 (Erase is semantics-preserving (for terms)).
If $\Gamma \vdash t : \tau$, then $\Gamma \vdash t \,\square\, \text{erase}(t) : \tau$.

An analogous result applies to program contexts:

Lemma 4.3 (Erase is semantics-preserving (for contexts)).
If $\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau$, then $\vdash \mathfrak{C} \,\square\, \text{erase}(\mathfrak{C}) : \Gamma', \tau' \to \Gamma, \tau$.

One should intuitively understand this result as "t behaves the same as $\text{erase}(t)$ when both are treated as values of type τ". The result does not specify what happens when we treat t as a value of a different type, like we did in Example 4.1 to demonstrate a full abstraction failure. Intuitively, it only specifies a kind of *equivalence reflection* for the erase function, not *preservation*.

Remember that a fully-abstract compiler must protect terms from being used in ways that are not allowed by their type, as in Example 4.1. This is taken care of by the second pass of the compiler.

We construct a family of dynamic typechecking wrappers protect_τ and confine_τ. protect_τ is a λ^u term that wraps an argument to enforce that it can only *be used* in ways that are valid according to type τ, as often done in secure compilation work [Patrignani et al., 2015, Bowman and Ahmed, 2015, Fournet et al., 2013, Ahmed and Blume, 2008]. Dually, confine_τ wraps its argument so that it can only *behave* in ways that are valid according to type τ. In the definition, the cases for product and coproduct types simply recursively descend on their subterms preserving the expected syntax of a product or coproduct argument. Protecting at

a function type means wrapping the function to confine its arguments and protect its results, and dually for confining at a function type. Finally, protecting at a base type (i.e., Unit or Bool) does nothing, simply because there is nothing one can do to a base value that is not allowed by its type. Confining a value at a base type is more interesting. Both for Unit and Bool values, we use the value in such a way that will only work when the value is actually of the correct type. If it is, we return the original value, otherwise the term will reduce to wrong.[3]

Remark on different inhabitants for Unit. In some lambda calculi, any value can be given type Unit and the sequencing operator does not require the term before the semicolon to be unit in order to reduce, but an arbitrary value. Thus, two syntactically-different source terms true and unit can be semantically equivalent at type Unit in this language. Interestingly, in such a language, the definitions of $\mathsf{protect}_{\mathsf{Unit}}$ and $\mathsf{confine}_{\mathsf{Unit}}$ would need to be swapped:

$$\mathsf{protect\text{-}alt}_{\mathsf{Unit}} \stackrel{\mathsf{def}}{=} \lambda\mathsf{y}.\,(\mathsf{y};\mathsf{unit}) \qquad\qquad \mathsf{confine\text{-}alt}_{\mathsf{Unit}} \stackrel{\mathsf{def}}{=} \lambda\mathsf{x}.\,\mathsf{x}$$

In fact, in the target language a context can distinguish what is syntactically different but semantically equivalent in the source. Thus the compiler needs to enforce that the chosen representation for terms of type Unit in the compiled code is unique. Conversely, any value received from a target context is now valid at type Unit, so no checks are made.

Example 4.4 (Protect and confine make a term secure)**.** Consider the protect wrapper $\mathsf{protect}_{\mathsf{Unit}\to\mathsf{Unit}}$ for type Unit → Unit, which is (roughly) equal to $\lambda\mathsf{y}.\,\lambda\mathsf{x}.\,\mathsf{y}\,(\mathsf{x};\mathsf{unit})$. Applying that wrapper to a function f (i.e. $\mathsf{protect}_{\mathsf{Unit}\to\mathsf{Unit}}$ f) reduces to $\lambda\mathsf{x}.\,\mathsf{f}\,(\mathsf{x};\mathsf{unit})$. Applying this value to a non-unit value will simply evaluate to wrong, therefore addressing the issues of Example 4.1.

For the second pass of the compiler, Lemma 4.5 holds.

Lemma 4.5 (Protect and confine are semantics-preserving)**.**
If $\Gamma \vdash \mathsf{t}\,\square_{\mathsf{n}}\,\mathsf{t} : \tau$, *then* $\Gamma \vdash \mathsf{t}\,\square_{\mathsf{n}}\,\mathsf{protect}_{\tau}\,\mathsf{t} : \tau$ *and* $\Gamma \vdash \mathsf{t}\,\square_{\mathsf{n}}\,\mathsf{confine}_{\tau}\,\mathsf{t} : \tau$.

Lemma 4.5 states that if t is related to t at type τ, then adding a $\mathsf{protect}_{\tau}$ or $\mathsf{confine}_{\tau}$ wrapper around t does not change that. In other words, the wrappers do not change the behaviour of t as long as they are treated as values of type τ. In Section 5.5, we will have more to say about the security of the wrappers.

This section concludes with the definition of the compiler used in this paper.

Definition 4.6 (The $[\![\cdot]\!]$ compiler)**.** If $\Gamma \vdash \mathsf{t} : \tau$, then t is compiled to $[\![\mathsf{t}]\!]$ and: $[\![\mathsf{t}]\!] \stackrel{\mathsf{def}}{=} \mathsf{protect}_{\tau}\,(\mathtt{erase}(\mathsf{t}))$.

Lemmas 4.2 and 4.5 about the first and second pass of the compiler can be combined into Lemma 4.7 to obtain that a λ^{τ} term of type τ behaves like its compilation when both are treated as terms of type τ.

Lemma 4.7 ($[\![\cdot]\!]$ is semantics-preserving)**.** *If* $\Gamma \vdash \mathsf{t} : \tau$, *then* $\Gamma \vdash \mathsf{t}\,\square\,[\![\mathsf{t}]\!] : \tau$.

---

[3]It would also be valid to diverge in this case, if λ^{u} had some form of dynamic type test which allowed us to do that.

5. Approximate Back-Translation

This section presents the core idea of our proof technique: the approximate back-translation. As explained in Section 1, the idea is to translate a target language program context $\mathfrak{C}$ to a source language program context $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ which conservatively n-approximates $\mathfrak{C}$. Intuitively, this means that $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ behaves like $\mathfrak{C}$ for up to n steps but it may diverge in cases where the original did not if $\mathfrak{C}$ takes more than n steps. We will make this more precise in Section 5.2.

At the core of the approximate back-translation is the λ^τ type UVal_n. The type is essentially a λ^τ encoding of the unitype of λ^u. Where the untyped context $\mathfrak{C}$ manipulates arbitrary λ^u values, its back-translation $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ manipulates values of type UVal_n. Section 5.1 defines UVal_n and the basic tools (constructors and destructors) for working with it. To explain how values in UVal_n model values in λ^u, Section 5.2 fills in the missing piece of the logical relations of Fig. 7 by defining $\mathcal{V}[\![\mathbf{EmulDV}_{n;p}]\!]_\square$.

The type UVal_n is sufficiently large to contain n-approximations of λ^u values. However, it also contains approximations of λ^u values *up to less than n steps*. Sometimes, values of type UVal_n will be *downgraded* to a type UVal_m with $m < n$. Dually, there will be cases where some values need to *upgrade*. Section 5.3 defines functions to perform value upgrading and downgrading.

With UVal_n and the related machinery introduced, Section 5.4 constructs the function $\mathrm{emulate}_n$, responsible for emulating a context such that it translates a λ^u term $\mathbf{t}$ into a λ^τ term of type UVal_n. This function is easily extended to work with program contexts, producing contexts with hole of type UVal_n as expected.

However, remember from Fig. 3 in Section 1 that the goal of the back-translation is generating a context $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ whose hole can be filled with λ^τ terms $\mathbf{t_1}$ and $\mathbf{t_2}$. Their type is not UVal_n but an arbitrary λ^τ type τ. Thus, there is a type mismatch between the hole of the emulated context $\mathrm{emulate}_n(\mathfrak{C})$ and the terms that we want to plug in there. Since the emulated contexts work with UVal_n values, we need a function that wraps terms of an arbitrary type τ into a value of type UVal_n. This is precisely what Section 5.5 defines, namely a function $\mathtt{inject}_{\tau;n}$ of type $\tau \to \mathrm{UVal}_n$.

Finally, Section 5.6 defines the approximate back-translation function $\langle\!\langle\cdot\rangle\!\rangle_{\tau;n}$, mapping a λ^u context $\mathfrak{C}$ to a λ^τ context $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;n}$. The additional index τ w.r.t. earlier discussions is needed to introduce an appropriate call to $\mathtt{inject}_{\tau;n}$ as discussed above, so that the hole of $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;n}$ is of type τ. Plugging a term $\mathbf{t_1}$ in $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;n}$ n-approximates plugging in the compilation $[\![\mathbf{t_1}]\!]$ in context $\mathfrak{C}$.

Immediately after the definition of each of the concepts discussed above (downgrade, upgrade, $\mathtt{inject}_{\tau;n}$ and $\mathrm{emulate}_n$), this section formalises the results about their behaviour. These results are expressed in terms of the logical relations of Fig. 7 and of the $\mathtt{EmulDV}_{n;p}$ pseudo-type; they will be used to prove equivalence preservation in Section 6.

5.1. UVal **and its Tools.** The family of types UVal_n is defined as follows:

$$\mathrm{UVal}_0 \stackrel{\mathrm{def}}{=} \mathtt{Unit}$$

$$\mathrm{UVal}_{n+1} \stackrel{\mathrm{def}}{=} \begin{array}{l} \mathtt{Unit} \uplus \mathtt{Unit} \uplus \mathtt{Bool} \uplus (\mathrm{UVal}_n \times \mathrm{UVal}_n)\uplus \\ (\mathrm{UVal}_n \uplus \mathrm{UVal}_n) \uplus (\mathrm{UVal}_n \to \mathrm{UVal}_n) \end{array}$$

UVal_n is the type that emulated λ^u terms have when back-translated into λ^τ. For every n, UVal_n is clearly a valid λ^τ type. At non-zero levels, the type UVal_{n+1} is a disjunct sum of base values (the second occurrence of $\mathtt{Unit}$ and $\mathtt{Bool}$), products and coproducts of

$$\mathtt{in}_{\mathtt{unk;n}} : \mathrm{UVal}_{\mathtt{n+1}}$$

$$\mathtt{in}_{\mathtt{Unit;n}} : \mathtt{Unit} \to \mathrm{UVal}_{\mathtt{n+1}}$$

$$\mathtt{in}_{\mathtt{Bool;n}} : \mathtt{Bool} \to \mathrm{UVal}_{\mathtt{n+1}}$$

$$\mathtt{in}_{\times;\mathtt{n}} : (\mathrm{UVal}_{\mathtt{n}} \times \mathrm{UVal}_{\mathtt{n}}) \to \mathrm{UVal}_{\mathtt{n+1}}$$

$$\mathtt{in}_{\uplus;\mathtt{n}} : (\mathrm{UVal}_{\mathtt{n}} \uplus \mathrm{UVal}_{\mathtt{n}}) \to \mathrm{UVal}_{\mathtt{n+1}}$$

$$\mathtt{in}_{\to;\mathtt{n}} : (\mathrm{UVal}_{\mathtt{n}} \to \mathrm{UVal}_{\mathtt{n}}) \to \mathrm{UVal}_{\mathtt{n+1}}$$

$$\mathrm{omega}_\tau : \tau$$

$$\mathrm{omega}_\tau \overset{\mathsf{def}}{=} \mathrm{fix}_{\mathtt{Unit}\to\tau} \ (\lambda \mathbf{x} : \mathtt{Unit} \to \tau. \, \mathbf{x}) \ \mathtt{unit}$$

$$\mathrm{unk}_{\mathtt{n}} : \mathrm{UVal}_{\mathtt{n}}$$

$$\mathrm{unk}_0 \overset{\mathsf{def}}{=} \mathtt{unit}$$

$$\mathrm{unk}_{\mathtt{n+1}} \overset{\mathsf{def}}{=} \mathtt{in}_{\mathtt{unk;n}}$$

$$\mathtt{case}_{\mathtt{Unit;n}} : \mathrm{UVal}_{\mathtt{n+1}} \to \mathtt{Unit}$$

$$\mathtt{case}_{\mathtt{Bool;n}} : \mathrm{UVal}_{\mathtt{n+1}} \to \mathtt{Bool}$$

$$\mathtt{case}_{\times;\mathtt{n}} : \mathrm{UVal}_{\mathtt{n+1}} \to (\mathrm{UVal}_{\mathtt{n}} \times \mathrm{UVal}_{\mathtt{n}})$$

$$\mathtt{case}_{\uplus;\mathtt{n}} : \mathrm{UVal}_{\mathtt{n+1}} \to (\mathrm{UVal}_{\mathtt{n}} \uplus \mathrm{UVal}_{\mathtt{n}})$$

$$\mathtt{case}_{\to;\mathtt{n}} : \mathrm{UVal}_{\mathtt{n+1}} \to \mathrm{UVal}_{\mathtt{n}} \to \mathrm{UVal}_{\mathtt{n}}$$

$$\mathtt{case}_{\mathtt{Unit;n}} \overset{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{\mathtt{n+1}}. \, \mathrm{case} \ \mathbf{x} \ \mathrm{of} \ \{\mathtt{in}_{\mathtt{Unit;n}} \ \mathbf{x} \mapsto \mathbf{x}; \, \_ \mapsto \mathrm{omega}\}$$

$$\mathtt{case}_{\mathtt{Bool;n}} \overset{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{\mathtt{n+1}}. \, \mathrm{case} \ \mathbf{x} \ \mathrm{of} \ \{\mathtt{in}_{\mathtt{Bool;n}} \ \mathbf{x} \mapsto \mathbf{x}; \, \_ \mapsto \mathrm{omega}\}$$

$$\mathtt{case}_{\times;\mathtt{n}} \overset{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{\mathtt{n+1}}. \, \mathrm{case} \ \mathbf{x} \ \mathrm{of} \ \{\mathtt{in}_{\times;\mathtt{n}} \ \mathbf{x} \mapsto \mathbf{x}; \, \_ \mapsto \mathrm{omega}\}$$

$$\mathtt{case}_{\uplus;\mathtt{n}} \overset{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{\mathtt{n+1}}. \, \mathrm{case} \ \mathbf{x} \ \mathrm{of} \ \{\mathtt{in}_{\uplus;\mathtt{n}} \ \mathbf{x} \mapsto \mathbf{x}; \, \_ \mapsto \mathrm{omega}\}$$

$$\mathtt{case}_{\to;\mathtt{n}} \overset{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{\mathtt{n+1}}. \, \lambda \mathbf{y} : \mathrm{UVal}_{\mathtt{n}}. \, \mathrm{case} \ \mathbf{x} \ \mathrm{of} \{\mathtt{in}_{\to;\mathtt{n}} \ \mathbf{z} \mapsto \mathbf{z} \ \mathbf{y}; \, \_ \mapsto \mathrm{omega}\}$$

Figure 11: Basic tools for working with $\mathrm{UVal}_{\mathtt{n}}$. The subscript of omega is omitted when it is clear from the context.

$\mathrm{UVal}_{\mathtt{n}}$s and functions mapping a $\mathrm{UVal}_{\mathtt{n}}$ to a $\mathrm{UVal}_{\mathtt{n}}$. All of these cases are used to emulate a corresponding λ^{u} value. Additionally, at every level including $n = 0$, the type $\mathrm{UVal}_{\mathtt{n}}$ contains a $\mathtt{Unit}$ case which is needed to represent an arbitrary λ^{u} value in cases where the precision of the approximate emulation is insufficient to provide more information. Note that the two occurrences of $\mathtt{Unit}$ in the definition of $\mathrm{UVal}_{\mathtt{n+1}}$ are not a typo. The first is used for imprecisely representing arbitrary λ^{u} terms while the second accurately represents λ^{u} $\mathtt{unit}$ values.

To work with $\mathrm{UVal}_{\mathtt{n}}$ values, we need basic tools for dealing with sum types: tag injections and case extractions (Fig. 11). Functions $\mathtt{in}_{\mathtt{unk;n}}$, $\mathtt{in}_{\mathtt{Unit;n}}$, $\mathtt{in}_{\mathtt{Bool;n}}$, $\mathtt{in}_{\times;\mathtt{n}}$, $\mathtt{in}_{\uplus;\mathtt{n}}$, $\mathtt{in}_{\to;\mathtt{n}}$ are convenient names for nested applications of coproduct injection functions for the nested coproduct in the definition of $\mathrm{UVal}_{\mathtt{n+1}}$. The term $\mathrm{unk}_{\mathtt{n}}$ produces either the single value of UVal_0 or uses $\mathtt{in}_{\mathtt{unk;n}}$ to produce a $\mathrm{UVal}_{\mathtt{n+1}}$ value representing a 0-precision approximate back-translation of an arbitrary untyped term. For using $\mathrm{UVal}_{\mathtt{n}}$ values, we define functions $\mathtt{case}_{\mathtt{Unit;n}}$, $\mathtt{case}_{\mathtt{Bool;n}}$, $\mathtt{case}_{\times;\mathtt{n}}$, $\mathtt{case}_{\uplus;\mathtt{n}}$, $\mathtt{case}_{\to;\mathtt{n}}$ using a somewhat liberal pattern matching syntax that can be easily desugared to nested case expressions. The functions are lambdas that inspect their $\mathrm{UVal}_{\mathtt{n+1}}$ argument and return the contained value if it is in the appropriate

branch of the coproduct, or diverge otherwise. To achieve divergence, we use a term omega_τ constructed using fix. We simply write omega when the type τ can be inferred from the context.

5.2. λ^{u} **Values vs.** UVal. To make the correspondence between a λ^{u} term and its emulation in UVal_n more exact, this section fills in the definition of $\mathcal{V}[\![\text{EmulDV}_{n;p}]\!]_\square$, the missing piece of the logical relations of Fig. 7. Intuitively, the previously presented cases of the logical relations define the relation between a λ^τ term and its compilation. The $\mathcal{V}[\![\text{EmulDV}_{n;p}]\!]_\square$ case defines the relation between a λ^{u} term and its UVal_n-typed back-translation, as motivated in Example 5.1. This relation depends on the index n of type UVal_n and additionally on a parameter $p ::= \texttt{precise} \mid \texttt{imprecise}$, that is explained below.

Example 5.1 (The need for EmulDV). Consider the term $\mathbf{t} \equiv \text{in}_{\text{Bool};1}\ \texttt{true}$. Since UVal_n is a sum type, according to the definition of $\mathcal{V}[\![\tau \uplus \tau']\!]$, it can be related only to terms that have the same tag. However, for the back-translation we do not want this, we want that term to be related to the t term that $\mathbf{t}$ approximates (in this case, $\texttt{true}$). Type $\text{EmulDV}_{n;p}$ serves the purpose of expressing this t-emulates-$\mathbf{t}$ relation (as opposed to the t-is-the-compilation-of-$\mathbf{t}$ relation expressed by the other types). In other words, $\text{in}_{\text{Bool};1}\ \texttt{true}$ and $\texttt{true}$ will be related at pseudo-type $\text{EmulDV}_{2;p}$.

Before explaining the definition of the logical relations for $\text{EmulDV}_{n;p}$, we should elaborate on the approximateness of the correspondence.

Example 5.2 (Approximate values unk_n). Consider the UVal_6 value

$$\text{in}_{\times;5}\ \langle \text{in}_{\uplus;4}\ (\text{inl unk}_4), \text{unk}_5 \rangle$$

This value might be used by the approximate back-translation to represent the λ^{u} term $\langle \text{inl}\ \langle \text{unit}, \texttt{true} \rangle, \lambda \mathsf{x}.\, \mathsf{x} \rangle$. Our $\mathcal{V}[\![\text{EmulDV}_{n;p}]\!]_\square$ specification will enforce that terms of the form $\text{in}_{\times;n}\ \langle \cdot, \cdot \rangle$ or $\text{in}_{\uplus;n}\ (\text{inl}\ \cdot)$ represent the corresponding λ^{u} constructs, but terms unk_4 and unk_5 can represent arbitrary terms (in this case: a pair of base values and a lambda).

The limited size of the type UVal_n sometimes forces us to resort to unk_n values in the back-translation, making it approximate. However, $\mathcal{V}[\![\text{EmulDV}_{n;p}]\!]_\square$ does not allow these unk_n values to occur just anywhere, because they could compromise the required precision of our approximate back-translation.

In fact, $\mathcal{V}[\![\text{EmulDV}_{n;p}]\!]_\square$ provides two different specifications for the occurrence of unk_n, depending on the value of p. The case where $p = \texttt{imprecise}$ is used when we are proving $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \lesssim \mathfrak{C}$, which means roughly that termination of $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$ in *any* number of steps implies termination of $\mathfrak{C}$. In this case, $\mathcal{V}[\![\text{EmulDV}_{n;p}]\!]_\square$ allows unk_n values to occur everywhere in a back-translation term, and they can correspond to arbitrary λ^{u} terms. These mild requirements on the correspondence of λ^{u} terms place a large burden on the code in a back-translation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$. This code must be able to deal with unk_n values and produce behaviour for them that approximates the behaviour of $\mathfrak{C}$ for the arbitrary values that the unk_ns correspond with. Luckily, when we are proving $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \lesssim \mathfrak{C}$, we can achieve this by simply making all the functions in our back-translation diverge whenever they try to use a UVal_n value that happens to be an unk_n. This is sufficient because the approximation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \lesssim \mathfrak{C}$ trivially holds when $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$ diverges: it essentially only requires that $\mathfrak{C}$ terminates whenever $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$ does, but nothing needs to be shown when the latter diverges.

$$\mathcal{V}[\![\texttt{EmulDV}_{0;p}]\!]_{\square} \overset{\text{def}}{=} \{(\underline{W}, \mathbf{v}, \mathsf{v}) \mid \mathbf{v} = \texttt{unit} \text{ and } \mathsf{p} = \texttt{imprecise}\}$$

$$\mathcal{V}[\![\texttt{EmulDV}_{n+1;p}]\!]_{\square} \overset{\text{def}}{=} \left\{ (\underline{W}, \mathbf{v}, \mathsf{v}) \;\middle|\; \begin{array}{l} \mathbf{v} \in \mathbf{oftype}(\text{UVal}_{n+1}) \text{ and one of the following holds:} \\[4pt] \left\{ \begin{array}{l} \mathbf{v} = \texttt{in}_{\text{unk};n} \text{ and } \mathsf{p} = \texttt{imprecise} \\[4pt] \exists \mathbf{v}'.\, \mathbf{v} = \texttt{in}_{\texttt{Unit};n}\ \mathbf{v}' \text{ and } (\underline{W}, \mathbf{v}', \mathsf{v}) \in \mathcal{V}[\![\texttt{Unit}]\!]_{\square} \\[4pt] \exists \mathbf{v}'.\, \mathbf{v} = \texttt{in}_{\texttt{Bool};n}\ \mathbf{v}' \text{ and } (\underline{W}, \mathbf{v}', \mathsf{v}) \in \mathcal{V}[\![\texttt{Bool}]\!]_{\square} \\[4pt] \exists \mathbf{v}'.\, \mathbf{v} = \texttt{in}_{\times;n}\ \mathbf{v}' \text{ and} \\ \quad (\underline{W}, \mathbf{v}', \mathsf{v}) \in \mathcal{V}[\![\texttt{EmulDV}_{n;p} \times \texttt{EmulDV}_{n;p}]\!]_{\square} \\[4pt] \exists \mathbf{v}'.\, \mathbf{v} = \texttt{in}_{\uplus;n}\ \mathbf{v}' \text{ and} \\ \quad (\underline{W}, \mathbf{v}', \mathsf{v}) \in \mathcal{V}[\![\texttt{EmulDV}_{n;p} \uplus \texttt{EmulDV}_{n;p}]\!]_{\square} \\[4pt] \exists \mathbf{v}'.\, \mathbf{v} = \texttt{in}_{\to;n}\ \mathbf{v}' \text{ and} \\ \quad (\underline{W}, \mathbf{v}', \mathsf{v}) \in \mathcal{V}[\![\texttt{EmulDV}_{n;p} \to \texttt{EmulDV}_{n;p}]\!]_{\square} \end{array} \right. \end{array} \right\}$$

Figure 12: Specifying the relation between λ^{u} values and their emulation in $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_{\square}$.

Example 5.3 (Relatedness with `imprecise`). Consider the term $\mathbf{t} \equiv \texttt{in}_{\times;42}\ \langle \text{unk}_{42}, \text{unk}_{42} \rangle$. This term will be related to $\langle \mathsf{t}_1, \mathsf{t}_2 \rangle$ at pseudo-type $\texttt{EmulDV}_{43;\texttt{imprecise}}$ for any terms t_1 and t_2 and in any world.

The case when $p = \texttt{precise}$ specifies where values unk_n are allowed when we are proving that $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \gtrsim_n \mathfrak{C}$, meaning roughly that termination of $\mathfrak{C}$ in less than n steps implies termination of $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$. In this case, the requirements on the back-translation correspondence are significantly stronger: unk_n is simply ruled out by the definition of $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_{\square}$. That does not mean, however, that unk_n cannot occur inside related terms, rather that unk_n can only occur at depths that cannot be reached using the number of steps in the world.

Example 5.4 (Relatedness with `precise`). Consider again the term $\mathbf{t} \overset{\text{def}}{=} \texttt{in}_{\times;42}\ \langle \text{unk}_{42}, \text{unk}_{42} \rangle$. This term will still be related by $\texttt{EmulDV}_{43;\texttt{precise}}$ to $\mathsf{t} \overset{\text{def}}{=} \langle \mathsf{t}_1, \mathsf{t}_2 \rangle$ for any terms t_1 and t_2, but only in worlds $\underline{W}$ such that $\textsf{lev}(\underline{W}) = 0$. More precisely, our specification will state that $(\underline{W}, \mathbf{t}, \mathsf{t}) \in \mathcal{V}[\![\texttt{EmulDV}_{43;\texttt{precise}}]\!]_{\square}$ iff

$$(\underline{W}, \langle \text{unk}_{42}, \text{unk}_{42} \rangle, \langle \mathsf{t}_1, \mathsf{t}_2 \rangle) \in \mathcal{V}[\![\texttt{EmulDV}_{42;\texttt{precise}} \times \texttt{EmulDV}_{42;\texttt{precise}}]\!]_{\square}.$$

By the definition in Fig. 7, this requires in turn that $(\underline{W}, \text{unk}_{42}, \mathsf{t}_1)$ and $(\underline{W}, \text{unk}_{42}, \mathsf{t}_2)$ are in $\triangleright \mathcal{V}[\![\texttt{EmulDV}_{42,\texttt{precise}}]\!]_{\square}$. However if $\textsf{lev}(\underline{W}) = 0$, then this is vacuously true by definition of the $\triangleright$ operator, independent of the requirements of $\mathcal{V}[\![\texttt{EmulDV}_{42,\texttt{precise}}]\!]_{\square}$.

Intuitively, it is sufficient to only forbid unk_n at depths lower than the number of steps left in the world because we are proving $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \gtrsim_n \mathfrak{C}$ (emphasis on the index $_n$ of $\gtrsim_n$). So, if $\mathfrak{C}$ terminates *in less than n steps*, then the evaluation of $\mathfrak{C}$ cannot have used values that are deeper than level n in any UVal$_n$. The corresponding execution of $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$ will also not have had a chance to encounter the unk_ns. Therefore, the executions must have behaved identically.

With this approximation aspect explained, Fig. 12 presents the definition of $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_{\square}$. For relating terms $\mathbf{v}$ and v in a world $\underline{W}$, the definition requires that $\mathbf{v}$ has the right type and that $p = \texttt{imprecise}$ if $\mathbf{v}$ is unk_n. Additionally, the structure of the λ^{τ} term stripped

$$\mathtt{toEmul}(\emptyset)_{\mathsf{n;p}} = \emptyset$$

$$\mathtt{toEmul}(\Gamma, \mathsf{x})_{\mathsf{n;p}} = \mathtt{toEmul}(\Gamma)_{\mathsf{n;p}}, (\mathsf{x} : \mathtt{EmulDV_{n;p}})$$

$$\mathtt{repEmul}(\emptyset) = \emptyset$$

$$\mathtt{repEmul}(\mathbf{\Gamma}, (\mathbf{x} : \hat{\tau})) = \mathtt{repEmul}(\mathbf{\Gamma}), (\mathbf{x} : \mathtt{repEmul}(\hat{\tau}))$$

$$\mathtt{repEmul}(\hat{\tau} \times \hat{\tau'}) = \mathtt{repEmul}(\hat{\tau}) \times \mathtt{repEmul}(\hat{\tau'})$$

$$\mathtt{repEmul}(\hat{\tau} \uplus \hat{\tau'}) = \mathtt{repEmul}(\hat{\tau}) \uplus \mathtt{repEmul}(\hat{\tau'})$$

$$\mathtt{repEmul}(\hat{\tau} \to \hat{\tau'}) = \mathtt{repEmul}(\hat{\tau}) \to \mathtt{repEmul}(\hat{\tau'})$$

$$\mathtt{repEmul}(\mathtt{EmulDV_{n;p}}) = \mathrm{UVal_n}$$

$$\mathtt{repEmul}(\mathtt{Bool}) = \mathtt{Bool}$$

$$\mathtt{repEmul}(\mathtt{Unit}) = \mathtt{Unit}$$

Figure 13: Helper functions for $\mathtt{EmulDV_{n;p}}$.

of its $\mathrm{UVal_n}$ tag and the structure of the λ^{u} term must coincide. Formally, this is expressed by the following conditions: $(\underline{\mathsf{W}}, \mathbf{v'}, \mathsf{v})$ are in $\mathcal{V}[\![\mathcal{B}]\!]_\square$ (recall that $\mathcal{B}$ are ground types), $\mathcal{V}[\![\mathtt{EmulDV_{n;p}} \times \mathtt{EmulDV_{n;p}}]\!]_\square$, $\mathcal{V}[\![\mathtt{EmulDV_{n;p}} \uplus \mathtt{EmulDV_{n;p}}]\!]_\square$ or $\mathcal{V}[\![\mathtt{EmulDV_{n;p}} \to \mathtt{EmulDV_{n;p}}]\!]_\square$ if $\mathbf{v} = \mathsf{in}_{\mathcal{B};n} \mathbf{v'}$, $\mathbf{v} = \mathsf{in}_{\times;n} \mathbf{v'}$, $\mathbf{v} = \mathsf{in}_{\uplus;n} \mathbf{v'}$ or $\mathbf{v} = \mathsf{in}_{\to;n} \mathbf{v'}$ respectively.

In addition to $\mathtt{EmulDV_{n;p}}$, we still need to define two helper functions (Fig. 13). The first, $\mathtt{repEmul}(\cdot)$, was left open in Fig. 7. It re-maps all variables of a $\mathbf{\Gamma}$ that are of type $\mathtt{EmulDV_{n;p}}$ to type $\mathrm{UVal_n}$. A second function, $\mathtt{toEmul}(\cdot)_{\mathsf{n;p}}$, turns an untyped Γ into one where all variables are mapped to $\mathtt{EmulDV_{n;p}}$.

The adequacy property of the logical relations (Lemma 3.1) holds for the complete definition of the logical relations, including the definition for $\mathcal{V}[\![\mathtt{EmulDV_{n;p}}]\!]$.

5.3. **Upgrading and Downgrading Values.** Figure 14 defines the functions $\mathrm{downgrade}_{\mathsf{n;d}} :$ $\mathrm{UVal_{n+d}} \to \mathrm{UVal_n}$ and $\mathrm{upgrade}_{\mathsf{n;d}} : \mathrm{UVal_n} \to \mathrm{UVal_{n+d}}$ (by induction on n) that we talked about before. Most cases simply work structurally over the type, but some are more interesting. There is a contravariance in the cases for function values in both $\mathrm{downgrade}_{\mathsf{n;d}}$ and $\mathrm{upgrade}_{\mathsf{n;d}}$: a function $\mathrm{UVal_n} \to \mathrm{UVal_n}$ is turned into a function of type $\mathrm{UVal_{n+d}} \to \mathrm{UVal_{n+d}}$ by constructing a wrapper that downgrades the argument and upgrades the result and vice versa. Unknown values are always mapped to unknown values, but additionally, the case for $\mathrm{downgrade}_{\mathsf{n;d}}$ when $n = 0$ will throw away the information contained in its argument of type $\mathrm{UVal_d}$ and simply returns the single unknown value in $\mathrm{UVal_0}$. Note that $\mathrm{downgrade}_{\mathsf{n;d}}$ and $\mathrm{upgrade}_{\mathsf{n;d}}$ are not inverse functions, since $\mathrm{downgrade}_{\mathsf{n;d}}$ throws away information that was previously there. Informally, while $\mathbf{t} \sim \mathrm{downgrade}_{\mathsf{n;d}} (\mathrm{upgrade}_{\mathsf{n;d}} \mathbf{t})$, the reverse ($\mathbf{t} \sim \mathrm{upgrade}_{\mathsf{n;d}} (\mathrm{downgrade}_{\mathsf{n;d}} \mathbf{t})$) is not true, since applying downgrade first reduces precision.

Example 5.5 (Downgrading terms)**.** Suppose that we want to emulate a λ^{u} term $\lambda\mathsf{x}. \langle\mathsf{x}, \mathsf{x}\rangle$ in $\mathrm{UVal.}$ for a sufficiently-large n. We would expect roughly the following λ^{τ} term:

$$\mathsf{in}_{\to;n-1} (\lambda\mathbf{x} : \mathrm{UVal_{n-1}}. \mathsf{in}_{\times;n-2} \langle\mathbf{x}, \mathbf{x}\rangle)$$

$$\text{downgrade}_{n;d} : \text{UVal}_{n+d} \to \text{UVal}_n$$

$$\text{downgrade}_{0;d} \overset{\text{def}}{=} \lambda\mathbf{v} : \text{UVal}_d.\,\text{unk}_0$$

$$\text{downgrade}_{n+1;d} \overset{\text{def}}{=} \lambda\mathbf{x} : \text{UVal}_{n+d+1}.\,\text{case }\mathbf{x}\text{ of}$$

$$\begin{array}{l}
\text{in}_{\text{unk};n+d} \mapsto \text{in}_{\text{unk};n} \\
\text{in}_{\text{Unit};n+d}\ \mathbf{y} \mapsto \text{in}_{\text{Unit};n}\ \mathbf{y} \\
\text{in}_{\text{Bool};n+d}\ \mathbf{y} \mapsto \text{in}_{\text{Bool};n}\ \mathbf{y} \\
\text{in}_{\times;n+d}\ \mathbf{y} \mapsto \text{in}_{\times;n}\ \langle\text{downgrade}_{n;d}\ \mathbf{y.1}, \text{downgrade}_{n;d}\ \mathbf{y.2}\rangle \\
\text{in}_{\uplus;n+d}\ \mathbf{y} \mapsto \text{in}_{\uplus;n}\ \text{case }\mathbf{y}\text{ of}\ \left|\begin{array}{l}\text{inl }\mathbf{x} \mapsto \text{inl }(\text{downgrade}_{n;d}\ \mathbf{x}) \\ \text{inr }\mathbf{x} \mapsto \text{inr }(\text{downgrade}_{n;d}\ \mathbf{x})\end{array}\right. \\
\text{in}_{\to;n+d}\ \mathbf{y} \mapsto \text{in}_{\to;n}\ (\lambda\mathbf{z} : \text{UVal}_n.\,\text{downgrade}_{n;d}(\mathbf{y}\ (\text{upgrade}_{n;d}\ \mathbf{z})))
\end{array}$$

$$\text{upgrade}_{n;d} : \text{UVal}_n \to \text{UVal}_{n+d}$$

$$\text{upgrade}_{0;d} \overset{\text{def}}{=} \lambda\mathbf{x} : \text{UVal}_0.\,\text{unk}_d$$

$$\text{upgrade}_{n+1;d} \overset{\text{def}}{=} \lambda\mathbf{x} : \text{UVal}_{n+1}.\,\text{case }\mathbf{x}\text{ of}$$

$$\begin{array}{l}
\text{in}_{\text{unk};n} \mapsto \text{in}_{\text{unk};n+d}; \\
\text{in}_{\text{Unit};n}\ \mathbf{y} \mapsto \text{in}_{\text{Unit};n+d}\ \mathbf{y}; \\
\text{in}_{\text{Bool};n}\ \mathbf{y} \mapsto \text{in}_{\text{Bool};n+d}\ \mathbf{y}; \\
\text{in}_{\times;n}\ \mathbf{y} \mapsto \text{in}_{\times;n+d}\ \langle\text{upgrade}_{n;d}\ \mathbf{y.1}, \text{upgrade}_{n;d}\ \mathbf{y.2}\rangle \\
\text{in}_{\uplus;n}\ \mathbf{y} \mapsto \text{in}_{\uplus;n+d}\ \text{case }y\text{ of}\ \left|\begin{array}{l}\text{inl }\mathbf{x} \mapsto \text{inl }(\text{upgrade}_{n;d}\ \mathbf{x}) \\ \text{inr }\mathbf{x} \mapsto \text{inr }(\text{upgrade}_{n;d}\ \mathbf{x})\end{array}\right. \\
\text{in}_{\to;n}\ \mathbf{y} \mapsto \text{in}_{\to;n+d}\ (\lambda\mathbf{z} : \text{UVal}_n.\,\text{upgrade}_{n;d}(\mathbf{y}\ (\text{downgrade}_{n;d}\ \mathbf{z})))
\end{array}$$

Figure 14: Upgrade and downgrade for UVal_n.

Indices $n-1$ and $n-2$ of the UVal_n constructors are imposed by the well-typedness constraints. However, even this is not enough to guarantee well-typedness. With a closer inspection, the variable $\mathbf{x}$ of type UVal_{n-1} is used where a term of type UVal_{n-2} is required (it is inside a pair tagged with $\text{in}_{\times;n-2}$). This is a problem of type safety, not precision of approximation. Since $\mathbf{x}$ appears inside a pair, inspecting $\mathbf{x}$ for any number of steps requires at least one additional step to first project it out of the pair. In other words, for the pair to be a precise approximation up to $\leq n-1$ steps, $\mathbf{x}$ needs only to be precise up to $n-2$ steps. It is then safe to throw away one level of precision and downgrade $\mathbf{x}$ from type UVal_{n-1} to UVal_{n-2}.

We will use the function downgrade for the situation of Example 5.5 and similar ones in the next sections. In dual situations we will need to upgrade terms from type UVal_n to UVal_{n+d}. This will neither increase precision of the approximation, nor decrease it.

The correctness property for downgrade and upgrade is stated in the following lemma.

Lemma 5.6 (Compatibility lemma for $\text{upgrade}_{n;d}$ and $\text{downgrade}_{n;d}$). *Suppose that*
- *either (*$n < m$ *and* $p = \texttt{precise}$*)*

$$\mathrm{emulate_n(t) : UVal_n}$$

$$\mathrm{emulate_n(unit)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{Unit;n}}\ \mathtt{unit})$$

$$\mathrm{emulate_n(true)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{Bool;n}}\ \mathtt{true})$$

$$\mathrm{emulate_n(false)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{Bool;n}}\ \mathtt{false})$$

$$\mathrm{emulate_n(x)} \overset{\mathsf{def}}{=} \mathbf{x}$$

$$\mathrm{emulate_n(\lambda x.\, t)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{\to;n}}\ (\lambda \mathbf{x} : \mathrm{UVal_n}.\, \mathrm{emulate_n(t)}))$$

$$\mathrm{emulate_n(t_1\ t_2)} \overset{\mathsf{def}}{=} \mathrm{case_{\to;n}}\ (\mathrm{upgrade_{n;1}}\ (\mathrm{emulate_n(t_1)}))\ \mathrm{emulate_n(t_2)}$$

$$\mathrm{emulate_n(\langle t_1, t_2 \rangle)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{\times;n}}\ \langle \mathrm{emulate_n(t_1)}, \mathrm{emulate_n(t_2)} \rangle)$$

$$\mathrm{emulate_n(inl\ t)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{\uplus;n}}\ (\mathrm{inl\ emulate_n(t)}))$$

$$\mathrm{emulate_n(inr\ t)} \overset{\mathsf{def}}{=} \mathrm{downgrade_{n;1}}\ (\mathrm{in_{\uplus;n}}\ (\mathrm{inr\ emulate_n(t)}))$$

$$\mathrm{emulate_n(t.1)} \overset{\mathsf{def}}{=} (\mathrm{case_{\times;n}}\ (\mathrm{upgrade_{n;1}}\ (\mathrm{emulate_n(t)}))).\mathbf{1}$$

$$\mathrm{emulate_n(t.2)} \overset{\mathsf{def}}{=} (\mathrm{case_{\times;n}}\ (\mathrm{upgrade_{n;1}}\ (\mathrm{emulate_n(t)}))).\mathbf{2}$$

$$\mathrm{emulate_n(t; t')} \overset{\mathsf{def}}{=} (\mathrm{case_{Unit;n}}\ (\mathrm{upgrade_{n;1}}(\mathrm{emulate_n(t)}))); \mathrm{emulate_n(t')}$$

$$\mathrm{emulate_n(wrong)} \overset{\mathsf{def}}{=} \mathrm{omega}$$

$$\mathrm{emulate_n(case\ t_1\ of\ inl\ x \mapsto t_2 \mid inr\ x \mapsto t_3)} \overset{\mathsf{def}}{=}$$

$$\mathrm{case}\ (\mathrm{case_{\uplus;n}}\ (\mathrm{upgrade_{n;1}}\ (\mathrm{emulate_n(t_1)})))\ \mathrm{of}\ \left| \begin{array}{l} inl\ x \mapsto \mathrm{emulate_n(t_2)} \\ inr\ x \mapsto \mathrm{emulate_n(t_3)} \end{array} \right.$$

$$\mathrm{emulate_n(if\ t\ then\ t_1\ else\ t_2)} \overset{\mathsf{def}}{=}$$

$$\mathrm{if}\ (\mathrm{case_{Bool;n}}\ (\mathrm{upgrade_{n;1}}\ (\mathrm{emulate_n t})))\ \mathrm{then}\ \mathrm{emulate_n(t_1)}\ \mathrm{else}\ \mathrm{emulate_n(t_2)}$$

Figure 15: Emulating λ^{u} terms in $\mathrm{UVal_n}$.

- *or ($\square = \lesssim$ and $\mathtt{p} = \mathtt{imprecise}$),*
then
- *If $\mathbf{\Gamma} \vdash \mathbf{t}\ \square_{\mathsf{n}}\ \mathsf{t} : \mathtt{EmulDV_{m+d;p}}$, then $\mathbf{\Gamma} \vdash \mathrm{downgrade_{m;d}}\ \mathbf{t}\ \square_{\mathsf{n}}\ \mathsf{t} : \mathtt{EmulDV_{m;p}}$.*
- *If $\mathbf{\Gamma} \vdash \mathbf{t}\ \square_{\mathsf{n}}\ \mathsf{t} : \mathtt{EmulDV_{m;p}}$, then $\mathbf{\Gamma} \vdash \mathrm{upgrade_{m;d}}\ \mathbf{t}\ \square_{\mathsf{n}}\ \mathsf{t} : \mathtt{EmulDV_{m+d;p}}$.*

This lemma covers both situations that we discussed previously. It requires that either $n < m$ (so that the results only hold in worlds $\underline{\mathsf{W}}$ with $\mathsf{lev}(\underline{\mathsf{W}}) \leq n < m$), in which case $p = \mathtt{precise}$, or $\square = \lesssim$ and $p = \mathtt{imprecise}$. If that is the case, the lemma says that if a term $\mathbf{t}$ is related to t by $\mathtt{EmulDV_{m+d;p}}$ (or $\mathtt{EmulDV_{m;p}}$) then it stays related to t after upgrading or downgrading.

5.4. **Emulation.** Having defined downgrade and upgrade, Fig. 15 defines the $\mathrm{emulate_n}$ function. That function maps arbitrary λ^{u} terms to their approximate back-translation: λ^{τ}

terms of type $\mathrm{UVal_n}$. $\mathrm{emulate_n}$ is defined by induction on t. The different cases follow the same pattern: every term t is mapped to a λ^τ term constructed recursively from the emulation of sub-terms, producing and consuming $\mathrm{UVal_n}$ terms wherever t works with untyped terms. Additionally, the definitions use $\mathrm{upgrade_{n;1}}$ and $\mathrm{downgrade_{n;1}}$ to make the resulting term type-check, as explained in Example 5.5. For example, the case for pairs applies $\mathtt{in_{\times;n}}$ to a pair constructed from the emulations of its components. Since this produces a $\mathrm{UVal_{n+1}}$, $\mathrm{downgrade_{n;1}}$ is used to downgrade this to a $\mathrm{UVal_n}$ term. Finally, the untyped term wrong is back-translated to a divergent term.

The back-translation produced by $\mathrm{emulate_n}$ is necessarily approximate, as the type $\mathrm{UVal_n}$ is not large enough for back-translating arbitrary terms. Inaccuracies in the back-translation are introduced in the calls to $\mathrm{downgrade_{n;1}}$ in several of the cases. The approximation is accurate enough for the following lemma to hold.

Lemma 5.7 (Emulate relates at `EmulDV`). *If $\Gamma \vdash \mathsf{t}$, and if*
- *either ($\mathsf{m} > \mathsf{n}$ and $\mathsf{p} = \mathtt{precise}$)*
- *or ($\square = \lesssim$ and $\mathsf{p} = \mathtt{imprecise}$),*

then we have that $\mathtt{toEmul}(\Gamma)_{\mathsf{m;p}} \vdash \mathrm{emulate_m}(\mathsf{t}) \,\square_\mathsf{n}\, \mathsf{t} : \mathtt{EmulDV}_{\mathsf{m;p}}$.

Like Lemma 5.6, Lemma 5.7 requires that either $n < m$ (so that the results only hold in worlds $\underline{\mathsf{W}}$ with $\mathsf{lev}(\underline{\mathsf{W}}) \leq n < m$), in which case $p = \mathtt{precise}$, or $\square = \lesssim$ and $p = \mathtt{imprecise}$. This again covers what we need for the two logical approximations of $\langle\!\langle \mathfrak{C} \rangle\!\rangle_\mathsf{n}$ in Fig. 3. The lemma states that the back-translation of any well-scoped term is related to the term by $\mathtt{EmulDV}_{\mathsf{m;p}}$, as intended.

An analogous result holds for contexts.

Lemma 5.8 (Emulate relates contexts at `EmulDV`). *If $\vdash \mathfrak{C} : \Gamma' \to \Gamma$, and if*
- *either ($m > n$ and $p = \mathtt{precise}$)*
- *or ($\square = \lesssim$ and $p = \mathtt{imprecise}$),*

then $\vdash \mathrm{emulate_m}(\mathfrak{C}) \,\square_\mathsf{n}\, \mathfrak{C} : \mathtt{toEmul}(\Gamma')_{\mathsf{m;p}}, \mathtt{EmulDV}_{\mathsf{m;p}} \to \mathtt{toEmul}(\Gamma)_{\mathsf{m;p}}, \mathtt{EmulDV}_{\mathsf{m;p}}$

5.5. **Injection and Extraction of Terms.** One final thing is missing to construct a back-translation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_\mathsf{n}$ of an untyped program context $\mathfrak{C}$. While $\mathrm{emulate_n}(\mathfrak{C})$ produces a λ^τ context that expects a $\mathrm{UVal_n}$ value (just like $\mathfrak{C}$ expects an arbitrary λ^u value), the back-translation should accept values of a given type τ (the type of the terms $\mathsf{t_1}$ and $\mathsf{t_2}$ that we are compiling). To bridge this difference, Fig. 16 defines a λ^τ function $\mathtt{inject}_{\tau;\mathsf{n}}$ of type $\tau \to \mathrm{UVal_n}$ which injects values of an arbitrary type τ into $\mathrm{UVal_n}$. We define it mutually recursively with a dual function $\mathtt{extract}_{\tau;\mathsf{n}} : \mathrm{UVal_n} \to \tau$ which is needed for contravariantly converting $\mathrm{UVal_n}$ arguments to the appropriate type in the $\mathtt{inject}_{\tau;\mathsf{n}}$ case for function types.

Generally, $\mathtt{inject}_{\tau;\mathsf{n}}$ converts a value v of type τ to a value of type $\mathrm{UVal_n}$ that behaves like the compilation $[\![\mathsf{v}]\!]$. The cases for base values use the appropriate tagging and case functions (e.g., $\mathtt{in_{Unit;n}}$ and $\mathtt{case_{Bool;n}}$) to achieve this. For pair and sum values, $\mathtt{inject}_{\tau;\mathsf{n}}$ and $\mathtt{extract}_{\tau;\mathsf{n}}$ simply recurse over the structure of the values, respectively applying $\mathtt{in_{\times;n}}$, $\mathtt{in_{\uplus;n}}$ and $\mathtt{case_{\times;n}}$, $\mathtt{case_{\uplus;n}}$ to construct and destruct $\mathrm{UVal_n}$s of a certain expected form. Note that when $\mathrm{UVal_n}$ values do not have the form expected for type τ, then $\mathtt{extract}_{\tau;\mathsf{n}}$ will diverge by definition of the $\mathtt{case_{\ldots;n}}$ functions. This divergence corresponds to the wrong that we get when an untyped context attempts to use λ^u values as pairs, disjunct sum values or base values when those values are of a different form.

$$\text{extract}_{\tau;n} : \text{UVal}_n \to \tau$$

$$\text{extract}_{\tau;0} \stackrel{\text{def}}{=} \lambda x : \text{UVal}_0 . \text{omega}$$

$$\text{extract}_{\text{Unit};n+1} \stackrel{\text{def}}{=} \lambda x : \text{UVal}_{n+1} . \text{case}_{\text{Unit};n} \, x$$

$$\text{extract}_{\text{Bool};n+1} \stackrel{\text{def}}{=} \lambda x : \text{UVal}_{n+1} . \text{case}_{\text{Bool};n} \, x$$

$$\text{extract}_{\tau_1 \to \tau_2;n+1} \stackrel{\text{def}}{=} \lambda x : \text{UVal}_{n+1} . \lambda y : \tau_1 . \text{extract}_{\tau_2;n} \, (\text{case}_{\to;n} \, x \, (\text{inject}_{\tau_1;n} \, y))$$

$$\text{extract}_{\tau_1 \times \tau_2;n+1} \stackrel{\text{def}}{=} \lambda x : \text{UVal}_{n+1} . \langle \text{extract}_{\tau_1;n} \, (\text{case}_{\times;n} \, x).1, \text{extract}_{\tau_2;n} \, (\text{case}_{\times;n} \, x).2 \rangle$$

$$\text{extract}_{\tau_1 \uplus \tau_2;n+1} \stackrel{\text{def}}{=} \lambda x : \text{UVal}_{n+1} . \text{case } \text{case}_{\uplus;n} \, x \text{ of} \begin{vmatrix} \text{inl } y \to \text{inl } (\text{extract}_{\tau_1;n} \, y) \\ \text{inr } y \to \text{inr } (\text{extract}_{\tau_2;n} \, y) \end{vmatrix}$$

$$\text{inject}_{\tau;n} : \tau \to \text{UVal}_n$$

$$\text{inject}_{\tau;0} \stackrel{\text{def}}{=} \lambda x : \tau . \text{omega}$$

$$\text{inject}_{\text{Unit};n+1} \stackrel{\text{def}}{=} \lambda x : \text{Unit} . \text{in}_{\text{Unit};n} \, x$$

$$\text{inject}_{\text{Bool};n+1} \stackrel{\text{def}}{=} \lambda x : \text{Bool} . \text{in}_{\text{Bool};n} \, x$$

$$\text{inject}_{\tau_1 \to \tau_2;n+1} \stackrel{\text{def}}{=} \lambda x : \tau_1 \to \tau_2 . \text{in}_{\to;n} \, (\lambda x : \text{UVal}_n . \text{inject}_{\tau_2;n} \, (x \, (\text{extract}_{\tau_1;n} \, x)))$$

$$\text{inject}_{\tau_1 \times \tau_2;n+1} \stackrel{\text{def}}{=} \lambda x : \tau_1 \times \tau_2 . \text{in}_{\times;n} \langle \text{inject}_{\tau_1;n} \, x.1, \text{inject}_{\tau_2;n} \, x.2 \rangle$$

$$\text{inject}_{\tau_1 \uplus \tau_2;n+1} \stackrel{\text{def}}{=} \lambda x : \tau_1 \uplus \tau_2 . \text{in}_{\uplus;n} \left(\text{case } x \text{ of} \begin{vmatrix} \text{inl } y \mapsto \text{inr } (\text{inject}_{\tau_1;n} \, y) \\ \text{inr } y \mapsto \text{inr } (\text{inject}_{\tau_2;n} \, y) \end{vmatrix} \right)$$

Figure 16: Injecting λ^τ values into UVal_n.

For function types, $\text{inject}_{\tau;n}$ and $\text{extract}_{\tau;n}$ produce lambdas that contravariantly extract and inject the argument and covariantly inject and extract the result. Finally, when $n = 0$, then the size of our type is insufficient for $\text{extract}_{\tau;n}$ and $\text{inject}_{\tau;n}$ to accurately perform their intended function. Luckily, to obtain the necessary precision of our approximate back-translation, it is sufficient for them to simply diverge in this case: they simply return omega terms of the expected type.

For a value $\mathbf{v}$ of type τ, $\text{inject}_{\tau;n}$ will produce a value UVal_n that behaves as the compilation of $\mathbf{v}$, $[\![\mathbf{v}]\!]$. More precisely and more generally, the following lemma states that if a term $\mathbf{t}$ is related to a term t at type τ (intuitively if $\mathbf{t}$ behaves as t when used in a way that is valid according to type τ), then $\text{inject}_{\tau;n} \, \mathbf{t}$ behaves as the emulation of $\text{protect}_\tau \, t$. A dual result about $\text{extract}_{\tau;n}$ and confine_τ states (intuitively) that if a term $\mathbf{t}$ behaves as an emulation of value t, then $\text{confine}_\tau \, t$ will behave as $\text{extract}_{\tau;n} \, \mathbf{t}$ when used in ways that are valid according to type τ.

Lemma 5.9 (Inject is protect and extract is confine). *If $\hat{\Gamma} \vdash \mathbf{t} \, \square_n \, t : \tau$ and if*
- *either ($m \geq n$ and $p = \texttt{precise}$)*
- *or ($\square = \lesssim$ and $p = \texttt{imprecise}$)*

then $\hat{\Gamma} \vdash \text{inject}_{\tau;m} \, \mathbf{t} \, \square_n \, \text{protect}_\tau \, t : \text{EmulDV}_{m;p}$.

If $\hat{\boldsymbol{\Gamma}} \vdash \mathbf{t} \,\square_n\, t : \text{EmulDV}_{m;p}$, *and if*

- *either* $(m \geq n$ *and* $p = \texttt{precise})$
- *or* $(\square = \lesssim$ *and* $p = \texttt{imprecise})$

then $\hat{\boldsymbol{\Gamma}} \vdash \text{extract}_{\tau;m}\, \mathbf{t} \,\square_n\, \text{confine}_\tau\, t : \tau$.

Example 5.10. Consider again Example 4.1. We have that

$$\emptyset \vdash \lambda\mathbf{x} : \text{Unit}.\, \mathbf{x} \,\square\, \lambda\text{x}.\, \text{x} : \text{Unit} \to \text{Unit}.$$

$\lambda\mathbf{x} : \text{Unit}.\, \mathbf{x}$ behaves like $\lambda\text{x}.\, \text{x}$, when the latter is used in ways that are valid for a value of type $\text{Unit} \to \text{Unit}$. Lemma 5.9 then yields:

$$\emptyset \vdash \text{inject}_{\tau;n}\, (\lambda\mathbf{x} : \text{Unit}.\, \mathbf{x}) \,\square_n\, \text{protect}_{\text{Unit}\to\text{Unit}}\, (\lambda\text{x}.\, \text{x}) : \text{EmulDV}_{m;n}.$$

For n sufficiently large and modulo some simplifications, these terms become:

$$\text{inject}_{\tau;n}\, (\lambda\mathbf{x} : \text{Unit}.\, \mathbf{x}) = \text{in}_{\to;n-1}\, (\lambda\mathbf{x} : \text{UVal}_{n-1}.\, \text{in}_{\text{Unit};n-2}\, (\text{case}_{\text{Unit};n-2}\, \mathbf{x}))$$

$$\text{protect}_{\text{Unit}\to\text{Unit}}\, (\lambda\text{x}.\, \text{x}) = \lambda\text{x}.\, \text{x}; \text{unit}$$

We invite the reader to verify that both expressions behave appropriately when applied to any values $\mathbf{v}$ and v that are related by $\text{EmulDV}_{n;p}$: for example $(\mathbf{v} = \text{in}_{\text{Unit};n-1}\, \text{unit}$ and $\text{v} = \text{unit})$, $(\mathbf{v} = \text{in}_{\to;n-1}\, (\lambda\mathbf{x} : \text{UVal}_{n-1}.\, \mathbf{x})$ and $\text{v} = \lambda\text{x}.\, \text{x})$ or $(\mathbf{v} = \text{unk}_n,\ \text{v}$ is any λ^{u} term and $\square = \lesssim)$.

5.6. **Approximate Back-Translation.** We are now ready to define the approximate back-translation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}$ of an arbitrary untyped context $\mathfrak{C}$ with a hole of type τ. However, before we do, we need to correct a few simplifications that were made in Fig. 3.

First, as we have already explained, the back-translation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n$ does not just depend on n but also on the type τ of the terms $\mathbf{t_1}$ and $\mathbf{t_2}$ that we are compiling. As such, we define the back-translation with τ as an additional parameter.

Definition 5.11 (n-approximate back-translation $\langle\!\langle \cdot \rangle\!\rangle_{\tau;n}$)**.** The n-approximate back-translation of a context $\mathfrak{C}$ with a hole of type τ is defined as follows. $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n} \overset{\text{def}}{=} \text{emulate}_{n+1}(\mathfrak{C})[\text{inject}_{\tau;n}\, \cdot]$

A second simplification in Fig. 3 was the fact that we claimed $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \gtrsim_n \mathfrak{C}$ and $\langle\!\langle \mathfrak{C} \rangle\!\rangle_n \lesssim \mathfrak{C}$. Fig. 17 shows a more accurate picture of the relations that we have. As we will see in the next section, this more accurate picture still allows us to conclude the facts that $\emptyset \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}[\mathbf{t_1}] \gtrsim_n \mathfrak{C}[[\![\mathbf{t_1}]\!]] : \text{EmulDV}_{n;\text{precise}}$ and $\emptyset \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}[\mathbf{t_2}] \lesssim_{n'} \mathfrak{C}[[\![\mathbf{t_2}]\!]] : \text{EmulDV}_{n;\text{imprecise}}$ so that the proof goes through unchanged.

The correctness of $\langle\!\langle \cdot \rangle\!\rangle_{\tau;n}$ is captured in Lemma 5.12.

Lemma 5.12 (Correctness of $\langle\!\langle \cdot \rangle\!\rangle_{\tau;n}$)**.** *If* $\boldsymbol{\Gamma} \vdash \mathbf{t} \,\square_n\, t : \tau$, *and if*

- *either* $(m \geq n$ *and* $p = \texttt{precise})$
- *or* $(\square = \lesssim$ *and* $p = \texttt{imprecise})$

then $\boldsymbol{\Gamma} \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;m}[\mathbf{t}] \,\square_n\, \mathfrak{C}[\text{protect}_\tau\, t] : \text{EmulDV}_{m;p}$.
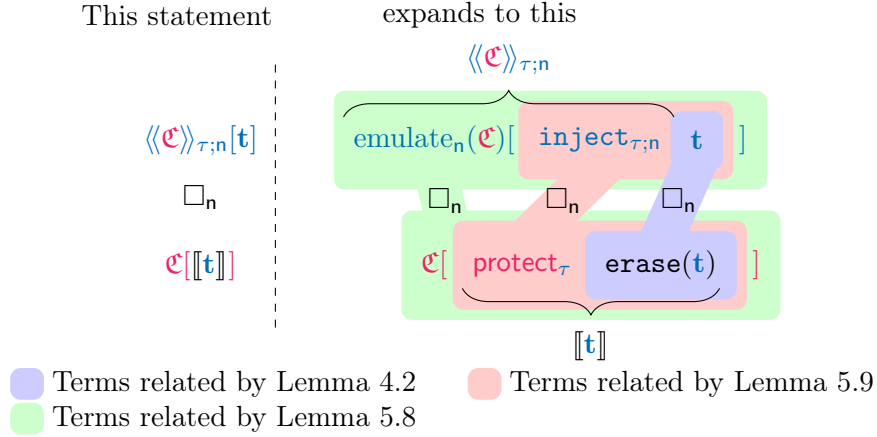
Proof. Follows from Lemmas 5.8 and 5.9. $\qquad\square$

Figure 17: A more accurate picture of related components of compiled term $\mathbf{t}$, program context $\mathfrak{C}$, compilation $[\![\mathbf{t}]\!]$ and emulation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}$ than in the simplified Fig. 3.

6. Compiler Full-Abstraction

This section presents the proof that the compiler $[\![\cdot]\!]$ is fully-abstract (Theorem 6.3) by relying on the logical relations of Fig. 7. As previously mentioned, this results in proving equivalence reflection (Theorem 6.1) and preservation (Theorem 6.2). As suggested by Fig. 1 in Section 1, the lemmas presented in Section 4 are enough to prove equivalence reflection for $[\![\cdot]\!]$. Dually, as suggested by Fig. 3 in Section 1, the lemmas presented in Section 5 are enough to prove equivalence preservation for $[\![\cdot]\!]$.

Recall from Definition 4.6 that $[\![\mathbf{t}]\!]$ is $\mathsf{protect}_\tau\ (\mathsf{erase}(\mathbf{t}))$.

Theorem 6.1 ($[\![\cdot]\!]$ is correct). *If $\emptyset \vdash \mathbf{t_1} : \tau$, $\emptyset \vdash \mathbf{t_2} : \tau$ and $\emptyset \vdash [\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$, then $\emptyset \vdash \mathbf{t_1} \simeq_{ctx} \mathbf{t_2} : \tau$.*

Proof. Take $\mathfrak{C}$ so that $\vdash \mathfrak{C} : \emptyset, \tau \to \emptyset, \tau'$. By definition of $\simeq_{ctx}$, we need to prove that $\mathfrak{C}[\mathbf{t_1}]\Downarrow$ iff $\mathfrak{C}[\mathbf{t_2}]\Downarrow$. By symmetry, it suffices to prove the $\Rightarrow$ direction. So, assume that $\mathfrak{C}[\mathbf{t_1}]\Downarrow$. We need to prove that $\mathfrak{C}[\mathbf{t_2}]\Downarrow$.

Define $\mathfrak{C} \overset{\mathsf{def}}{=} \mathsf{erase}(\mathfrak{C})$, Lemma 4.3 yields $\vdash \mathfrak{C} \,\square\, \mathfrak{C} : \emptyset, \tau \to \emptyset, \tau'$. By Lemma 4.7, we get $\emptyset \vdash \mathbf{t_1} \,\square\, [\![\mathbf{t_1}]\!] : \tau$ and $\emptyset \vdash \mathbf{t_2} \,\square\, [\![\mathbf{t_2}]\!] : \tau$. By definition of $\vdash \mathfrak{C} \,\square\, \mathfrak{C} : \emptyset, \tau \to \emptyset, \tau'$, we get (specifically) that $\emptyset \vdash \mathfrak{C}[\mathbf{t_1}] \gtrsim \mathfrak{C}[[\![\mathbf{t_1}]\!]] : \tau'$ and $\emptyset \vdash \mathfrak{C}[\mathbf{t_2}] \lesssim \mathfrak{C}[[\![\mathbf{t_2}]\!]] : \tau'$.

$\mathfrak{C}[\mathbf{t_1}]\Downarrow$ and $\emptyset \vdash \mathfrak{C}[\mathbf{t_1}] \,\square\, \mathfrak{C}[[\![\mathbf{t_1}]\!]] : \tau'$ imply that $\mathfrak{C}[[\![\mathbf{t_1}]\!]]\Downarrow$ by Lemma 3.1. From $[\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$ and $\mathfrak{C}[[\![\mathbf{t_1}]\!]]\Downarrow$, we get that $\mathfrak{C}[[\![\mathbf{t_2}]\!]]\Downarrow$. $\emptyset \vdash \mathfrak{C}[\mathbf{t_2}] \,\square\, \mathfrak{C}[[\![\mathbf{t_2}]\!]] : \tau'$ and $\mathfrak{C}[[\![\mathbf{t_2}]\!]]\Downarrow$ yield $\mathfrak{C}[\mathbf{t_2}]\Downarrow$ by Lemma 3.1. $\qquad\square$

Theorem 6.2 ($[\![\cdot]\!]$ is secure). *If $\emptyset \vdash \mathbf{t_1} : \tau$, $\emptyset \vdash \mathbf{t_2} : \tau$ and $\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} : \tau$, then $[\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$.*

Proof. Note that $[\![\mathbf{t_1}]\!] = \mathsf{protect}_\tau\ (\mathsf{erase}(\mathbf{t_1}))$ by definition and similarly for $\mathbf{t_2}$.

Take a $\vdash \mathfrak{C} : \emptyset \to \emptyset$ and suppose that $\mathfrak{C}[\mathsf{protect}_\tau(\mathsf{erase}(\mathbf{t_1}))]\Downarrow$, then we need to show that $\mathfrak{C}[\mathsf{protect}_\tau(\mathsf{erase}(\mathbf{t_2}))]\Downarrow$.

Take n larger than the number of steps in the termination of $\mathfrak{C}[\mathsf{protect}_\tau(\mathsf{erase}(\mathbf{t_1}))]\Downarrow$. By Lemma 4.2, we have that $\emptyset \vdash \mathbf{t_1} \gtrsim_n \mathsf{erase}(\mathbf{t_1}) : \tau$.

By Lemma 5.12 (taking $m = n \geq n$, $p = \mathtt{precise}$ and $\square = \gtrsim$), we then have that

$$\emptyset \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}[\mathbf{t_1}] \gtrsim_n \mathfrak{C}[\mathsf{protect}_\tau\ (\mathsf{erase}(\mathbf{t_1}))] : \mathtt{EmulDV}_{\mathbf{n;precise}}.$$

Now by Lemma 3.1, by $\mathfrak{C}[\mathtt{protect}_\tau(\mathtt{erase}(\mathbf{t_1}))]\!\Downarrow$, and by the choice of n, we have that $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}[\mathbf{t_1}]\!\Downarrow$.

It now follows from $\emptyset \vdash \mathbf{t_1} \simeq_{ctx} \mathbf{t_2} : \tau$ and $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}[\mathbf{t_1}]\!\Downarrow$ that $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}[\mathbf{t_2}]\!\Downarrow$.

Now take n' the number of steps in the termination of $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}[\mathbf{t_2}]\!\Downarrow$. We have from Lemma 4.2 that $\emptyset \vdash \mathbf{t_2} \lesssim_{n'} \mathtt{erase}(\mathbf{t_2}) : \tau$.

By Lemma 5.12, we then have (taking $m = n$, $n = n'$, $p = \mathtt{imprecise}$ and $\square = \lesssim$) that

$$\emptyset \vdash \langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}[\mathbf{t_2}] \lesssim_{n'} \mathfrak{C}[\mathtt{protect}_\tau\ (\mathtt{erase}(\mathbf{t_2}))] : \mathtt{EmulDV}_{\mathbf{n};\mathtt{imprecise}}$$

Now by Lemma 3.1, by the choice of n' and by the fact that $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}[\mathbf{t_2}]\!\Downarrow$, we get that $\mathfrak{C}[\mathtt{protect}_\tau\ (\mathtt{erase}(\mathbf{t_2}))]\!\Downarrow$ as required. $\square$

Theorem 6.3 ($[\![\cdot]\!]$ is fully-abstract). *If $\emptyset \vdash \mathbf{t_1} : \tau$ and $\emptyset \vdash \mathbf{t_2} : \tau$, then*

$$\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} \iff [\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$$

Proof. Theorem 6.1 provides the $\Leftarrow$ direction while Theorem 6.2 provides the $\Rightarrow$ one. $\square$

7. Modular Fully-Abstract Compilation

For the sake of simplicity, so far we only considered compilers that take a whole program as input, keeping modular compilers (and thus linking of compiled programs) out of the picture. However, for the proof technique to be applicable and useful in real-world scenarios, it must scale to modular compilers. Modular compilers compile different parts of a program separately, leading to faster re-compilation times since only the fragments that changed since the last compilation are recompiled. This section extends the presented proof technique to modular compilers.

More in detail, a modular compiler considers source programs that are *open*; it compiles these open programs independently and *links* the result to form the runnable program. Open programs are those that have dependencies on other ones, e.g., a secure transactions program could rely on third-party cryptographic-signing function to accomplish its task. The program does not implement the signing function itself, rather it relies on such a function to be provided at link time. Linking is the process of taking open programs and fulfilling their dependencies with the other programs they are linked against. When the aforementioned secure transaction program is linked against the code that provides the signing function, the linker ensures that whenever the program calls that function, the call is dispatched to the actual implementation.

Full-abstraction as stated in Theorem 6.3 is not a correct criterion for modular compilers, as it is stated for closed terms. Instead, a generalisation exists for modular compilers: modular full-abstraction [Patrignani et al., 2016]. Modular full-abstraction forces one to reason about linking of programs when developing a fully-abstract compiler. Modular full-abstraction can be derived from compiler modularity and full-abstraction stated with an open environment (so not as in Theorem 6.3).

In the remainder of this section, we explain how to turn the compiler developed so far into a modular one and prove it to be modularly fully-abstract. In order to discuss modularity and linking, this section first defines what open terms are and introduces a notion of linking in both source and target languages (Section 7.1). Then it extends the compiler to work for open terms (Section 7.2) so that it can be proven to be modularly fully-abstract (Section 7.3). The proofs are all carried out using the machinery developed in the previous section, which furthers our belief in the strength of the proof technique.

7.1. **Open Terms and Linking.** Open terms are already part of the model since they are those that are type checked against a non-empty environment. For the sake of simplicity, we only consider linking two terms $\mathbf{t_1}$ and $\mathbf{t_2}$ (linking an arbitrary number of terms simply adds an inductive step to the formal development but no additional insight). Both $\mathbf{t_1}$ and $\mathbf{t_2}$ have a single free variable that the other term is intended to fulfill, i.e. $\mathbf{t_1}$ has a free variable $\mathbf{x_2}$ of the same type as $\mathbf{t_2}$, and $\mathbf{t_2}$ has a free variable $\mathbf{x_1}$ of the same type as $\mathbf{t_1}$. We allow $\mathbf{t_1}$ and $\mathbf{t_2}$ to be mutually dependent, but the case for non-mutually dependent terms follows as a special case.

Intuitively, the linker must return the pair of $\mathbf{t_1}$ and $\mathbf{t_2}$ where the free variable of each term is replaced with the other term. Because these two terms have mutual dependencies, linking is encoded by using a fixpoint to produce a pair containing versions of $\mathbf{t_1}$ and $\mathbf{t_2}$ with the occurrences of the free variable of $\mathbf{t_1}$ filled in with $\mathbf{t_2}$ (and vice-versa). Recall that fixpoint is a syntactic form that exists in both languages.

Since we are in a call-by-value setting, fixpoints are a bit delicate. Specifically, if we just feed any two arbitrary terms $\mathbf{t_1}$ and $\mathbf{t_2}$ to the fixpoints, it is not possible to produce the fixpoint without risking divergence. To address this (known) problem, we restrict the compiler to lambdas $\lambda \mathbf{x'_1} : \tau'_1. \, \mathbf{t_1}$ and $\lambda \mathbf{x'_2} : \tau'_2. \, \mathbf{t_2}$, as one would expect from a call-by-value program.

However, we cannot simply use a fixpoint to produce the pair that we want because we had to encode fix in λ^{u} as the Z combinator (which can only produce fixpoints that are functions). While intuitively the linker should just use fix to produce the pair of the two terms, in this case it needs to be wrapped into a lambda that discards its argument. We choose to supply Unit-type values to such a lambda.

Definition 7.1 presents linking in λ^{τ} and λ^{u}.

Definition 7.1 (Linking). If

$$\mathbf{x_2} : \tau'_2 \to \tau_2 \vdash \mathbf{t_1} : \tau'_1 \to \tau_1$$
$$\mathbf{x_1} : \tau'_1 \to \tau_1 \vdash \mathbf{t_2} : \tau'_2 \to \tau_2$$

then

$$\mathbf{t_1} + \mathbf{t_2} \stackrel{\text{def}}{=} \left(\begin{array}{l} \text{fix}_{\texttt{Unit} \to ((\tau'_1 \to \tau_1) \times (\tau'_2 \to \tau_2))} \\ (\lambda p : \texttt{Unit} \to ((\tau'_1 \to \tau_1) \times (\tau'_2 \to \tau_2)). \, \lambda\_ : \texttt{Unit}. \\ \left\langle \begin{array}{l} \lambda x'_1 : \tau'_1. \, ((\lambda x_2 : \tau'_2 \to \tau_2. \, t_1) \; ((p \; \texttt{unit}).2)) \; x'_1, \\ \lambda x'_2 : \tau'_2. \, ((\lambda x_1 : \tau'_1 \to \tau_1. \, t_2) \; ((p \; \texttt{unit}).1)) \; x'_2 \end{array} \right\rangle)) \end{array} \right) \texttt{unit}$$

We can show that the this produces a well-typed term:

$$\mathbf{t_1} + \mathbf{t_2} : ((\tau'_1 \to \tau_1) \times (\tau'_2 \to \tau_2))$$

If

$$\mathsf{x_2} \vdash \mathsf{t_1}$$
$$\mathsf{x_1} \vdash \mathsf{t_2}$$

then

$$\mathsf{t_1} + \mathsf{t_2} \stackrel{\text{def}}{=} \left(\textit{fix} \left(\lambda p. \, \lambda\_. \left\langle \begin{array}{l} \lambda x'_1. \, ((\lambda x_2. \, t_1) \; (p \; \texttt{unit}).2) \; x'_1, \\ \lambda x'_2. \, ((\lambda x_1. \, t_2) \; (p \; \texttt{unit}).1) \; x'_2 \end{array} \right\rangle \right) \right) \right) \texttt{unit}$$

Both linkers are defined analogously. They use the recursive fix arguments ($\mathbf{p}$ and $\mathbf{p}$ respectively) inside the lambda term, binding the projections of that argument to the corresponding free variable (for instance, binding the second projection of $\mathbf{p}$ to $\mathbf{x_2}$). As stated before, the recursive application is done after an eta-expansion to prevent the term from diverging.

In the context of modular full abstraction, it is important that for any term $\mathbf{t}$, linking with $\mathbf{t}$ produces a valid program context $\cdot + \mathbf{t}$. This way, a program context (representing an adversary) can link the program with an arbitrary term of its choosing, i.e. a compiled program cannot trust what it is being linked against. As a result of this, compiled programs must perform checks against that code too; these checks are the modifications to the compiler to which we turn next. The advantage of the fact that linking produces valid contexts is that if we take $\mathfrak{C}[\mathbf{t_1} + \mathbf{t_2}]$, i.e. we let a linked program $\mathbf{t_1} + \mathbf{t_2}$ interact with an adversary context $\mathfrak{C}$, then if $\mathbf{t_2}$ contains a security bug, we can still change our point of view and consider $\mathfrak{C}[\cdot + \mathbf{t_2}]$ as an adversary context that trusted program $\mathbf{t_1}$ is being linked against. In other words, modular full abstraction implies a form of compartmentalisation: security bugs in a component do not expose other components' internals.

7.2. **A Secure Compiler for Open Terms.** The compiler definition changes as in Definition 7.2 to account for open terms and compiled terms being in a lambda-form.

Definition 7.2 (A Modular Compiler $\llbracket \cdot \rrbracket_{\lambda^u}^{\lambda^\tau}$). Assuming
- $\mathbf{x_2} : \tau_2' \to \tau_2 \vdash \lambda \mathbf{x_1'} : \tau_1'. \mathbf{t_1} : \tau_1' \to \tau_1$,

then:
$$\llbracket \lambda \mathbf{x_1'} : \tau_1'. \mathbf{t_1} \rrbracket_{\lambda^u}^{\lambda^\tau} = \mathsf{protect}_{\tau_1' \to \tau_1}(\lambda \mathsf{x_1'}.((\lambda \mathsf{x_2}.\, \mathtt{erase}(\mathbf{t_1}))(\mathsf{confine}_{\tau_2' \to \tau_2}\, \mathsf{x_2})))$$

The compiler knows that $\mathtt{erase}(\lambda \mathbf{x_1'}.\mathbf{t})$ will generate an open term with an open variable $\mathsf{x_2}$. So it closes that variable with a $\lambda \mathsf{x_2}.\cdot$ just to open it again with the argument of that lambda (the term $(\mathsf{confine}_{\tau_2' \to \tau_2}\mathsf{x_2})$). The point of this is to force a $\mathsf{confine.}$ around the free variables.

If one considers two source terms being compiled with this compiler and then linked, then the extra $\mathsf{confine.}$ is redundant. However, linking at the target level can be done with arbitrary terms, so they need to be restricted on how they interoperate with these terms by calling $\mathsf{confine.}$ on them. The term supplied for the open variable is like the argument of a function (the linker really treats it that way), thus the choice of $\mathsf{confine.}$. Adding this additional $\mathsf{confine.}$ does not disrupt the functionality of compiled code, as proved by Lemma 7.3.

Lemma 7.3 (An extra confine is just fine). *If* $\Gamma, \mathbf{x} : \tau' \vdash \mathbf{t} : \tau$, *then*
$$\Gamma, \mathbf{x} : \tau' \vdash \mathbf{t} \,\Box_n\, (\lambda \mathsf{x}.\, \mathtt{erase}(\mathbf{t}))(\mathsf{confine}_{\tau'}\, \mathsf{x}) : \tau$$

7.3. **Modular Full-Abstraction for $\llbracket \cdot \rrbracket_{\lambda^u}^{\lambda^\tau}$.** The property that $\llbracket \cdot \rrbracket_{\lambda^u}^{\lambda^\tau}$ must have is modular full-abstraction [Patrignani et al., 2016], which is the combination of compiler full-abstraction with an open environment (Theorem 7.7 in Section 7.3.1) and compiler modularity (Theorem 7.8 in Section 7.3.2).

7.3.1. *Full-Abstraction with an Open Environment.* The first step to re-prove compiler full-abstraction is compiler correctness for open lambda-terms (Theorem 7.4).

Theorem 7.4 ($[\![\cdot]\!]_{\lambda^u}^{\lambda^\tau}$ is correct). *If*

$$\mathbf{x_2} : \tau_2' \to \tau_2 \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} : \tau_1' \to \tau_1$$

then

$$\mathbf{x_2} : \tau_2' \to \tau_2 \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} \;\square_n\; \mathsf{protect}_{\tau_1' \to \tau_1}(\lambda\mathsf{x_1'}.\,((\lambda\mathsf{x_2}.\,\mathtt{erase}(\mathbf{t_1}))(\mathsf{confine}_{\tau_2' \to \tau_2}\,\mathsf{x_2}))) : \tau_1' \to \tau_1.$$

We then re-state Theorem 6.1 and Theorem 6.2 to work for open lambda terms only and to work for the new definition of $[\![\cdot]\!]_{\lambda^u}^{\lambda^\tau}$.

Theorem 7.5 ($[\![\cdot]\!]_{\lambda^u}^{\lambda^\tau}$ reflects equivalence). *If*

$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} : \tau_1' \to \tau_1,$$
$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_2'} : \tau_1'.\, \mathbf{t_2} : \tau_1' \to \tau_1,$$
$$\mathsf{x} \vdash [\![\lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1}]\!]_{\lambda^u}^{\lambda^\tau} \simeq_{ctx} [\![\lambda\mathbf{x_2'} : \tau_1'.\, \mathbf{t_2}]\!]_{\lambda^u}^{\lambda^\tau}$$

then

$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} \simeq_{ctx} \lambda\mathbf{x_2'} : \tau_1'.\, \mathbf{t_2} : \tau_1' \to \tau_1.$$

Theorem 7.6 ($[\![\cdot]\!]_{\lambda^u}^{\lambda^\tau}$ preserves equivalence). *If*

$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} : \tau_1' \to \tau_1,$$
$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_2'} : \tau_1'.\, \mathbf{t_2} : \tau_1' \to \tau_1,$$
$$\mathbf{x} : \tau' \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} \simeq_{ctx} \lambda\mathbf{x_1'} : \tau_2'.\, \mathbf{t_2} : \tau,$$

then

$$\mathsf{x} \vdash [\![\lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1}]\!]_{\lambda^u}^{\lambda^\tau} \simeq_{ctx} [\![\lambda\mathbf{x_2'} : \tau_2'.\, \mathbf{t_2}]\!]_{\lambda^u}^{\lambda^\tau}.$$

The proofs of Theorem 7.5 and of Theorem 7.6 are analogous to their closed-environments analogues except that they rely on Theorem 7.4. They are reported in the companion tech report.

Theorem 7.7 (Compiler Full Abstraction). *If*

$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} : \tau_1' \to \tau_1,$$
$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_1'} : \tau_1'.\, \mathbf{t_2} : \tau_1' \to \tau_1,$$

then

$$\mathbf{x} : \tau' \to \tau \vdash \lambda\mathbf{x_1'}.\, \mathbf{t_1} \simeq_{ctx} \lambda\mathbf{x_2'}.\, \mathbf{t_2} : \tau_1' \to \tau_1 \iff \mathsf{x} \vdash [\![\lambda\mathbf{x_1'}.\, \mathbf{t_1}]\!]_{\mathcal{T}}^{\mathcal{S}} \simeq_{ctx} [\![\lambda\mathbf{x_2'}.\, \mathbf{t_2}]\!]_{\mathcal{T}}^{\mathcal{S}}.$$

Proof. By Theorem 7.6 and Theorem 7.5. $\qquad\square$

$$\mathbf{t_1} + \mathbf{t_2} \simeq_{ctx} \mathbf{t_1} + \mathbf{t_2}$$

$$\langle\!\langle \mathfrak{C} \rangle\!\rangle_\mathsf{n} \big[\mathbf{t_1} + \mathbf{t_2}\big] \Downarrow_\_ \;\Rightarrow\; \langle\!\langle \mathfrak{C} \rangle\!\rangle_\mathsf{n} \big[\mathbf{t_1} + \mathbf{t_2}\big] \Downarrow_\_$$
$$(2)$$

$$\begin{array}{ccc} \langle\!\langle \mathfrak{C} \rangle\!\rangle_\mathsf{n} \gtrsim_\mathsf{n} \mathfrak{C} & & \langle\!\langle \mathfrak{C} \rangle\!\rangle_\mathsf{n} \lesssim_\_ \mathfrak{C} \\ \mathbf{t_1} + \mathbf{t_2} \gtrsim_\_ [\![\mathbf{t_1} + \mathbf{t_2}]\!] & \Uparrow(1) \qquad (3)\Downarrow & \mathbf{t_1} + \mathbf{t_2} \lesssim_\_ [\![\mathbf{t_1}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} + [\![\mathbf{t_2}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} \end{array}$$

$$\mathfrak{C}\Big[[\![\mathbf{t_1} + \mathbf{t_2}]\!]\Big]\Downarrow_\mathsf{n} \;\overset{?}{\Rightarrow}\; \mathfrak{C}\Big[[\![\mathbf{t_1}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} + [\![\mathbf{t_2}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau}\Big]\Downarrow_\_$$

$$[\![\mathbf{t_1} + \mathbf{t_2}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} \overset{?}{\simeq}_{ctx} [\![\mathbf{t_1}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} + [\![\mathbf{t_2}]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau}$$

Figure 18: Proving compiler modularity. Only one direction of this half is presented ($\Rightarrow$), the other one follows by symmetry.

7.3.2. *Compiler Modularity.* Compiler modularity is a property that is stated just between λ^u terms. Intuitively, it states that linking two source terms $\mathbf{t_1}$ and $\mathbf{t_2}$ and compiling the result is contextually-equivalent to compiling $\mathbf{t_1}$ and $\mathbf{t_2}$ individually and then linking the result in the target. Formally, this is captured by Theorem 7.8, where compiler modularity is written with a closed environment as linking is generally a global step.

Theorem 7.8 (Compiler modularity). *If*

$$\mathbf{x_2} : \tau_2' \to \tau_2 \vdash \lambda \mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} : \tau_1' \to \tau_1$$
$$\mathbf{x_1} : \tau_1' \to \tau_1 \vdash \lambda \mathbf{x_2'} : \tau_2'.\, \mathbf{t_2} : \tau_2' \to \tau_2$$

then

$$\emptyset \vdash \big[\![\lambda \mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} + \lambda \mathbf{x_2'} : \tau_2'.\, \mathbf{t_2}\big]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} \simeq_{ctx} \big[\![\lambda \mathbf{x_1'} : \tau_1'.\, \mathbf{t_1}\big]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} + \big[\![\lambda \mathbf{x_2'} : \tau_2'.\, \mathbf{t_2}\big]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau}.$$

The formal setup developed so far (i.e., the logical relation) however, is only built for cross-language reasoning. Since we do not really have a λ^u logical relation, nor do we want to build one, we resort to an analogous of the proof of compiler security – the part that relies on the back translation, except that we will have the same term on both sides of the source contextual equivalence (Figure 18).

Step 1 is given by the correctness of $[\![\cdot]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau}$ (Theorem 7.4) and Step 2 is trivial since a term is equivalent to itself. All that remains to be proven is Step 3, as captured by Lemma 7.9.

Lemma 7.9 (Source linking is related to target linking). *If*

$$\mathbf{x_2} : \tau_2' \to \tau_2 \vdash \lambda \mathbf{x_1'} : \tau_1'.\, \mathbf{t_1} : \tau_1' \to \tau_1$$
$$\mathbf{x_1} : \tau_1' \to \tau_1 \vdash \lambda \mathbf{x_2'} : \tau_2'.\, \mathbf{t_2} : \tau_2' \to \tau_2$$

then

$$\emptyset \vdash (\lambda \mathbf{x_1'} : \tau_1'.\, \mathbf{t_1}) + (\lambda \mathbf{x_2'} : \tau_2'.\, \mathbf{t_2}) \,\square_\mathsf{n}$$

$$\big[\![\lambda \mathbf{x_1'} : \tau_1'.\, \mathbf{t_1}\big]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} + \big[\![\lambda \mathbf{x_2'} : \tau_2'.\, \mathbf{t_2}\big]\!]_{\lambda^\mathsf{u}}^{\lambda^\tau} : (\tau_1' \to \tau_1) \times (\tau_2' \to \tau_2).$$

Given that source and target linking are defined to be syntactically duals, this proof is a mere application of several compatibility lemmas and Lemma 7.3.

8. Mechanically verified proof

Proofs of full abstraction for non-trivial compiler passes are very often only given on paper. The reason is that they are quite involved and require a significant effort to mechanically verify. This is unfortunate because the proofs are often lengthy and non-trivial, so they would benefit from the extra assurance offered by mechanical verification. In this section, we report on our succesful mechanical verification using Coq of the full abstraction proof presented in Section 7.2.

This proof has been a significant undertaking ($\pm$ 2-3 man-months, 11k lines of code excluding comments) resulting in a medium-sized coq development, available online[4]. The proof only assumes a single axiom: functional extensionality, i.e. the property that two functions are provably equal if they produce the same result for all inputs. This is used in the substitution machinery and for proving propositional equality for the folding/unfolding of the well-founded fixpoints that define the logical relations. In this section, we provide an overview of the construction of the proof and some discussion of our experience constructing it.

The Coq version of the proof largely follows the structure of the "on-paper" version. As is often the case in proofs about properties of lambda calculi, a lot of the overhead arises from the definition of the syntax and more precisely, its use of variables and variable binding. Since the POPLmark challenge [Aydemir et al., 2005], the literature has seen a lot of proposals for encoding such definitions in proof assistants. However, our requirements on the encoding go beyond those of the challenge in some places, so that our options for the encoding were a bit more limited. In particular, we require a notion of *simultaneous closing substitutions* (i.e. substitutions that simultaneously substitute closed values for *all* the free variables of a term), we need to deal with two separate lambda calculi (λ^τ and λ^u) without duplicating all the binding infrastructure and finally, we want to keep our options open for extending the proof to System F (see Section 9).

Because of these requirements, we decided to rely on traditional but proven technology and we chose a standard encoding using de Bruijn indices (with a separate well-scopedness judgement). We used the UniDB library[5] for de Bruijn encodngs in Coq (developed by the last author using his experiences working on the Knot framework Keuchel et al. [2016]). UniDB is instantiated with language specific traversal functions of our source and target language and properties of these traversal, from which the variable binding boilerplate like simultaneous substitution and its properties are derived. The library defines a set of Coq type classes as an interface that allows us to use the same notation for the source and target language. On top of what UniDB provides, we have constructed a number of Ltac tactics for automating the construction of language-specific well-typedness, well-scopedness and evaluation proofs.

Encoding the logical relation did not cause major concerns. The step-indexing in the LR could be expressed without many problems using a library for well-founded induction over natural numbers from Coq's standard library. We did not encounter major problems in the original on-paper proof, although we did run into some minor problems, like the need to be more explicit about the required closedness of untyped terms in the value relation.

In summary, this mechanical verification obviously strengthens our trust in the proof. However, we point out that, to the best of our knowledge, it is the first full proof of full

---

[4]`http://people.cs.kuleuven.be/dominique.devriese/permanent/facomp-stlc-coq.tar.xz`

[5]Available at: https://github.com/skeuchel/unidb-coq

abstraction for a non-trivial compiler pass (see Section 10 for a discussion of related work). As such, it demonstrates that such proofs are within reach of current tools like Coq, although there is room for improvement: 2-3 man-months is more effort than we would hope to spend for a proof that we have already done in much detail on paper. Conversely, the fact that our proof did not uncover any major problem and most of the effort went into dealing with variable binding, well-typedness proofs etc., also shows that this sort of logical relation-based proofs of full abstraction lend themselves well to mechanisation and the level of detail of our original proof is a good level to aim for.

9. Discussion and Future Work

Our interest in fully-abstract compilation comes from a security perspective. We think that a fully-abstract compiler from realistic source languages to a form of assembly that is efficiently executable by processors has important security applications (combining trusted and untrusted code at the assembly level and compartmentalising applications). So far, it remains unclear precisely which security properties are preserved by fully abstract compilers, although it seems that at least important security properties like noninterference [Bowman and Ahmed, 2015] are. Unless targeting typed assembly language [Morrisett et al., 1999], a crucial step of a secure compiler is a form of secure type erasure. The contribution of this paper is mostly the proof technique that proves the type erasure step secure. We intend to reuse this proof technique in other settings.

There are a number of important problems that need to be solved in order to develop a realistic fully-abstract compiler. Several widely-implemented high-level language features present significant challenges: parametric polymorphism, (higher-order) references, exceptions etc. Generally, we believe that low-level assembly languages should be defined that are not only efficiently executable but also provide sufficient abstraction features to enable fully abstract compilation of such standard programming language features. For now, it remains an open question whether this is feasible. Let us zoom in on some of these features in more detail. A long-standing open problem is fully-abstract compilation of parametric polymorphism to a form of operational sealing primitives [Sumii and Pierce, 2007, Matthews and Ahmed, 2008, Neis et al., 2009]. More concretely, several researchers have developed interesting results about fully-abstract compilation from System F to λ^{seal} (an untyped lambda calculus with sealing primitives), but a fully-abstract compiler in this setting has so far only been conjectured. We believe that the problem is quite related to the one tackled in this paper. Without providing details (for space reasons), an exact back-translation from λ^{seal} to System F seems possible, but only if we assume a form of generally recursive type constructors of kind $* \to *$, which we cannot add to System F without causing other problems for the compilation. We have been working on a proof using an approximate back-translation, but we ran into an unexpected problem: the conjectured full abstraction of Sumii-Pierce's compiler is false. We will report on this further in future work.

In other settings, it is also not clear whether it is possible to construct a fully-abstract compiler. For example, if we add typed, higher-order references to λ^τ and untyped references to λ^u, it is not clear if a fully-abstract compiler can be devised. The problem is essentially to choose a representation for typed references and a way of manipulating them that reconciles a number of requirements: (1) trusted code reading from a reference always produces a type-correct value, (2) trusted code writing a type-correct value to a reference always works, (3) untrusted code should be able to read/write type-correct values from references, (4)

dynamic type checks or wrappers may only be added where the context could also choose to fail for other reasons (i.e. not at the time of reading/writing a reference by trusted code), (5) efficiency: we do not want to check the contents of all references every time control is passed from trusted code to the context. Several obvious solutions do not work: representing references as objects with read and write methods violates requirement (4), just checking the contents of a reference when it is received from the context is not enough to guarantee (1) and (2). We intend to explore a solution based on trusted but abstract read/write/alloc methods (using sealing primitives as used for parametric polymorphism) but this remains speculation for the moment.

Another interesting problem when compiling to an assembly language is the enforcement of well-bracketed control flow. The question is essentially how to represent return pointers at the assembly level. Even if we prevent functions from accessing parts of the stack and only give them access to an opaque invokable return pointer, they still have ways to misuse them [Patrignani et al., 2016]. Imagine a trusted assembly function f invoking an untrusted g. Additionally, assume that g in turn re-invokes f and f simply re-invokes g again. Now g might attempt to invoke the wrong return pointer, returning on its first invocation without first returning on the second. Such an attack breaks the well-bracketedness of control flow that trusted code may rely on in languages without call/cc primitives [Dreyer et al., 2010]. [Ahmed and Blume, 2011] have demonstrated a solution for this problem which exploits parametric polymorphism to enforce the invocation of the correct continuation, and it is interesting to see if their work can be reused as an intermediate step on the way to assembly language.

On a technical level, we expect few problems for applying our technique of approximate back-translation to all of these settings. The Hur-Dreyer-inspired cross-language logical relations can be applied in diverse settings including ML and assembly and support references (through Kripke worlds), parametric polymorphism (through quantification over abstract type interpretations as relations) and well-bracketed control flow guarantees (through public/private transitions in the transition systems stored in the worlds). We have also shown in this paper that they can be easily modified to an asymmetric setting.

10. Related Work

Secure compilation through full-abstraction was pioneered by Abadi [1999] and successfully applied to many different settings [Patrignani et al., 2015, Fournet et al., 2013, Bowman and Ahmed, 2015, Ahmed and Blume, 2011, 2008, Tse and Zdancewic, 2004, Shikuma and Igarashi, 2007, Abadi and Plotkin, 2012, Jagadeesan et al., 2011, Riecke, 1993, Ritter and Pitts, 1995, Mitchell, 1993, McCusker, 1996, Smith, 1998, Larmuseau et al., 2015].

Recently, Gorla and Nestman [2014] and Parrow [2014] have argued against the use of the mere existence of fully-abstract translations as a measure of language expressiveness, because very often fully abstract translations exist but are in some sense degenerate, uninteresting and/or unrealistic. These arguments are not directly relevant to our work, because we are not interested in the mere existence of a fully abstract compiler as a measure of language expressiveness, but we prove the fully abstractness of a specific, realistic compiler.

Some secure compilation works prove compiler full-abstraction using logical relations. Ahmed and Blume [2011] and Ahmed and Blume [2008] proved that typed closure conversion and CPS transformation are fully-abstract when compiling from System F and the simply-typed λ-calculus (respectively) to System F. Tse and Zdancewic [2004] started a line of

work to compile the dependency core calculus of Abadi et al. [1999] (DCC) into System F, effectively proving that non interference can be encoded with parametricity. They achieve a property analogous to fully-abstract compilation where contextual equivalence is replaced with non-interference. Due to an imprecision in their proof, the result of Tse and Zdancewic does not hold; Shikuma and Igarashi [2007] refined their result for a weaker form of DCC. A fully-abstract translation from DCC to System F was provided by Bowman and Ahmed [2015], and that is the closest work to what is presented here. The formal machinery adopted by Bowman and Ahmed does appear a bit heavier than the one presented here. Specifically, we do not need a new logical relation to prove well-foundedness of the back-translation. The secure compilation of DCC to System F is quite different from our setting, since our target language is untyped and our source and target languages are both non-terminating.

In a paper that is closely related to our work, New et al. [2016] prove full-abstraction of closure conversion of a simply-typed lambda calculus with recursive types into a simply typed language with exceptions and an effect system to track exceptions. To achieve this, they apply a back-translation using a universal type, similar to our UVal. They present a very interesting comparison to a previous version of this work. They explain how one can see our approximation of target-language terms as an underapproximation because we only back-translate a part of the behaviour of the term. While they do not need this under-approximation, because their source language includes recursive types, they apply an over-approximation because their universal type can embed more than just their target language: like our UVal, it can embed the full untyped lambda calculus, rather than just the subset of terms that are well-typed in the target language.

Independently from our work, the idea of using an *approximate* back-translation was also mentioned recently by Schmidt-Schauß et al. [2015]. In this work, Schmidt-Schauß et al. present a framework for defining and reasoning about fully abstract compilation and related notions in a wide variety of languages. Using the name *families of translations*, they define what we call an approximate back-translation (in relation to full abstraction of a language embedding). 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. The idea is to use an approximate back-translation that unrolls applications of fix n times in the nth approximation. The proof is not very detailed, but seems a lot simpler than ours. Partly this is 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 the proof in this paper may be simplified by factoring our compiler into two separate compilation passes: (1) embedding λ^τ into λ^τ with recursive types (using an approximate back-translation to prove full abstraction of the embedding) and (2) compiling λ^τ with recursive types into λ^u as we do here (using a full, non-approximate back-translation to prove full abstraction).

Many other secure compilation works prove full-abstraction by replacing target-level contextual equivalence with another equivalent equivalence (most times it is trace equivalence or bisimilarity) [Fournet et al., 2013, Abadi and Plotkin, 2012, Jagadeesan et al., 2011, Patrignani et al., 2015]. These works rely on additional results of the equivalence used for full-abstraction to hold, and this can complicate and lengthen proofs relying on this other technique. Earlier, McCusker [1996] has shown that proving full abstraction of a compiler can be simplified by limiting the back-translation to contexts that are in a certain sense *compact*. This is related to our approximate back-translations, though not quite the same. A downside of McCusker's approach is that it does not always seem clear how to characterize the compact elements in a language.

To the best of our knowledge, the only paper on secure compilation that comes with a mechanised proof of full abstraction (or a variant of it), is by Juglaret et al. [2016]. They propose and study *Secure Compartmentalizing Compilation*: a variant of full abstraction that supports unsafe source languages (where full abstraction cannot be expected to hold for components that exhibit undefined behaviour), and includes a notion of modularity. They report on a Coq mechanization of some of their results, but it is not a "full mechanization" like ours, in the sense that they keep many lemmas and results as unproven assumptions.

As mentioned in Section 1, the presented proof technique borrows from recent results in compiler correctness [Hur and Dreyer, 2011, Benton and Hur, 2010, 2009]. These results build cross-language logical relation based on a common language specification in order to prove compiler correctness. Benton and Hur [2009] provided a correct compiler from a call-by-value λ-calculus as well as for System F with recursion to a SECD machine [Benton and Hur, 2010]. Hur and Dreyer [2011] devised a correct compiler from an idealised ML to assembly. The techniques devised in these works were further developed into Relational Transition Systems (RTS) and Parametric Inter-Language Simulations (PILS) in order to prove both vertically- and horizontally-composable compiler correctness [Hur et al., 2012, Neis et al., 2015]. A different approach to cross-language relations could have been adopting a Matthews and Findler-style multi-language semantics, where source and target language are combined [Matthews and Findler, 2009]. For example, Perconti and Ahmed [2014] devised a two-step correct compiler for System F with existential and recursive types to typed assembly language using multi-language logical relations. As the focus of this work is compiler full-abstraction for a compiler that is not multi-pass, there was no necessity to use RTS nor multi-language systems. However, were our compiler to be multi-pass, we would have had to resort to a different proof technique like those described above. This is because even if compiler full abstraction scales to multi-pass compilers (i.e., it is vertically composable), compiler correctness proven with logical relations does not. As compiler full abstraction ought to always be accompanied by a compiler correctness result (Theorem 6.1 in our case), and since both can be proven with a single proof technique, we believe that to prove compiler full abstraction for a multipass compiler one would have to port the findings of this paper to one of the techniques above.

Some elements of our proof technique are reminiscent of techniques from the field of denotational semantics. First, our family of types $UVal_n$ can be seen as a kind of syntactical version of an iteratively constructed Scott model for the untyped lambda calculus [Scott, 1976]. In fact, the analogue of our $UVal_n$ used by New et al. [2016] extends this correspondence to a language with effects (using an exception monad to model a target language with exceptions). We note also that using a family of finite approximations (like our $UVal_n$ types) to interpret a recursive type (like the type $UVal$ discussed in the introduction) is quite standard in denotational semantics [MacQueen et al., 1984].

11. Conclusion

This paper presented a novel proof technique for proving compiler full-abstraction based on asymmetric, cross-language logical relations. The proof technique revolves around an approximate back-translation from target terms (and contexts) to source terms (and contexts). The back-translation is approximate in the sense that the context generated by the back-translation may diverge when the target-level counterpart would not, but not vice versa. The proof technique is demonstrated for a compiler from a simply-typed λ-calculus without

recursive types to the untyped λ-calculus. That compiler is proven to be fully-abstract in Coq, and this is the first such result for fully abstract compilation proofs. Although logical relations have been used for full-abstraction proofs, this is the first usage of cross-language logical relations for compiler full-abstraction targeting an untyped language. We believe the techniques developed in this paper scale to languages with more advanced functionalities and they can be used to prove compiler full-abstraction in richer settings.

Acknowledgements

References

M. Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, 1999. ISBN 3-540-66130-1.

M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15:8:1–8:29, July 2012. ISSN 1094-9224. doi: 10.1145/2240276.2240279.

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Principles of Programming Languages*, pages 147–160. ACM, 1999. doi: 10.1145/292540.292555.

P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations*, pages 171–185, 2012.

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming*, pages 157–168. ACM, 2008. doi: 10.1145/1411204.1411227.

A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming*, pages 431–444. ACM, 2011. doi: 10.1145/2034773.2034830.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Theorem Proving in Higher Order Logics*, pages 50–65. Springer Berlin Heidelberg, 2005. doi: 10.1007/11541868_4.

N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming*, volume 44, pages 97–108. ACM, 2009. doi: 10.1145/1596550.1596567.

N. Benton and C.-K. Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical report, MSR, 2010.

W. J. Bowman and A. Ahmed. Noninterference for free. In *International Conference on Functional Programming*. ACM, 2015.

P.-L. Curien. Definability and full abstraction. *Electron. Notes Theor. Comput. Sci.*, 172:301–310, 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.02.011.

D. Devriese, M. Patrignani, and F. Piessens. Fully abstract compilation by approximate back-translation. In *Principles of Programming Languages*. ACM, 2016.

D. Devriese, M. Patrignani, and F. Piessens. Modular fully abstract compilation by approximate back-translation: Technical appendix. Technical Report CW 702, Dept. of Computer Science, KU Leuven, 2017.

D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *International Conference on Functional Programming*, pages 143–156, 2010. doi: 10.1145/1863543.1863566.

C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *Principles of Programming Languages*, pages 371–384. ACM, 2013. doi: 10.1145/2429069.2429114.

D. Gorla and U. Nestman. Full abstraction for expressiveness: History, myths and facts. *Math. Struct. Comp. Science*, 2014.

C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Principles of Programming Languages*, pages 133–146. ACM, 2011. doi: 10.1145/1926385.1926402.

C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *Principles of Programming Languages*, pages 59–72. ACM, 2012. doi: 10.1145/2103656.2103666.

R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *Computer Security Foundations Symposium*, pages 161–174. IEEE Computer Society, 2011. doi: 10.1109/CSF.2011.18.

Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *CSF*. IEEE Computer Society Press, July 2016. URL https://arxiv.org/abs/1602.04503.

A. Kennedy. Securing the .NET programming model. *Theor. Comput. Sci.*, 364(3):311–317, Nov. 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.08.014.

S. Keuchel, S. Weirich, and T. Schrijvers. Needle & knot: Binder boilerplate tied up. In *European Symposium on Programming*, pages 419–445. Springer Berlin Heidelberg, 2016. doi: 10.1007/978-3-662-49498-1_17.

A. Larmuseau, M. Patrignani, and D. Clarke. A secure compiler for ML modules. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 29–48, 2015. doi: 10.1007/978-3-319-26529-2_3. URL http://dx.doi.org/10.1007/978-3-319-26529-2_3.

D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Principles of Programming Languages*, pages 165–174. ACM, 1984. doi: 10.1145/800017.800528.

J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Programming Languages and Systems*, volume 4960 of *LNCS*, pages 16–31. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-78739-6_2.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31:12:1–12:44, Apr. 2009. ISSN 0164-0925. doi: 10.1145/1498926.1498930.

G. McCusker. Full abstraction by translation. *Advances in Theory and Formal Methods of Computing*, 1996.

J. C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141 – 163, 1993. ISSN 0167-6423. doi: 10.1016/0167-6423(93)90004-9.

G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on*

Compiler Support for System Software, pages 25–35, 1999.

G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *International Conference on Functional Programming*, pages 135–148. ACM, 2009. doi: 10.1145/1596550. 1596572.

G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *International Conference on Functional Programming*. ACM, 2015.

M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming*, pages 103–116. ACM, 2016. doi: 10.1145/2951913.2951941.

J. Parrow. General conditions for full abstraction. *Math. Struct. Comp. Science*, 2014.

M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, Apr. 2015. ISSN 0164-0925. doi: 10.1145/2699503.

M. Patrignani, D. Devriese, and F. Piessens. On Modular and Fully Abstract Compilation. In *CSF 2016*, 2016.

J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 128–148, 2014.

B. C. Pierce. *Types and programming languages*. MIT press, 2002.

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977. doi: 10.1016/0304-3975(77)90044-5.

J. G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3:387–415, 12 1993. ISSN 1469-8072. doi: 10.1017/ S0960129500000293.

E. Ritter and A. M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 397–413. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-59048-4. doi: 10.1007/BFb0014067.

M. Schmidt-Schauß, D. Sabel, J. Niehren, and J. Schwinghammer. Observational program calculi and the correctness of translations. *Theoretical Computer Science*, 577:98 – 124, 2015. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2015.02.027.

D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976. doi: 10.1137/0205037.

N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed λ-calculus. In M. Okada and I. Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *LNCS*, pages 301–315. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-77505-8_24.

S. F. Smith. The coverage of operational semantics. In *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 307–346. Cambridge University Press, 1998.

E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *Theor. Comput. Sci.*, 375 (1-3):169–192, Apr. 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.032.

S. Tse and S. Zdancewic. Translating dependency into parametricity. In *International Conference on Functional Programming*, pages 115–125. ACM, 2004. doi: 10.1145/1016850. 1016868.