Fully-Abstract Compilation by Approximate Back-Translation

Dominique Devriese Marco Patrignani Frank Piessens

iMinds-Distrinet, Computer Science dept., KU Leuven, Belgium

first.last @ cs.kuleuven.be

Abstract

A compiler is *fully-abstract* if the compilation from source language programs to target language programs preserves and reflects 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 the simply-typed λ-calculus (λ^{τ}) to the 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 proof 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. We believe this proof technique can scale to challenging settings and enable simpler, more scalable proofs of compiler full-abstraction.

Keywords Fully-abstract compilation, cross-language logical relations, step-indexing, compiler security, secure compilation

We use **green** and pink to typeset resp. source and target language terms; we recommend to 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 [1, 12]. 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 [1, 2, 11, 16, 24]. 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.

POPL '16 Month d–d, 2016, City, ST, Country

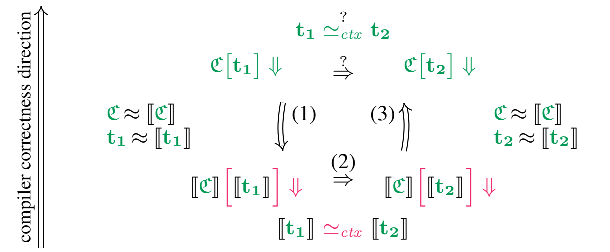

Figure 1. Proving one half of full-abstraction: compiler correctness. Only one direction of this half is presented ($\Rightarrow$), the other one follows by symmetry.

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 **green**, **bold** font, while elements of the target language are typeset in a pink, $\mathsf{sans\text{-}serif}$ 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 ($[\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$) [1]. 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: $t \simeq_{ctx} t' \overset{\mathsf{def}}{=} \forall \mathfrak{C}, \mathfrak{C}[t] \Downarrow \iff \mathfrak{C}[t'] \Downarrow$. The universal quantification over contexts $\mathfrak{C}$ ensures that the results produced by t and t' are the same [9, 27].

The full-abstraction property can be split into two parts: the right-to-left implication and the left-to-right implication, which we call compiler *correctness* and compiler *security* respectively.

Compiler correctness ($\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} \Leftarrow [\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$) requires that if the compiler produces equivalent target programs, then the source programs must have been equivalent. Intuitively, it is clear that if programs with different source language behaviour become equivalent after compilation, the compiler must have incorrectly compiled at least one of them.

We build on cross-language logical relations: a technique that has recently been proposed for proving compiler correctness [6, 7, 13]. 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 needs 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 $\mathfrak{C}[\mathbf{t_1}] \Downarrow \Rightarrow \mathfrak{C}[\mathbf{t_2}] \Downarrow$.

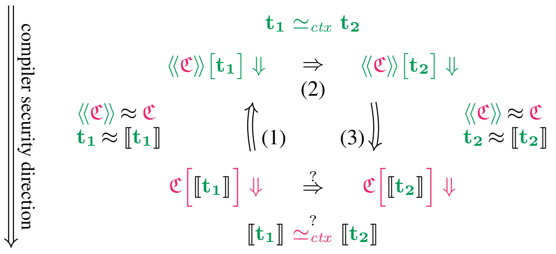

Figure 2. Proving the other half of full-abstraction: compiler security.

The idea of the approach is to define a cross-language logical relation $\mathbf{t} \approx \mathsf{t}$ that expresses when a compiled term t behaves as a target-level version of source-level term $\mathbf{t}$. This logical relation is not compiler-specific: it should be understood as a specification of a target-level calling convention rather than a precise compiler result. 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 compiler correctness follows. 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 $\mathfrak{C}[\mathbf{t_1}] \approx [\![\mathfrak{C}]\!][[\![\mathbf{t_1}]\!]]$ and the same for $\mathbf{t_2}$. If the logical relations are constructed adequately, two terms are related if they 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 compiler correctness is finished.

Compiler security ($\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} \Rightarrow [\![\mathbf{t_1}]\!] \simeq_{ctx} [\![\mathbf{t_2}]\!]$) 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 compiler correctness for proving compiler security, as depicted in Fig. 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 a sufficiently-powerful source language, we can construct a *back-translation* $\langle\langle\mathfrak{C}\rangle\rangle$ for any target-level context $\mathfrak{C}$ that returns a valid source-level context $\mathfrak{C}$. Using the same logical relation as above, it then suffices to prove that $\langle\langle\mathfrak{C}\rangle\rangle \approx \mathfrak{C}$ for any valid context. If we have that $\langle\langle\mathfrak{C}\rangle\rangle \approx \mathfrak{C}$ and $\mathbf{t_1} \approx [\![\mathbf{t_1}]\!]$, and similarly for $\mathbf{t_2}$, compositionality and adequacy of the logical relation 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:

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

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 where the latter work with values in UVal where the original λ^{u} context worked 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?

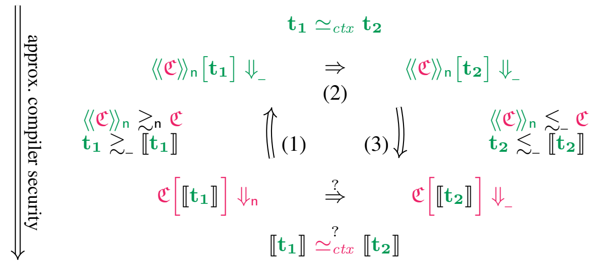

Figure 3. Proving compiler security using an n-*approximate* back-translation. An _ subscript indicates *any* number of steps.

In this paper we answer positively to this question and develop a general proof 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 emulation 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.

The proof starts, again, from an arbitrary target-level context $\mathfrak{C}$ and the knowledge that $\mathfrak{C}[\mathsf{t}] \Downarrow_{\mathsf{n}}$ and we call n the number of reduction steps in this execution. 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 $\mathfrak{C}[\mathsf{t}]$ 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 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 has to be *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 approximate back-translation $\langle\langle\mathfrak{C}\rangle\rangle_{\mathbf{n}}$ is not sufficient for the context to accurately simulate the behavior 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 $\mathfrak{C}[\mathsf{t}]$. 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:

$$\mathrm{UVal}_0 \overset{\mathsf{def}}{=} \mathtt{Unit}$$
$$\mathrm{UVal}_{\mathsf{n+1}} \overset{\mathsf{def}}{=} \mathtt{Unit} \uplus \mathcal{B} \uplus (\mathrm{UVal}_{\mathsf{n}} \times \mathrm{UVal}_{\mathsf{n}}) \uplus (\mathrm{UVal}_{\mathsf{n}} \uplus \mathrm{UVal}_{\mathsf{n}}) \uplus (\mathrm{UVal}_{\mathsf{n}} \to \mathrm{UVal}_{\mathsf{n}}).$$

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 $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{n} \lesssim \mathfrak{C}$ 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.

To summarise, the contributions of this work are:

- a new and general proof technique for proving compiler full-abstraction using asymmetric, cross-language logical relations and targeting untyped languages;
- an instantiation of that proof technique to fully-abstractly compile 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. Firstly it formalises the source and target languages λ^τ and λ^u (Section 2). Secondly it presents the cross-language logical relations that are used 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. The paper then presents the approximate back-translation (Section 5) which is used when proving compiler full-abstraction (Section 6). Finally, we discuss the presented results (Section 7), related work (Section 8) and we conclude in Section 9.

2. Source and Target Languages

The source language λ^τ is presented in Fig. 4. It is a standard, 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 $\mathbb{C}$. 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 e.g. [26]): 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) [26, §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^μ as well keeps the compiler simpler. Working with the more standard 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.

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

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

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

$$\triangleright (0) \stackrel{\mathsf{def}}{=} (0) \qquad (k) \sqsupset_{\triangleright} (k') \stackrel{\mathsf{def}}{=} k < k'$$

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

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

Figure 6. Logical relations: Worlds.

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 an error term wrong in case of type errors (e.g. projecting from a non-pair value, case splitting on a non-sum value etc.). The well-scopedness rules are unsurprising and the evaluation rules again use evaluation contexts to impose a strict evaluation order. Again, we omit the definition of program contexts $\mathfrak{C}$ (expressions with a single hole in place of a subterm) and their well-scopedness judgement $\vdash \mathfrak{C} : \Gamma' \to \Gamma$, whose inductive definition guarantees that substituting a well-scoped term $\Gamma' \vdash \mathsf{t}$ for the hole produces a well-scoped result term $\Gamma \vdash \mathfrak{C}[\mathsf{t}]$.

The interested reader can find full formalisation and proofs in the companion technical report.

3. Logical relations

This section presents the Kripke, step-indexed logical relations that are used to prove compiler full-abstraction. Firstly, 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. Note that part of these logical relations, namely the novel insights needed to provide compiler full-abstraction, are postponed until Section 5.2. 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 [13]. Essentially, we instantiate their language-generic logical relations to λ^τ and λ^u and simplify the complexities deriving from the System F type system, public/private transitions, references and garbage collections.

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 $\triangleright$ 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 kind of *token type* $\mathtt{EmulDV}_{\mathsf{n};\mathsf{p}}$, indexed by a non-negative number n and a value p $::=$ `precise` $\mid$ `imprecise`. This token type

$$
\begin{aligned}
\mathbf{t} &::= \mathbf{v} \mid \mathbf{x} \mid \mathbf{t}\ \mathbf{t} \mid \mathbf{t}.1 \mid \mathbf{t}.2 \mid \langle \mathbf{t}, \mathbf{t} \rangle \mid \text{inl}\ \mathbf{t} \mid \text{inr}\ \mathbf{t} \mid \text{case}\ \mathbf{t}\ \text{of inl}\ \mathbf{x_1} \mapsto \mathbf{t} \mid \text{inr}\ \mathbf{x_2} \mapsto \mathbf{t} \mid \mathbf{t}; \mathbf{t} \mid \text{if}\ \mathbf{t}\ \text{then}\ \mathbf{t}\ \text{else}\ \mathbf{t} \mid \text{fix}_{\tau_1 \to \tau_2}\ \mathbf{t} \\
\mathbf{v} &::= \mathtt{unit} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda \mathbf{x} : \tau.\ \mathbf{t} \mid \langle \mathbf{v}, \mathbf{v} \rangle \mid \text{inl}\ \mathbf{v} \mid \text{inr}\ \mathbf{v} \\
\tau &::= \mathtt{Unit} \mid \mathtt{Bool} \mid \tau \to \tau \mid \tau \times \tau \mid \tau \uplus \tau \qquad\qquad \mathbf{\Gamma} ::= \emptyset \mid \mathbf{\Gamma}, \mathbf{x} : \tau \\
\mathbb{C} &::= [\cdot] \mid \mathbb{C}\ \mathbf{t} \mid \mathbf{v}\ \mathbb{C} \mid \mathbb{C}.1 \mid \mathbb{C}.2 \mid \langle \mathbb{C}, \mathbf{t} \rangle \mid \langle \mathbf{v}, \mathbb{C} \rangle \mid \text{inl}\ \mathbb{C} \mid \text{inr}\ \mathbb{C} \mid \text{case}\ \mathbb{C}\ \text{of inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \mid \text{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2} \mid \mathbb{C}; \mathbf{t} \mid \text{if}\ \mathbb{C}\ \text{then}\ \mathbf{t}\ \text{else}\ \mathbf{t} \\
&\quad \mid \text{fix}_{\tau_1 \to \tau_2}\ \mathbb{C}
\end{aligned}
$$

$$
\frac{}{\mathbf{\Gamma} \vdash \mathtt{unit} : \mathtt{Unit}} \qquad
\frac{}{\mathbf{\Gamma} \vdash \mathtt{true} : \mathtt{Bool}} \qquad
\frac{(\mathbf{x} : \tau) \in \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash \mathbf{x} : \tau} \qquad
\frac{\mathbf{\Gamma}, (\mathbf{x} : \tau) \vdash \mathbf{t} : \tau'}{\mathbf{\Gamma} \vdash \lambda \mathbf{x} : \tau.\ \mathbf{t} : \tau'} \qquad
\frac{\mathbf{\Gamma} \vdash \mathbf{t_1} : \tau_1 \quad \mathbf{\Gamma} \vdash \mathbf{t_2} : \tau_2}{\mathbf{\Gamma} \vdash \langle \mathbf{t_1}, \mathbf{t_2} \rangle : \tau_1 \times \tau_2}
$$

$$
\frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau_1 \times \tau_2}{\mathbf{\Gamma} \vdash \mathbf{t}.1 : \tau_1} \qquad
\frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau' \to \tau \quad \mathbf{\Gamma} \vdash \mathbf{t}' : \tau'}{\mathbf{\Gamma} \vdash \mathbf{t}\ \mathbf{t}' : \tau} \qquad
\frac{\mathbf{\Gamma} \vdash \mathbf{t} : \tau}{\mathbf{\Gamma} \vdash \text{inl}\ \mathbf{t} : \tau \uplus \tau'} \qquad
\frac{\mathbf{\Gamma} \vdash \mathbf{t} : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)}{\mathbf{\Gamma} \vdash \text{fix}_{\tau_1 \to \tau_2}\ \mathbf{t} : \tau_1 \to \tau_2}
$$

$$
\frac{\begin{array}{c}\mathbf{\Gamma} \vdash \mathbf{t} : \tau_1 \uplus \tau_2 \\ \mathbf{\Gamma}, (\mathbf{x_1} : \tau_1) \vdash \mathbf{t_1} : \tau \quad \mathbf{\Gamma}, (\mathbf{x_2} : \tau_2) \vdash \mathbf{t_2} : \tau\end{array}}{\mathbf{\Gamma} \vdash \text{case}\ \mathbf{t}\ \text{of inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \mid \text{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2} : \tau} \qquad
\frac{\begin{array}{c}\mathbf{\Gamma} \vdash \mathbf{t} : \mathtt{Bool} \\ \mathbf{\Gamma} \vdash \mathbf{t_1} : \tau \quad \mathbf{\Gamma} \vdash \mathbf{t_2} : \tau\end{array}}{\mathbf{\Gamma} \vdash \text{if}\ \mathbf{t}\ \text{then}\ \mathbf{t_1}\ \text{else}\ \mathbf{t_2} : \tau} \qquad
\frac{\begin{array}{c}\mathbf{\Gamma}, \vdash \mathbf{t_1} : \mathtt{Unit} \\ \mathbf{\Gamma}, \vdash \mathbf{t_2} : \tau\end{array}}{\mathbf{\Gamma} \vdash \mathbf{t_1}; \mathbf{t_2} : \tau}
$$

$$
\frac{\mathbf{t} \hookrightarrow \mathbf{t}'}{\mathbb{C}[\mathbf{t}] \hookrightarrow \mathbb{C}[\mathbf{t}']} \qquad
\frac{}{(\lambda \mathbf{x} : \tau.\ \mathbf{t})\ \mathbf{v} \hookrightarrow \mathbf{t}[\mathbf{v}/\mathbf{x}]} \qquad
\frac{}{\langle \mathbf{v_1}, \mathbf{v_2} \rangle.1 \hookrightarrow \mathbf{v_1}} \qquad
\frac{}{\begin{array}{l}\text{case inl}\ \mathbf{v}\ \text{of inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \\ \qquad \mid \text{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2}\end{array} \hookrightarrow \mathbf{t_1}[\mathbf{v}/\mathbf{x_1}]}
$$

$$
\frac{\begin{array}{c}\mathbf{v} \equiv \mathtt{true} \Rightarrow \mathbf{t}' \equiv \mathbf{t_1} \\ \mathbf{v} \equiv \mathtt{false} \Rightarrow \mathbf{t}' \equiv \mathbf{t_2}\end{array}}{\text{if}\ \mathbf{v}\ \text{then}\ \mathbf{t_1}\ \text{else}\ \mathbf{t_2} \hookrightarrow \mathbf{t}'} \qquad
\frac{}{\text{unit}; \mathbf{t} \hookrightarrow \mathbf{t}} \qquad
\frac{}{\begin{array}{l}\text{fix}_{\tau_1 \to \tau_2}\ (\lambda \mathbf{x} : \tau_1 \to \tau_2.\ \mathbf{t}) \hookrightarrow \\ \quad \mathbf{t}[(\lambda\ \mathbf{y} : \tau_1.\ \text{fix}_{\tau_1 \to \tau_2}\ (\lambda \mathbf{x} : \tau_1 \to \tau_2.\ \mathbf{t})\ \mathbf{y})/\mathbf{x}]\end{array}}
$$

Figure 4. Syntax, static and dynamic semantics of the source language λ^{τ} (selection of).

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

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

$$
\frac{}{\begin{array}{l}\text{case inl}\ \mathsf{v}\ \text{of inl}\ \mathsf{x_1} \mapsto \mathsf{t_1} \\ \qquad \mid \text{inr}\ \mathsf{x_2} \mapsto \mathsf{t_2}\end{array} \hookrightarrow \mathsf{t_1}[\mathsf{v}/\mathsf{x_1}]} \qquad
\frac{\begin{array}{c}\mathsf{v} \equiv \mathtt{true} \Rightarrow \mathsf{t}' \equiv \mathsf{t_1} \quad \mathsf{v} \equiv \mathtt{false} \Rightarrow \mathsf{t}' \equiv \mathsf{t_2} \\ (\mathsf{v} \not\equiv \mathtt{true} \wedge \mathsf{v} \not\equiv \mathtt{false}) \Rightarrow \mathsf{t}' \equiv \mathsf{wrong}\end{array}}{\text{if}\ \mathsf{v}\ \text{then}\ \mathsf{t_1}\ \text{else}\ \mathsf{t_2} \hookrightarrow \mathsf{t}'} \qquad
\frac{}{\mathtt{unit}; \mathsf{t} \hookrightarrow \mathsf{t}}
$$

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

is not an λ^{τ} type; it is needed because of the approximate back-translation. When necessary, we use a function `repEmul`() for converting a pseudo-type to a λ^{τ} type. The function replaces all occurrences of $\mathtt{EmulDV}_{\mathsf{n;p}}$ with a concrete λ^{τ} type. We postpone the definitions and explanations of $\mathtt{EmulDV}_{\mathsf{n;p}}$ and of $\mathcal{V}[\![\mathtt{EmulDV}_{\mathsf{n;p}}]\!]_{\square}$ to Section 5.2, after we have given some more information about the back-translation. We will sometimes silently use a normal type where a pseudo-type is expected, which makes sense since the syntax for the latter is a superset of the former.

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., [6]). 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 $\mathcal{G}[\![\mathbf{\Gamma}]\!]_{\square}$ relates substitutions instantiating a context $\mathbf{\Gamma}$, which simply requires that substitutions for all variables in the context are related at their types. For open terms, we define a logical relation $\hat{\mathbf{\Gamma}} \vdash \mathbf{t} \ \square_{\mathsf{n}} \ \mathsf{t} : \hat{\tau}$. 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 $\hat{\mathbf{\Gamma}}$ if the first is well-typed and if closing $\mathbf{t}$ and t with substitutions related at pseudo-context $\hat{\mathbf{\Gamma}}$ produces terms related at pseudo-type $\hat{\tau}$, in any world $\underline{W}$ such that $\mathsf{lev}(\underline{W}) \leq n$. If $\hat{\mathbf{\Gamma}} \vdash \mathbf{t} \ \square_{\mathsf{n}} \ \mathsf{t} : \hat{\tau}$ for any n, then we write

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

$$\hat{\tau} ::= \mathtt{Bool} \mid \mathtt{Unit} \mid \hat{\tau} \times \hat{\tau} \mid \hat{\tau} \uplus \hat{\tau} \mid \hat{\tau} \to \hat{\tau} \mid \mathtt{EmulDV}_{\mathsf{n;p}} \qquad\qquad \hat{\mathbf{\Gamma}} ::= \emptyset \mid \hat{\mathbf{\Gamma}}, \mathbf{x} : \hat{\tau}$$

$$\mathtt{repEmul}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \cdots \text{ (to be defined later, in Fig. 12)}$$

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

$$\mathsf{oftype}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \left\{ \mathsf{v} \;\middle|\; \begin{array}{ll} \mathsf{v} = \mathsf{unit} & \text{if } \hat{\tau} = \mathtt{Unit} \\ \mathsf{v} = \mathsf{true} \text{ or } \mathsf{v} = \mathsf{false} & \text{if } \hat{\tau} = \mathtt{Bool} \\ \exists \mathsf{t}.\, \mathsf{v} = \lambda \mathsf{x}.\, \mathsf{t} & \text{if } \exists \hat{\tau_1}, \hat{\tau_2}.\, \hat{\tau} = \hat{\tau_1} \to \hat{\tau_2} \\ \exists \mathsf{v}_1 \in \mathsf{oftype}(\hat{\tau_1}), \mathsf{v}_2 \in \mathsf{oftype}(\hat{\tau_2}).\, \mathsf{v} = \langle \mathsf{v}_1, \mathsf{v}_2 \rangle & \text{if } \exists \hat{\tau_1}, \hat{\tau_2}.\, \hat{\tau} = \hat{\tau_1} \times \hat{\tau_2} \\ \exists \mathsf{v}_1 \in \mathsf{oftype}(\hat{\tau_1}).\, \mathsf{v} = \mathsf{inl}\ \mathsf{v}_1 \text{ or } \exists \mathsf{v}_2 \in \mathsf{oftype}(\hat{\tau_2}).\, \mathsf{v} = \mathsf{inr}\ \mathsf{v}_2 & \text{if } \exists \hat{\tau_1}, \hat{\tau_2}.\, \hat{\tau} = \hat{\tau_1} \uplus \hat{\tau_2} \end{array} \right\}$$

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

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

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

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

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

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

$$\mathcal{V}[\![\hat{\tau_1} \times \hat{\tau_2}]\!]_\square \stackrel{\mathsf{def}}{=} \left\{ (\underline{W}, \mathbf{v}, \mathsf{v}) \;\middle|\; \begin{array}{r} (\mathbf{v}, \mathsf{v}) \in \mathtt{oftype}(\hat{\tau_1} \times \hat{\tau_2}) \text{ and } \exists \mathbf{v_1}, \mathbf{v_2}, \mathsf{v}_1, \mathsf{v}_2.\, \mathbf{v} = \langle \mathbf{v_1}, \mathbf{v_2} \rangle \text{ and } \mathsf{v} = \langle \mathsf{v}_1, \mathsf{v}_2 \rangle \text{ and} \\ (\underline{W}, \mathbf{v_1}, \mathsf{v}_1) \in \triangleright \mathcal{V}[\![\hat{\tau_1}]\!]_\square \text{ and } (\underline{W}, \mathbf{v_2}, \mathsf{v}_2) \in \triangleright \mathcal{V}[\![\hat{\tau_2}]\!]_\square \end{array} \right\}$$

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

$$\mathcal{V}[\![\mathtt{EmulDV}_{\mathsf{n;p}}]\!]_\square \stackrel{\mathsf{def}}{=} \cdots \text{ (to be defined later, in Fig. 11)}$$

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

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

Logical relations for substitutions, open terms and program contexts.

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

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

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

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

$$\vdash \mathfrak{C} \;\square\; \mathfrak{C} : \hat{\mathbf{\Gamma}}', \hat{\tau}' \to \hat{\mathbf{\Gamma}}, \hat{\tau} \stackrel{\mathsf{def}}{=} \begin{array}{r} \vdash \mathfrak{C} : \mathtt{repEmul}(\hat{\mathbf{\Gamma}}'), \mathtt{repEmul}(\hat{\tau}') \to \mathtt{repEmul}(\hat{\mathbf{\Gamma}}), \mathtt{repEmul}(\hat{\tau}) \text{ and} \\ \text{for all } \mathbf{t}, \mathsf{t}.\ \text{ if } \hat{\mathbf{\Gamma}}' \vdash \mathbf{t} \;\square\; \mathsf{t} : \hat{\tau}', \text{ then } \hat{\mathbf{\Gamma}} \vdash \mathfrak{C}[\mathbf{t}] \;\square\; \mathfrak{C}[\mathsf{t}] : \hat{\tau} \end{array}$$

Figure 7. Logical relations (partial, the missing definition can be found in Figs. 11 and 12).

$\hat{\mathbf{\Gamma}} \vdash \mathbf{t} \;\square\; \mathsf{t} : \hat{\tau}$. Finally, we define a logical relation for program contexts $\vdash \mathfrak{C} \;\square\; \mathfrak{C} : \hat{\mathbf{\Gamma}}', \hat{\tau}' \to \hat{\mathbf{\Gamma}}, \hat{\tau}$ which requires that substituting terms related at the appropriate pseudo-type 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.

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 1).

Lemma 1 (Adequacy for $\lesssim$ and $\gtrsim$)**.** *If $\emptyset \vdash \mathbf{t} \lesssim_{\mathsf{n}} \mathsf{t} : \tau$, and if $\mathbf{t} \hookrightarrow^{\mathbf{m}} \mathbf{v}$ with $n \ge m$, then also $\mathsf{t} \Downarrow$. If $\emptyset \vdash \mathbf{t} \gtrsim_{\mathsf{n}} \mathsf{t} : \tau$ and if $\mathsf{t} \hookrightarrow^{\mathsf{m}} \mathsf{v}$ with $n \ge m$, then also $\mathbf{t} \Downarrow$.*

4. The Compiler

This section presents our compiler from λ^{τ} to λ^{u}. The compiler proceeds in two passes: type erasure (Fig. 8) and dynamic type-checking wrappers (Fig. 9).

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

$$\texttt{erase}(\texttt{unit}) \overset{\text{def}}{=} \texttt{unit} \qquad \texttt{erase}(\langle \mathbf{t_1}, \mathbf{t_2} \rangle) \overset{\text{def}}{=} \langle \texttt{erase}(\mathbf{t_1}), \texttt{erase}(\mathbf{t_2}) \rangle$$

$$\texttt{erase}(\texttt{false}) \overset{\text{def}}{=} \texttt{false} \qquad \texttt{erase}(\mathbf{t_1}; \mathbf{t_2}) \overset{\text{def}}{=} \texttt{erase}(\mathbf{t_1}); \texttt{erase}(\mathbf{t_2})$$

$$\texttt{erase}(\mathbf{x}) \overset{\text{def}}{=} \mathsf{x} \qquad \texttt{erase}(\mathbf{t_1}\ \mathbf{t_2}) \overset{\text{def}}{=} \texttt{erase}(\mathbf{t_1})\ \texttt{erase}(\mathbf{t_2})$$

$$\texttt{erase}(\mathbf{t}.1) \overset{\text{def}}{=} \texttt{erase}(\mathbf{t}).1$$

$$\texttt{erase}(\texttt{true}) \overset{\text{def}}{=} \texttt{true}$$

$$\texttt{erase}(\mathbf{t}.2) \overset{\text{def}}{=} \texttt{erase}(\mathbf{t}).2$$

$$\texttt{erase}(\lambda \mathbf{x} : \tau.\, \mathbf{t}) \overset{\text{def}}{=} \lambda \mathsf{x}.\, \texttt{erase}(\mathbf{t})$$

$$\texttt{erase}(\mathrm{inl}\ \mathbf{t}) \overset{\text{def}}{=} \mathrm{inl}\ \texttt{erase}(\mathbf{t})$$

$$\texttt{erase}(\mathrm{fix}_{\tau_1 \to \tau_2}\ \mathbf{t}) \overset{\text{def}}{=} \mathit{fix}\ \texttt{erase}(\mathbf{t})$$

$$\texttt{erase}(\mathrm{inr}\ \mathbf{t}) \overset{\text{def}}{=} \mathrm{inr}\ \texttt{erase}(\mathbf{t})$$

$$\texttt{erase}(\mathrm{if}\ \mathbf{t}\ \mathrm{then}\ \mathbf{t_1}\ \mathrm{else}\ \mathbf{t_2}) \overset{\text{def}}{=} \begin{array}{l} \mathrm{if}\ \texttt{erase}(\mathbf{t})\ \mathrm{then}\ \texttt{erase}(\mathbf{t_1}) \\ \qquad\qquad \mathrm{else}\ \texttt{erase}(\mathbf{t_2}) \end{array}$$

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

Figure 8. Type erasure: the first pass of the compiler.

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

The $\texttt{erase}$ 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 [11, 24])**.** Consider the following, contextually equivalent λ^τ functions of type $\texttt{Unit} \to \texttt{Unit}$:

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

The $\texttt{erase}$ function will map these to the following λ^{u} functions:

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

The results of $\texttt{erase}$ are *not* contextually equivalent, essentially because applying them to a non-unit value like $\texttt{true}$ will produce $\texttt{true}$ for the left lambda and $\texttt{unit}$ for the right lambda. In this example, contextual equivalence is not preserved because the original functions are only defined for $\texttt{Unit}$ values, but their compilations can be applied to other values too.

Lemma 2 states that every λ^τ term is related to its erased term at its type.

Lemma 2 (Erase is semantics-preserving (for terms))**.** *If* $\mathbf{\Gamma} \vdash \mathbf{t} : \tau$, *then* $\mathbf{\Gamma} \vdash \mathbf{t} \,\Box\, \texttt{erase}(\mathbf{t}) : \tau$.

An analogous result applies to program contexts:

Lemma 3 (Erase is semantics-preserving (for context))**.** *For all* $\mathfrak{C}$, *if* $\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau$ *then* $\vdash \mathfrak{C} \,\Box\, \texttt{erase}(\mathfrak{C}) : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau$.

One should intuitively understand this result as "$\mathbf{t}$ behaves the same as $\texttt{erase}(\mathbf{t})$ 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 *correctness* of the $\texttt{erase}$ function, not its *security*.

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 $\mathsf{protect}_\tau$ and $\mathsf{confine}_\tau$. $\mathsf{protect}_\tau$ is a λ^{u} lambda 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 [4, 8, 11, 24]. Dually, $\mathsf{confine}_\tau$ 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., $\texttt{Unit}$ or $\texttt{Bool}$) 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 $\texttt{Unit}$ and $\texttt{Bool}$ 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.

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

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

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

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

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

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

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

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

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

Figure 9. Dynamic type checking wrappers: the second pass of the compiler.

Example 4.2 (Protect and confine make a term secure)**.** Consider the protect wrapper for type $\texttt{Unit} \to \texttt{Unit}$ $\mathsf{protect}_{\texttt{Unit} \to \texttt{Unit}}$, which is (roughly) equal to $\lambda \mathsf{y}.\, \lambda \mathsf{x}.\, \mathsf{y}\ (\mathsf{x}; \texttt{unit})$. Applying that wrapper to a function f (i.e. $\mathsf{protect}_{\texttt{Unit} \to \texttt{Unit}}\ \mathsf{f}$) reduces to $\lambda \mathsf{x}.\, \mathsf{f}\ (\mathsf{x}; \texttt{unit})$. Applying this value to a non-$\texttt{unit}$ value will simply evaluate to wrong, therefore addressing the issues of Example 4.1.

For the second pass of the compiler, Lemma 4 holds.

Lemma 4 (Protect and confine are semantics-preserving)**.** *If* $\mathbf{\Gamma} \vdash \mathbf{t} \,\Box_{\mathsf{n}}\, \mathsf{t} : \tau$ *then* $\mathbf{\Gamma} \vdash \mathbf{t} \,\Box_{\mathsf{n}}\, \mathsf{protect}_\tau\ \mathsf{t} : \tau$ *and* $\mathbf{\Gamma} \vdash \mathbf{t} \,\Box_{\mathsf{n}}\, \mathsf{confine}_\tau\ \mathsf{t} : \tau$.

Lemma 4 states that if $\mathbf{t}$ 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 1 (The $[\![\cdot]\!]$ compiler)**.** *If* $\mathbf{\Gamma} \vdash \mathbf{t} : \tau$, *then* $\mathbf{t}$ *is compiled to* $[\![\mathbf{t}]\!]$ *and:* $[\![\mathbf{t}]\!] \overset{\text{def}}{=} \mathsf{protect}_\tau\ (\texttt{erase}(\mathbf{t}))$.

Lemmas 2 and 4 about the first and second pass of the compiler can be combined into Lemma 5 to obtain that a λ^τ term of type τ behaves like its compilation when both are treated as terms of type τ.

Lemma 5 ($[\![\cdot]\!]$ is semantics-preserving). *For all* $\mathbf{t}$*, if* $\mathbf{\Gamma} \vdash \mathbf{t} : \tau$ *then* $\mathbf{\Gamma} \vdash \mathbf{t} \;\square\; [\![\mathbf{t}]\!] : \tau$.

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 $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\mathsf{n}}$ which conservatively n-approximates $\mathfrak{C}$. Intuitively, this means that $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\mathsf{n}}$ 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 $\mathrm{UVal}_{\mathsf{n}}$. 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 emulation $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\mathsf{n}}$ manipulates values of type $\mathrm{UVal}_{\mathsf{n}}$. Section 5.1 defines $\mathrm{UVal}_{\mathsf{n}}$ and the basic tools (constructors and destructors) for working with it. To explain how values in $\mathrm{UVal}_{\mathsf{n}}$ model values in λ^{u}, Section 5.2 fills in the missing piece of the logical relations of Fig. 7 by defining $\mathcal{V}[\![\mathtt{EmulDV}_{\mathsf{n;p}}]\!]_{\square}$.

The type $\mathrm{UVal}_{\mathsf{n}}$ 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*. This is crucial, as for a term to be well-typed the accuracy of the approximation can be less than n. In these cases values of type $\mathrm{UVal}_{\mathsf{n}}$ will be *downgraded* to a type $\mathrm{UVal}_{\mathsf{m}}$ 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 the definition of upgrading and downgrading, we have defined all the machinery that revolves around $\mathrm{UVal}_{\mathsf{n}}$. Section 5.4 constructs the function responsible for the emulating a context, $\mathrm{emulate}_{\mathsf{n}}$, such that it translates a λ^{u} term t into a λ^{τ} term of type $\mathrm{UVal}_{\mathsf{n}}$. This function is easily extended to work with program contexts, producing contexts with hole of type $\mathrm{UVal}_{\mathsf{n}}$ as expected.

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

Finally, Section 5.6 defines the approximate back-translation function $\langle\!\langle\cdot\rangle\!\rangle_{\tau;\mathsf{n}}$, mapping a λ^{u} context $\mathfrak{C}$ to a λ^{τ} context $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}$. 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 $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}$ is of type τ. Plugging a term $\mathbf{t_1}$ in $\langle\!\langle\mathfrak{C}\rangle\!\rangle_{\tau;\mathsf{n}}$ n-approximates plugging in the compilation $[\![\mathbf{t_1}]\!]$ in context $\mathfrak{C}$.

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

5.1 UVal and its Tools

The family of types $\mathrm{UVal}_{\mathsf{n}}$ is defined as follows:

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

$$\mathrm{UVal}_{\mathsf{n+1}} \stackrel{\mathsf{def}}{=} \mathtt{Unit} \uplus \mathtt{Unit} \uplus \mathtt{Bool} \uplus (\mathrm{UVal}_{\mathsf{n}} \times \mathrm{UVal}_{\mathsf{n}}) \uplus (\mathrm{UVal}_{\mathsf{n}} \uplus \mathrm{UVal}_{\mathsf{n}}) \uplus (\mathrm{UVal}_{\mathsf{n}} \to \mathrm{UVal}_{\mathsf{n}})$$

$\mathrm{UVal}_{\mathsf{n}}$ is the type that emulated λ^{u} terms have when back-translated into λ^{τ}. For every n, $\mathrm{UVal}_{\mathsf{n}}$ is clearly a valid λ^{τ} type. At non-zero levels, the type $\mathrm{UVal}_{\mathsf{n+1}}$ is a disjunct sum of base values (the second occurrence of $\mathtt{Unit}$ and $\mathtt{Bool}$), products and coproducts of $\mathrm{UVal}_{\mathsf{n}}$s and functions mapping a $\mathrm{UVal}_{\mathsf{n}}$ to a $\mathrm{UVal}_{\mathsf{n}}$. All of these cases are used to emulate a corresponding λ^{u} value. Additionally, at every level including $n = 0$, the type $\mathrm{UVal}_{\mathsf{n}}$ 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 $\mathrm{UVal}_{\mathsf{n+1}}$ are intentional. The first is used for imprecisely representing arbitrary λ^{u} terms while the second accurately represents λ^{u} $\mathtt{unit}$ values.

To work with $\mathrm{UVal}_{\mathsf{n}}$ values, we need basic tools for dealing with sum types: tag injections and case extractions (Fig. 10). Functions $\mathtt{in}_{\mathrm{unk},n}$, $\mathtt{in}_{\mathtt{Unit},n}$, $\mathtt{in}_{\mathtt{Bool},n}$, $\mathtt{in}_{\times,n}$, $\mathtt{in}_{\uplus,n}$, $\mathtt{in}_{\to,n}$ are convenient names for nested applications of coproduct injection functions for the nested coproduct in the definition of $\mathrm{UVal}_{\mathsf{n+1}}$. The term $\mathrm{unk}_{\mathsf{n}}$ produces either the single value of UVal_0 or uses $\mathtt{in}_{\mathrm{unk},n}$ to produce a $\mathrm{UVal}_{\mathsf{n}}$ value representing a 0-precision approximate back-translation of an arbitrary untyped term. For using $\mathrm{UVal}_{\mathsf{n}}$ values, we define functions $\mathtt{case}_{\mathtt{Unit},n}$, $\mathtt{case}_{\mathtt{Bool},n}$, $\mathtt{case}_{\times,n}$, $\mathtt{case}_{\uplus,n}$, $\mathtt{case}_{\to,n}$ using a somewhat liberal pattern matching syntax that can be easily desugared to nested case expressions. The functions are lambdas that inspect their $\mathrm{UVal}_{\mathsf{n+1}}$ 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.

$$\mathtt{in}_{\mathrm{unk},n} : \mathrm{UVal}_{\mathsf{n+1}}$$
$$\mathtt{in}_{\mathtt{Unit},n} : \mathtt{Unit} \to \mathrm{UVal}_{\mathsf{n+1}}$$
$$\mathtt{in}_{\mathtt{Bool},n} : \mathtt{Bool} \to \mathrm{UVal}_{\mathsf{n+1}}$$
$$\mathtt{in}_{\times,n} : (\mathrm{UVal}_{\mathsf{n}} \times \mathrm{UVal}_{\mathsf{n}}) \to \mathrm{UVal}_{\mathsf{n+1}}$$
$$\mathtt{in}_{\uplus,n} : (\mathrm{UVal}_{\mathsf{n}} \uplus \mathrm{UVal}_{\mathsf{n}}) \to \mathrm{UVal}_{\mathsf{n+1}}$$
$$\mathtt{in}_{\to,n} : (\mathrm{UVal}_{\mathsf{n}} \to \mathrm{UVal}_{\mathsf{n}}) \to \mathrm{UVal}_{\mathsf{n+1}}$$

$$\mathrm{unk}_{\mathsf{n}} : \mathrm{UVal}_{\mathsf{n}}$$
$$\mathrm{unk}_0 \stackrel{\mathsf{def}}{=} \mathtt{Unit}$$
$$\mathrm{unk}_{\mathsf{n+1}} \stackrel{\mathsf{def}}{=} \mathtt{in}_{\mathrm{unk},n}$$

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

$$\mathtt{case}_{\mathtt{Unit},n} : \mathrm{UVal}_{\mathsf{n+1}} \to \mathtt{Unit} \qquad \mathtt{case}_{\mathtt{Bool},n} : \mathrm{UVal}_{\mathsf{n+1}} \to \mathtt{Bool}$$
$$\mathtt{case}_{\times,n} : \mathrm{UVal}_{\mathsf{n+1}} \to (\mathrm{UVal}_{\mathsf{n}} \times \mathrm{UVal}_{\mathsf{n}})$$
$$\mathtt{case}_{\uplus,n} : \mathrm{UVal}_{\mathsf{n+1}} \to (\mathrm{UVal}_{\mathsf{n}} \uplus \mathrm{UVal}_{\mathsf{n}})$$
$$\mathtt{case}_{\to,n} : \mathrm{UVal}_{\mathsf{n+1}} \to \mathrm{UVal}_{\mathsf{n}} \to \mathrm{UVal}_{\mathsf{n}}$$

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

Figure 10. Basic tools for working with $\mathrm{UVal}_{\mathsf{n}}$. The subscript of $\mathrm{omega}$is omitted when it is clear from the context.

5.2 λ^{u} values vs UVal

To make the correspondence between a λ^{u} term and its emulation in $\mathrm{UVal}_{\mathsf{n}}$ more exact, this section fills in the definition of $\mathcal{V}[\![\mathtt{EmulDV}_{\mathsf{n;p}}]\!]_{\square}$, the missing piece of the logical relations of Fig. 7. Recall that the intuition is that $\mathtt{EmulDV}_{\mathsf{n;p}}$ is a token type that is used to relate λ^{u} terms to their λ^{τ} emulation of type $\mathrm{UVal}_{\mathsf{n}}$. This relation is done up to an approximateness degree, denoted with $p ::= \mathtt{precise} \mid \mathtt{imprecise}$, that is explained below. Intuitively, the previously presented cases of the logical relations define the relation between a λ^τ term and its compilation. The $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$ case defines the relation between a λ^u term and its UVal_n-typed back-translation, as motivated in Example 5.1.

Example 5.1 (The need of $\texttt{EmulDV}$)**.** Consider the term $\mathbf{t} \equiv \texttt{in}_{\texttt{Bool};1}(\texttt{true})$. 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 $\mathtt{t}$ term that $\mathbf{t}$ approximates (in this case, $\texttt{true}$).

Type $\texttt{EmulDV}_{n;p}$ serves the purpose of bridging the syntactic difference, allowing $\texttt{in}_{\texttt{Bool};1}(\texttt{true})$ and $\texttt{true}$ to be related.

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

Example 5.2. Consider the UVal_6 value $\texttt{in}_{\times;5}\ \langle \texttt{in}_{\uplus;4}\ (\mathrm{inl}\ \mathrm{unk}_4), \mathrm{unk}_5\rangle$. This value might be used by the approximate back-translation to represent the λ^u term $\langle \mathrm{inl}\ \langle \mathrm{unit}, \mathrm{true}\rangle, \lambda x.\, x\rangle$. Our $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$ specification will enforce that terms of the form $\texttt{in}_{\times;n}\ \langle\cdot,\cdot\rangle$ or $\texttt{in}_{\uplus;n}\ (\mathrm{inl}\ \cdot)$ 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, $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$ does not allow these unk_n values to occur anywhere, because they could compromise the required precision of our approximate back-translation.

In fact, $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$ 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, $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$ 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.

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

The increased index 43 is needed because $\mathbf{t}$ is tagged at 42, so we need an additional step to unfold the tagging.

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

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

By the definition in Fig. 7, this requires in turn that $(\underline{W}, \mathrm{unk}_{42}, \mathtt{t}_1)$ and $(\underline{W}, \mathrm{unk}_{42}, \mathtt{t}_2)$ are in $\triangleright \mathcal{V}[\![\texttt{EmulDV}_{42,\texttt{precise}}]\!]_\square$. However if $\mathsf{lev}(\underline{W}) = 0$, then this is 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. 11 presents the definition of $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$. For relating terms $\mathbf{v}$ and $\mathtt{v}$ 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 of its UVal_n tag and the structure of the λ^u term must coincide. Formally, this is expressed by the following conditions: $(\underline{W}, \mathbf{v}', \mathtt{v})$ are in $\mathcal{V}[\![\mathcal{B}]\!]_\square$, $\mathcal{V}[\![\texttt{EmulDV}_{n;p} \times \texttt{EmulDV}_{n;p}]\!]_\square$, $\mathcal{V}[\![\texttt{EmulDV}_{n;p} \uplus \texttt{EmulDV}_{n;p}]\!]_\square$ or $\mathcal{V}[\![\texttt{EmulDV}_{n;p} \to \texttt{EmulDV}_{n;p}]\!]_\square$ if $\mathbf{v} = \texttt{in}_{\mathcal{B};n}(\mathbf{v}')$, $\mathbf{v} = \texttt{in}_{\times;n}(\mathbf{v}')$, $\mathbf{v} = \texttt{in}_{\uplus;n}(\mathbf{v}')$ or $\mathbf{v} = \texttt{in}_{\to;n}(\mathbf{v}')$ respectively.

In addition to $\texttt{EmulDV}_{n;p}$, we still need to define two helper functions (Fig. 12) that were left open in Fig. 7. The first one, $\texttt{toEmul}(\cdot)_{n;p}$, turns an untyped Γ into one where all variables are mapped to $\texttt{EmulDV}_{n;p}$. The second one, $\texttt{repEmul}(\cdot)$, re-maps all variables of a $\mathbf{\Gamma}$ that are of type $\texttt{EmulDV}_{n;p}$ to type UVal_n.

The adequacy property of the logical relations (Lemma 1) hold for the complete definition of the logical relations, including the definition for $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]$.

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

$$\mathcal{V}[\![\texttt{EmulDV}_{n+1;p}]\!]_\square \stackrel{\mathsf{def}}{=}$$

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

Figure 11. Specifying the relation between λ^u values and their emulation in $\mathcal{V}[\![\texttt{EmulDV}_{n;p}]\!]_\square$.

$$\texttt{toEmul}(\emptyset)_{n;p} = \emptyset \qquad \texttt{toEmul}(\Gamma, x)_{n;p} = \texttt{toEmul}(\Gamma)_{n;p}, (\mathbf{x} : \texttt{EmulDV}_{n;p})$$

$$\texttt{repEmul}(\emptyset) = \emptyset \qquad \texttt{repEmul}(\mathbf{\Gamma}, (\mathbf{x} : \hat{\tau})) = \texttt{repEmul}(\mathbf{\Gamma}), (\mathbf{x} : \texttt{repEmul}(\hat{\tau}))$$

$$\texttt{repEmul}(\tau) = \tau \qquad \texttt{repEmul}(\texttt{EmulDV}_{n;p}) = \mathrm{UVal}_n$$

Figure 12. Helper functions for $\texttt{EmulDV}_{n;p}$.

$$
\begin{array}{l}
\mathrm{downgrade}_{n;d} : \mathrm{UVal}_{n+d} \to \mathrm{UVal}_n \\
\mathrm{downgrade}_{0;d} \stackrel{\mathsf{def}}{=} \lambda \mathbf{v} : \mathrm{UVal}_d\, \mathtt{unit} \\
\mathrm{downgrade}_{n+1;d} \stackrel{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{n+d+1}.\ \mathrm{case}\ \mathbf{x}\ \mathrm{of} \\
\left|\ \begin{array}{l}
\mathtt{in}_{\mathrm{unk},n+d} \mapsto \mathtt{in}_{\mathrm{unk},n} \\
\mathtt{in}_{\mathtt{Unit},n+d}\ \mathbf{y} \mapsto \mathtt{in}_{\mathtt{Unit},n}\ \mathbf{y} \\
\mathtt{in}_{\mathtt{Bool},n+d}\ \mathbf{y} \mapsto \mathtt{in}_{\mathtt{Bool},n}\ \mathbf{y} \\
\mathtt{in}_{\times,n+d}\ \mathbf{y} \mapsto \mathtt{in}_{\times,n}\ \langle \mathrm{downgrade}_{n;d}\ \mathbf{y}.\mathbf{1}, \mathrm{downgrade}_{n;d}\ \mathbf{y}.\mathbf{2} \rangle \\
\mathtt{in}_{\uplus,n+d}\ \mathbf{y} \mapsto \mathtt{in}_{\uplus,n}\ \mathrm{case}\ \mathbf{y}\ \mathrm{of} \left|\ \begin{array}{l} \mathrm{inl}\ \mathbf{x} \mapsto \mathrm{inl}\ (\mathrm{downgrade}_{n;d}\ \mathbf{x}) \\ \mathrm{inr}\ \mathbf{x} \mapsto \mathrm{inr}\ (\mathrm{downgrade}_{n;d}\ \mathbf{x}) \end{array}\right. \\
\mathtt{in}_{\to,n+d}\ \mathbf{y} \mapsto \begin{array}{l} \mathtt{in}_{\to,n}\ (\lambda \mathbf{z} : \mathrm{UVal}_n.\ \mathrm{downgrade}_{n;d} \\ \qquad (\mathbf{y}\ (\mathrm{upgrade}_{n;d}\ \mathbf{z}))) \end{array}
\end{array}\right.
\end{array}
$$

$$
\begin{array}{l}
\mathrm{upgrade}_{n;d} : \mathrm{UVal}_n \to \mathrm{UVal}_{n+d} \\
\mathrm{upgrade}_{0;d} \stackrel{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_0.\ \mathrm{unk}_d \\
\mathrm{upgrade}_{n+1;d} \stackrel{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{n+1}.\ \mathrm{case}\ \mathbf{x}\ \mathrm{of} \\
\left|\ \begin{array}{l}
\mathtt{in}_{\mathrm{unk},n} \mapsto \mathtt{in}_{\mathrm{unk},n+d}; \\
\mathtt{in}_{\mathtt{Unit},n}\ \mathbf{y} \mapsto \mathtt{in}_{\mathtt{Unit},n+d}\ \mathbf{y}; \\
\mathtt{in}_{\mathtt{Bool},n}\ \mathbf{y} \mapsto \mathtt{in}_{\mathtt{Bool},n+d}\ \mathbf{y}; \\
\mathtt{in}_{\times,n}\ \mathbf{y} \mapsto \mathtt{in}_{\times,n+d}\ \langle \mathrm{upgrade}_{n;d}\ \mathbf{y}.\mathbf{1}, \mathrm{upgrade}_{n;d}\ \mathbf{y}.\mathbf{2} \rangle \\
\mathtt{in}_{\uplus,n}\ \mathbf{y} \mapsto \mathtt{in}_{\uplus,n+d}\ \mathrm{case}\ y\ \mathrm{of} \left|\ \begin{array}{l} \mathrm{inl}\ \mathbf{x} \mapsto \mathrm{inl}\ (\mathrm{upgrade}_{n;d}\ \mathbf{x}) \\ \mathrm{inr}\ \mathbf{x} \mapsto \mathrm{inr}\ (\mathrm{upgrade}_{n;d}\ \mathbf{x}) \end{array}\right. \\
\mathtt{in}_{\to,n}\ \mathbf{y} \mapsto \begin{array}{l} \mathtt{in}_{\to,n+d}\ (\lambda \mathbf{z} : \mathrm{UVal}_n.\ \mathrm{upgrade}_{n;d} \\ \qquad (\mathbf{y}\ (\mathrm{downgrade}_{n;d}\ \mathbf{z}))) \end{array}
\end{array}\right.
\end{array}
$$

Figure 13. Upgrade and downgrade for UVal_n.

5.3 Upgrading and Downgrading Values

Figure 13 defines the functions $\mathrm{downgrade}_{n+d} : \mathrm{UVal}_{n+d} \to \mathrm{UVal}_n$ and $\mathrm{upgrade}_{n+d} : \mathrm{UVal}_n \to \mathrm{UVal}_{n+d}$ (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 $\mathrm{downgrade}_{n+d}$ and $\mathrm{upgrade}_{n+d}$: 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 $\mathrm{downgrade}_{n+d}$ when $n = 0$ will throw away the information contained in its argument of type UVal_d and simply returns the single unknown value in UVal_0. Note that $\mathrm{downgrade}_{n+d}$ and $\mathrm{upgrade}_{n+d}$ are not inverse functions, since $\mathrm{downgrade}_{n+d}$ throws away information that was previously there. While $\mathbf{t} \equiv \mathrm{downgrade}_{n-d}\mathrm{upgrade}_{n+d}\mathbf{t}$, the reverse ($\mathbf{t} \equiv \mathrm{upgrade}_{n+d}\mathrm{downgrade}_{n-d}\mathbf{t}$) 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 UVal_n for a sufficiently-large n. We would expect roughly the following λ^{τ} term:

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

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 $\mathtt{in}_{\times;n-2}$). 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 $\mathrm{downgrade}$ 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.

$$
\begin{array}{rcl}
\mathrm{emulate}_n(\mathsf{t}) & : & \mathrm{UVal}_n \\
\mathrm{emulate}_n(\mathtt{unit}) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\mathtt{Unit},n}\ \mathtt{unit}) \\
\mathrm{emulate}_n(\mathtt{true}) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\mathtt{Bool},n}\ \mathtt{true}) \\
\mathrm{emulate}_n(\mathtt{false}) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\mathtt{Bool},n}\ \mathtt{false}) \\
\mathrm{emulate}_n(\mathsf{x}) & \stackrel{\mathsf{def}}{=} & \mathbf{x} \\
\mathrm{emulate}_n(\lambda \mathsf{x}.\ \mathsf{t}) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\to,n}\ (\lambda \mathbf{x} : \mathrm{UVal}_n.\ \mathrm{emulate}_n(\mathsf{t}))) \\
\mathrm{emulate}_n(\mathsf{t}_1\ \mathsf{t}_2) & \stackrel{\mathsf{def}}{=} & \mathtt{case}_{\to,n}\ (\mathrm{upgrade}_{n+1}\ (\mathrm{emulate}_n(\mathsf{t}_1)))\ \mathrm{emulate}_n(\mathsf{t}_2) \\
\mathrm{emulate}_n(\langle \mathsf{t}_1, \mathsf{t}_2 \rangle) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\times,n}\ \langle \mathrm{emulate}_n(\mathsf{t}_1), \mathrm{emulate}_n(\mathsf{t}_2) \rangle) \\
\mathrm{emulate}_n(\mathrm{inl}\ \mathsf{t}) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\uplus,n}\ (\mathrm{inl}\ \mathrm{emulate}_n(\mathsf{t}))) \\
\mathrm{emulate}_n(\mathrm{inr}\ \mathsf{t}) & \stackrel{\mathsf{def}}{=} & \mathrm{downgrade}_{n+1}\ (\mathtt{in}_{\uplus,n}\ (\mathrm{inr}\ \mathrm{emulate}_n(\mathsf{t}))) \\
\mathrm{emulate}_n(\mathsf{t}.1) & \stackrel{\mathsf{def}}{=} & (\mathtt{case}_{\times,n}\ (\mathrm{upgrade}_{n+1}\ (\mathrm{emulate}_n(\mathsf{t})))).\mathbf{1} \\
\mathrm{emulate}_n(\mathsf{t}.2) & \stackrel{\mathsf{def}}{=} & (\mathtt{case}_{\times,n}\ (\mathrm{upgrade}_{n+1}\ (\mathrm{emulate}_n(\mathsf{t})))).\mathbf{2} \\
\mathrm{emulate}_n(\mathsf{t}; \mathsf{t}') & \stackrel{\mathsf{def}}{=} & (\mathtt{case}_{\mathtt{Unit},n}\ (\mathrm{upgrade}_{n+1}(\mathrm{emulate}_n(\mathsf{t})))); \mathrm{emulate}_n(\mathsf{t}') \\
\mathrm{emulate}_n(\mathsf{wrong}) & \stackrel{\mathsf{def}}{=} & \mathrm{omega}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{emulate}_n(\mathrm{case}\ \mathsf{t}_1\ \mathrm{of}\ \mathrm{inl}\ \mathsf{x} \mapsto \mathsf{t}_2 \mid \mathrm{inr}\ \mathsf{x} \mapsto \mathsf{t}_3) \stackrel{\mathsf{def}}{=} \\
\qquad \mathrm{case}\ \mathtt{case}_{\uplus,n}\ (\mathrm{upgrade}_{n+1}\ (\mathrm{emulate}_n(\mathsf{t}_1)))\ \mathrm{of} \\
\qquad\qquad \mathrm{inl}\ \mathbf{x} \mapsto \mathrm{emulate}_n(\mathsf{t}_2) \mid \mathrm{inr}\ \mathbf{x} \mapsto \mathrm{emulate}_n(\mathsf{t}_3) \\
\mathrm{emulate}_n(\mathrm{if}\ \mathsf{t}\ \mathrm{then}\ \mathsf{t}_1\ \mathrm{else}\ \mathsf{t}_2) \stackrel{\mathsf{def}}{=} \\
\quad \mathrm{if}\ (\mathtt{case}_{\mathtt{Bool},n}(\mathrm{upgrade}_{n+1}(\mathrm{emulate}_n\mathbf{t})))\ \mathrm{then}\ \mathrm{emulate}_n(\mathsf{t}_1)\ \mathrm{else} \\
\qquad\qquad\qquad \mathrm{emulate}_n(\mathsf{t}_2)
\end{array}
$$

Figure 14. Emulating λ^{u} terms in UVal_n.

Lemma 6 (Compatibility lemma for $\mathrm{upgrade}_{n+d}$ and $\mathrm{downgrade}_{n+d}$)**.** *Suppose that either ($n < m$ and $p = \mathtt{precise}$) or ($\square = \lesssim$ and $p = \mathtt{imprecise}$). Then*

- *If* $\mathbf{\Gamma} \vdash \mathbf{t}\ \square_n\ \mathsf{t} : \mathtt{EmulDV}_{m+d;p}$, *then* $\mathbf{\Gamma} \vdash \mathrm{downgrade}_{m+d}\ \mathbf{t}\ \square_n\ \mathsf{t} : \mathtt{EmulDV}_{m;p}$.
- *If* $\mathbf{\Gamma} \vdash \mathbf{t}\ \square_n\ \mathsf{t} : \mathtt{EmulDV}_{m;p}$, *then* $\mathbf{\Gamma} \vdash \mathrm{upgrade}_{m+d}\ \mathbf{t}\ \square_n\ \mathsf{t} : \mathtt{EmulDV}_{m+d;p}$.

This lemma covers both situations that we discussed previously. It requires that either $n < m$ (so that the results only hold in worlds $\underline{W}$ with $\mathrm{lev}(\underline{W}) \leq n < m$), 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 $\mathrm{downgrade}$ and $\mathrm{upgrade}$, Fig. 14 defines the $\mathrm{emulate}_n$ function. That function maps arbitrary λ^{u} terms to their approximate back-translation: λ^{τ} terms of type UVal_n. $\mathrm{emulate}_n$ is defined by induction on t. The different cases follow the same

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

$$\mathtt{extract}_{\tau;0} \stackrel{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_0 .\, \mathrm{omega}$$

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

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

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

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

$$\mathtt{extract}_{\tau_1 \uplus \tau_2;n+1} \stackrel{\mathsf{def}}{=} \lambda \mathbf{x} : \mathrm{UVal}_{n+1} .\, \text{case } \mathtt{case}_{\uplus;n}\ \mathbf{x} \text{ of} \left| \begin{array}{l} \text{inl } \mathbf{y} \to \text{inl } (\mathtt{extract}_{\tau_1;n}\ \mathbf{y}) \\ \text{inr } \mathbf{y} \to \text{inr } (\mathtt{extract}_{\tau_2;n}\ \mathbf{y}) \end{array} \right.$$

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

$$\mathtt{inject}_{\tau;0} \stackrel{\mathsf{def}}{=} \lambda \mathbf{x} : \tau .\, \mathrm{omega}$$

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

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

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

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

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

Figure 15. Injecting λ^{τ} values into UVal_n.

pattern: every term t is mapped to a λ^{τ} term constructed recursively from the emulation of sub-terms, producing and consuming 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 discussed in the previous section. 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 UVal_{n+1}, $\mathrm{downgrade}_{n;1}$ is used to downgrade this to a 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 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 7 (Emulate relates at $\mathtt{EmulDV}$)**.** *If* $\Gamma \vdash \mathsf{t}$*, and if* ($m > n$ *and* $p = \mathtt{precise}$) *or* ($\square =\lesssim$ *and* $p = \mathtt{imprecise}$)*, then we have that* $\mathtt{toEmul}(\Gamma)_{m;p} \vdash \mathrm{emulate}_m(\mathsf{t})\ \square_n\ \mathsf{t} : \mathtt{EmulDV}_{m;p}$.

Like Lemma 6, Lemma 7 requires that either $n < m$ (so that the results only hold in worlds $\underline{W}$ with $\mathsf{lev}(\underline{W}) \leq n < m$), 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 $\langle\langle \mathfrak{C} \rangle\rangle_n$ in Fig. 3. The lemma states that the back-translation of any well-scoped term is related to the term by $\mathtt{EmulDV}_{m;p}$, as intended.

An analogous result holds for contexts.

Lemma 8 (Emulated relates contexts at $\mathtt{EmulDV}$)**.** *If* $\vdash \mathfrak{C} : \Gamma' \to \Gamma$*, if* ($m > n$ *and* $p = \mathtt{precise}$) *or* ($\square =\lesssim$ *and* $p = \mathtt{imprecise}$)*, then* $\vdash\ \mathrm{emulate}_m(\mathfrak{C})\ \square_n\ \mathfrak{C} : \mathtt{toEmul}(\Gamma')_{m;p}, \mathtt{EmulDV}_{m;p} \to \mathtt{toEmul}(\Gamma)_{m;p}, \mathtt{EmulDV}_{m;p}$

5.5 Injection and Extraction of Terms

One final thing is missing to construct a back-translation $\langle\langle \mathfrak{C} \rangle\rangle_n$ of an untyped program context $\mathfrak{C}$. While $\mathrm{emulate}_n(\mathfrak{C})$ produces a λ^{τ} context that expects a 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 $\mathbf{t_1}$ and $\mathbf{t_2}$ that we are compiling). To bridge this difference, Fig. 15 defines a λ^{τ} function $\mathtt{inject}_{\tau;n}$ of type $\tau \to \mathrm{UVal}_n$ which injects values of an arbitrary type τ into UVal_n. We define it mutually recursively with a dual function $\mathtt{extract}_{\tau;n} : \mathrm{UVal}_n \to \tau$ which is needed for contravariantly converting UVal_n arguments to the appropriate type in the $\mathtt{inject}_{\tau;n}$ case for function types.

Generally, $\mathtt{inject}_{\tau;n}$ converts a value $\mathbf{v}$ of type τ to a value of type UVal_n that behaves like the compilation $[\![\mathbf{v}]\!]$. The cases for base values use the related tagging and case (e.g., $\mathtt{in}_{\mathtt{Unit};n}$ and $\mathtt{case}_{\mathtt{Bool};n}$) to achieve this. For pair and sum values, $\mathtt{inject}_{\tau;n}$ and $\mathtt{extract}_{\tau;n}$ 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 UVal_ns of a certain expected form. Note that when UVal_n values do not have the form expected for type τ, then $\mathtt{extract}_{\tau;n}$ 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.

For function types, $\mathtt{inject}_{\tau;n}$ and $\mathtt{extract}_{\tau;n}$ 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 $\mathtt{extract}_{\tau;n}$ and $\mathtt{inject}_{\tau;n}$ 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, $\mathtt{inject}_{\tau;n}$ 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 t behaves as $\mathbf{t}$ when used in a way that is valid according to type τ), then $\mathtt{inject}_{\tau;n}\ \mathbf{t}$ behaves as the emulation of $\mathsf{protect}_{\tau}\ \mathsf{t}$. A dual result about $\mathtt{extract}_{\tau;n}$ and $\mathsf{confine}_{\tau}$ states (intuitively) that if a term $\mathbf{t}$ behaves as an emulation of value t, then $\mathsf{confine}_{\tau}\ \mathsf{t}$ will behave as $\mathtt{extract}_{\tau;n}\ \mathbf{t}$ when used in ways that are valid according to type τ.

Lemma 9 (Inject is protect and extract is confine)**.** *If* ($m \geq n$ *and* $p = \mathtt{precise}$) *or* ($\square =\lesssim$ *and* $p = \mathtt{imprecise}$) *and if* $\hat{\Gamma} \vdash \mathbf{t}\ \square_n\ \mathsf{t} : \tau$*, then*

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

If ($m \geq n$ *and* $p = \mathtt{precise}$) *or* ($\square =\lesssim$ *and* $p = \mathtt{imprecise}$) *and if* $\hat{\Gamma} \vdash \mathbf{t}\ \square_n\ \mathsf{t} : \mathtt{EmulDV}_{m;p}$ *then*

$$\hat{\Gamma} \vdash \mathtt{extract}_{\tau;m}\ \mathbf{t}\ \square_n\ \mathsf{confine}_{\tau}\ \mathsf{t} : \tau.$$

Example 5.6. Consider again Example 4.1. We have that

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

$\lambda \mathbf{x} : \mathtt{Unit}.\, \mathbf{x}$ behaves like $\lambda \mathsf{x}.\, \mathsf{x}$, when the latter is used in ways that are valid for a value of type $\mathtt{Unit} \to \mathtt{Unit}$. Lemma 9 then yields:

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

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

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

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

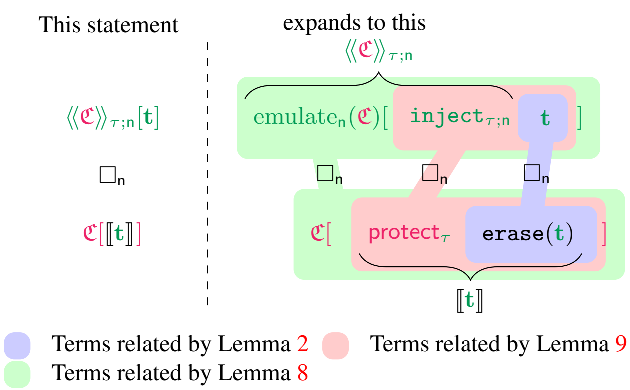

Figure 16. A more accurate picture of related components of compiled term $\mathtt{t}$, program context $\mathfrak{C}$, compilation $[\![\mathtt{t}]\!]$ and emulation $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}$ than in the simplified Fig. 3.

We invite the reader to verify that both expressions behave appropriately when applied to any values $\mathbf{v}$ and v that are related by $\mathtt{EmulDV}_{n;p}$: for example ($\mathbf{v} = \mathtt{in}_{\mathtt{Unit};n-1}\ \mathtt{unit}$ and $\mathsf{v} = \mathtt{unit}$), ($\mathbf{v} = \mathtt{in}_{\to;n-1}\ (\lambda \mathbf{x} : \mathrm{UVal}_{n-1}.\, \mathbf{x})$ and $\mathsf{v} = \lambda \mathsf{x}.\, \mathsf{x}$) or ($\mathbf{v} = \mathrm{unk}_n$, 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 2 (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.* $\langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n} \stackrel{\text{def}}{=} \mathrm{emulate}_n(\mathfrak{C})[\mathtt{inject}_{\tau;n}\ \cdot]$

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. 16 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 $\emptyset \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}[\mathbf{t}_1] \gtrsim_n \mathfrak{C}[[\![\mathbf{t}_1]\!]] : \mathtt{EmulDV}_{n;precise}$ and $\emptyset \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}[\mathbf{t}_2] \lesssim_{n'} \mathfrak{C}[[\![\mathbf{t}_2]\!]] : \mathtt{EmulDV}_{n;imprecise}$ so that the proof goes through unchanged.

The correctness of $\langle\!\langle \cdot \rangle\!\rangle_{\tau;n}$ is captured in Lemma 10.

Lemma 10 (Correctness of $\langle\!\langle \cdot \rangle\!\rangle_{\tau;n}$). *If* $\boldsymbol{\Gamma} \vdash \mathbf{t} \square_n \mathsf{t} : \tau$ *then* $\boldsymbol{\Gamma} \vdash \langle\!\langle \mathfrak{C} \rangle\!\rangle_{\tau;n}[\mathbf{t}] \square_n \mathfrak{C}[\mathsf{t}] : \tau$.

6. Compiler Full-Abstraction

This section presents the proof that the compiler $[\![\cdot]\!]$ is fully-abstract (Theorem 3) in terms of the logical relations of Fig. 7. As previously mentioned, this results in proving compiler correctness (Theorem 1) and compiler security (Theorem 2). As suggested by Fig. 1 in Section 1, the lemmas presented in Section 4 are enough to prove compiler correctness of $[\![\cdot]\!]$. Dually, as suggested by Fig. 3 in Section 1, the lemmas presented in Section 5 are enough to prove compiler security of $[\![\cdot]\!]$.

Recall from Definition 1 that $[\![\mathbf{t}]\!]$ is $\mathtt{protect}_\tau\ (\mathtt{erase}(\mathbf{t}))$.

Theorem 1 ($[\![\cdot]\!]$ is correct). *If* $\emptyset \vdash \mathbf{t}_1 : \tau$ *and* $\emptyset \vdash \mathbf{t}_2 : \tau$ *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 $\mathfrak{C}[\mathbf{t}_2]$. 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 $\mathfrak{C} \stackrel{\text{def}}{=} \mathtt{erase}(\mathfrak{C})$, Lemma 3 yields $\vdash \mathfrak{C} \square \mathfrak{C} : \emptyset, \tau \to \emptyset, \tau'$.
By Lemma 5, 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 $\mathfrak{C}[[\![\mathbf{t}_1]\!]]\Downarrow$ by Lemma 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 1. □

Theorem 2 ($[\![\cdot]\!]$ is secure). *If* $\emptyset \vdash \mathbf{t}_1 : \tau$ *and* $\emptyset \vdash \mathbf{t}_2 : \tau$ *and* $\mathbf{t}_1 \simeq_{ctx} \mathbf{t}_2 : \tau$, *then* $[\![\mathbf{t}_1]\!] \simeq_{ctx} [\![\mathbf{t}_2]\!]$.

Proof. By definition of $\simeq_{ctx}$, we need to prove that $\mathfrak{C}[[\![\mathbf{t}_1]\!]]\Downarrow$ iff $\mathfrak{C}[[\![\mathbf{t}_2]\!]]$. By symmetry, it suffices to prove the $\Rightarrow$ direction.
Take a $\vdash \mathfrak{C} : \emptyset \to \emptyset$ and suppose that $\mathfrak{C}[[\![\mathbf{t}_1]\!]]\Downarrow$.
We need to show that $\mathfrak{C}[[\![\mathbf{t}_2]\!]]\Downarrow$.
Take n larger than the number of steps in $\mathfrak{C}[[\![\mathbf{t}_1]\!]]\Downarrow$.
By Lemma 2, we have that $\emptyset \vdash \mathbf{t}_1 \gtrsim_n \mathtt{erase}(\mathbf{t}_1) : \tau$.
By Lemma 9, we then have (taking $m = n \geq n$) that $\emptyset \vdash \mathtt{inject}_{\tau;n}\ \mathbf{t}_1 \gtrsim_n [\![\mathbf{t}_1]\!] : \mathtt{EmulDV}_{n;precise}$.
Define $\mathfrak{C} \stackrel{\text{def}}{=} \mathrm{emulate}_n(\mathfrak{C})$. From Lemma 8, we get that $\vdash \mathfrak{C} \gtrsim_n \mathfrak{C} : \emptyset, \mathtt{EmulDV}_{n;precise} \to \emptyset, \mathtt{EmulDV}_{n;precise}$.
It follows that $\emptyset \vdash \mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_1] \gtrsim_n \mathfrak{C}[[\![\mathbf{t}_1]\!]] : \mathtt{EmulDV}_{n;precise}$.
By Lemma 1 applied to $\mathfrak{C}[[\![\mathbf{t}_1]\!]]\Downarrow$, and by the choice of n, we have that $\mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_1]\Downarrow$.
If we take $\mathfrak{C}' = \mathfrak{C}[\mathtt{inject}_{\tau;n}\ \cdot]$, then from $\mathbf{t}_1 \simeq_{ctx} \mathbf{t}_2$ and $\mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_1]\Downarrow$ it follows that $\mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_2]\Downarrow$.
Take n' the number of steps in the termination of $\mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_2]\Downarrow$.
From Lemma 2 we have that $\emptyset \vdash \mathbf{t}_2 \lesssim_{n'} \mathtt{erase}(\mathbf{e}_2) : \tau$.
By Lemma 9, we have that $\emptyset \vdash \mathtt{inject}_{\tau;n}\ \mathbf{t}_2 \lesssim_{n'} [\![\mathbf{t}_2]\!] : \mathtt{EmulDV}_{n;imprecise}$.
From Lemma 8, we get that $\vdash \mathfrak{C} \lesssim_{n'} \mathfrak{C} : \emptyset, \mathtt{EmulDV}_{n;imprecise} \to \emptyset, \mathtt{EmulDV}_{n;imprecise}$.
It follows that $\emptyset \vdash \mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_2] \lesssim_{n'} \mathfrak{C}[[\![\mathbf{t}_2]\!]] : \mathtt{EmulDV}_{n;imprecise}$.
By Lemma 1, by $\mathfrak{C}[\mathtt{inject}_{\tau;n}\ \mathbf{t}_2]\Downarrow$, and by the choice of n', we have that $\mathfrak{C}[[\![\mathbf{t}_2]\!]]\Downarrow$ as required. □

Theorem 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 1 provides the $\Leftarrow$ direction while Theorem 2 provides the $\Rightarrow$ one. □

7. 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). Unless targeting typed assembly language [19], 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. A longstanding open problem is fully-abstract compilation of parametric polymorphism to a form of operational sealing primitives [17, 20, 29]. 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 $* \rightarrow *$, which we cannot add to System F without causing other problems for the compilation. We conjecture an approximate back-translation is what is needed to provide a fully-abstract compilation in this setting and we hope to confirm it in future work.

In other settings, it is not clear whether it is even 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 [23]. 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 [10]. Ahmed and Blume have demonstrated a solution for this problem which exploits parametric polymorphism to enforce the invocation of the correct continuation [4], 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.

8. Related Work

Secure compilation through full-abstraction was pioneered by Abadi [1] and successfully applied to many different settings [2, 4, 5, 8, 11, 15, 24, 28, 30]. Recently, Parrow has proven which conditions must hold in the source and target languages in order to provide a fully-abstract compiler between the two [22]. Moreover Gorla and Nestmann have investigated fully-abstract translations for language expressiveness, concluding that full-abstraction is meaningful when it entails properties such as security [12].

Some secure compilation works prove compiler full-abstraction using logical relations [4, 5, 8, 28, 30]. Ahmed and Blume 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 [4, 5]. Tse and Zdancewic started a line of work to compile the dependency core calculus of Abadi *et al.* (DCC [3]) into System F, effectively proving that non interference can be encoded with parametricity [30]. They achieve an analogous of 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 refined their result for a weaker form of DCC [28]. A fully-abstract translation from DCC to System F was provided by Bowman and Ahmed [8], and that is the closest work to what presented here. While they prove an important result, we believe the formal machinery adopted by Bowman and Ahmed is heavier than the one presented here. For example, we do not need a new logical relation to prove correctness of the back-translation and can reuse the one for terms. Moreover, all the works that securely compile DCC target System F, so they have a typed target language for their secure compilers while in this paper the target language is untyped.

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) [2, 11, 15, 24]. 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.

As mentioned in Section 1, the presented proof technique borrows from recent results in compiler correctness [6, 7, 13]. These results build cross-language logical relation based on a common language specification in order to prove compiler correctness. Benton and Hur provided a correct compiler from a call-by-value λ-calculus [6] as well as for System F with recursion to to a SECD machine [7]. Hur and Dreyer devised a correct compiler between an idealised ML to assembly [13]. The techniques devised in these works were further developed into Relational Transition Systems (RTS) in order to prove both vertically- and horizontally-composable compiler correctness [14, 21]. 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 [18]. For example, Perconti and Ahmed devised a two-step correct compiler for System F with existential and recursive types to typed assembly language using multi-language logical relations [25]. As compiler full-abstraction does scale to multi-pass compilers (i.e., it is vertically composable), there was no necessity to use RTS nor multi-language systems.

9. 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-translations 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. 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.

References

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

[2] 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.

[3] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292555.

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

[5] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 431–444, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034830.

[6] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.*, 44:97–108, Aug. 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596567.

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

[8] W. J. Bowman and A. Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, New York, NY, USA, 2015. ACM.

[9] 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.

[10] 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.

[11] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *POPL '13*, pages 371–384, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429114.

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

[13] 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.

[14] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and kripke logical relations. *SIGPLAN Not.*, 47:59–72, Jan. 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103666.

[15] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 161–174, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4365-9. doi: 10.1109/CSF.2011.18.

[16] 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.

[17] 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 *Lecture Notes in Computer Science*, pages 16–31. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78738-9. doi: 10.1007/978-3-540-78739-6_2.

[18] 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.

[19] 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.

[20] G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 135–148. ACM, 2009. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596572.

[21] 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 *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, New York, NY, USA, 2015. ACM.

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

[23] M. Patrignani. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. PhD thesis, KU Leuven, Leuven, Belgium, May 2015.

[24] 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.

[25] 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.

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

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

[28] 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 *Lecture Notes in Computer Science*, pages 301–315. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-77504-1. doi: 10.1007/978-3-540-77505-8_24.

[29] 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.

[30] S. Tse and S. Zdancewic. Translating dependency into parametricity. *SIGPLAN Not.*, 39:115–125, Sept. 2004. ISSN 0362-1340. doi: 10.1145/1016848.1016868.