ON THE SEMANTIC EXPRESSIVENESS OF ISO- AND EQUI-RECURSIVE TYPES

DOMINIQUE DEVRIESE, ERIC M. MARTIN, AND MARCO PATRIGNANI

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

Jane Street Capital
e-mail address: emartin@janestreet.com

University of Trento
e-mail address: mp@cs.stanford.edu

Abstract. Recursive types extend the simply-typed lambda calculus (STLC) with the additional expressive power to enable diverging computation and to encode recursive data-types (e.g., lists). Two formulations of recursive types exist: iso-recursive and equi-recursive. The relative advantages of iso- and equi-recursion are well-studied when it comes to their impact on type-inference. However, the relative semantic expressiveness of the two formulations remains unclear so far.

This paper studies the semantic expressiveness of STLC with iso- and equi-recursive types, proving that these formulations are *equally expressive*. In fact, we prove that they are both as expressive as STLC with only term-level recursion. We phrase these equi-expressiveness results in terms of full abstraction of three canonical compilers between these three languages (STLC with iso-, with equi-recursive types and with term-level recursion). Our choice of languages allows us to study expressiveness when interacting over both a simply-typed and a recursively-typed interface. The three proofs all rely on a typed version of a proof technique called approximate backtranslation.

Together, our results show that there is no difference in semantic expressiveness between STLCs with iso- and equi-recursive types. In this paper, we focus on a simply-typed setting but we believe our results scale to more powerful type systems like System F.

To present notions more clearly, this paper uses syntax highlighting accessible to both colourblind and black & white readers [Patrignani, 2020]. For a better experience, please print or view this in colour. Specifically, we use a blue, sans-serif font for STLC with the fix operator, a **red, bold** *font for* **STLC** *with* **iso-recursive** *types, and pink, italics font for STLC with coinductive equi-recursive types. Elements common to all languages are typeset in a black, italic font (to avoid repetition).*

2012 ACM CCS: [**Theory of computation Lambda calculus**]: 300; [**Theory of computation Type theory**]: 300; [**Software and its engineering Recursion**]: 300 .

Key words and phrases: Fully abstract compilation, cross-language logical relation, modular compilation.

[*] extended version of the paper in POPL'21, now including a fixed and mechanized proof. More details are in Section 1.3.

1. Introduction

Recursive types were first proposed by Morris [1968] as a way to recover divergence from the untyped lambda calculus in a simply-typed lambda calculus. They also enable the definition of recursive data-types such as lists, trees, and Lisp S-expressions in typed languages.

Morris' original formulation was equi-recursive: a type $\mu\alpha.\,\tau$ was regarded as an infinite type and considered equal to its unfolding $\tau[\mu\alpha.\,\tau/\alpha]$. Subsequent formulations (e.g., Abadi and Fiore [1996]) use different type equality relations. In this paper we will work with λ^{μ}_{E}: a standard simply-typed lambda calculus with coinductive equi-recursive types [e.g., Cai et al., 2016].

Years after Morris' formulation of recursive types, a different one appeared [e.g., Gordon et al., 1979; Harper and Mitchell, 1993], where the two types are not considered equal, but *isomorphic*: values can be converted from $\boldsymbol{\mu\alpha.\,\tau}$ to $\boldsymbol{\tau[\mu\alpha.\,\tau/\alpha]}$ and back using explicit **fold** and **unfold** annotations in terms. These annotations are used to guide typechecking, but they also have a significance at runtime: an explicit reduction step is needed to cancel them out: $\mathbf{unfold}_{\mu\alpha.\tau}\ (\mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v}) \hookrightarrow \mathbf{v}$. In this paper, we work with a standard iso-recursive calculus $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$.

The relation between these two formulations has been studied by Abadi and Fiore [1996] and Urzyczyn [1995] (the latter focusing on positive recursive types). Specifically, they show that any term typable in one formulation can also be typed in the other, possibly by adding extra **unfold** or **fold** annotations. Additionally, Abadi and Fiore prove that for types considered equal in the equi-recursive system, there exist coercion functions in the iso-recursive formulation that are mutually inverse in the (axiomatised) program logic. The isomorphism properties are proved in a logic for the iso-recursive language (which is only conjectured to be sound), and the authors do not consider an operational semantics.

The relative semantic expressiveness of the two formulations, however, has remained yet unexplored. In principle, executions that are converging in the equi-recursive language may become diverging in the iso-recursive setting because of the extra fold-unfold reductions. Because of this, it is unclear whether the two formulations of recursive types produce equally expressive languages.

Concretely, in this paper, we study the expressive power of $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ and λ^{μ}_{E} when interacting over two kinds of language interfaces. The first is characterized by simply-typed lambda calculus types, which do not mention recursive types themselves. We consider implementations of this interface in λ^{fx}, a simply typed lambda calculus with term-level recursion in the form of a primitive fixpoint operator. We embed these implementations into both $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ and λ^{μ}_{E} using two so-called canonical compilers, i.e., compilers that map any construct of the source language into the same – or the closest – construct of the target. We show that if two λ^{fx} terms cannot be distinguished by λ^{fx} contexts, then the same is true for both $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ and λ^{μ}_{E} contexts, i.e., the compiler is fully abstract. Additionally, we consider STLC types that contain recursive types themselves as interfaces. We take implementations of them in $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ and a canonical compiler for them into λ^{μ}_{E}. We show that this compiler is also fully abstract. These three fully-abstract compilation results establish the equi-expressiveness of $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$, λ^{μ}_{E}, and λ^{fx} contexts, interacting over simply-typed interfaces with and without recursive types. Moreover, these three fully-abstract compilation results have been completely formalised in the Coq proof assistant.

Proving full abstraction for a compiler is notoriously hard, particularly in the preservation direction, i.e., showing that equivalent source terms get compiled to equivalent target terms.

Informally, it requires showing that any behaviour (e.g., termination) of target program contexts can be replicated by source program contexts. Demonstrating such a claim is particularly complicated in our setting since λ^{μ}_{E} contexts have coinductive (and thus infinite) type equality derivations. To be able to prove fully-abstract compilation, we adopt the approximate backtranslation proof technique of Devriese et al. [2017]. This technique relies on two key components: a cross-language approximation relation between source and target terms (and source and target program contexts) and a backtranslation function from target to source program contexts. Intuitively, the approximation relation is used to tell when a source and a target term (or program context) equi-terminate; we use step-indexed logical relations to define this and rely on the step as the measure for the approximation. The backtranslation is a function that takes a target program context and produces a source program context that approximates the target one. This is particularly appropriate for backtranslating λ^{μ}_{E} program contexts, since we show that it is sufficient to approximate their coinductive derivations instead of replicating them precisely.

We construct three backtranslations: from $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ and λ^{μ}_{E} contexts respectively into λ^{fx} ones and from λ^{μ}_{E} contexts into $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ ones. We do so by defining a family of types for backtranslated terms that is not just indexed by the approximation level but also by the target type of the backtranslated term. To the best of our knowledge, this is a novel approach, since all existing work relies on a single type for backtranslated terms [Devriese et al., 2017; New et al., 2016].

For proving the correctness of these backtranslations, we define a step-indexed logical relation to express when compiled and backtranslated terms approximate each other. While the logical relation is largely the same for the different compilers and backtranslations, differences in the language semantics impose that we treat backtranslated $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ terms differently from λ^{μ}_{E}.

Like previous work [Devriese et al., 2017; New et al., 2016], we use a step-indexed logical relation that relates terms (and values) across languages so long as they equi-terminate. In previous work, the step-indexed logical relation approximates (or, relates) terms (and values) up to an index that is related to the amount of steps that are required for termination. In this work, we change that approximation to also consider an additional bound on the size of terms encountered during termination. To provide this new bound, we introduce a novel notion of termination, called size-bound termination, and state that terms are related when size-bound termination of one term implies termination of the other. The need for an additional bound (and thus for size-bound termination) arose while mechanising these proofs in the Coq proof assistant, as this led to the discovery of a bug in the previous proofs (as we describe in Section 1.3). The additional bound lets us reason explicitly about the finiteness of values encountered during reductions, and it lets us go through those cases that broke certain proofs (as we describe in detail in Example 9 in Section 4.2.2).

1.1. Using Fully Abstract Compilation to Compare Language Expressiveness.

To study language expressiveness meaningfully, it is important to phrase the question properly. If we just consider programs that receive a natural number and return a boolean, then both languages will allow expressing the same set of algorithms, simply by their Turing completeness [Mitchell, 1993].

The question of comparing language expressiveness is more interesting if we consider programs that interact over a richer interface. Consider, for example, a term t from the simply-typed lambda calculus embedded into either the $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ or λ^{μ}_{E} calculus. An interesting question is whether there are ways in which λ^{μ}_{E} contexts (i.e., larger programs) can interact with t that contexts in $\lambda^{\mu}_{\mathrm{I}}$ cannot. The use of contexts in different languages interacting with a common term as a way of measuring language expressiveness has a long history [Felleisen, 1991; Mitchell, 1993], mostly in the study of process calculi [Parrow, 2008]. In this setting, equal expressiveness of programming languages is sometimes argued for by proving the existence of a fully-abstract compiler from one language to the other [Gorla and Nestmann, 2016]. Such a compiler translates contextually-equivalent terms in a source language (indicated as L_{src}) to contextually-equivalent terms in a target language (indicated as L_{trg}) [Abadi, 1998; Patrignani et al., 2019]. That is, if contexts cannot distinguish two terms in L_{src}, they will also not be able to distinguish them after the compilation to L_{trg}.

Let us now argue why the choice of fully-abstract compilation as a measure of the relative expressiveness of programming languages is the right one in our setting. After all, several researchers have pointed out that the mere existence of a fully-abstract compilation is not in itself meaningful and only compilers that are sufficiently well-behaved should be considered [Gorla and Nestmann, 2016; Parrow, 2008]. The reason for this is that one can build a degenerate fully-abstract compiler that shows both languages having an equal amount (cardinality) of equivalence classes for terms. This would indicate that the languages are equally-expressive, but unfortunately this is also trivial to satisfy [Parrow, 2008]. These degenerate examples, as such, clarify the necessity for well-behavedness of the compiler. However, we have not found a clear argument explaining why well-behaved fully-abstract compilation implies equi-expressiveness of languages, so here it is.

In our opinion (and we believe this point has not yet been made in the literature), the issue is that fully-abstract compilation results measure language expressiveness *not* by verifying that they can express the same *terms*, but that they can express the same *contexts*. Defining when a context in L_{src} is the same as a context in L_{trg} is hard, and therefore fully-abstract compilation simply requires that L_{trg} contexts can express the interaction of L_{src} contexts with any term that is shared between both languages. The role of the compiler, the translation from L_{src} to L_{trg}, is simply to obtain this common term against which expressiveness of contexts in both languages can be measured.

In other words, expressiveness of a programming language is only meaningful with respect to a certain interface and the role of the compiler is to map L_{src} implementations of this interface to L_{trg} implementations. In a sense, the L_{src} implementation of the interface should be seen as an expressiveness challenge for L_{src} contexts and the compiler translates it to the corresponding challenge in L_{trg}. As such, the compiler should be seen as part of the definition of equi-expressiveness and the well-behavedness requirement is there to make sure the L_{src} challenge is translated to "the same" challenge in L_{trg}. Fortunately, in this work we only rely on canonical compilers that provide the most intuitive translation for a term in our source languages into "the same" term in our target ones. Thus, we believe that in our setting using fully-abstract compilation is the right tool to measure the relative expressiveness of programming languages.

1.2. **Contributions and Outline.**

To summarize, the key contribution of this paper is the proof that iso- and coinductive equi-recursive typing are equally expressive. This result is achieved via the following contributions (depicted in Figure 1).

- An adaptation of the approximate backtranslation proof technique to operate on families of backtranslation types that are type-indexed on target types
- An adaptation of the proof technique to be more precise when relating terms cross-language by relying on the notion of size-bound termination;

- A proof that the compiler from λ^{fx} to $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ is fully abstract with an approximate backtranslation;
- A proof that the compiler from $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ to λ^{μ}_{E} is fully abstract with an approximate backtranslation;
- A proof that the compiler from λ^{fx} to λ^{μ}_{E} is fully abstract with an approximate backtranslation;
- The mechanisation of these three proofs in the Coq proof assistant.

$$\begin{tikzcd}
 & \lambda^{\mathsf{fx}} \arrow[dl, "\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}"'] \arrow[dr, "\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}"] & \\
\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}} \arrow[ur, dotted, "\langle\!\langle\cdot\rangle\!\rangle^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}"'] \arrow[rr, "\llbracket\cdot\rrbracket^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}"] & & \lambda^{\mu}_{E} \arrow[ul, dotted, "\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{E}}_{\lambda^{\mathsf{fx}}}"] \arrow[ll, dotted, bend left, "\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{E}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}"]
\end{tikzcd}$$

FIGURE 1. Our contributions, visually. Full arrows indicate canonical embeddings $\llbracket\cdot\rrbracket$ while dotted ones are (approximate) backtranslations $\langle\!\langle\cdot\rangle\!\rangle$. Translations' superscripts indicate input languages while their subscripts indicate output languages.

Note that technically, we can derive the compiler and backtranslation between λ^{fx} and λ^{μ}_{E} by composing the compilers and backtranslations through $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$. We present this result as a stand-alone one because it offers insights on proofs of fully-abstract compilation for languages with coinductive notions.

The remainder of this paper is organised as follows. We first formalise the languages we use (λ^{fx}, $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E}) as well as the cross-language logical relations which express when two terms in those languages are semantically equivalent (Section 2). Next, we present fully-abstract compilation and describe our approximate backtranslation proof technique in detail (Section 3). Then we define the three compilers (from λ^{fx} to $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$, from λ^{fx} to λ^{μ}_{E} and from $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ to λ^{μ}_{E}) and prove that they are fully abstract using three approximate backtranslations (Section 4). These compilers and their fully-abstract compilation proofs are all formalised in Coq, so we also present the most useful insights into this formalisation (Section 5). After a discussion of the presented results (Section 6, we present related work (Section 7) and conclude (Section 8).

For the sake of simplicity we omit some elements of the formalisation such as auxiliary lemmas and proofs. The Coq mechanisation of this work is available at:

`https://github.com/dominiquedevriese/fixismu-coq`

1.3. **Comparison with the Previous Version.** This work extends the work of Patrignani et al. [2021] presented at POPL'21 in the following way:

- We fix a bug in the original proof that broke Lemma 13. The bug is addressed by making the approximate logical relation rely on an additional bound on the size of terms encountered during reductions, as mentioned before. This, in turn changes the observation relation of the logical relation, i.e., the part that tells when two terms are related. Previously (as well as in related work [Devriese et al., 2016]), a term t was related to another one t at level n if termination of t in at most n steps implied termination of t in some steps (and vice versa). Here, we introduce a new notion of bounded termination (called size-bound termination) that a term fulfils for some steps m if the term terminates in at most m steps and (roughly) terms encountered during this reduction have at most size m (in terms of the depth of the AST of the term). We rely on size-bound termination in the observation relation and state that a term t is related to another t at level j if size-bound termination of t in at most j steps implies termination of t in some steps (and vice versa). Intuitively, in the previous formulation the step index n imposes a bound on the amount of steps required for termination. Here instead, the step index j imposes a bound both on the steps required for termination and on the size of terms encountered during such termination.

We explain in more detail the problem with the old formulation and how this new idea lets Lemma 13 go through in Section 4.2.2, where we discuss Example 9.

- We mechanise the three fully-abstract compilation proofs in the Coq proof assistant and report on the formalisation in Section 5.

2. Languages and Cross-Language Logical Relations

This section presents the simply-typed lambda calculus (λ) and its extensions with a typed fixpoint operator (λ^{fx}), with iso-recursive types ($\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$) and with coinductive equi-recursive types (λ^{μ}_{E}). We first define the syntax (Section 2.1), then the static semantics (Section 2.2) and then the operational semantics of these languages (Section 2.3). Finally, this section presents the cross-language logical relations used to reason about the expressiveness of terms in different languages (Section 2.5). Note that these logical relations are partial, the key addition needed to attain fully-abstract compilation is presented in Section 3.3 only after said addition is justified.

2.1. Syntax.

All languages include standard terms (t) and values (v) from the simply-typed lambda calculus: lambda abstractions, applications, pairs, projections, tagged unions, case destructors, booleans, branching, unit and sequencing. Additionally, λ^{fx} has a fix operator providing general recursion, while $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ has **fold** and **unfold** annotations; λ^{μ}_{E} requires no additional syntactic construct.

Regarding types, both $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E} add recursive types according to the same syntax. In $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E}, recursive types are syntactically constrained to be *contractive*. Note however that for simplicity of presentation we will indicate a type as τ and simply report the contractiveness constraints when meaningful. A recursive type $\mu\alpha.\,\tau$ is contractive if, the use of the recursion variable α in τ occurs under a type constructor such as $\rightarrow$ or $\times$ [MacQueen et al., 1984]. Non-contractive types (e.g., $\mu\alpha.\,\alpha$) are not inhabited by any value, so it is reasonable to elide them. Moreover, they do not have an infinite unfolding and (without restrictions on the type equality relation) can be proven equivalent to any other type [Im et al., 2013], which is undesirable.

All languages have evaluation contexts ($\mathbb{E}$), which indicate where the next reduction will happen, and program contexts ($\mathfrak{C}$), which are larger programs to link terms with.

$$
\begin{aligned}
\tau, \sigma &::= \mathit{Unit} \mid \mathit{Bool} \mid \tau^s \to \tau^s \mid \tau^s \times \tau^s \mid \tau^s \uplus \tau^s \mid \boldsymbol{\mu\alpha.\,\tau} \mid \mu\alpha.\,\tau \\
\tau^s &::= \boldsymbol{\alpha} \mid \alpha \mid \tau \\
\Gamma &::= \emptyset \mid \Gamma, x : \tau \\
v &::= \mathit{unit} \mid \mathit{true} \mid \mathit{false} \mid \lambda x : \tau.\, t \mid \langle v, v \rangle \mid \mathit{inl}\ v \mid \mathit{inr}\ v \mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v} \\
t &::= \mathit{unit} \mid \mathit{true} \mid \mathit{false} \mid \lambda x : \tau.\, t \mid x \mid t\ t \mid t.\mathit{1} \mid t.\mathit{2} \mid \langle t, t \rangle \\
&\quad \mid \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t \mid \mathit{inr}\ x_2 \mapsto t \mid \mathit{inl}\ t \mid \mathit{inr}\ t \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \mid t; t \\
&\quad \mid \mathsf{fix}_{\tau\to\tau}\ \mathsf{t} \mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{t} \mid \mathbf{unfold}_{\mu\alpha.\tau}\ \mathbf{t} \\
\mathbb{E} &::= [\cdot] \mid \mathbb{E}\ t \mid v\ \mathbb{E} \mid \mathbb{E}.\mathit{1} \mid \mathbb{E}.\mathit{2} \mid \langle \mathbb{E}, t \rangle \mid \langle v, \mathbb{E} \rangle \mid \mathit{case}\ \mathbb{E}\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t_1 \mid \mathit{inr}\ x_2 \mapsto t_2 \\
&\quad \mid \mathit{inl}\ \mathbb{E} \mid \mathit{inr}\ \mathbb{E} \mid \mathbb{E}; t \mid \mathit{if}\ \mathbb{E}\ \mathit{then}\ t\ \mathit{else}\ t \mid \mathsf{fix}_{\tau\to\tau}\ \mathbb{E} \mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathbb{E} \mid \mathbf{unfold}_{\mu\alpha.\tau}\ \mathbb{E} \\
\mathfrak{C} &::= [\cdot] \mid \lambda x : \tau.\, \mathfrak{C} \mid \mathfrak{C}\ t \mid t\ \mathfrak{C} \mid \mathfrak{C}.\mathit{1} \mid \mathfrak{C}.\mathit{2} \mid \langle \mathfrak{C}, t \rangle \mid \langle t, \mathfrak{C} \rangle \mid \mathit{case}\ \mathfrak{C}\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t \mid \mathit{inr}\ x_2 \mapsto t \\
&\quad \mid \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto \mathfrak{C} \mid \mathit{inr}\ x_2 \mapsto t \mid \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t \mid \mathit{inr}\ x_2 \mapsto \mathfrak{C} \\
&\quad \mid \mathit{inl}\ \mathfrak{C} \mid \mathit{inr}\ \mathfrak{C} \mid \mathfrak{C}; t \mid t; \mathfrak{C} \mid \mathit{if}\ \mathfrak{C}\ \mathit{then}\ t\ \mathit{else}\ t \mid \mathit{if}\ t\ \mathit{then}\ \mathfrak{C}\ \mathit{else}\ t \\
&\quad \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ \mathfrak{C} \mid \mathsf{fix}_{\tau\to\tau}\ \mathfrak{C} \mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathfrak{C} \mid \mathbf{unfold}_{\mu\alpha.\tau}\ \mathfrak{C}
\end{aligned}
$$

As mentioned in Section 1, we need a measure to define size-bound termination as the new logical relation requires. The measure we rely on is the size of a term t, which we calculate via function $\mathtt{size}(\cdot) : t \to n \in \mathbb{N}$. Intuitively, the size of a measure counts the number of nodes in the term's AST, ignoring the bodies of lambdas. As a result, apart from bodies of lambdas, any sub-term t' has size smaller than the super-term t that contains t'.

$$\mathtt{size}(\mathit{unit}) = 1 \quad \mathtt{size}(\mathit{true}) = 1 \quad \mathtt{size}(\mathit{false}) = 1$$

$$
\begin{aligned}
\mathtt{size}(x) &= 1 & \mathtt{size}(\lambda x : \tau.\, t) &= 1 \\
\mathtt{size}(t\ t') &= \mathtt{size}(t) + \mathtt{size}(t') + 1 & \mathtt{size}(t.1) &= \mathtt{size}(t) + 1 \\
\mathtt{size}(t.2) &= \mathtt{size}(t) + 1 & \mathtt{size}(\langle t, t' \rangle) &= \mathtt{size}(t) + \mathtt{size}(t') + 1 \\
\mathtt{size}(\mathit{inl}\ t) &= \mathtt{size}(t) + 1 & \mathtt{size}(\mathit{inr}\ t) &= \mathtt{size}(t) + 1 \\
\mathtt{size}(t; t') &= \mathtt{size}(t) + \mathtt{size}(t') + 1 & \mathtt{size}(\mathsf{fix}_{\tau\to\tau}\ \mathsf{t}) &= \mathtt{size}(\mathsf{t}) + 1 \\
\mathtt{size}(\mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{t}) &= \mathtt{size}(\mathbf{t}) + 1 & \mathtt{size}(\mathbf{unfold}_{\mu\alpha.\tau}\ \mathbf{t}) &= \mathtt{size}(\mathbf{t}) + 1
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{size}(\mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t' \mid \mathit{inr}\ x_2 \mapsto t'') &= \mathtt{size}(t) + \mathtt{size}(t') + \mathtt{size}(t'') + 1 \\
\mathtt{size}(\mathit{if}\ t\ \mathit{then}\ t'\ \mathit{else}\ t'') &= \mathtt{size}(t) + \mathtt{size}(t') + \mathtt{size}(t'') + 1
\end{aligned}
$$

2.2. **Static Semantics.** This section presents the (fairly standard) static semantics of our languages, we delay discussing alternative formulations of equi-recursive types to Section 7. The static semantics for terms follows the canonical judgement $\Gamma \vdash t : \tau$, which attributes type τ to term t under environment Γ and occasionally relies on function $\mathtt{ftv}(\tau)$, which returns the free type variables of τ. The only difference in the typing rules regards **fold**/**unfold** terms (Rules $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$-Type-fold and $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$-Type-unfold) and the introduction of the type equality ($\triangleq$ in Rule λ^{μ}_{E}-Type-eq).

$\boxed{\Gamma \vdash t : \tau}$

$$\text{(Type-var)}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(Type-unit)}\ \frac{}{\Gamma \vdash \mathit{unit} : \mathit{Unit}} \qquad \text{(Type-true)}\ \frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool}} \qquad \text{(Type-false)}\ \frac{}{\Gamma \vdash \mathit{false} : \mathit{Bool}}$$

$$\text{(Type-p1)}\ \frac{\Gamma \vdash t : \tau \times \tau'}{\Gamma \vdash t.1 : \tau} \qquad \text{(Type-p2)}\ \frac{\Gamma \vdash t : \tau' \times \tau}{\Gamma \vdash t.2 : \tau} \qquad \text{(Type-inl)}\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathit{inl}\ t : \tau \uplus \tau'} \qquad \text{(Type-inr)}\ \frac{\Gamma \vdash t : \tau'}{\Gamma \vdash \mathit{inr}\ t : \tau \uplus \tau'}$$

$$\text{(Type-case)}\ \frac{\begin{array}{c}\Gamma \vdash t : \tau' \uplus \tau'' \\ \Gamma, x_1 : \tau' \vdash t' : \tau \qquad \Gamma, x_2 : \tau'' \vdash t'' : \tau\end{array}}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t' \mid \mathit{inr}\ x_2 \mapsto t'' : \tau} \qquad \text{(Type-if)}\ \frac{\begin{array}{c}\Gamma \vdash t : \mathit{Bool} \\ \Gamma \vdash t' : \tau \qquad \Gamma \vdash t'' : \tau\end{array}}{\Gamma \vdash \mathit{if}\ t\ \mathit{then}\ t'\ \mathit{else}\ t'' : \tau}$$

$$\text{(Type-seq)}\ \frac{\begin{array}{c}\Gamma \vdash t : \mathit{Unit} \\ \Gamma \vdash t' : \tau\end{array}}{\Gamma \vdash t; t' : \tau} \qquad \text{(Type-lam)}\ \frac{\begin{array}{c}\Gamma, x : \tau \vdash t : \tau' \\ \mathtt{ftv}\,(\tau) = \emptyset\end{array}}{\Gamma \vdash \lambda x : \tau.\, t : \tau \to \tau'} \qquad \text{(Type-app)}\ \frac{\begin{array}{c}\Gamma \vdash t : \tau' \to \tau \\ \Gamma \vdash t' : \tau'\end{array}}{\Gamma \vdash t\ t' : \tau}$$

$$\text{(Type-pair)}\ \frac{\begin{array}{c}\Gamma \vdash t : \tau \\ \Gamma \vdash t' : \tau'\end{array}}{\Gamma \vdash \langle t, t' \rangle : \tau \times \tau'} \qquad (\lambda^{\mathsf{fx}}\text{-Type-fix})\ \frac{\Gamma \vdash \mathsf{t} : (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2}{\Gamma \vdash \mathsf{fix}_{\tau_1 \to \tau_2}\ \mathsf{t} : \tau_1 \to \tau_2}$$

$$(\lambda^{\mu}_{\mathrm{I}}\text{-Type-fold})\ \frac{\boldsymbol{\Gamma \vdash \mathbf{t} : \tau[\mu\alpha.\,\tau/\alpha]}}{\boldsymbol{\Gamma \vdash \mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{t} : \mu\alpha.\,\tau}} \qquad (\lambda^{\mu}_{\mathrm{I}}\text{-Type-unfold})\ \frac{\boldsymbol{\Gamma \vdash \mathbf{t} : \mu\alpha.\,\tau}}{\boldsymbol{\Gamma \vdash \mathbf{unfold}_{\mu\alpha.\tau}\ \mathbf{t} : \tau[\mu\alpha.\,\tau/\alpha]}}$$

$$(\lambda^{\mu}_{E}\text{-Type-eq})\ \frac{\Gamma \vdash t : \tau \qquad \tau \mathrel{\mathring{=}} \sigma}{\Gamma \vdash t : \sigma}$$

Program contexts have an important role in fully-abstract compilation. They follow the usual typing judgement ($\mathfrak{C} \vdash \Gamma, \tau \to \Gamma', \tau'$), i.e., program context $\mathfrak{C}$ is well typed with a hole of type τ that use free variables in Γ, and overall $\mathfrak{C}$ returns a term of type τ' and uses variables in Γ'.

$$\boxed{\mathfrak{C} \vdash \Gamma, \tau \to \Gamma', \tau'}$$

$$\text{(Type-Ctx-Hole)}\ \frac{}{\vdash \cdot : \Gamma, \tau \to \Gamma, \tau} \qquad \text{(Type-Ctx-Lam)}\ \frac{\vdash \mathfrak{C} : \Gamma'', \tau'' \to (\Gamma, x : \tau'), \tau}{\vdash \lambda x : \tau'.\, \mathfrak{C} : \Gamma'', \tau'' \to \Gamma, \tau' \to \tau}$$

$$\text{(Type-Ctx-Pair1)}\ \frac{\begin{array}{c}\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1 \\ \Gamma \vdash t_2 : \tau_2\end{array}}{\vdash \langle \mathfrak{C}, t_2 \rangle : \Gamma', \tau' \to \Gamma, \tau_1 \times \tau_2} \qquad \text{(Type-Ctx-Pair2)}\ \frac{\begin{array}{c}\Gamma \vdash t_1 : \tau_1 \\ \vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_2\end{array}}{\vdash \langle t_1, \mathfrak{C} \rangle : \Gamma', \tau' \to \Gamma, \tau_1 \times \tau_2}$$

$$\text{(Type-Ctx-Inl)}\ \frac{\vdash \mathfrak{C} : \Gamma'', \tau'' \to \Gamma, \tau}{\vdash \mathit{inl}\ \mathfrak{C} : \Gamma'', \tau'' \to \Gamma, \tau \uplus \tau'} \qquad \text{(Type-Ctx-Inr)}\ \frac{\vdash \mathfrak{C} : \Gamma'', \tau'' \to \Gamma, \tau'}{\vdash \mathit{inr}\ \mathfrak{C} : \Gamma'', \tau'' \to \Gamma, \tau \uplus \tau'}$$

$$\text{(Type-Ctx-App1)}\ \frac{\begin{array}{c}\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1 \to \tau_2 \\ \Gamma \vdash t_2 : \tau_1\end{array}}{\vdash \mathfrak{C}\ t_2 : \Gamma', \tau' \to \Gamma, \tau_2} \qquad \text{(Type-Ctx-App2)}\ \frac{\begin{array}{c}\Gamma \vdash t_1 : \tau_1 \to \tau_2 \\ \vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1\end{array}}{\vdash t_1\ \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_2}$$

(Type-Ctx-Proj1)
$$\frac{\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1 \uplus \tau_2}{\vdash \mathfrak{C}.1 : \Gamma', \tau' \to \Gamma, \tau_1}$$

(Type-Ctx-Proj2)
$$\frac{\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1 \uplus \tau_2}{\vdash \mathfrak{C}.2 : \Gamma', \tau' \to \Gamma, \tau_2}$$

(Type-Ctx-Case1)
$$\frac{\begin{array}{c}\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1 \uplus \tau_2 \\ \Gamma, x_1 : \tau_1 \vdash t_1 : \tau_3 \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau_3\end{array}}{\vdash \mathit{case}\ \mathfrak{C}\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t_1 \mid \mathit{inr}\ x_2 \mapsto t_2 : \Gamma', \tau' \to \Gamma, \tau_3}$$

(Type-Ctx-Case2)
$$\frac{\Gamma \vdash t : \tau_1 \uplus \tau_2 \qquad \vdash \mathfrak{C} : \Gamma', \tau' \to (\Gamma, x_1 : \tau_1), \tau_3 \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau_3}{\vdash \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto \mathfrak{C} \mid \mathit{inr}\ x_2 \mapsto t_2 : \Gamma', \tau' \to \Gamma, \tau_3}$$

(Type-Ctx-Case3)
$$\frac{\Gamma \vdash t : \tau_1 \uplus \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau_3 \qquad \vdash \mathfrak{C} : \Gamma', \tau' \to (\Gamma, x_2 : \tau_2), \tau_3}{\vdash \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t_1 \mid \mathit{inr}\ x_2 \mapsto \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_3}$$

(Type-Ctx-If1)
$$\frac{\begin{array}{c}\vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \mathit{Bool} \\ \Gamma' \vdash t_1 : \tau' \qquad \Gamma' \vdash t_2 : \tau'\end{array}}{\vdash \mathit{if}\ \mathfrak{C}\ \mathit{then}\ t_1\ \mathit{else}\ t_2 : \Gamma, \tau \to \Gamma', \tau'}$$

(Type-Ctx-If2)
$$\frac{\begin{array}{c}\Gamma \vdash t : \mathit{Bool} \\ \vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau' \qquad \Gamma \vdash t_2 : \tau'\end{array}}{\vdash \mathit{if}\ t\ \mathit{then}\ \mathfrak{C}\ \mathit{else}\ t_2 : \Gamma, \tau \to \Gamma', \tau'}$$

(Type-Ctx-If3)
$$\frac{\Gamma \vdash t : \mathit{Bool} \qquad \Gamma \vdash t_1 : \tau' \qquad \vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau'}{\vdash \mathit{if}\ t\ \mathit{then}\ t_1\ \mathit{else}\ \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau'}$$

(Type-Ctx-Seq1)
$$\frac{\begin{array}{c}\mathfrak{C} : \Gamma, \tau \to \Gamma', \mathit{Unit} \\ \Gamma' \vdash t : \tau''\end{array}}{\vdash \mathfrak{C}; t : \Gamma, \tau \to \Gamma', \tau''}$$

(Type-Ctx-Seq2)
$$\frac{\begin{array}{c}\Gamma \vdash t : \mathit{Unit} \\ \vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau'\end{array}}{\vdash t; \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau'}$$

(λ^{fx}-Type-Ctx-Fix)
$$\frac{\mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau \to \tau}{\vdash \mathsf{fix}_{\tau \to \tau}\ \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau}$$

($\lambda^{\mu}_{\mathrm{I}}$-Type-Ctx-Fold)
$$\frac{\vdash \boldsymbol{\mathfrak{C}} : \boldsymbol{\Gamma', \tau' \to \Gamma, \tau[\mu\alpha.\,\tau/\alpha]}}{\vdash \mathbf{fold}_{\mu\alpha.\tau}\ \boldsymbol{\mathfrak{C}} : \boldsymbol{\Gamma', \tau' \to \Gamma, \mu\alpha.\,\tau}}$$

($\lambda^{\mu}_{\mathrm{I}}$-Type-Ctx-Unfold)
$$\frac{\vdash \boldsymbol{\mathfrak{C}} : \boldsymbol{\Gamma', \tau' \to \Gamma, \mu\alpha.\,\tau}}{\vdash \mathbf{unfold}_{\mu\alpha.\tau} \boldsymbol{\mathfrak{C}} : \boldsymbol{\Gamma', \tau' \to \Gamma, \tau[\mu\alpha.\,\tau/\alpha]}}$$

(λ^{μ}_{E}-Type-Eq)
$$\frac{\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau \qquad \tau \doteq \sigma}{\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \sigma}$$

We use the same coinductive type equality relation of Cai et al. [2016], with a cosmetic difference only. Two types are equal if they are the same base type ι or variable (Rules $\doteq$-prim and $\doteq$-var). If the types are composed of two types, the connectors must be the same and each sub-type must be equivalent (Rule $\doteq$-bin). If the left type starts with a μ (or if that does not but the right one does), then we unfold the type for checking the equality (Rules $\doteq$-μ_l and $\doteq$-μ_r). Note that these last two rules are defined in an asymmetric fashion to make equality derivation deterministic. Finally, we make explicit the rules for reflexivity, symmetry and transitivity (Rule $\doteq$-refl, Rules $\doteq$-symm and $\doteq$-trans) whose derivations we have proved from the other rules.

$\boxed{\tau \doteq \tau'}$

$$\frac{\iota = \mathit{Unit} \;\vee\; \iota = \mathit{Bool}}{\iota \doteq \iota}\;(\doteq\text{-prim}) \qquad \frac{}{\alpha \doteq \alpha}\;(\doteq\text{-var}) \qquad \frac{\star \in \{\rightarrow, \times, \uplus\} \quad \tau_1 \doteq \sigma_1 \quad \tau_2 \doteq \sigma_2}{\tau_1 \star \tau_2 \doteq \sigma_1 \star \sigma_2}\;(\doteq\text{-bin})$$

$$\frac{\tau[\mu\alpha.\,\tau/\alpha] \doteq \sigma}{\mu\alpha.\,\tau \doteq \sigma}\;(\doteq\text{-}\mu_l) \qquad \frac{\tau \doteq \sigma[\mu\alpha.\,\sigma/\alpha]}{\tau \doteq \mu\alpha.\,\sigma}\;(\doteq\text{-}\mu_r) \qquad \frac{}{\tau \doteq \tau}\;(\doteq\text{-refl}) \qquad \frac{\sigma \doteq \tau}{\tau \doteq \sigma}\;(\doteq\text{-symm}) \qquad \frac{\tau \doteq \sigma \quad \sigma \doteq \tau'}{\tau \doteq \tau'}\;(\doteq\text{-trans})$$

To prove results about this equality relation, we will often induct on the "leading-mu-count" (lmc) measure. Intuitively, that measure counts the amount of μs that a λ^{μ}_{E} type has before a different connector is found. This is almost the same as the number of times a type can be unfolded before it is no longer recursive at the top level (e.g. $\texttt{lmc}\,(\mathit{Unit}) = 0$, $\texttt{lmc}\,(\mu\alpha.\,\alpha \uplus \mathit{Unit}) = 1$).

$$\texttt{lmc}\,(\tau) \overset{\mathsf{def}}{=} \begin{cases} \texttt{lmc}\,(\tau') + 1 & \tau = \mu\alpha.\,\tau' \\ 0 & \text{otherwise} \end{cases}$$

Non-contractive types such as $\mu\alpha.\,\alpha$, however, create problems here, for they always unfold into another top level recursive type. This motivates our restriction to contractive types only: a contractive type τ can be unfolded exactly $\texttt{lmc}\,(\tau)$ times.

2.3. **Dynamic Semantics.** All our languages are given a small-step, contextual, call-by-value, operational semantics. We highlight primitive reductions as $\hookrightarrow_p$ and non-primitive ones as $\hookrightarrow$. We indicate the capture-avoiding substitution of variable (or type variable) x in t with value (or type) v as $t[v/x]$. Note that since λ^{μ}_{E} has no peculiar syntactic construct, it also has no specific reduction rule.

$$\boxed{t \hookrightarrow t' \quad \text{and} \quad t \hookrightarrow_p t'}$$

$$\frac{t \hookrightarrow_p t'}{\mathbb{E}\,[t] \hookrightarrow \mathbb{E}\,[t']}\;(\text{Eval-ctx}) \qquad \frac{}{(\lambda x : \tau.\, t)\ v \hookrightarrow_p t[v/x]}\;(\text{Eval-beta}) \qquad \frac{i \in 1..2}{\langle v_1, v_2\rangle .i \hookrightarrow_p v_i}\;(\text{Eval-pi})$$

$$\frac{}{\mathit{unit}; t \hookrightarrow_p t}\;(\text{Eval-seq}) \qquad \frac{}{\mathit{case}\ \mathit{inl}\ v\ \mathit{of} \left| \begin{array}{l} \mathit{inl}\ x_1 \mapsto t \\ \mathit{inr}\ x_2 \mapsto t' \end{array} \right. \hookrightarrow_p t[v/x_1]}\;(\text{Eval-inl})$$

$$\frac{}{\mathit{case}\ \mathit{inr}\ v\ \mathit{of} \left| \begin{array}{l} \mathit{inl}\ x_1 \mapsto t \\ \mathit{inr}\ x_2 \mapsto t' \end{array} \right. \hookrightarrow_p t'[v/x_2]}\;(\text{Eval-inr}) \qquad \frac{v = \mathit{true} \vee \mathit{false}}{\mathit{if}\ v\ \mathit{then}\ t_{\mathit{true}}\ \mathit{else}\ t_{\mathit{false}} \hookrightarrow_p t_v}\;(\text{Eval-if})$$

$$\frac{}{\mathsf{fix}_{\tau\to\tau}\ (\lambda \mathsf{x} : \tau.\, \mathsf{t}) \hookrightarrow_{\mathsf{p}} \mathsf{t}\,[\mathsf{fix}_{\tau\to\tau}\ (\lambda \mathsf{x} : \tau.\, \mathsf{t})/\mathsf{x}]}\;(\lambda^{\mathsf{fx}}\text{-Eval-fix}) \qquad \frac{}{\mathbf{unfold}_{\mu\alpha.\tau}\ (\mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v}) \hookrightarrow_{\mathbf{p}} \mathbf{v}}\;(\lambda^{\mu}_{\mathbf{I}}\text{-Eval-fold})$$

2.4. **Notions of Termination.** For technical reasons, we need to define two notions of termination for our languages. To define contextual equivalence (which is required for fully-abstract compilation), we rely on the canonical definition of termination, which tells that a term eventually reduces to a value in some number of steps.

Definition 1 (Termination)**.**

$$t\Downarrow \overset{\mathsf{def}}{=} \exists n \in \mathbb{N}, v.\, t\Downarrow_n v$$

We rely on the auxiliary judgement for bounded termination in order to say that a term t reduces to a value v in n steps.

$$\text{(Bounded termination-value)}\ \frac{}{v\Downarrow_0 v} \qquad \text{(Bounded termination-term)}\ \frac{t \hookrightarrow t' \quad t'\Downarrow_n v}{t\Downarrow_{n+1} v}$$

As mentioned in Section 1, to make the logical relation more precise, we need another notion of bounded termination that not only bounds the number of steps needed for reaching a value but also the size of intermediate terms encountered during these steps.

$$\text{(size-bound termination-value)}\ \frac{\mathtt{size}\,(v) \leq n}{v\lightning_n v} \qquad \text{(size-bound termination-term)}\ \frac{t \hookrightarrow t' \quad t'\lightning_n v \quad \mathtt{size}\,(t) \leq n}{t\lightning_{n+1} v}$$

It is worth noting that this definition does not apply the same size bound to all terms encountered during execution, but the bound decreases as execution progresses. This approach has minor technical benefits in the definition, but we think a definition with a single bound on all terms would work as well.

The two termination notions are related by Theorem 1 below. For any term t that terminates there exists a n such that size-bound termination holds for t in n steps. Conversely, if size-bound termination holds for a term then it also terminates.

Theorem 1 (Relation between Termination and Size-Bound Termination)**.**

$$\text{if } t\Downarrow \text{ then } \exists n \in \mathbb{N}, v.\, t\lightning_n v$$
$$\text{if } t\lightning_{\_} \text{ then } t\Downarrow$$

Although this theorem is quite easy to prove, it does capture a non-trivial property of the programming language, namely the fact that it only contains finite values. If we would define a variant of the language with infinite values (e.g. if we had interpreted μ as producing a coinductive fixpoint rather than an inductive one, perhaps with a call-by-need semantics), then the property would no longer hold.

2.5. **Logical Relations Between Our Languages.** As mentioned in Section 1, we need cross-language relations that indicate when related source and target terms approximate each other. Intuitively, one such relation is needed by each one of the compilers we define later. Thus, we need to define three logical relations:

A one between λ^{fx} and $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$, which we dub $LR^{\mathsf{fx}}_{\boldsymbol{\mu}\mathbf{I}}$;

B one between $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E}, which we dub $LR^{\boldsymbol{\mu}\mathbf{I}}_{\mu E}$;

C one between λ^{fx} and λ^{μ}_{E}, which we dub $LR^{\mathsf{fx}}_{\mu E}$.

These relations are all indexed by a step and then by the source type, so logical relations (A) and (C) look the same. For brevity we present only one of them. Additionally, given that $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ has the same types of λ^{fx} plus recursive types, we only show that case for logical relation

(B). Ours are Kripke, step-indexed logical relations that are based on those of Devriese et al. [2017]; Hur and Dreyer [2011]. The step-indexing is not inherently needed for relations (A) and (C), which could be defined just by induction on λ^{fx} types (since they do not include recursive types). However, all of our relations are step-indexed anyway because the steps also determine for how many steps one term should approximate the other and this detail is key for the backtranslation proof technique.

Before presenting the details, note that the relations we show here are *not* complete. Specifically they only talk about the terms needed to conclude reflection of fully-abstract compilation but not preservation (admittedly, the most interesting part). Completing the logical relations relies on technical insights regarding the backtranslations, so we do this later in Section 3.3. The goal of this section is to provide an understanding of what it means for two terms to approximate each other.

$$W \overset{\mathsf{def}}{=} n \in \mathbb{N} \quad lev(n) = n \quad \triangleright(0) = 0 \quad \triangleright(n+1) = n$$

$$W \sqsupseteq W' = lev(W) \leq lev(W') \quad W \sqsupset_{\triangleright} W' = lev(W) < lev(W')$$

$$O(W)_{\lesssim} \overset{\mathsf{def}}{=} \{(\mathsf{t}, \mathbf{t}) \mid \text{if } lev(W) > n \text{ and } \mathsf{t}\lightning_{n}\mathsf{v} \text{ then } \exists \mathbf{k}, \mathbf{v}.\ \mathbf{t}\Downarrow_{k}\mathbf{v}\}$$

$$O(W)_{\gtrsim} \overset{\mathsf{def}}{=} \{(\mathsf{t}, \mathbf{t}) \mid \text{if } lev(W) > n \text{ and } \mathbf{t}\lightning_{n}\mathbf{v} \text{ then } \exists \mathsf{k}, \mathsf{v}.\ \mathsf{t}\Downarrow_{\mathsf{k}}\mathsf{v}\}$$

$$O(W)_{\approx} \overset{\mathsf{def}}{=} O(W)_{\lesssim} \cap O(W)_{\gtrsim}$$

FIGURE 2. Worlds, observations and related technicalities. These are typeset for the relation between λ^{fx} and $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ but the other ones do not change.

All three relations rely on the same notion of very simple Kripke worlds W (Fig. 2). Worlds consist of just a step-index k that is accessed via function $lev(W)$. The use of this function is intended to facilitate future extensions of the Kripke worlds with additional information, but we do not currently make use of this extra generality. The $\triangleright$ modality and future world relation $\sqsupseteq$ express that future worlds allow programs to take fewer reduction steps. We define two different observation relations, one for each direction of the approximations we are interested in: $O(W)_{\lesssim}$ and $O(W)_{\gtrsim}$ while $O(W)_{\approx}$ indicates the intersection of those approximations. The former defines that a source term approximates a target term if shrinking of the first in $lev(W)$ steps or less implies termination of the second (in any number of steps). The latter requires the reverse. All of our logical relations will be defined in terms of either $O(W)_{\lesssim}$ or $O(W)_{\gtrsim}$. For definitions and lemmas or theorems that apply for both instantiations, we use the symbol $\triangledown$ as a metavariable that can be instantiated to either $\lesssim$ or $\gtrsim$.

Note that our logical relations are not indexed by source types, but by *pseudo-types* $\hat{\tau}$. Pseudo-types contain all the constructs of source types, plus an additional type which we indicate for now as *EmulT*. This type is not a source type; it is needed because of the approximate backtranslation, so we defer explaining its details until Section 3.3. Function $\texttt{repEmul}^{\mathsf{f}\mathbf{I}}(\cdot)$ converts a pseudo-type to an actual source type by replacing all occurrences of *EmulT* with a concrete source type.[1] We will sometimes silently use a normal source type where a pseudo-type is expected; this makes sense since the syntax for the latter is

[1]As a convention, superscripts of these auxiliary functions indicate the initials of the two languages involved.

a superset of the former. Function $\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\cdot)$ converts a λ^{fx} pseudo-type into its $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ correspondent; this is needed because unlike the previous work of Devriese et al. [2017], all of our target languages are typed. The formal details of both these functions are deferred until *EmulT* is defined (Section 3.3) but we report their types below for clarity. Finally, function $\mathtt{oftype}^{\mathsf{f}\mathbf{I}}(\cdot)$ checks that terms have the correct form according to the rules of syntactic typing (Section 2.2).

$$\hat{\tau} ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \hat{\tau} \to \hat{\tau} \mid \hat{\tau} \times \hat{\tau} \mid \hat{\tau} \uplus \hat{\tau} \mid \mathit{EmulT} \text{ (to be defined in Section 3.3)}$$

$$\mathtt{oftype}^{\mathsf{f}\mathbf{I}}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \left\{(\mathsf{v}, \mathbf{v}) \;\middle|\; \mathsf{v} \in \mathsf{oftype}\left(\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}(\hat{\tau})\right) \text{ and } \mathbf{v} \in \mathbf{oftype}\left(\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\hat{\tau})\right)\right\}$$

$$\mathsf{oftype}(\tau) \stackrel{\mathsf{def}}{=} \{\mathsf{v} \mid \emptyset \vdash \mathsf{v} : \tau\} \qquad \mathbf{oftype}(\boldsymbol{\tau}) \stackrel{\mathsf{def}}{=} \{\mathbf{v} \mid \boldsymbol{\emptyset} \vdash \mathbf{v} : \boldsymbol{\tau}\}$$

$$\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}(\cdot) : \hat{\tau} \to \tau \text{ (see Section 3.3)} \quad \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\cdot) : \hat{\tau} \to \boldsymbol{\tau} \text{ (see Section 3.3)}$$

These definitions are used in the $LR^{\mathsf{fx}}_{\boldsymbol{\mu}\mathbf{I}}$ relation and similar ones are used in the other ones, so we report their definitions and signatures below. Function $\mathtt{oftype}^{\mathbf{I}E}(\cdot)$ does the analogous syntactic typecheck but for terms of $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E} and $\mathtt{oftype}^{\mathsf{f}E}(\cdot)$ does it for terms of λ^{fx} and λ^{μ}_{E}. Functions $\mathtt{repEmul}^{\mathsf{f}E}(\cdot)$ and $\mathtt{repEmul}^{\mathbf{I}E}(\cdot)$ do the analogous conversion from pseudo types to actual types. Function $\mathtt{convTy}^{\mathsf{f}\to E}(\cdot)$ and $\mathtt{convTy}^{\mathbf{I}\to E}(\cdot)$ do the analogous conversion from source pseudo types to target actual types. As we clarify later, *EmulT* is indexed by target types, so essentially we have a set of pseudo types for the λ^{fx} to $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ compilation and a different set for the λ^{fx} to λ^{μ}_{E} compilation, and thus we need two different conversion functions (whose signatures look the same for now).

$$\mathtt{oftype}^{\mathbf{I}E}(\hat{\boldsymbol{\tau}}) \stackrel{\mathsf{def}}{=} \left\{(\mathbf{v}, v) \mid \mathbf{v} \in \mathbf{oftype}\left(\mathtt{repEmul}^{\mathbf{I}E}(\hat{\boldsymbol{\tau}})\right) \text{ and } v \in \mathit{oftype}\left(\mathtt{convTy}^{\mathbf{I}\to E}(\hat{\boldsymbol{\tau}})\right)\right\}$$

$$\mathtt{oftype}^{\mathsf{f}E}(\hat{\tau}) \stackrel{\mathsf{def}}{=} \left\{(\mathsf{v}, v) \;\middle|\; \mathsf{v} \in \mathsf{oftype}\left(\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}(\hat{\tau})\right) \text{ and } v \in \mathit{oftype}\left(\mathtt{convTy}^{\mathsf{f}\to E}(\hat{\tau})\right)\right\}$$

$$\mathit{oftype}(\tau) \stackrel{\mathsf{def}}{=} \{v \mid \emptyset \vdash v : \tau\}$$

$$\mathtt{repEmul}^{\mathsf{f}E}(\cdot) : \hat{\tau} \to \tau \qquad \mathtt{repEmul}^{\mathbf{I}E}(\cdot) : \hat{\boldsymbol{\tau}} \to \boldsymbol{\tau} \qquad \text{(see Section 3.3)}$$

$$\mathtt{convTy}^{\mathsf{f}\to E}(\cdot) : \hat{\tau} \to \tau \qquad \mathtt{convTy}^{\mathbf{I}\to E}(\cdot) : \hat{\boldsymbol{\tau}} \to \tau \qquad \text{(see Section 3.3)}$$

The value relation $\mathcal{V}[\![\hat{\tau}]\!]_{\triangledown}$ (Figure 3) is defined inductively on source pseudo-types and it is quite standard save for an additional premise in the value relation for function types. *Unit* and *Bool* values are related in any world so long as they are the same value. Function values are related if they are well-typed, if both are lambdas, and if substituting related values in the bodies yields related terms in any strictly-future world. Additionally, when the approximation direction is $\gtrsim$, we require that the world W' contains enough steps to bound the size of the target argument $\mathbf{v}'$. This is a technicality that is required to complete the proof of Lemma 13, as we explain at the end of Section 4.2.2. Pair values are related if both are pairs and each projection is related in strictly-future worlds and sum values are related if they have the same tag (*inl* or *inr*) and the tagged values are related in strictly-future worlds. Finally, the value relation for recursive types used by $LR^{\boldsymbol{\mu}\mathbf{I}}_{\mu E}$ is not defined on strictly-future worlds because in an equi-recursive language, values of recursive type can be inspected without consuming a step. However, this does not compromise well-foundedness of the relation because our recursive types $\mu\alpha.\,\tau$ are contractive, so the recursion variable α in τ must occur under a type constructor such as $\to$ and the relation for these constructors recurses only at strictly-future worlds.

$$\triangleright R \overset{\mathsf{def}}{=} \{(W, \mathsf{v}, \mathbf{v}) \mid \text{if } \mathit{lev}(W) > 0 \text{ then } (\triangleright(W), \mathsf{v}, \mathbf{v}) \in R\}$$

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

$$\mathcal{V}[\![\mathsf{Bool}]\!]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \mathsf{v}, \mathbf{v}) \mid (\mathsf{v} = \mathsf{true} \text{ and } \mathbf{v} = \mathbf{true}) \text{ or } (\mathsf{v} = \mathsf{false} \text{ and } \mathbf{v} = \mathbf{false})\}$$

$$\mathcal{V}\left[\!\!\left[\hat{\tau} \to \hat{\tau}'\right]\!\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \left\{ (W, \mathsf{v}, \mathbf{v}) \;\middle|\; \begin{array}{l} (\mathsf{v}, \mathbf{v}) \in \mathtt{oftype}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau} \to \hat{\tau}'\right) \text{ and} \\ \exists \mathsf{t}, \mathbf{t}.\ \mathsf{v} = \lambda \mathsf{x} : \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}(\hat{\tau}).\,\mathsf{t}, \mathbf{v} = \boldsymbol{\lambda} \mathbf{x} : \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\hat{\tau}).\,\mathbf{t} \text{ and} \\ \forall W', \mathsf{v}', \mathbf{v}'.\ \text{if } W' \sqsupseteq_{\triangleright} W \text{ and } (W', \mathsf{v}', \mathbf{v}') \in \mathcal{V}[\![\hat{\tau}]\!]_{\triangledown} \text{ and} \\ \ (\text{if } \triangledown = \gtrsim \text{ then } \mathtt{size}(\mathbf{v}') \leq \mathit{lev}(W')) \text{ then } (W', \mathsf{t}[\mathsf{v}'/\mathsf{x}], \mathbf{t}[\mathbf{v}'/\mathbf{x}]) \in \mathcal{E}\left[\!\!\left[\hat{\tau}'\right]\!\!\right]_{\triangledown} \end{array} \right\}$$

$$\mathcal{V}\left[\!\!\left[\hat{\tau} \times \hat{\tau}'\right]\!\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \left\{ (W, \mathsf{v}, \mathbf{v}) \;\middle|\; \begin{array}{l} (\mathsf{v}, \mathbf{v}) \in \mathtt{oftype}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau} \times \hat{\tau}'\right) \text{ and} \\ \exists \mathsf{v}_1, \mathsf{v}_2, \mathbf{v_1}, \mathbf{v_2}.\ \mathsf{v} = \langle \mathsf{v}_1, \mathsf{v}_2 \rangle, \mathbf{v} = \langle \mathbf{v_1}, \mathbf{v_2} \rangle \text{ and} \\ (W, \mathsf{v}_1, \mathbf{v_1}) \in \triangleright \mathcal{V}[\![\hat{\tau}]\!]_{\triangledown} \text{ and } (W, \mathsf{v}_2, \mathbf{v_2}) \in \triangleright \mathcal{V}\left[\!\!\left[\hat{\tau}'\right]\!\!\right]_{\triangledown} \end{array} \right\}$$

$$\mathcal{V}\left[\!\!\left[\hat{\tau} \uplus \hat{\tau}'\right]\!\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \left\{ (W, \mathsf{v}, \mathbf{v}) \;\middle|\; \begin{array}{l} (\mathsf{v}, \mathbf{v}) \in \mathtt{oftype}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau} \uplus \hat{\tau}'\right) \text{ and either} \\ \exists \mathsf{v}', \mathbf{v}'.\ (W, \mathsf{v}', \mathbf{v}') \in \triangleright \mathcal{V}[\![\hat{\tau}]\!]_{\triangledown} \text{ and } \mathsf{v} = \mathsf{inl}\ \mathsf{v}', \mathbf{v} = \mathbf{inl}\ \mathbf{v}' \text{ or} \\ \exists \mathsf{v}', \mathbf{v}'.\ (W, \mathsf{v}', \mathbf{v}') \in \triangleright \mathcal{V}\left[\!\!\left[\hat{\tau}'\right]\!\!\right]_{\triangledown} \text{ and } \mathsf{v} = \mathsf{inr}\ \mathsf{v}', \mathbf{v} = \mathbf{inr}\ \mathbf{v}' \end{array} \right\}$$

$$\mathcal{V}[\![\mathit{EmulT}]\!]_{\triangledown} \overset{\mathsf{def}}{=} \text{to be defined in Section 3.3}$$

$$\mathcal{K}[\![\hat{\tau}]\!]_{\triangledown} \overset{\mathsf{def}}{=} \left\{ (W, \mathbb{E}, \mathbb{E}) \;\middle|\; \begin{array}{l} \forall W', \mathsf{v}, \mathbf{v}.\ \text{if } W' \sqsupseteq W \text{ and } (W', \mathsf{v}, \mathbf{v}) \in \mathcal{V}[\![\hat{\tau}]\!]_{\triangledown} \text{ then} \\ (\mathbb{E}[\mathsf{v}], \mathbb{E}[\mathbf{v}]) \in O(W')_{\triangledown} \end{array} \right\}$$

$$\mathcal{E}[\![\hat{\tau}]\!]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \mathsf{t}, \mathbf{t}) \mid \forall \mathbb{E}, \mathbb{E}.\ \text{if } (W, \mathbb{E}, \mathbb{E}) \in \mathcal{K}[\![\hat{\tau}]\!]_{\triangledown} \text{ then } (\mathbb{E}[\mathsf{t}], \mathbb{E}[\mathbf{t}]) \in O(W)_{\triangledown}\}$$

$$\mathcal{G}[\![\emptyset]\!]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \emptyset, \boldsymbol{\emptyset})\}$$

$$\mathcal{G}\left[\!\!\left[\hat{\Gamma}, \mathsf{x} : \hat{\tau}\right]\!\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \left\{(W, \gamma[\mathsf{v}/\mathsf{x}], \boldsymbol{\gamma}[\mathbf{v}/\mathbf{x}]) \;\middle|\; (W, \gamma, \boldsymbol{\gamma}) \in \mathcal{G}\left[\!\!\left[\hat{\Gamma}\right]\!\!\right]_{\triangledown} \text{ and } (W, \mathsf{v}, \mathbf{v}) \in \mathcal{V}[\![\hat{\tau}]\!]_{\triangledown}\right\}$$

$$\mathcal{V}[\![\boldsymbol{\mu\hat{\alpha}.\,\tau}]\!]_{\triangledown} \overset{\mathsf{def}}{=} \left\{ (W, \mathbf{v}, v) \;\middle|\; \begin{array}{l} (\mathbf{v}, v) \in \mathtt{oftype}^{\mathbf{I}E}(\boldsymbol{\mu\hat{\alpha}.\,\tau}) \text{ and} \\ \exists \mathbf{v}'.\ (W, \mathbf{v}', v) \in \mathcal{V}\left[\!\!\left[\boldsymbol{\tau}[\boldsymbol{\mu\hat{\alpha}.\,\tau}/\boldsymbol{\alpha}]\right]\!\!\right]_{\triangledown} \text{ and } \mathbf{v} = \mathbf{fold}_{\boldsymbol{\mu\alpha.\tau}}\ \mathbf{v}' \end{array} \right\}$$

The rest of $\mathcal{V}[\![\hat{\boldsymbol{\tau}}]\!]_{\triangledown}$ is analogous to the cases presented for $\mathcal{V}[\![\hat{\tau}]\!]_{\triangledown}$

The $\mathcal{K}[\![\hat{\boldsymbol{\tau}}]\!]_{\triangledown}$, $\mathcal{E}[\![\hat{\boldsymbol{\tau}}]\!]_{\triangledown}$, and $\mathcal{G}\left[\!\!\left[\hat{\boldsymbol{\Gamma}}\right]\!\!\right]_{\triangledown}$ relations are analogous to the presented ones

---

The $\mathcal{V}[\![\hat{\tau}]\!]_{\triangledown}$, $\mathcal{K}[\![\hat{\tau}]\!]_{\triangledown}$, $\mathcal{E}[\![\hat{\tau}]\!]_{\triangledown}$, and $\mathcal{G}\left[\!\!\left[\hat{\Gamma}\right]\!\!\right]_{\triangledown}$ relations for $LR^{\mathsf{fx}}_{\mu E}$ are analogous to the presented ones

FIGURE 3. Part of the three cross-language logical relations we rely on (classical bits) and its auxiliary functions.

The value, evaluation context and term relations are defined by mutual recursion, using a technique called biorthogonality (see, e.g., [Benton and Hur, 2009]). Evaluation contexts $\mathcal{K}[\![\hat{\tau}]\!]_{\triangledown}$ are related in a world if plugging in related values in any future world yields terms that are related according to the observation relation of the world. Similarly, terms are related

$\mathcal{E}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown}$ if plugging the terms in related evaluation contexts yields terms related according to the observation relation of the world. Relation $\mathcal{G}\left[\!\left[\hat{\Gamma}\right]\!\right]_{\triangledown}$ relates substitutions; this simply requires that substitutions for all variables in the context are for related values.

We indicate open terms to be logically related according to the three relations as follows (Definition 3, Definitions 4 and 5). Those definitions rely on terms being related up to n steps (Definition 2) which we present for $LR^{\mathsf{fx}}_{\mu\mathbf{I}}$ only since the other definitions are analogous. Here, when we apply $\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\cdot)$ to typing contexts, we mean the application of $\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\cdot)$ to all bindings in the context.

Definition 2 (Logical relation up to n steps for $LR^{\mathsf{fx}}_{\mu\mathbf{I}}$)**.**

$$\begin{aligned} \hat{\Gamma} \vdash \mathsf{t} \triangledown_n \mathbf{t} : \hat{\tau} \overset{\mathsf{def}}{=}\ & \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\Gamma}\right) \vdash \mathsf{t} : \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}(\hat{\tau}) \\ \text{and}\ \ & \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\Gamma}\right) \vdash \mathbf{t} : \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\hat{\tau}) \\ \text{and}\ \ & \forall W. \\ \text{if}\ \ & \mathit{lev}(W) \leq n \\ \text{then}\ \ & \forall \gamma, \boldsymbol{\gamma}.\ (W, \gamma, \boldsymbol{\gamma}) \in \mathcal{G}\left[\!\left[\hat{\Gamma}\right]\!\right]_{\triangledown}, \\ & (W, \mathsf{t}\gamma, \mathbf{t}\boldsymbol{\gamma}) \in \mathcal{E}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \end{aligned}$$

Definition 3 ($LR^{\mathsf{fx}}_{\mu\mathbf{I}}$ Logical relation)**.**

$$\hat{\Gamma} \vdash \mathsf{t} \triangledown \mathbf{t} : \hat{\tau} \overset{\mathsf{def}}{=} \forall n \in \mathbb{N}.\ \hat{\Gamma} \vdash \mathsf{t} \triangledown_n \mathbf{t} : \hat{\tau}$$

Definition 4 ($LR^{\mu\mathbf{I}}_{\mu E}$ Logical relation)**.**

$$\hat{\mathbf{\Gamma}} \vdash \mathbf{t} \triangledown t : \hat{\boldsymbol{\tau}} \overset{\mathsf{def}}{=} \forall n \in \mathbb{N}.\ \hat{\mathbf{\Gamma}} \vdash \mathbf{t} \triangledown_n t : \hat{\boldsymbol{\tau}}$$

Definition 5 ($LR^{\mathsf{fx}}_{\mu E}$ Logical relation)**.**

$$\hat{\Gamma} \vdash \mathsf{t} \triangledown t : \hat{\tau} \overset{\mathsf{def}}{=} \forall n \in \mathbb{N}.\ \hat{\Gamma} \vdash \mathsf{t} \triangledown_n t : \hat{\tau}$$

An open source term is related up to n steps at pseudo-type $\hat{\tau}$ in pseudo-context $\hat{\Gamma}$ to a target open term if both are well-typed and closing both terms with substitutions related in $\hat{\Gamma}$ produces terms related at $\hat{\tau}$ in any world that has at least n steps. If terms are related for any number of steps, we simply omit the n index and write $\hat{\Gamma} \vdash \mathsf{t} \triangledown \mathbf{t} : \hat{\tau}$. Since we have to also relate program contexts across languages, we define what it means for them to be related as follows.

Definition 6 ($LR^{\mathsf{fx}}_{\mu\mathbf{I}}$ Logical relation for program contexts)**.**

$$\begin{aligned} \vdash \mathfrak{C} \triangledown \boldsymbol{\mathfrak{C}} : \hat{\Gamma}, \hat{\tau} \to \hat{\Gamma}', \hat{\tau}' \overset{\mathsf{def}}{=}\ & \vdash \mathfrak{C} : \hat{\Gamma}, \hat{\tau} \to \hat{\Gamma}', \hat{\tau}' \\ \text{and}\ \ & \vdash \boldsymbol{\mathfrak{C}} : \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\Gamma}\right), \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\hat{\tau}) \to \\ & \qquad \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\Gamma}'\right), \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}'\right) \\ \text{and}\ \ & \forall \mathsf{t}, \mathbf{t} \\ \text{if}\ \ & \hat{\Gamma} \vdash \mathsf{t} \triangledown \mathbf{t} : \hat{\tau} \\ \text{then}\ \ & \hat{\Gamma}' \vdash \mathfrak{C}[\mathsf{t}] \triangledown \boldsymbol{\mathfrak{C}}[\mathbf{t}] : \hat{\tau}' \end{aligned}$$

Definition 7 ($LR^{\mu\mathbf{I}}_{\mu E}$ Logical relation for program contexts)**.**

$$\begin{aligned}
\vdash \mathfrak{C} \triangledown \mathfrak{C} : \boldsymbol{\Gamma}, \boldsymbol{\tau} \to \boldsymbol{\Gamma}', \boldsymbol{\tau}' \overset{\mathsf{def}}{=}\ & \vdash \mathfrak{C} : \boldsymbol{\Gamma}, \boldsymbol{\tau} \to \boldsymbol{\Gamma}', \boldsymbol{\tau}' \\
\text{and}\ \ & \vdash \mathfrak{C} : \mathtt{convTy}^{\mathbf{I}\to E}\left(\hat{\boldsymbol{\Gamma}}\right), \mathtt{convTy}^{\mathbf{I}\to E}\left(\hat{\boldsymbol{\tau}}\right) \to \\
& \qquad \mathtt{convTy}^{\mathbf{I}\to E}\left(\hat{\boldsymbol{\Gamma}}'\right), \mathtt{convTy}^{\mathbf{I}\to E}\left(\hat{\boldsymbol{\tau}}'\right) \\
\text{and}\ \ & \forall \mathbf{t}, t \\
\text{if}\ \ & \hat{\boldsymbol{\Gamma}} \vdash \mathbf{t} \triangledown t : \hat{\boldsymbol{\tau}} \\
\text{then}\ \ & \hat{\boldsymbol{\Gamma}}' \vdash \mathfrak{C}[\mathbf{t}] \triangledown \mathfrak{C}[t] : \hat{\boldsymbol{\tau}}'
\end{aligned}$$

Definition 8 ($LR^{\mathsf{fx}}_{\mu E}$ Logical relation for program contexts)**.**

$$\begin{aligned}
\vdash \mathfrak{C} \triangledown \mathfrak{C} : \hat{\Gamma}, \hat{\tau} \to \hat{\Gamma}', \hat{\tau}' \overset{\mathsf{def}}{=}\ & \vdash \mathfrak{C} : \hat{\Gamma}, \hat{\tau} \to \hat{\Gamma}', \hat{\tau}' \\
\text{and}\ \ & \vdash \mathfrak{C} : \mathtt{convTy}^{\mathsf{f}\to E}\left(\hat{\Gamma}\right), \mathtt{convTy}^{\mathsf{f}\to E}\left(\hat{\tau}\right) \to \\
& \qquad \mathtt{convTy}^{\mathsf{f}\to E}\left(\hat{\Gamma}'\right), \mathtt{convTy}^{\mathsf{f}\to E}\left(\hat{\tau}'\right) \\
\text{and}\ \ & \forall \mathsf{t}, t \\
\text{if}\ \ & \hat{\Gamma} \vdash \mathsf{t} \triangledown t : \hat{\tau} \\
\text{then}\ \ & \hat{\Gamma}' \vdash \mathfrak{C}[\mathsf{t}] \triangledown \mathfrak{C}[t] : \hat{\tau}'
\end{aligned}$$

Program contexts are related if they are well-typed and if plugging terms related at the pseudo-type of the hole ($\hat{\tau}$) in each of them produces terms related at the pseudo-type of the result ($\hat{\tau}'$).

All our logical relations are constructed so that for related terms, termination of one term implies termination of the other according to the direction of the approximation ($\lesssim$ or $\gtrsim$) (Lemma 1).

Lemma 1 (Adequacy for $\approx$ for $LR^{\mathsf{fx}}_{\mu\mathbf{I}}$)**.**

$$\text{if } \emptyset \vdash \mathsf{t} \lesssim_n \mathbf{t} : \tau \text{ and } \mathsf{t} \not\downarrow_{\mathsf{m}} \mathsf{v} \text{ with } n \geq m \text{ then } \mathbf{t}\Downarrow$$
$$\text{if } \emptyset \vdash \mathsf{t} \gtrsim_n \mathbf{t} : \tau \text{ and } \mathbf{t} \not\downarrow_{\boldsymbol{m}} \mathbf{v} \text{ with } n \geq m \text{ then } \mathsf{t}\Downarrow$$

3. Fully-abstract compilation and Approximate Backtranslations

This section provides an overview of fully-abstract compilation and of the approximate backtranslation proof technique that we use (Section 3.1). The approximate backtranslation requires defining the backtranslation type, i.e., the type that represents backtranslated values (Section 3.2). This type provides the insights needed to complete the definitions of our logical relations and to understand how to reason about backtranslated terms cross-languages (Section 3.3).

3.1. A Primer on Fully-Abstract Compilation and Approximate Backtranslations.

A compiler is fully abstract if it preserves and reflects contextual equivalence between source and target language [Abadi, 1998]. Many compiler passes have been proven to satisfy this criterion [Ahmed and Blume, 2008, 2011; Devriese et al., 2017; Fournet et al., 2013; New et al., 2016; Patrignani et al., 2015; Skorstengaard et al., 2019; Van Strydonck et al., 2019], we refer the interested reader to the survey of Patrignani et al. [2019].

Two programs are contextually equivalent if they produce the same behaviour no matter the larger program (i.e., program context) they interact with [Plotkin, 1977]. As commonly done, we define "producing the same behaviour" as equi-termination (one terminates iff the other does). We use a complete formulation of contextual equivalence for typed programs, which enforces that contexts are well-typed and their types match that of the terms considered.

Definition 9 (Contextual Equivalence)**.**

$$\Gamma \vdash t_1 \simeq_{\text{ctx}} t_2 : \tau \overset{\mathsf{def}}{=} \Gamma \vdash t_1 : \tau \text{ and } \Gamma \vdash t_2 : \tau \text{ and}$$
$$\forall \mathfrak{C}.\, \mathfrak{C} : \Gamma, \tau \to \emptyset, \tau'.\, \mathfrak{C}[t_1]\Downarrow \iff \mathfrak{C}[t_2]\Downarrow$$

Quantifying over all contexts in Definition 9 ensures that contextually-equivalent terms do not just equi-terminate, but that any value the context can obtain from them is indistinguishable.

For a compiler $[\![\cdot]\!]$ from language L_{src} to L_{trg}, we define full abstraction as follows:

Definition 10 (Fully-abstract compilation)**.**

$$\vdash [\![\cdot]\!] : FA \overset{\mathsf{def}}{=} \forall \mathrm{t}_1, \mathrm{t}_2 \in L_{src}.\, \emptyset \vdash \mathrm{t}_1 \simeq_{\text{ctx}} \mathrm{t}_2 : \tau \iff \emptyset \vdash [\![\mathrm{t}_1]\!] \simeq_{\text{ctx}} [\![\mathrm{t}_2]\!] : [\![\tau]\!]$$

For simplicity, we instantiate Definition 10 for closed terms only (i.e., well-typed under empty environments). Opening the environment to a non-empty set of term variables is straightforward and therefore omitted [Devriese et al., 2017].

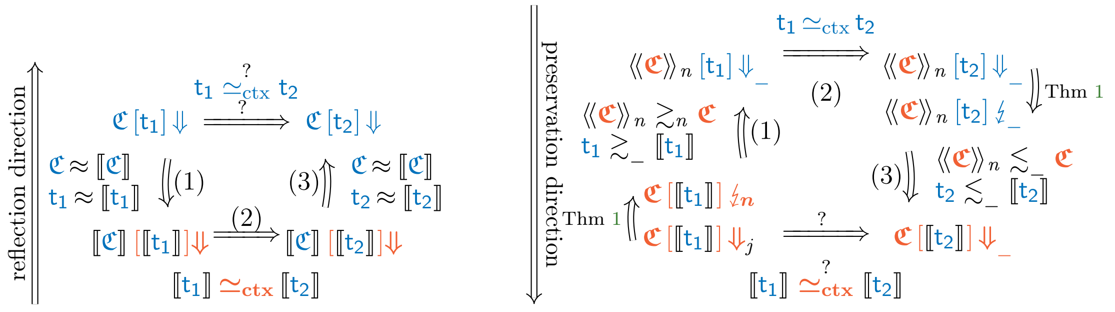


FIGURE 4. Diagram breakdown of the reflection (left) and preservation (right) proofs of fully-abstract compilation.

3.1.1. *Proving Fully-Abstract Compilation: Reflection (or, the Easy Part).*

The reflection part of fully-abstract compilation requires that the compiler produces equivalent target programs only if their source counterparts were equivalent. Contrapositively, inequivalent source programs must be compiled to inequivalent target programs. This proof can often be derived as a corollary of standard compiler correctness (i.e., refinement) [Patrignani et al., 2019].

As mentioned, we prove the reflection direction by relying on the cross-language logical relations. Our logical relations are compiler-agnostic—they simply state when terms approximate each other (recall that $\approx$ is the intersection of both approximations $\lesssim$ and $\gtrsim$). However, we use them to show that any term (and program context) is related to its compilation. With this fact, by relying on the adequacy of logical relations (Lemma 1), we know that related terms equi-terminate. Thus, we can apply the reasoning depicted in Figure 4 (left) to conclude this part of fully-abstract compilation.

3.1.2. *Proving Fully-Abstract Compilation: Preservation (or, the Hard Part).* Fully-abstract compilation proofs are notorious and their complexity resides in the *preservation* direction. That is, starting from contextually-equivalent programs in the source, prove that their compiled counterparts are contextually-equivalent in the target. For our three fully-abstract compilation results we rely on the approximate backtranslation proof technique [Devriese et al., 2017], depicted in Figure 4 (right).

We rely on both directions of the cross-language approximation relating terms for this proof. Recall that $\mathsf{t} \gtrsim_n \mathbf{t}$ is used to know that if $\mathbf{t}$ shrinks in n steps in the target, then t also terminates (in arbitrary steps) in the source. The converse, $\mathsf{t} \lesssim_n \mathbf{t}$ is used to know that if t shrinks in n steps in the source, then $\mathbf{t}$ also terminates (again in arbitrary steps) in the target. We start with source term t approximating (in both directions) its compilation $[\![\mathsf{t}]\!]$. Then, to prove target contextual equivalence (the ?-decorated equivalence), we start by assuming that a target context $\mathfrak{C}$ linked with $[\![\mathsf{t}_1]\!]$ terminates in some steps ($\Downarrow_n$). By relying on Theorem 1, we know that $\mathfrak{C}$ linked with $[\![\mathsf{t}_1]\!]$ size-bound terminates in some steps ($\lightning_{n'}$). Eventually, we need to show that the same target context linked with $[\![\mathsf{t}_2]\!]$ also terminates in any steps ($\Downarrow\_$). This is the ?-decorated implication, the reverse direction holds by symmetry. To progress, we construct a *backtranslation* $\langle\!\langle\cdot\rangle\!\rangle_n$, i.e., a function that takes a target context $\mathfrak{C}$ and returns a source context that approximates $\mathfrak{C}$ in both directions. With the backtranslation and this direction of the approximation $\gtrsim_n$, we prove implication (1): the backtranslated context $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ linked with t_1 terminates in the source. At this point, the assumption of source contextual equivalence yields implication (2): the same backtranslated context $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ linked with t_2 also terminates ($\Downarrow$). Here we apply again Theorem 1 to know that $\langle\!\langle\mathfrak{C}\rangle\!\rangle_n$ linked with t_2 size-bound terminates ($\lightning\_$). Now we rely on the another direction of the approximation between the target context and its backtranslation (as well as between source terms and their compilation): $\lesssim\_$. This other approximation lets us conclude implication (3): the original target context $\mathfrak{C}$ linked with $[\![\mathsf{t}_2]\!]$ terminates in the target. This is what we prove for a compiler to be fully abstract.

3.2. **A Family of Backtranslation Types.** Backtranslated contexts must be valid source contexts, i.e., they need to be well typed in the source. However, λ^{fx} does not have recursive types, so what is the source-level correspondent of $\mu\alpha.\,\tau$?

We adapt the same intuition of previous work [Devriese et al., 2016, 2017] in our setting too: it is not necessary to precisely embed target types into the source language in order to backtranslate terms. In fact, we need to reason for *up to n steps*, which means that we can approximate target types *n-levels deep*. Thus, concretely, we do not need recursive types in λ^{fx}. Given a target recursive type, we unfold it n times and backtranslate its unfolding to model the n target reductions required.

According to this strategy, the backtranslation of a term of type τ should have type *unfold τ n times*. During this unfolding, however, things can go wrong. Specifically, the

$$\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{0;\tau} \overset{\mathsf{def}}{=} \mathsf{Unit}$$

$$\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n+1;\tau} \overset{\mathsf{def}}{=} \begin{cases} \mathsf{Unit} \uplus \mathsf{Unit} & \text{if } \tau = \mathbf{Unit} \\ \mathsf{Bool} \uplus \mathsf{Unit} & \text{if } \tau = \mathbf{Bool} \\ (\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau} \to \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau'}) \uplus \mathsf{Unit} & \text{if } \tau = \tau \to \tau' \\ (\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau} \times \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau'}) \uplus \mathsf{Unit} & \text{if } \tau = \tau \times \tau' \\ (\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau} \uplus \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau'}) \uplus \mathsf{Unit} & \text{if } \tau = \tau \uplus \tau' \\ \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau'[\mu\alpha.\tau'/\alpha]} \uplus \mathsf{Unit} & \text{if } \tau = \mu\alpha.\,\tau' \end{cases}$$

$$\mathbf{BtT}^{\mathbf{I}E}_{n;\tau} \overset{\mathsf{def}}{=} \text{ as } \mathsf{BtT}^{\mathsf{f}E}_{n;\tau}$$

$$\mathsf{BtT}^{\mathsf{f}E}_{n+1;\tau} \overset{\mathsf{def}}{=} \begin{cases} \text{omitted cases are as above} \\ \mathsf{BtT}^{\mathsf{f}E}_{n+1;\tau'[\mu\alpha.\tau'/\alpha]} & \text{if } \tau = \mu\alpha.\,\tau' \end{cases}$$

FIGURE 5. The type of backtranslated terms.

backtranslated code does not know at runtime the level of unfolding we are dealing with, i.e., it cannot inspect n at runtime. Thus, we need a way to model the term reaching more than n unfoldings, because in that case the backtranslated code needs to diverge. Recall in fact that one of the two terms ($[\![t_1]\!]$ and $[\![t_2]\!]$) is guaranteed to terminate within n steps. Therefore, if that termination does not happen, the backtranslated code to diverge; this ensures that contextually-equivalent terms remain equivalent, i.e., they equi-terminate. Thus at each level of unfolding, we backtranslate τ into "$\tau \uplus \mathsf{Unit}$" (we will make this formal below), where the right Unit models failure. Then any time the backtranslation code receives a value which inhabits the 'right Unit' type of the backtranlation type, it will diverge, knowing that it is not dealing with the term that had to terminate within the n unfoldings.

We make these intuitions concrete and formalise the type for $\lambda^{\mu}_{\mathbf{I}}$ values backtranslated into λ^{fx} as $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau}$ in Figure 5 (for Backtranslation Type; the superscript indicates the languages involved, the subscripts are effectively parameters of this type). Type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau}$ is defined inductively on n and it backtranslates the structure of τ in the source type it creates. At no steps ($\mathsf{n}=0$), the backtranslation is not needed any more because intuitively we already performed the n steps, so the only type is Unit. Otherwise, the backtranslated type maintains the same structure of the target type. In the case for $\mu\alpha.\,\tau$, the backtranslated type is the unfolding of $\mu\alpha.\,\tau$, but at a decremented index (n). Intuitively, this is to match the reduction step that will happen in the target for eliminating $\mathbf{unfold}_{\mu\alpha.\tau}$ $\mathbf{fold}_{\mu\alpha.\tau}$ annotations.

The type of λ^{μ}_{E} terms backtranslated in λ^{fx} ($\mathsf{BtT}^{\mathsf{f}E}_{n;\tau}$, still in Figure 5) has an important difference. The case for $\mu\alpha.\,\tau$ does not lose a step in the index and simply performs the unfolding of the recursive type without an additional $\uplus\mathsf{Unit}$. This difference matches the fact that in λ^{μ}_{E} there is no additional reduction rule in the semantics. Additionally, this difference affects the helper functions needed to deal with values of backtranslation type, as we discuss later.

Intuitively, the fact that the backtranslation of a recursive type is its n-level deep unfolding is possible because $\mu\alpha.\,\tau$ is contractive in α. This is sufficient because we need to only replicate n steps in order to differentiate terms, so a n-level deep unfolding of the type suffices in order to reach the differentiation. For example, let us take the type of list of booleans in λ^{μ}_{E}:

$$\mu\alpha.\ Unit \uplus (Bool \times \alpha) \text{ (which we dub } List_B\text{)}$$

and its first unfolding:

$$Unit \uplus (Bool \times List_B) \text{ (which we dub } List^{1}_{B}\text{)}$$

the backtranslation (for $n = 3$) for this type is:

$$\begin{aligned}
\mathsf{BtT}^{\mathsf{f}E}_{3;List_B} &= \mathsf{BtT}^{\mathsf{f}E}_{3;Unit\uplus(Bool\times List_B)}\\
&= ((\mathsf{BtT}^{\mathsf{f}E}_{2;Unit}) \uplus \mathsf{BtT}^{\mathsf{f}E}_{2;Bool\times List_B}) \uplus Unit\\
&= ((Unit \uplus Unit) \uplus (((\mathsf{BtT}^{\mathsf{f}E}_{1;Bool}) \times \mathsf{BtT}^{\mathsf{f}E}_{1;List_B}) \uplus Unit)) \uplus Unit\\
&= ((Unit \uplus Unit) \uplus (((Bool \uplus Unit) \times \mathsf{BtT}^{\mathsf{f}E}_{0;List^{1}_{B}}) \uplus Unit)) \uplus Unit\\
&= ((Unit \uplus Unit) \uplus (((Bool \uplus Unit) \times Unit) \uplus Unit)) \uplus Unit
\end{aligned}$$

Formally, the measure that ensures that this type is well founded is the precision n together with $\texttt{lmc}\,(\mu\alpha.\,\tau)$ i.e., the number of leading μs in type τ, for reasons analogous to those discussed in Section 2.2.

The type of λ^{μ}_{E} terms backtranslated in $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ ($\mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n};\tau}$) is the same as the one just presented ($\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\tau}$). Intuitively, this is because the n-level deep unfolding of τ in the backtranslation type does not rely on recursive types in $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$.

3.2.1. *Working with the Backtranslation Type.* In order to work with values of backtranslated type, we need a way to create and destruct them. Additionally, we need a way to increase and decrease the approximation level (the n index), for reasons we explain below. This is what we present now mainly for terms of type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}}$, though we report the most interesting cases for the other backtranslation types too. Recall that the definitions of the other two backtranslation types are the same, so these helpers are also the same and we report only one.

Given a target value $\mathbf{v}$ of type $\boldsymbol{\tau}$, in order to *create* a source term of type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}}$ it suffices to create $\mathsf{inl\ v}$ (informally). However, in order to *use* a source term of type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}}$ at the expected type $\boldsymbol{\tau}$, we need to destroy it according to $\boldsymbol{\tau}$: this is done by the family of source functions $\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}}$.

$$\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\boldsymbol{\tau}}.\,\mathsf{case\ x\ of\ inl\ x_1} \mapsto \mathsf{x_1} \mid \mathsf{inr\ x_2} \mapsto \mathsf{omega}_{\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}}}$$

Intuitively, all these functions strip the value of type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\boldsymbol{\tau}}$ they take in input of the inl tag and return the underlying value. Thus, at arrow type, the returned value has type $(\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}} \to \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}'})$ while at recursive type it has type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}[\boldsymbol{\mu\alpha.\tau}/\boldsymbol{\alpha}]}$. In case the wrong value is passed in (i.e., it is an inr), these functions diverge via term $\mathsf{omega}_{\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}}}$, which is easily encodable in λ^{fx}.

Recall that the $\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\tau}$ for $\tau = \mu\alpha.\,\tau$ is different: it is just $\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\tau[\mu\alpha.\tau/\alpha]}$ so the type is unfolded and the index is the same. The destructor used for this backtranslation type ($\mathsf{case}^{\mathsf{f}E}_{\mathsf{n};\mu\alpha.\tau}$) is therefore different than the one above. Specifically, we do not need to destruct a backtranslated type indexed with $\mu\alpha.\,\tau$ because that never arises (i.e., the type is unfolded). Consider type $\mathsf{BtT}^{\mathsf{f}E}_{3;List_B}$ from before: at index 3 the backtranslation does not handle values of recursive type but of type $\mathsf{BtT}^{\mathsf{f}E}_{3;List^{1}_{B}}$. That is, it handles values whose top-level connector

$$\boxed{\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} \to \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\tau}}$$

$$\begin{aligned}
\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{0;\mathsf{d};\tau} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{0;\tau}.\, \mathsf{unk}_{\mathsf{d}} \\
\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{d};\mathbf{Unit}} &= \lambda \mathsf{x} : \mathsf{Unit} \uplus \mathsf{Unit}.\, \mathsf{x} \qquad \mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{d};\mathbf{Bool}} = \lambda \mathsf{x} : \mathsf{Bool} \uplus \mathsf{Unit}.\, \mathsf{x} \\
\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{d};\tau\times\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\tau\times\tau'}. \\
&\qquad \mathsf{case}\ \mathsf{x}\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ \left\langle \mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau}\ \mathsf{x}_1.1, \mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau'}\ \mathsf{x}_1.2 \right\rangle \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2 \end{array} \right. \\
\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{d};\tau\uplus\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\tau\uplus\tau'}. \\
&\qquad \mathsf{case}\ \mathsf{x}\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ \mathsf{case}\ \mathsf{x}_1\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau}\ \mathsf{x}_1) \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ (\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau'}\ \mathsf{x}_2) \end{array} \right. \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2 \end{array} \right. \\
\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{d};\tau\to\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\tau\to\tau'}. \\
&\qquad \mathsf{case}\ \mathsf{x}\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ \lambda \mathsf{z} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\tau}.\, \mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau'}\ (\mathsf{x}_1\ (\mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau}\ \mathsf{z})) \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2 \end{array} \right. \\
\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{d}\mu\alpha.\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mu\alpha.\tau'}.\, \mathsf{case}\ \mathsf{x}\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{d};\tau'[\mu\alpha.\tau'/\alpha]}\ \mathsf{x}_1) \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2 \end{array} \right.
\end{aligned}$$

$$\mathbf{upgrade}^{\mathbf{I}E}_{\mathbf{n};\tau} = \text{as } \mathsf{upgrade}^{\mathsf{f}E}_{\mathsf{n};\tau}$$

$$\mathsf{upgrade}^{\mathsf{f}E}_{\mathsf{n}+1;\mu\alpha.\tau} = \mathsf{upgrade}^{\mathsf{f}E}_{\mathsf{n}+1;\tau[\mu\alpha.\tau/\alpha]} \qquad \mathsf{upgrade}^{\mathsf{f}E}_{\mathsf{n};\tau} = \text{as above}$$

FIGURE 6. Definition of the upgrade function.

is the $\uplus$ of $List_B$. Finally, the destructor used for $\mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n};\mu\alpha.\tau}$ ($\mathbf{case}^{\mathbf{I}E}_{\mathbf{n};\mu\alpha.\tau}$) is analogous to this last one ($\mathsf{case}^{\mathsf{f}E}_{\mathsf{n};\mu\alpha.\tau}$).

$$\begin{aligned}
\mathsf{case}^{\mathsf{f}E}_{\mathsf{n};\tau} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n}+1;\tau}.\, \mathsf{case}\ \mathsf{x}\ \mathsf{of}\ \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{x}_1 \mid \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{omega}_{\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\tau}} && \tau \neq \mu\alpha.\tau \\
\mathbf{case}^{\mathbf{I}E}_{\mathbf{n};\tau} &= \boldsymbol{\lambda} \mathbf{x} : \mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n}+1;\tau}.\, \mathbf{case}\ \mathbf{x}\ \mathbf{of}\ \mathbf{inl}\ \mathbf{x}_1 \mapsto \mathbf{x}_1 \mid \mathbf{inr}\ \mathbf{x}_2 \mapsto \mathbf{omega}_{\mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n};\tau}} && \tau \neq \mu\alpha.\tau
\end{aligned}$$

The second piece of formalism that we need is functions to increase or decrease the approximation level of backtranslated terms. We exemplify their necessity with an example from Devriese et al. [2016].

Example 1 (The need for downgrade)**.** Consider $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ term $\boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}.\, \mathbf{inr}\ \mathbf{x}$, intuitively its backtranslation (for a sufficiently-large n) is: $\mathsf{inl}\ \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}-1;\tau}.\, \mathsf{inl}\ \mathsf{inr}\ \mathsf{x}$ If we try to typecheck this, though, we see that x has type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}-1;\tau}$ while it is expected to have type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}-2;\tau}$, i.e., its index should be lower. This concern is about well-typedness, not precision of the backtranslation. Since x is inside an inr , inspecting it for any number of steps requires at least an additional step, to 'case' x out of the inr . In other words, for the inr to be a precise approximation up to $n - 1$ steps, x needs to only be precise up to $n - 2$ steps. Thus, it is safe to throw away one level of precision and *downgrade* x from type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}-1;\tau}$ to $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}-2;\tau}$.
⊡

$$\boxed{\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n};\tau} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1};\tau} \to \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}}$$

$$\begin{aligned}
\mathsf{downgrade}^{\mathbf{fI}}_{0;\mathsf{d};\tau} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{d};\tau}.\,\mathsf{unit} \\
\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n+1;d};\mathbf{Unit}} &= \lambda \mathsf{x} : \mathsf{Unit} \uplus \mathsf{Unit}.\,\mathsf{x} \qquad \mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n+1;d};\mathbf{Bool}} = \lambda \mathsf{x} : \mathsf{Bool} \uplus \mathsf{Unit}.\,\mathsf{x} \\
\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n+1;d};\tau\times\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1+d};\tau\times\tau'}. \\
&\qquad \mathsf{case\ x\ of} \left| \begin{array}{l} \mathsf{inl\ x_1} \mapsto \mathsf{inl}\ \left\langle \mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau}\ \mathsf{x_1.1}, \mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau'}\ \mathsf{x_1.2} \right\rangle \\ \mathsf{inr\ x_2} \mapsto \mathsf{inr\ x_2} \end{array} \right. \\
\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n+1;d};\tau\uplus\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1+d};\tau\uplus\tau'}. \\
&\qquad \mathsf{case\ x\ of} \left| \begin{array}{l} \mathsf{inl\ x_1} \mapsto \mathsf{inl\ case\ x_1\ of} \left| \begin{array}{l} \mathsf{inl\ x_1} \mapsto \mathsf{inl}\ (\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau}\ \mathsf{x_1}) \\ \mathsf{inr\ x_2} \mapsto \mathsf{inr}\ (\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau'}\ \mathsf{x_2}) \end{array} \right. \\ \mathsf{inr\ x_2} \mapsto \mathsf{inr\ x_2} \end{array} \right. \\
\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n+1;d};\tau\to\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1+d};\tau\to\tau'}. \\
&\qquad \mathsf{case\ x\ of} \left| \begin{array}{l} \mathsf{inl\ x_1} \mapsto \mathsf{inl}\ \lambda \mathsf{z} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}.\,\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau'}\ (\mathsf{x_1}\ (\mathsf{upgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau}\ \mathsf{z})) \\ \mathsf{inr\ x_2} \mapsto \mathsf{inr\ x_2} \end{array} \right. \\
\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n+1;d};\mu\alpha.\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1+d};\mu\alpha.\tau'}.\,\mathsf{case\ x\ of} \left| \begin{array}{l} \mathsf{inl\ x_1} \mapsto \mathsf{inl}\ (\mathsf{downgrade}^{\mathbf{fI}}_{\mathsf{n;d};\tau'[\mu\alpha.\tau'/\alpha]}\ \mathsf{x_1}) \\ \mathsf{inr\ x_2} \mapsto \mathsf{inr\ x_2} \end{array} \right.
\end{aligned}$$

$$\mathbf{downgrade}^{\mathbf{I}E}_{\mathbf{n};\tau} = \text{as } \mathsf{downgrade}^{\mathsf{f}E}_{\mathsf{n};\tau}$$

$$\mathsf{downgrade}^{\mathsf{f}E}_{\mathsf{n+1};\mu\alpha.\tau} = \mathsf{downgrade}^{\mathsf{f}E}_{\mathsf{n+1};\tau[\mu\alpha.\tau/\alpha]} \qquad \mathsf{downgrade}^{\mathsf{f}E}_{\mathsf{n};\tau} = \text{as above}$$

FIGURE 7. Definition of the downgrade function.

However, downgrading is not sufficient, as demonstrated by the next example regarding function types.

Example 2 (The need for upgrade)**.** Consider how we can downgrade a value of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1};\tau\to\tau'}$ to one of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau\to\tau'}$. We need to convert a function of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1};\tau} \to \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1};\tau'}$ into one of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau} \to \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau'}$. To do this, we need to upgrade the argument value of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}$ into one of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n+1};\tau}$. Fortunately, this does not mean we need to magically improve the approximation precision of the value concerned. Type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}$ has an "error box" $(\cdots \uplus \mathsf{Unit})$ at every level so we can simply construct the value such that it simply does not use the additional level of precision in $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}$. ⊡

Finally, another reason we need to upgrade and downgrade a value is that type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}$ must be sufficiently large to contain approximations of target values *up to less than n steps*. In fact, for a term to be well-typed the accuracy of the approximation can be less than n. In these cases (i.e, for $m < n$), values of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}$ will be downgraded to type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{m};\tau}$. Dually, there will be cases where some values need to be upgraded.

Functions $\mathsf{upgrade}^{\mathbf{fI}}_{\cdot}$ and $\mathsf{downgrade}^{\mathbf{fI}}_{\cdot}$ perform what we just discussed; their types and formalisation is presented in Figures 6 and 7. Their definition closely follows the structure of the type approximations $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}$ and essentially just transfers an approximated

$\boxed{\mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} \quad \text{and} \quad \mathsf{case\text{-}up}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}}$

$$\mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Unit}} = \lambda \mathsf{x} : \mathsf{Unit}.\, \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Unit}}\ (\mathsf{inl}\ \mathsf{x}) \qquad \mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Bool}} = \lambda \mathsf{x} : \mathsf{Bool}.\, \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Bool}}\ (\mathsf{inl}\ \mathsf{x})$$

$$\mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau\to\tau'} = \begin{array}{l}\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} \to \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}.\\ \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau\to\tau'}\ (\mathsf{inl}\ \mathsf{x})\end{array} \qquad \mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau\times\tau'} = \begin{array}{l}\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} \times \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}.\\ \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau\times\tau'}\ (\mathsf{inl}\ \mathsf{x})\end{array}$$

$$\mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau\uplus\tau'} = \begin{array}{l}\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} \uplus \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}.\\ \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau\uplus\tau'}\ (\mathsf{inl}\ \mathsf{x})\end{array} \qquad \mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mu\alpha.\tau} = \begin{array}{l}\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau[\mu\alpha.\tau/\alpha]}.\\ \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mu\alpha.\tau}\ (\mathsf{inl}\ \mathsf{x})\end{array}$$

$$\mathsf{case\text{-}up}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}.\, \mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ (\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ (\mathsf{x}))$$

$\mathbf{in\text{-}dn}^{\mathbf{I}E}_{\mathbf{n};\tau}$ and $\mathbf{case\text{-}up}^{\mathbf{I}E}_{\mathbf{n};\tau}$ = as above, without a case for $\tau = \mu\alpha.\,\tau$

$\mathsf{in\text{-}dn}^{\mathsf{f}E}_{\mathsf{n};\tau}$ and $\mathsf{case\text{-}up}^{\mathsf{f}E}_{\mathsf{n};\tau}$ = as above, without a case for $\tau = \mu\alpha.\,\tau$

FIGURE 8. Compacted functions used to manipulate backtranslated values.

value to the corresponding value in a deeper or shallower approximation of type τ. The cases for Unit and Bool are optimised based on the fact that $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Unit}} = \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m};\mathbf{Unit}}$ (resp. $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Bool}} = \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m};\mathbf{Bool}}$) so long as $n, m > 0$. As mentioned, downgrade 'forgets' information about the approximation, effectively dropping *1* level of precision in the backtranslation. Dually, upgrade adds *1* level of information in the approximation. Adding this information is, however, not precise, because those additional levels are unknown (unk). Effectively, while $\mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ (\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \mathsf{t})$ reduces to t, term $\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ (\mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \mathsf{t})$ does not reduce to t because information was lost (Example 3).

Example 3 (Upgrading after downgrading forgets information)**.** Consider the following term: $\mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{0;\mathbf{Bool}}\ \mathsf{inl}\ \mathsf{true}$, which reduces to unit. If we apply $\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}_{0;\mathbf{Bool}}$ to it, we do not obtain back inl true but unk, which is inr unit. That is because downgrade forgets the shape of the value it received (inl true) and upgrade cannot possibly recover that information. ⊡

Finally, we need to define these functions for the other backtranslations that rely on the other backtranslation types $\mathsf{BtT}^{\mathsf{f}E}$ and $\mathbf{BtT}^{\mathbf{I}E}$. As mentioned, the main difference between these last two backtranslation types and $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}$ is the case for target recursive types. Recall that these last two backtranslation types for recursive types perform the unfolding of the type without decrementing the index. This affects these functions too: upgrading or downgrading a term at a recursive type is like upgrading or downgrading at the unfolding of that type but at the same index.

In the backtranslation, we generally use creation of a backtranslated value together with a $\mathsf{downgrade}^{\mathsf{f}\mathbf{I}}$, while we use destruction of backtranslated values together with an $\mathsf{upgrade}^{\mathsf{f}\mathbf{I}}$. Thus, we provide compacted functions that do exactly this, $\mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}$ and $\mathsf{case\text{-}up}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}$ (Figure 8). Note that the arguments to the first function is not ill-typeset: they indeed take a parameter whose type is the *inl* projection of type $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Unit}}$. As for the previous helpers, the compacted versions that operate on terms of type $\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\mu\alpha.\tau}$ (and $\mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n};\mu\alpha.\tau}$) are different. Since there is no destructor for $\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\mu\alpha.\tau}$, there also is no need for a compacted version.

At this point we may ask ourselves: how can we reason about these functions, as well as about backtranslated terms? This is what we explain next.

3.3. **Relating Backtranslated Terms.** If we were to use the logical relations of Figure 3 to relate a term and its backtranslation, this would simply not work. Consider $\lambda^{\mu}_{\mathbf{I}}$ type **Unit**, that is backtranslated (at any approximation $n > 0$) into $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Unit}}$, i.e., $\mathsf{Unit} \uplus \mathsf{Unit}$. Value **unit** should normally be backtranslated to $\mathsf{inl}\ \mathsf{unit}$. Following the value relation in $LR^{\mathsf{fx}}_{\mu\mathbf{I}}$ for $\uplus$ types, both terms need to have an *inl* tag, so this does not work. More importantly, it *should not* work: we are not relating terms of $\uplus$ type, we are relating backtranslated terms, where the backtranslation performs a modification on the type (and thus the term) by inserting the *inl* .

$$\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{0;\mathtt{imprecise};\tau}\right]\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \mathsf{v}, \mathbf{v}) \mid \mathsf{v} = \mathsf{unit}\} \qquad \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{0;\mathtt{precise};\tau}\right]\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \emptyset$$

$$\begin{aligned}
&\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{p};\tau}\right]\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \mathsf{v}, \mathbf{v}) \mid \mathsf{v} \in \mathsf{oftype}\left(\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{p};\tau}\right) \text{ and } \mathbf{v} \in \mathbf{oftype}\left(\tau\right) \text{ and}\\
&\quad \text{either } \cdot\ \mathsf{v} = \mathsf{inr}\ \mathsf{unit} \text{ and } \mathsf{p} = \mathsf{imprecise}\\
&\quad \text{or } \cdot \begin{cases}
\cdot\ \ \tau = \mathbf{Unit} \text{ and } \exists \mathsf{v}'.\ \mathsf{v} = \mathsf{inl}\ \mathsf{v}' \text{ and } (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}\left[\!\left[\mathsf{Unit}\right]\!\right]_{\triangledown}\\
\cdot\ \ \tau = \mathbf{Bool} \text{ and } \exists \mathsf{v}'.\ \mathsf{v} = \mathsf{inl}\ \mathsf{v}' \text{ and } (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}\left[\!\left[\mathsf{Bool}\right]\!\right]_{\triangledown}\\
\cdot\ \ \tau = \tau_1 \to \tau_2 \text{ and } \exists \mathsf{v}'.\ \mathsf{v} = \mathsf{inl}\ \mathsf{v}' \text{ and } (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau_1} \to \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau_2}\right]\!\right]_{\triangledown}\\
\cdot\ \ \tau = \tau_1 \times \tau_2 \text{ and } \exists \mathsf{v}'.\ \mathsf{v} = \mathsf{inl}\ \mathsf{v}' \text{ and } (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau_1} \times \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau_2}\right]\!\right]_{\triangledown}\\
\cdot\ \ \tau = \tau_1 \uplus \tau_2 \text{ and } \exists \mathsf{v}'.\ \mathsf{v} = \mathsf{inl}\ \mathsf{v}' \text{ and } (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau_1} \uplus \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau_2}\right]\!\right]_{\triangledown}\\
\cdot\ \ \tau = \mu\alpha.\tau \text{ and } \exists \mathsf{v}'.\ \mathsf{v} = \mathsf{inl}\ \mathsf{v}' \text{ and}\\
\quad\ \exists \mathbf{v}'.\ \mathbf{v} = \mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v}' (W, \mathsf{v}', \mathbf{v}') \in \triangleright \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau[\mu\alpha.\tau/\alpha]}\right]\!\right]_{\triangledown}
\end{cases}
\end{aligned}$$

$\mathcal{V}\left[\!\left[\mathbf{EmulT}^{\mathbf{I}E}_{\mathbf{n};\mathbf{p};\tau}\right]\!\right]_{\triangledown}$ is defined analogously to $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{n};\mathsf{p};\tau}\right]\!\right]_{\triangledown}$

$$\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}E}_{0;\mathtt{imprecise};\tau}\right]\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \mathsf{v}, v) \mid \mathsf{v} = \mathsf{unit}\} \qquad \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}E}_{0;\mathtt{precise};\tau}\right]\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \emptyset$$

$$\begin{aligned}
&\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{n}+1;\mathsf{p};\tau}\right]\!\right]_{\triangledown} \overset{\mathsf{def}}{=} \{(W, \mathsf{v}, v) \mid \mathsf{v} \in \mathsf{oftype}\left(\mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{n}+1;\mathsf{p};\tau}\right) \text{ and } v \in \mathit{oftype}\left(\tau\right) \text{ and}\\
&\quad \text{either } \cdot\ \mathsf{v} = \mathsf{inr}\ \mathsf{unit} \text{ and } \mathsf{p} = \mathsf{imprecise}\\
&\quad \text{or } \cdot \begin{cases}
\cdot\ \ \text{omitted parts are as above}\\
\cdot\ \ \tau = \mu\alpha.\tau \text{ and } \tau \text{ contractive in } \alpha \text{ and } (W, \mathsf{v}, v) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}+1;\mathsf{p};\tau[\mu\alpha.\tau/\alpha]}\right]\!\right]_{\triangledown}
\end{cases}
\end{aligned}$$

FIGURE 9. Missing bits of the logical relation: value relation for backtranslation type (excerpts). Note that p can be either `precise` or `imprecise` in the second clause (the 'or') of the $\mathsf{n}+1$ case.

This is the reason we have pseudotypes and, in particular, the reason we have *EmulT*. We have three *EmulT*s—one per backtranslation—and each follows the same intuition, which we explain starting with $\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau}$, the type of backtranslated $\lambda^{\mu}_{\mathbf{I}}$ terms into λ^{fx} (top of Figure 9). $\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathsf{p};\tau}$ is indexed by a non-negative number n, a value p ::= `precise` | `imprecise` and the original target type τ. The number tracks the depth of type that are being related, index p tracks the precision of the approximation (as explained below) and the original type carries precise information of the type to expect in the backtranslation. As seen, sometimes we have unk values (i.e., $\mathsf{inr\ unit}$) in the backtranslation, the intuition behind their meaning is presented in Example 4

Example 4 (Approximate values unk)**.** Consider the $\mathsf{BtT}^{\mathbf{fI}}_{6,\mathbf{Bool}}$ value: $\mathsf{inl}\ \langle \mathsf{inl}\ (\mathsf{inl}\ \mathsf{unk}_4), \mathsf{unk}_5\rangle$. This value might be used by the approximate back-translation to represent the term $\langle \mathbf{inl}\ \langle \mathbf{unit}, \mathbf{true}\rangle, \boldsymbol{\lambda}\mathbf{x} : \mathbf{Bool}.\, \mathbf{x}\rangle$. Our $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{\cdot}\right]\!\right]_{\triangledown\square}$ specification will enforce that terms of the form $\mathsf{inl}\ \langle\cdot,\cdot\rangle$ or $\mathsf{inl}\ (\mathsf{inl}\ \cdot)$ represent the corresponding target constructs, but terms unk_4 and unk_5 can represent arbitrary terms (in this case: a pair of base values and a lambda). ⊡

Thus, $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{n;p;}\boldsymbol{\tau}}\right]\!\right]_{\triangledown}$ regulates how these unk values occur depending on the precision index. $\mathsf{p} = \mathtt{imprecise}$ will only be used in the $\lesssim$ direction of the approximation, i.e., we have that source termination in *any* number of steps implies target termination. Here, $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{n;p;}\boldsymbol{\tau}}\right]\!\right]_{\triangledown}$ allows unk values to occur anywhere in a backtranslated term, and they can correspond to arbitrary target terms. These constraints are simple to enforce because with $\lesssim$ we can achieve this by making backtranslated terms diverge whenever they try to use a unk value. This is sufficient because the $\lesssim$ approximation trivially holds when the source term diverges.

On the other hand, $\mathsf{p} = \mathtt{precise}$ will be used for the other direction of approximation: $\gtrsim$. Recall that for this direction, termination of target terms in less than n steps implies termination of source terms. In this case, the requirements on backtranslated terms are stronger: unk is ruled out by the definition of $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{n;p;}\boldsymbol{\tau}}\right]\!\right]_{\triangledown}$ within depth n, i.e., we cannot reach unk in the steps of the world.

Example 5 (Relatedness with $\mathtt{imprecise}$)**.** Consider the term $\mathsf{t} \equiv \mathsf{inl}\ \langle \mathsf{unk}_{42}, \mathsf{unk}_{42}\rangle$. This term will be related to $\langle \mathbf{t_1}, \mathbf{t_2}\rangle$ at pseudo-type $\mathsf{EmulT}^{\mathbf{fI}}_{43;\mathtt{imprecise};\boldsymbol{\tau_1}\times\boldsymbol{\tau_2}}$ for any terms $\mathbf{t_1}$ and $\mathbf{t_2}$ and in any world. ⊡

Example 6 (Relatedness with $\mathtt{precise}$)**.** Consider again the term $\mathsf{t} \overset{\mathsf{def}}{=} \mathsf{inl}\ \langle \mathsf{unk}_{42}, \mathsf{unk}_{42}\rangle$. This term will still be related by $\mathsf{EmulT}^{\mathbf{fI}}_{43;\mathtt{precise};\boldsymbol{\tau}\times\boldsymbol{\tau}'}$ to $\mathbf{t} \overset{\mathsf{def}}{=} \langle \mathbf{t_1}, \mathbf{t_2}\rangle$ for any terms $\mathbf{t_1}$ and $\mathbf{t_2}$, but only in worlds W such that $lev(W) = 0$. More precisely, our specification will state that $(W, \mathsf{t}, \mathbf{t}) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{43;\mathtt{precise};\boldsymbol{\tau_1}\times\boldsymbol{\tau_2}}\right]\!\right]_{\triangledown}$ iff

$$(W, \langle \mathsf{unk}_{42}, \mathsf{unk}_{42}\rangle, \langle \mathbf{t_1}, \mathbf{t_2}\rangle) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{42;\mathtt{precise};\boldsymbol{\tau_1}} \times \mathsf{EmulT}^{\mathbf{fI}}_{42;\mathtt{precise};\boldsymbol{\tau_2}}\right]\!\right]_{\triangledown}$$

By the definition of the logical relation, this requires in turn that $(W, \mathsf{unk}_{42}, \mathbf{t_1})$ and $(W, \mathsf{unk}_{42}, \mathbf{t_2})$ are in $\triangleright\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{42;\mathtt{precise};\boldsymbol{\tau_1}}\right]\!\right]_{\triangledown}$ and in $\triangleright\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{42;\mathtt{precise};\boldsymbol{\tau_2}}\right]\!\right]_{\triangledown}$ respectively. However if $lev(W) = 0$, then this is vacuously true by definition of the $\triangleright$ operator, independent of the requirements of $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{42;\mathtt{precise};\cdot}\right]\!\right]_{\triangledown}$. ⊡

The pseudotype for the λ^{μ}_{E} to λ^{fx} backtranslation ($\mathsf{EmulT}^{\mathsf{f}E}_{\cdot}$) follows the same pattern as $\mathsf{BtT}^{\mathsf{f}E}_{\cdot}$: it does not lose a step in the $\mu\alpha.\,\tau$ case (Figure 9). At a cursory glance, it appears that a non-contractive $\mu\alpha.\,\tau$ ruins the well-foundedness of our induction as without decrementing our step index, a non-contractive type seems to infinitely recurse under this definition. Fortunately, however, the condition $v \in \mathit{oftype}(\tau)$, which with the fact that no values exist of non-contractive types prevents this concern from arising. As before, the pseudotype for the λ^{μ}_{E} to $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ backtranslation ($\mathbf{EmulT}^{\mathbf{I}E}_{\cdot}$) follows the same approach as $\mathsf{EmulT}^{\mathsf{f}E}_{\cdot}$.

$$
\begin{array}{rlrl}
\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n;p;}\boldsymbol{\tau}}\right) = \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n;}\boldsymbol{\tau}} & & \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_1 \to \hat{\tau}_2\right) = \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_1\right) \to \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_2\right) \\
\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\mathsf{Bool}\right) = \mathsf{Bool} & & \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_1 \times \hat{\tau}_2\right) = \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_1\right) \times \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_2\right) \\
\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\mathsf{Unit}\right) = \mathsf{Unit} & & \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_1 \uplus \hat{\tau}_2\right) = \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_1\right) \uplus \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\hat{\tau}_2\right) \\
\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n;p;}\boldsymbol{\tau}}\right) = \boldsymbol{\tau} & & \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_1 \to \hat{\tau}_2\right) = \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_1\right) \to \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_2\right) \\
\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\mathsf{Unit}\right) = \mathbf{Unit} & & \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_1 \times \hat{\tau}_2\right) = \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_1\right) \times \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_2\right) \\
\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\mathsf{Bool}\right) = \mathbf{Bool} & & \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_1 \uplus \hat{\tau}_2\right) = \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_1\right) \uplus \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\hat{\tau}_2\right) \\
\mathtt{repEmul}^{\mathsf{f}E}\left(\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n;p;}\tau}\right) = \mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n;}\tau} & & \mathtt{repEmul}^{\mathsf{f}E}\left(\cdots\right) = \text{ as the other cases for } \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\cdot\right) \\
\mathtt{repEmul}^{\mathbf{I}E}\left(\mathbf{EmulT}^{\mathbf{I}E}_{\mathbf{n},\tau}\right) = \mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n},\tau} & & \mathtt{repEmul}^{\mathbf{I}E}\left(\cdots\right) = \text{ as the other cases for } \mathtt{repEmul}^{\mathsf{f}\mathbf{I}}\left(\cdot\right) \\
\mathtt{convTy}^{\mathsf{f}\to E}\left(\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n;p;}\tau}\right) = \tau & & \mathtt{convTy}^{\mathsf{f}\to E}\left(\cdots\right) = \text{ as the other cases for } \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\cdot\right) \\
\mathtt{convTy}^{\mathbf{I}\to E}\left(\mathbf{EmulT}^{\mathbf{I}E}_{\mathbf{n;p;}\tau}\right) = \tau & & \mathtt{convTy}^{\mathbf{I}\to E}\left(\cdots\right) \text{ as the other cases for } \mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}\left(\cdot\right)
\end{array}
$$

FIGURE 10. Missing auxiliary functions of the logical relation.

Finally, we can define function $\mathtt{repEmul}^{\mathsf{f}\mathbf{I}}(\cdot)$ that translate from source pseudo-types into plain source types and function $\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\cdot)$, that translates source pseudotypes into target types (Figure 10). As expected, these functions exists for all backtranslations and they follow the same pattern presented here; for the sake of brevity, we only report the names and types of the omitted ones.

4. The Three Compilers and Their Backtranslations

Our compilers (Section 4.1) and backtranslations (Section 4.2) translate between languages as depicted in Figure 1. After showing their formalisation and proving that they relate terms cross-language, this section proves the compilers are fully abstract (Section 4.3).

4.1. Compilers and Reflection of Fully-Abstract Compilation.

The compilers (Figure 11) are all mostly homomorphic apart from what we describe below. We overload the compilation notation and express the compiler for types and terms in the same way (we omit the compiler for types since it is the identity). Compiler $\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}$ translates $\mathsf{fix}_{\_}$ into the Z-combinator annotated with **fold** and **unfold** for $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$. We cannot use the Y combinator since it does not work in call-by-value [Devriese et al., 2017; New et al., 2016], but fortunately the Z-combinator does [Pierce, 2002, Sec. 5]. Compiler $\llbracket\cdot\rrbracket^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$ erases **fold** and **unfold** annotations since λ^{μ}_{E} does not have them. Compiler $\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ is just the composition of the previous two.

Correctness of the compilation (Lemmas 3 to 5 below) is proven via a series of standard compatibility lemmas (Lemma 2, we report just the case for lambda since the others follow the same structure). These, in turn, rely on a series of standard results for these kinds of logical relations such as the fact that related terms plugged in related contexts are still related and antireduction (i.e., if two terms step to related terms, then they are themselves related).

Lemma 2 (Compatibility for λ).

$$\text{if } \Gamma, \mathsf{x} : \tau' \vdash \mathsf{t} \mathrel{\nabla_n} \mathbf{t} : \tau \text{ then } \Gamma \vdash \lambda\mathsf{x} : \tau'.\,\mathsf{t} \mathrel{\nabla_n} \boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}'.\,\mathbf{t} : \tau' \to \tau$$

$$\boxed{[\![\cdot]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} : t \to t \quad \text{and} \quad [\![\cdot]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}} : t \to t \quad \text{and} \quad [\![\cdot]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{E}} : t \to t}$$

$$[\![\mathsf{unit}]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{unit} \qquad [\![\lambda x : \tau.\, t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \lambda x : [\![\tau]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\, [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} \qquad [\![t.1]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\mathbf{1} \qquad [\![x]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = x$$

$$[\![\mathsf{true}]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{true} \qquad [\![t\ t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}\ [\![t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} \qquad [\![t.2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\mathbf{2} \qquad [\![\mathsf{inl}\ t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{inl}\ [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$$

$$[\![\mathsf{false}]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{false} \qquad [\![\langle t, t'\rangle]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \left\langle [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}, [\![t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} \right\rangle \qquad [\![t; t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}; [\![t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} \qquad [\![\mathsf{inr}\ t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{inr}\ [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$$

$$[\![\mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\ x_1 \mapsto t' \mid \mathsf{inr}\ x_2 \mapsto t'']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{case}\ [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}\ \mathbf{of}\ \mathbf{inl}\ x_1 \mapsto [\![t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} \mid \mathbf{inr}\ x_2 \mapsto [\![t'']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$$

$$[\![\mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \mathbf{if}\ [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}\ \mathbf{then}\ [\![t']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}\ \mathbf{else}\ [\![t'']\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$$

$$[\![\mathsf{fix}_{\tau_1 \to \tau_2}\ t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} = \left(\begin{array}{l} \lambda f : [\![(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}. \\ \left(\lambda x : \mu\alpha.\, \alpha \to [\![\tau_1 \to \tau_2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\, f\ (\lambda y : [\![\tau_1]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\, ((\mathbf{unfold}_{\mu\alpha.\alpha \to [\![\tau_1 \to \tau_2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}}\ x)\ x)\ y) \right) \\ \mathbf{fold}_{\mu\alpha.\alpha \to [\![\tau_1 \to \tau_2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}} \\ \quad \left(\lambda x : \mu\alpha.\, \alpha \to [\![\tau_1 \to \tau_2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\, f\ (\lambda y : [\![\tau_1]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}.\, ((\mathbf{unfold}_{\mu\alpha.\alpha \to [\![\tau_1 \to \tau_2]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}}\ x)\ x)\ y) \right) \end{array} \right) [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$$

$$[\![\cdots]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}} = \begin{array}{c}\text{omitted rules are} \\ \text{as above}\end{array} \qquad [\![\mathbf{fold}_{\mu\alpha.\tau}\ t]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}} = [\![t]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}} \qquad [\![\mathbf{unfold}_{\mu\alpha.\tau}\ t]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}} = [\![t]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}}$$

$[\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{E}} = \left[\!\!\left[[\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} \right]\!\!\right]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}}$, i.e., as above, without **fold**/**unfold** annotations in the compilation of fix

FIGURE 11. Definition of our compilers.

Lemma 3 ($[\![\cdot]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$ is semantics preserving)**.**

$$\text{if } \Gamma \vdash t : \tau \text{ then } \Gamma \vdash t \mathrel{\nabla_n} [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} : \tau$$

Lemma 4 ($[\![\cdot]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}}$ is semantics preserving)**.**

$$\text{if } \boldsymbol{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau} \text{ then } \boldsymbol{\Gamma} \vdash \mathbf{t} \mathrel{\nabla_n} [\![\mathbf{t}]\!]^{\lambda^{\mu}_{I}}_{\lambda^{\mu}_{E}} : \boldsymbol{\tau}$$

Lemma 5 ($[\![\cdot]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{E}}$ is semantics preserving)**.**

$$\text{if } \Gamma \vdash t : \tau \text{ then } \Gamma \vdash t \mathrel{\nabla_n} [\![t]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{E}} : \tau$$

Since fully-abstract compilation requires reasoning about program contexts, we extend the compiler to operate on them too. This follows the same structure of the compilers above and therefore we omit this definition. Correctness of the compiler scales to contexts too (Lemmas 6 to 8).

Lemma 6 ($[\![\cdot]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}}$ is semantics preserving for contexts)**.**

$$\text{if } \vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau' \text{ then } \vdash \mathfrak{C} \mathrel{\nabla_n} [\![\mathfrak{C}]\!]^{\lambda^{fx}}_{\lambda^{\mu}_{I}} : \Gamma, \tau \to \Gamma', \tau'$$

Lemma 7 ($[\![\cdot]\!]^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$ is semantics preserving for contexts)**.**

$$\text{if } \vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau' \text{ then } \vdash \mathfrak{C} \mathrel{\nabla_n} [\![\mathfrak{C}]\!]^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} : \Gamma, \tau \to \Gamma', \tau'$$

Lemma 8 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ is semantics preserving for contexts)**.**

$$\text{if } \vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau' \text{ then } \vdash \mathfrak{C} \mathrel{\nabla_n} [\![\mathfrak{C}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : \Gamma, \tau \to \Gamma', \tau'$$

With these results, we can already prove the reflection direction of fully-abstract compilation (Theorems 2 to 4). The proof follows the structure depicted in the left part of Figure 4.

Theorem 2 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}$ reflects equivalence)**.**

$$\text{If } \emptyset \vdash [\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} \simeq_{\mathbf{ctx}} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} \text{ then } \emptyset \vdash \mathsf{t}_1 \simeq_{\mathrm{ctx}} \mathsf{t}_2 : \tau$$

Theorem 3 ($[\![\cdot]\!]^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$ reflects equivalence)**.**

$$\text{If } \emptyset \vdash [\![\mathbf{t_1}]\!]^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} [\![\mathbf{t_2}]\!]^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} : [\![\tau]\!]^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} \text{ then } \emptyset \vdash \mathbf{t_1} \simeq_{\mathbf{ctx}} \mathbf{t_2} : \boldsymbol{\tau}$$

Theorem 4 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ reflects equivalence)**.**

$$\text{If } \emptyset \vdash [\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \text{ then } \emptyset \vdash \mathsf{t}_1 \simeq_{\mathrm{ctx}} \mathsf{t}_2 : \tau$$

Since this last compiler is the composition of the other two, the proof of Theorem 4 trivially follows from composing the proofs of the other two compilers.

4.2. **Backtranslations and Preservation of Fully-Abstract Compilation.**

Function $\mathsf{emulate}^{\mathbf{fI}}(\cdot)$ is responsible for translating a target term of type $\boldsymbol{\tau}$ into a source one of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}$ (Section 4.2.1) by relying on the machinery needed for working with $\mathsf{BtT}^{\mathbf{fI}}$ terms from Section 3.2. This function is easily extended to work with program contexts, producing contexts with hole of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}$. However, recall that the goal of the backtranslation is generating a source context whose hole can be filled with source terms t_1 and t_2 and their type is not $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}$ but τ. Thus, there is a mismatch between the type of the hole of the emulated context and that of the terms to be plugged there. Since emulated contexts work with $\mathsf{BtT}^{\mathbf{fI}}$ values, we need a function that wraps terms of an arbitrary type τ into a value of type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}$. This function is called $\mathsf{inject}^{\mathbf{fI}}$ (Section 4.2.2) and it is the last addition we need before the backtranslations (Section 4.2.3).

4.2.1. *Emulation of Terms and Contexts.*

Like the compiler, the emulation must not just operate on types and terms, but also on program contexts. Unlike the compiler, the emulation operates on *type derivations* for terms and contexts since all our target languages are typed. Thus, the emulation of a lambda would look like the following (using $\mathbf{D}$ as a metavariable to range over derivations and omitting functions to work with $\mathsf{BtT}^{\mathbf{fI}}$).

$$\mathsf{emulate}^{\mathbf{fI}}\left(\dfrac{\dfrac{\mathbf{D}}{\boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau} \vdash \mathbf{t} : \boldsymbol{\tau}'}}{\boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}.\, \mathbf{t} : \boldsymbol{\tau} \to \boldsymbol{\tau}'}\right) = \lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}.\, \mathsf{emulate}^{\mathbf{fI}}\left(\dfrac{\mathbf{D}}{\boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau} \vdash \mathbf{t} : \boldsymbol{\tau}'}\right)$$

However, note that each judgement uniquely identifies which typing rule is being applied and the underlying derivation. Thus, for compactness, we only write the judgement in the emulation and implicitly apply the related typing rule to obtain the underlying judgements for recursive calls.

$$\boxed{\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\cdot) : \mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau} \to \mathsf{t}}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{unit} : \mathbf{Unit}) \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\mathbf{Unit}}\ \mathsf{unit} \qquad \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{true} : \mathbf{Bool}) \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\mathbf{Bool}}\ \mathsf{true}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{false} : \mathbf{Bool}) \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\mathbf{Bool}}\ \mathsf{false} \qquad \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{x} \colon \boldsymbol{\tau}) \overset{\mathsf{def}}{=} \mathsf{x}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}.\, \mathbf{t} \colon \boldsymbol{\tau} \to \boldsymbol{\tau}') \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}\to\boldsymbol{\tau}'}\ (\lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}.\, \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma}, \mathbf{x} : \boldsymbol{\tau} \vdash \mathbf{t} \colon \boldsymbol{\tau}'))$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t}\ \mathbf{t}' \colon \boldsymbol{\tau}) \overset{\mathsf{def}}{=} \left(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}'\to\boldsymbol{\tau}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} \colon \boldsymbol{\tau}' \to \boldsymbol{\tau})\right) (\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t}' : \boldsymbol{\tau}'))$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \langle \mathbf{t}, \mathbf{t}' \rangle \colon \boldsymbol{\tau} \times \boldsymbol{\tau}') \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}\times\boldsymbol{\tau}'}\ \langle \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}), \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t}' : \boldsymbol{\tau}') \rangle$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t.1} \colon \boldsymbol{\tau}) \overset{\mathsf{def}}{=} \left(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}\times\boldsymbol{\tau}'}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau} \times \boldsymbol{\tau}')\right).1$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t.2} \colon \boldsymbol{\tau}) \overset{\mathsf{def}}{=} \left(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}'\times\boldsymbol{\tau}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}' \times \boldsymbol{\tau})\right).2$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\left(\begin{array}{l} \mathbf{\Gamma} \vdash \mathbf{case\ t\ of} \\ \left|\begin{array}{l} \mathbf{inl\ x_1} \mapsto \mathbf{t}' \\ \mathbf{inr\ x_2} \mapsto \mathbf{t}'' \end{array}\right. : \boldsymbol{\tau} \end{array}\right) \overset{\mathsf{def}}{=} \begin{array}{l} \mathsf{case}\ (\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau_1}\uplus\boldsymbol{\tau_2}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau_1} \uplus \boldsymbol{\tau_2})) \\ \mathsf{of} \left|\begin{array}{l} \mathsf{inl\ x_1} \mapsto \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma}, (\mathbf{x_1} : \boldsymbol{\tau_1}) \vdash \mathbf{t}' : \boldsymbol{\tau}) \\ \mathsf{inr\ x_2} \mapsto \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma}, (\mathbf{x_2} : \boldsymbol{\tau_2}) \vdash \mathbf{t}'' : \boldsymbol{\tau}) \end{array}\right. \end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{inl\ t} \colon \boldsymbol{\tau} \uplus \boldsymbol{\tau}') \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}\uplus\boldsymbol{\tau}'}\ (\mathsf{inl}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}))$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{inr\ t} \colon \boldsymbol{\tau} \uplus \boldsymbol{\tau}') \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}\uplus\boldsymbol{\tau}'}\ (\mathsf{inr}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}'))$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\left(\mathbf{\Gamma} \vdash \begin{array}{l} \mathbf{if\ t\ then\ t1} \\ \mathbf{else\ t2} \end{array} : \boldsymbol{\tau}\right) \overset{\mathsf{def}}{=} \begin{array}{l} \mathsf{if}\ \ (\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\mathbf{Bool}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \mathbf{Bool})) \\ \mathsf{then}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t1} : \boldsymbol{\tau})\ \mathsf{else}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t2} : \boldsymbol{\tau}) \end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t}; \mathbf{t}' \colon \boldsymbol{\tau}) \overset{\mathsf{def}}{=} (\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\mathbf{Unit}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \mathbf{Unit}))\,; \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t}' : \boldsymbol{\tau})$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{fold}_{\boldsymbol{\mu\alpha.\tau}}\mathbf{t} \colon \boldsymbol{\mu\alpha.\,\tau}) \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}[\boldsymbol{\mu\alpha.\tau}/\boldsymbol{\alpha}]}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}[\boldsymbol{\mu\alpha.\,\tau}/\boldsymbol{\alpha}])$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\left(\begin{array}{l} \mathbf{\Gamma} \vdash \mathbf{unfold}_{\boldsymbol{\mu\alpha.\tau}}\ \mathbf{t} \\ \ : \boldsymbol{\tau}[\boldsymbol{\mu\alpha.\,\tau}/\boldsymbol{\alpha}] \end{array}\right) \overset{\mathsf{def}}{=} \mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\mu\alpha.\tau}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\mu\alpha.\,\tau})$$

$$\mathbf{emulate}^{\mathbf{I}E}_{\mathbf{n}}(\cdots) \overset{\mathsf{def}}{=} \text{ as } \mathsf{emulate}^{\mathsf{f}E}_{\mathsf{n}}(\cdots)$$

$$\mathsf{emulate}^{\mathsf{f}E}_{\mathsf{n}}\left(\frac{\mathit{\Gamma} \vdash t : \tau \qquad \tau \overset{\circ}{=} \sigma}{\mathit{\Gamma} \vdash t : \sigma}\right) \overset{\mathsf{def}}{=} \mathsf{emulate}^{\mathsf{f}E}_{\mathsf{n}}(\ \mathit{\Gamma} \vdash t : \tau\ ) \qquad \mathsf{emulate}^{\mathsf{f}E}_{\mathsf{n}}(\cdots) \overset{\mathsf{def}}{=} \begin{array}{l}\text{other cases} \\ \text{are as above}\end{array}$$

FIGURE 12. Emulation of target terms into source ones.

Function $\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\cdot)$ (Figures 12 and 13) is indexed by the approximation index n in order to know which $\mathsf{BtT}^{\mathbf{fI}}$-helper functions to use. There are few interesting bits in the emulation of terms (and of contexts). When emulating constructors for terms of type $\boldsymbol{\tau}$, we create a value of the corresponding backtranslation type $\mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\boldsymbol{\tau}}$ and, in order to be well-typed, we $\mathsf{downgrade}^{\mathbf{fI}}$ that value by 1. Dually, emulating destructors for terms of type

$$\boxed{\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\cdot) : \ (\vdash \mathfrak{C} : \mathbf{\Gamma}, \tau \to \mathbf{\Gamma}', \tau') \to \mathfrak{C}}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}([\cdot]) \overset{\mathsf{def}}{=} [\cdot]$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \boldsymbol{\lambda}\mathbf{x} : \tau'.\, \mathfrak{C} : \\ \mathbf{\Gamma}'', \tau'' \to \mathbf{\Gamma}, \tau' \to \tau\end{pmatrix} \overset{\mathsf{def}}{=} \begin{array}{l}\mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\tau\to\tau'} \\ \quad (\lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{\mathsf{n};\tau}.\, \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}'', \tau'' \to \mathbf{\Gamma}, \mathbf{x} : \tau', \tau))\end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C}\ \mathbf{t_2} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_2) \overset{\mathsf{def}}{=} \begin{array}{l}\left(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\tau'\to\tau}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1 \to \tau_2)\right) \\ (\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t_2} : \tau_1))\end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \mathfrak{C}.\mathbf{1} : \\ \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1\end{pmatrix} \overset{\mathsf{def}}{=} \left(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\tau\times\tau'}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1 \times \tau_2)\right).2$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \mathfrak{C}.\mathbf{2} : \\ \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_2\end{pmatrix} \overset{\mathsf{def}}{=} \left(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\tau\times\tau'}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1 \times \tau_2)\right).1$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \langle \mathfrak{C}, \mathbf{t_2}\rangle : \\ \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1 \times \tau_2\end{pmatrix} \overset{\mathsf{def}}{=} \begin{array}{l}\mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\tau_1\times\tau_2} \\ \quad \langle \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1), \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t_2} : \tau_2)\rangle\end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \mathbf{inl}\ \mathfrak{C} : \\ \mathbf{\Gamma}'', \tau'' \to \mathbf{\Gamma}, \tau \uplus \tau'\end{pmatrix} \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\tau\uplus\tau'}\ (\mathsf{inl}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}'', \tau'' \to \mathbf{\Gamma}, \tau))$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \mathbf{inr}\ \mathfrak{C} : \\ \mathbf{\Gamma}'', \tau'' \to \mathbf{\Gamma}, \tau \uplus \tau'\end{pmatrix} \overset{\mathsf{def}}{=} \mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\tau\uplus\tau'}\ (\mathsf{inr}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}'', \tau'' \to \mathbf{\Gamma}, \tau'))$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \mathbf{case}\ \mathfrak{C}\ \mathbf{of} \left|\begin{array}{l}\mathbf{inl}\ \mathbf{x_1} \mapsto \mathbf{t_1} \\ \mathbf{inr}\ \mathbf{x_2} \mapsto \mathbf{t_2}\end{array}\right. : \\ \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_3\end{pmatrix} \overset{\mathsf{def}}{=} \begin{array}{l}\mathsf{case}\ (\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\tau_1\uplus\tau_2}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau_1 \uplus \tau_2)) \\ \mathsf{of} \left|\begin{array}{l}\mathsf{inl}\ \mathsf{x_1} \mapsto \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma}, (\mathbf{x_1} : \tau_1) \vdash \mathbf{t_1} : \tau_3) \\ \mathsf{inr}\ \mathsf{x_2} \mapsto \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma}, (\mathbf{x_2} : \tau_2) \vdash \mathbf{t_2} : \tau_3)\end{array}\right.\end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}\begin{pmatrix}\vdash \mathfrak{C}; \mathbf{t} : \\ \mathbf{\Gamma}, \tau \to \mathbf{\Gamma}', \tau''\end{pmatrix} \overset{\mathsf{def}}{=} \begin{array}{l}(\mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\mathbf{Unit}}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}, \tau \to \mathbf{\Gamma}', \mathbf{Unit}));\\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathbf{\Gamma} \vdash \mathbf{t}' : \tau)\end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathbf{fold}_{\mu\alpha.\tau}\, \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \mu\alpha.\,\tau) \overset{\mathsf{def}}{=} \begin{array}{l}\mathsf{in\text{-}dn}^{\mathbf{fI}}_{\mathsf{n};\tau[\mu\alpha.\tau/\alpha]} \\ \quad \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau[\mu\alpha.\,\tau/\alpha])\end{array}$$

$$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathbf{unfold}_{\mu\alpha.\tau}\, \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \tau[\mu\alpha.\,\tau/\alpha]) \overset{\mathsf{def}}{=} \mathsf{case\text{-}up}^{\mathbf{fI}}_{\mathsf{n};\mu\alpha.\tau}\ \mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \tau' \to \mathbf{\Gamma}, \mu\alpha.\,\tau)$$

$$\boxed{\mathbf{emulate}^{\mathbf{I}E}_{\mathbf{n}}(\cdot) : \ (\vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau') \to \mathfrak{C}}$$

Analogous to the case above since $\mathfrak{C}$ are a subset of $\mathfrak{C}$

$$\boxed{\mathsf{emulate}^{\mathsf{f}E}_{\mathsf{n}}(\cdot) : \ (\vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau') \to \mathfrak{C}}$$

Analogous to the case above since $\mathfrak{C}$ are a subset of $\mathfrak{C}$

FIGURE 13. Emulation of target contexts into source ones (excerpts).

τ requires upgrading the term for 1 level of precision because they are then destructed to access the underlying type. When emulating $\lambda^{\mu}_{\mathbf{I}}$ derivations into λ^{fx}, we need to consider the case when $\mathbf{fold}_{\mu\alpha.\tau}$ and $\mathbf{unfold}_{\mu\alpha.\tau}$ annotations are encountered. There, we know that the backtranslation will work with terms typed at the unfolding of $\mu\alpha.\,\tau$, so we simply perform the recursive call and insert the appropriate helper function to ensure the resulting term is well-typed. Concretely, Example 7 shows what the emulation of a simple term is.

Example 7 (Emulating a term)**.** Consider the term $\emptyset \vdash \mathbf{true} : \mathbf{Bool}$, its emulation is:

$$
\begin{aligned}
& \mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Bool}}\ \mathsf{true} \\
& \quad \text{then by unfolding the definition of } \mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\cdot} \\
= {} & (\lambda \mathsf{y} : \mathsf{Bool}.\, \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathbf{Bool}}\ (\mathsf{inl}\ \mathsf{y}))\ \mathsf{true} \\
& \quad \text{then by unfolding the definition of } \mathsf{downgrade}^{\mathsf{f}\mathbf{I}}_{\cdot}\,() \\
= {} & (\lambda \mathsf{y} : \mathsf{Bool}.\, (\lambda \mathsf{z} : \mathsf{Bool} \uplus \mathsf{Unit}.\, \mathsf{z})\mathsf{inl}\ \mathsf{y})\ \mathsf{true}
\end{aligned}
$$

Which eventually reduces to value $\mathsf{inl}\ \mathsf{true}$, as expected. ⊡

When emulating λ^{μ}_{E} derivations (in the other two emulates in Figure 12), we need to consider the case when term t is given type τ knowing it had type σ and that $\sigma \triangleq \tau$ (Rule λ^{μ}_{E}-Type-eq). Here we rely on a crucial observation: given two equivalent types, their backtranslation types are *the same* (Theorem 5). To understand why this is the case, consider how the definition of $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}$ simply unfolds recursive types without losing precision, i.e. it essentially only looks at the depth-n unfolding of type τ and these unfoldings are equal for equal types $\tau \triangleq \sigma$. With this fact, we can get away with just performing the recursive call on the sub-derivation for t at type σ.

Theorem 5 (Equivalent types are backtranslated to the same type)**.**

$$\text{If } \tau \triangleq \sigma \text{ then } \mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\tau} = \mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\sigma}$$

Finally, consider $\mathbf{emulate}^{\mathbf{I}E}_{\cdot}(\cdot)$, i.e., the emulation of λ^{μ}_{E} terms into $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$: there is no construct that adds $\mathbf{fold}/\mathbf{unfold}$ annotations. This is due to the same intuition presented before regarding the unfolding of the backtranslation type $\mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n};\mu\alpha.\tau}$, which is $\mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n};\tau[\mu\alpha.\tau/\alpha]}$ i.e, the indexing type is unfolded but the step is not decreased. Intuitively, the backtranslation performs an n-level deep unfolding of the recursive types and operates on those. Thus, backtranslated contexts do not use recursive types but just their n-level deep unfolding, so their annotations are not needed.

In order to state that $\mathsf{emulate}^{\mathsf{f}\mathbf{I}}_{\cdot}(\cdot)$ is correct, we rely on compatibility lemmas akin to those used for compiler correctness (recall Lemma 2). First, note that all our logical relations relate a source and target term at a source pseudo-type. We have extended the logical relation to express the relation between a source and target term at pseudotype $\mathsf{EmulT}^{\mathsf{f}\mathbf{I}}$, so we should use this to relate a target term and its backtranslation. Second, all logical relations require a source environment to relate terms, and in this case we are given a target environment (the one for the typing of the backtranslated term). To create a source environment starting from this target environment, we take each bound variable and give it backtranslation type using function $\mathtt{toEmul}\,(\cdot)$. Finally, in these lemmas we need to account for the different directions of the approximation we have. Thus, these compatibility lemmas require that either $n < m$ (so that the results only hold in worlds W with $lev(W) \leq n < m$) and $p = \mathtt{precise}$ or $\triangledown = \lesssim$ and $p = \mathtt{imprecise}$, for m being the approximation level of interest.

The intuition behind these constraints is that when $p = \mathtt{imprecise}$, there is no lower bound on the emulation depth m. However, in that case, we only get a left-to-right approximation $\lesssim$, since the emulated term may have insufficient precision to emulate the original term accurately (as in Example 5) and may diverge in cases where the emulation precision runs out. In the case where $p = \texttt{precise}$, the lemma requires that the emulation depth m is sufficiently large. Specifically, m is required to be at least as large as the step bound n up to which the approximation in both directions $\triangledown\, n$ is guaranteed. Intuitively, this covers the scenario where the depth of the approximation is larger than the amount of steps taken by a back-translated program. In such a scenario, the back-translation is guaranteed to accurately emulate the behaviour of the target term and we get approximations in both directions, but only up to the amount of steps n.

Thus, a typical compatibility lemma for emulate looks like Lemma 9.

Lemma 9 (Compatibility for λ Emulation)**.**

$$\text{if } (m > n \text{ and } p = \texttt{precise}) \text{ or } (\triangledown = \lesssim \text{ and } p = \texttt{imprecise})$$

$$\text{then} \quad \text{if } \texttt{toEmul}_{\mathsf{m;p}}(\mathbf{\Gamma}, \mathbf{x} : \boldsymbol{\tau}) \vdash \mathsf{t} \triangledown_n \mathbf{t} : \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m;p};\boldsymbol{\tau}'}$$

$$\text{then } \texttt{toEmul}_{\mathsf{m;p}}(\mathbf{\Gamma}) \vdash \mathsf{in\text{-}dn}^{\mathsf{f}\mathbf{I}}_{\mathsf{m};\boldsymbol{\tau}\to\boldsymbol{\tau}'} \left(\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m};\boldsymbol{\tau}}.\, \mathsf{t}\right) \triangledown_n \boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}.\, \mathbf{t} : \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m;p};\boldsymbol{\tau}\to\boldsymbol{\tau}'}$$

The compatibility lemma for terms typed using type equality (Lemma 10) is the most interesting of these. The proof of this lemma is surprisingly simple because most of the heavy lifting is done by a corollary of Theorem 5, which proves that equivalent types have not only the same backtranslation type but also the same term relation.

Lemma 10 (Compatibility lemma for emulation of type equality)**.**

$$\text{if } (m > n \text{ and } p = \texttt{precise}) \text{ or } (\triangledown = \lesssim \text{ and } p = \texttt{imprecise})$$

$$\text{then if } \texttt{toEmul}^{\mathsf{f}E}_{\mathsf{m;p}}(\mathit{\Gamma}) \vdash \mathsf{t} \triangledown_n \mathit{t} : \mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{m;p};\tau} \text{ and } \tau \mathrel{\mathring{=}} \sigma \text{ then } \texttt{toEmul}^{\mathsf{f}E}_{\mathsf{m;p}}(\mathit{\Gamma}) \vdash \mathsf{t} \triangledown_n \mathit{t} : \mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{m;p};\sigma}$$

Corollary 1 (Equivalent types have the same term relation)**.**

$$\text{if } \tau \mathrel{\mathring{=}} \sigma \text{ then } \forall n.\, \mathcal{E}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{n;p};\tau}\right]\!\right]_{\triangledown} = \mathcal{E}\left[\!\left[\mathsf{EmulT}^{\mathsf{f}E}_{\mathsf{n;p};\sigma}\right]\!\right]_{\triangledown}$$

Given a series of these kinds of compatibility lemmas, we can state that emulate is correct.

Lemma 11 (Emulate is semantics-preserving)**.**

$$\text{if } (m > n \text{ and } p = \texttt{precise}) \text{ or } (\triangledown = \lesssim \text{ and } p = \texttt{imprecise}) \text{ and } \mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}$$

$$\text{then } \texttt{toEmul}_{\mathsf{m;p}}(\mathbf{\Gamma}) \vdash \mathsf{emulate}^{\mathsf{f}\mathbf{I}}_{\mathsf{m}}(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}) \triangledown_n \mathbf{t} : \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m;p};\boldsymbol{\tau}}$$

The key property we rely on for fully-abstract compilation though, is that emulation of contexts is correct (this relies on correctness of emulation for terms though).

Lemma 12 (Emulate is semantics preserving for contexts)**.**

$$\text{if } (m > n \text{ and } p = \texttt{precise}) \text{ or } (\triangledown = \lesssim \text{ and } p = \texttt{imprecise}) \text{ and } \vdash \mathfrak{C} : \mathbf{\Gamma}', \boldsymbol{\tau}' \to \mathbf{\Gamma}, \boldsymbol{\tau}$$

$$\text{then } \vdash \mathsf{emulate}^{\mathsf{f}\mathbf{I}}_{\mathsf{m}}(\vdash \mathfrak{C} : \mathbf{\Gamma}', \boldsymbol{\tau}' \to \mathbf{\Gamma}, \boldsymbol{\tau}) \triangledown_n \mathfrak{C} : \texttt{toEmul}_{\mathsf{m;p}}(\mathbf{\Gamma}'), \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m;p};\boldsymbol{\tau}'} \to \texttt{toEmul}_{\mathsf{m;p}}(\mathbf{\Gamma}), \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{\mathsf{m;p};\boldsymbol{\tau}}$$

4.2.2. *Inject and Extract.* As mentioned, the backtranslated target context must be a valid source context in order to be linked with a source term. Specifically, it must have a hole whose type is the compilation of some source type τ. Backtranslated terms, however, have backtranslation type $\mathsf{BtT}^{\mathsf{f}E}_{\mathsf{n};\tau}$, so we need to convert values of source type into values of backtranslation type (and back). To do this conversion we rely on functions $\mathsf{inject}^{\mathsf{f}\mathbf{I}}$ and $\mathsf{extract}^{\mathsf{f}\mathbf{I}}$ whose types and definitions are in Figure 14. Function $\mathsf{inject}^{\mathsf{f}\mathbf{I}}$ takes a source value of type τ and converts it into "the same" value at the backtranslation type so that backtranslated terms can use that value. Since the backtranslation type is indexed by target types, we use function $\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\cdot)$ to generate the target type related to τ. Function $\mathsf{extract}^{\mathsf{f}\mathbf{I}}$ does the dual and takes a value of backtranslation type and converts it into a type of some source type. These functions are defined mutually inductively in order to contravariantly convert function arguments to the appropriate type.

$$\boxed{\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} : \tau \to \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)} \quad \text{and} \quad \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)} \to \tau}$$

$$\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{0;\tau} = \lambda \mathsf{x} : \tau.\, \mathsf{unit} \qquad \mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1;Unit}} = \lambda \mathsf{x} : \mathsf{Unit}.\, \mathsf{inl}\ \mathsf{x} \qquad \mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1;Bool}} = \lambda \mathsf{x} : \mathsf{Bool}.\, \mathsf{inl}\ \mathsf{x}$$

$$\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\tau\to\tau'} = \lambda \mathsf{x} : \tau \to \tau'.\mathsf{inl}\ \lambda \mathsf{y} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)}.\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}\ (\mathsf{x}\ (\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \mathsf{y}))$$

$$\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\tau\times\tau'} = \lambda \mathsf{x} : \tau \times \tau'.\mathsf{inl}\ \left\langle \mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ (\mathsf{x}.1), \mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}\ (\mathsf{x}.2)\right\rangle$$

$$\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\tau\uplus\tau'} = \lambda \mathsf{x} : \tau \uplus \tau'.\mathsf{inl}\ \mathsf{case}\ \mathsf{x}\ \mathsf{of}\ \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \mathsf{x}_1) \mid \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ (\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}\ \mathsf{x}_2)$$

$$\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{0;\tau} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)}.\, \mathsf{omega}_\tau$$

$$\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1;Unit}} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathbf{Unit}}.\, \mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathbf{Unit}}\ \mathsf{x} \qquad \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1;Bool}} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathbf{Bool}}.\, \mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathbf{Bool}}\ \mathsf{x}$$

$$\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\tau\to\tau'} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau\to\tau')}.\, \lambda \mathsf{y} : \tau.\, \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}\ \left(\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau\to\tau')}\ \mathsf{x}\ (\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \ \mathsf{y})\right)$$

$$\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\tau\times\tau'} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau\times\tau')}.\, \left\langle \begin{array}{l} \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \left(\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)}\ \mathsf{x}.1\right), \\ \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau'}\ \left(\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau')}\ \mathsf{x}.2\right) \end{array} \right\rangle$$

$$\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\tau\uplus\tau'} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n+1};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau\uplus\tau')}.\, \mathsf{case}\ \left(\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau\uplus\tau')}\ \mathsf{x}\right)\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)}\ \mathsf{x}_1 \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau')}\ \mathsf{x}_2 \end{array}\right.$$

$$\mathbf{inject}^{\mathbf{I}E}_{\mathbf{n+1};\boldsymbol{\mu\alpha.\tau}} = \boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\mu\alpha.\,\tau}.\, \mathbf{inject}^{\mathbf{I}E}_{\mathbf{n+1};\boldsymbol{\tau}[\boldsymbol{\mu\alpha.\tau}/\boldsymbol{\alpha}]}\ (\mathbf{unfold}_{\boldsymbol{\mu\alpha.\tau}}\ \mathbf{x})$$

$$\mathbf{extract}^{\mathbf{I}E}_{\mathbf{n+1};\boldsymbol{\mu\alpha.\tau}} = \boldsymbol{\lambda}\mathbf{x} : \mathbf{BtT}^{\mathbf{I}E}_{\mathbf{n+1};\mathtt{convTy}^{\mathbf{I}\to E}(\boldsymbol{\mu\alpha.\tau})}.\, \mathbf{extract}^{\mathbf{I}E}_{\mathbf{n+1};\boldsymbol{\mu\alpha.\tau}}\ \mathbf{fold}_{\boldsymbol{\mu\alpha.\tau}}\ (\mathbf{case}^{\mathbf{I}E}_{\mathbf{n+1};\mathtt{convTy}^{\mathbf{I}\to E}(\boldsymbol{\mu\alpha.\tau})}\ \ \mathbf{x})$$
omitted cases are as above

$$\mathsf{inject}^{\mathsf{f}E}_{\mathsf{n};\tau} \overset{\mathsf{def}}{=} \text{as above} \qquad\qquad \mathsf{extract}^{\mathsf{f}E}_{\mathsf{n};\tau} \overset{\mathsf{def}}{=} \text{as above}$$

FIGURE 14. Definition of the inject and extract functions.

For values of the base type, these functions use the already introduced constructors and destructors for backtranslation type to perform their conversion. For pair and sum types, these functions operate recursively on the structure of the values they take in input. For arrow type, these functions convert the argument contravariantly before converting the result after the application of the function. When the size of the type is insufficient for these functions to behave as expected (i.e., when n is 0) it is sufficient for $\mathsf{inject}^{\mathsf{f}\mathbf{I}}$ to return unit and for $\mathsf{extract}^{\mathsf{f}\mathbf{I}}$ to just diverge.

Example 8 (The need for $\mathsf{extract}^{\mathsf{f}\mathbf{I}}$)**.** Consider the emulated term from Example 7: $\mathsf{inl}\ \mathsf{true}$, which is the result of emulating $\emptyset \vdash \mathbf{true} : \mathbf{Bool}$. Ideally, we want to extract that term into type Bool at index 1, in order to strip the underlying true of the outer $\mathsf{inl}\ \cdot$. That is precisely what $\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{1;\mathsf{Bool}}$ does:

$$(\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{1;\mathsf{Bool}}\ )\ \mathsf{inl}\ \mathsf{true} =$$

$\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{1;\mathbf{Bool}}.\ \mathsf{case}^{\mathsf{f}\mathbf{I}}_{2;\mathbf{Bool}}\ \mathsf{x}\ \mathsf{inl}\ \mathsf{true}$

which by definition of $\mathsf{case}^{\mathsf{f}\mathbf{I}}_{\cdot}$ becomes

$(\lambda \mathsf{y} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{1;\mathbf{Bool}}.\ (\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{2;\mathbf{Bool}}.\ \mathsf{case}\ \mathsf{x}\ \mathsf{of}\ \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{x}_1 \mid \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{omega}_{\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{2;\mathbf{Bool}}})\mathsf{y})\ \mathsf{inl}\ \mathsf{true}$

After two reduction steps, this term becomes the expected true, which can be used at the expected Bool type. ⊡

Note that these functions are indexed by *source* types since they convert between them and the backtranslation type. Thus, while two of our compilers have the same source language (and therefore the same inject/extract), the third compiler has a different source language, with more types: $\mu\alpha.\,\tau$. Thus, for the third backtranslation, we have a different, extended version of $\mathbf{inject}^{\mathbf{I}E}$ /$\mathbf{extract}^{\mathbf{I}E}$ that converts values of recursive types into values of backtranslation type and back. Additionally, the hole of the first two backtranslations cannot have a recursive type, since the source type for those backtranslations is λ^{fx}.

As for the emulation of terms, we prove that these functions are correct according to the logical relations. Terms that are related at a source type are related at backtranslation type after an $\mathsf{inject}^{\mathsf{f}\mathbf{I}}$ while terms that are related at backtranslation type are related at source type after an $\mathsf{extract}^{\mathsf{f}\mathbf{I}}$.

Lemma 13 (Inject and extract are semantics preserving)**.**

$$\begin{aligned}
&\text{If}\ \ (m \geq n \text{ and } p = \texttt{precise}) \text{ or } (\nabla = \lesssim \text{ and } p = \texttt{imprecise})\\
\text{then}\ &\text{if}\ \ \Gamma \vdash \mathsf{t}\ \nabla_n\ \mathbf{t} : \tau \text{ then } \Gamma \vdash \mathsf{inject}^{\mathsf{f}\mathbf{I}}_{m;\tau}\ \ \mathsf{t}\ \nabla_n\ \mathbf{t} : \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{m;p;\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)}\\
&\text{if}\ \ \Gamma \vdash \mathsf{t}\ \nabla_n\ \mathbf{t} : \mathsf{EmulT}^{\mathsf{f}\mathbf{I}}_{m;p;\mathtt{convTy}^{\mathsf{f}\to\mathbf{I}}(\tau)} \text{ then } \Gamma \vdash \mathsf{extract}^{\mathsf{f}\mathbf{I}}_{m;\tau}\ \ \mathsf{t}\ \nabla_n\ \mathbf{t} : \tau
\end{aligned}$$

As mentioned in Section 1, Lemma 13 broke with the logical relation that does not define the observation relation $O(W)_{\approx}$ in terms of size-bound termination. Example 9 below argues why this technical change is needed and what the differences are in the technical development as opposed to the old one of Devriese et al. [2017].

Example 9 (The Need for Size-Bound Termination)**.** In this example, assume the logical relation does not rely on $O(\cdot)$, but on the equi-termination observation relation defined below ($\mathcal{W}(\cdot)$ for $\mathcal{W}$rong).

$$\begin{aligned}
\mathcal{W}(W)_{\lesssim} &\overset{\mathsf{def}}{=} \{(\mathsf{t}, \mathbf{t}) \mid \text{if } lev(W) > n \text{ and } \mathsf{t}{\Downarrow}_{\mathsf{n}}\mathsf{v} \text{ then } \exists \mathbf{k}.\ \mathbf{t}{\Downarrow}_{\boldsymbol{k}}\mathbf{v}\}\\
\mathcal{W}(W)_{\gtrsim} &\overset{\mathsf{def}}{=} \{(\mathsf{t}, \mathbf{t}) \mid \text{if } lev(W) > n \text{ and } \mathbf{t}{\Downarrow}_{\boldsymbol{n}}\mathbf{v} \text{ then } \exists \mathsf{k}.\ \mathsf{t}{\Downarrow}_{\mathsf{k}}\mathsf{v}\}\\
\mathcal{W}(W)_{\approx} &\overset{\mathsf{def}}{=} \mathcal{W}(W)_{\lesssim} \cap \mathcal{W}(W)_{\gtrsim}
\end{aligned}$$

Now take the following two terms (for $m >= 1$):

$$\begin{array}{ll}
\mathsf{t} : \mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{m;(\mathbf{Bool}\uplus\mathbf{Unit})\uplus\mathbf{Unit}} & \mathbf{t} : (\mathbf{Bool} \uplus \mathbf{Unit}) \uplus \mathbf{Unit}\\
\mathsf{t} = \mathsf{inl}\ (\mathsf{inl}\ (\mathsf{inl}\ (\mathsf{inl}\ (\mathsf{inr}\ \mathsf{unit})))) & \mathbf{t} = \mathbf{inl}\ (\mathbf{inl}\ \mathbf{true})
\end{array}$$

Intuitively, t correctly emulates **t** but only one level deep: it correctly emulates the outer two **inl** constructors as two inl inl · but then bails out by using inr unit, i.e., the right branch of the $\cdots \uplus \mathsf{Unit}$ in the definition of $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{n;\tau}$, which models approximation failure. For these two terms, we can fulfil the premise of Lemma 13 for specific n and m and prove that t and **t** are related, but unfortunately we cannot prove the conclusion of the lemma, which amounts to proving that if **t** terminates, then $\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\cdot}$ t terminates as well.

Let us first show that the premise of the lemma is satisfied for $\triangledown{=}\gtrsim$ and $n = 1$. This amounts to proving that t and $\mathbf{t}$ are in the term relation for $\mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{m;p;}\boldsymbol{\tau}}$, where $\boldsymbol{\tau} = (\mathbf{Bool} \uplus \mathbf{Unit}) \uplus \mathbf{Unit}$, $\mathsf{m} = 3$ and $\mathsf{p} = \mathtt{precise}$. For this, we have to prove that they are in the term relation for any world W whose level is at most n, i.e., 1. In the case where the level is *0* the relation is trivial, since any term is related in a world with no steps. Since the term relation includes the value relation, it suffices to show that: $(W, \mathsf{t}, \mathbf{t}) \in \mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{3;\mathsf{p};(\mathbf{Bool}\uplus\mathbf{Unit})\uplus\mathbf{Unit}}\right]\!\right]_{\triangledown}$. From the definition of that value relation ($n + 1$ case) it suffices to strip t of one $\mathsf{inl}\ \cdot$ and show that the terms are in $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{2;\mathsf{p};\mathbf{Bool}\uplus\mathbf{Unit}} \times \mathsf{EmulT}^{\mathbf{fI}}_{2;\mathsf{p};\mathbf{Unit}}\right]\!\right]_{\triangledown}$. From the definition of the value relation for $\uplus$ it suffices to strip each term of an *inl* $\cdot$, decrease the level of W by *1* (which becomes *0*) and show that the resulting terms ($\mathsf{inl\ inl\ inr\ unit}$ and $\mathbf{inl\ true}$) are in $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{2;\mathsf{p};\mathbf{Bool}\uplus\mathbf{Unit}}\right]\!\right]_{\triangledown}$. Again from the definition of the value relation for $\mathsf{EmulT}^{\mathbf{fI}}_{\cdot}$ ($n + 1$ case) it suffices to strip t of one $\mathsf{inl}\ \cdot$ and show that the terms are in $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{1;\mathsf{p};\mathbf{Bool}} \times \mathsf{EmulT}^{\mathbf{fI}}_{1;\mathsf{p};\mathbf{Unit}}\right]\!\right]_{\triangledown}$. From the definition of the value relation for $\uplus$ it suffices to strip each term of an *inl* $\cdot$ and prove that ($\mathsf{inr\ unit}$ and $\mathbf{true}$) are in $\triangleright\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{1;\mathsf{p};\mathbf{Bool}}\right]\!\right]_{\triangledown}$. This is vacuously true from the definition of $\triangleright\mathcal{V}[\![]\!]_{\triangledown}$ since the world has *0* steps. It is worth noting that if we had taken $n > 1$, we would not be able to prove that t and $\mathbf{t}$ are related, since the premise of $\triangleright\mathcal{V}[\![\cdot]\!]_{\triangledown}$ would be true, but the conclusion would not be (i.e., $\mathsf{inr\ unit}$ and $\mathbf{true}$ are not in $\mathcal{V}\left[\!\left[\mathsf{EmulT}^{\mathbf{fI}}_{1;\mathsf{p};\mathbf{Bool}}\right]\!\right]_{\triangledown}$ for any world).

We now focus on the reductions for the problematic case of extract, for which the conclusion of Lemma 13 does not hold (note that $\tau = (\mathsf{Bool} \uplus \mathsf{Unit}) \uplus \mathsf{Unit}$).

$$
\begin{aligned}
& \mathsf{extract}^{\mathbf{fI}}_{2,\tau}\ \ \mathsf{t} \\
\hookrightarrow^{3}\ & \mathsf{inl}\ (\mathsf{extract}^{\mathbf{fI}}_{2,\mathsf{Bool}\uplus\mathsf{Unit}}\ \ (\mathsf{inl}\ (\mathsf{inl}\ (\mathsf{inr\ unit})))) \\
\stackrel{\mathsf{def}}{=}\ & \mathsf{inl}\ \left(\left(\lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{2;\tau}.\ \mathsf{case}\ \left(\mathsf{case}^{\mathbf{fI}}_{1;\tau}\ \mathsf{x}\right)\ \mathsf{of} \left| \begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{extract}^{\mathbf{fI}}_{1;\mathsf{Bool}}\ \mathsf{x}_1) \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ (\mathsf{extract}^{\mathbf{fI}}_{1;\mathsf{Unit}}\ \mathsf{x}_2) \end{array}\right.\right)\ (\mathsf{inl}\ (\mathsf{inl}\ (\mathsf{inr\ unit})))\right) \\
\hookrightarrow^{3}\ & \mathsf{inl}\ \left(\mathsf{inl}\ (\mathsf{extract}^{\mathbf{fI}}_{1;\mathsf{Bool}}\ (\mathsf{inr\ unit}))\right) \\
\stackrel{\mathsf{def}}{=}\ & \mathsf{inl}\ \left(\mathsf{inl}\ \left(\left(\lambda \mathsf{x} : \mathsf{BtT}^{\mathbf{fI}}_{1;\mathsf{Bool}}.\ \mathsf{case}^{\mathbf{fI}}_{0;\mathsf{Bool}}\ \mathsf{x}\right)\ (\mathsf{inr\ unit})\right)\right) \\
\hookrightarrow^{3}\ & \mathsf{inl}\ \left(\mathsf{inl\ omega}_{\mathsf{BtT}^{\mathbf{fI}}_{0;\mathbf{Bool}}}\right) \text{ which diverges}
\end{aligned}
$$

This breaks Lemma 13, since our goal was to prove that $\mathsf{extract}^{\mathbf{fI}}_{2,\tau}\ \ \mathsf{t}$ terminates.

Intuitively, the problem here is that applying extract to a value like t will diverge whenever there is an approximation failure in the value, no matter how deep in the value. This approximation failure is ruled out by the value relation, but only for worlds with a sufficiently large step index. For smaller worlds, whose step index is not large enough to look at the full depth of the term, the lemma simply does not hold as demonstrated by our example.

Fortunately, the observation relation $O(\cdot)$. from Figure 2 resolves this issue, so that we can prove the conclusion of Lemma 13. Specifically, given that W has level 1, by the definition of $O(W)_{\gtrsim}$, we need to show that if $\mathbf{t} \not\Downarrow_{0} \mathbf{v}$ then t terminates. This holds vacuously since the premise of the implication is false: it is not true that $\mathbf{t} \not\Downarrow_{0} \mathbf{v}$ since $\mathtt{size}\,(\mathbf{t}) = 2$ and

$2 \not\leq 0$. In other words, the new observation relation simply rules out worlds whose step index is not large enough to look at the full depth of the term, leaving us with only larger step indices where the problem does not exist. $\boxdot$

The size-bound termination hypothesis of $O(\cdot)$. shows up in the technical development in only a few places. For the interested reader, we now give a brief, very technical and succinct overview of where the change impacts the technical development. Readers who are not experts or not interested are encouraged to skip ahead to Section 4.2.3.

Concretely, Lemma 13 relies on two auxiliary lemmas, one for $\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\cdot}$ and one for $\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\cdot}$. The latter is extended with an additional hypothesis that if $\triangledown{=}\gtrsim$, then $\mathtt{size}\,(\mathbf{t}) \leq \mathit{lev}(W)$, which comes in handy in all the cases for constructors. For example, when proving relatedness of two terms, knowing $\mathtt{size}\,(\mathbf{inl}\ \mathbf{t}) \leq \mathit{lev}(W)$ lets us rule out the case when $\mathit{lev}(W) = 0$.

Dually, in the case for $\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\cdot}$ for function types, $\mathsf{extract}^{\mathsf{f}\mathbf{I}}_{\cdot}$ is called on the argument of the function. In that case we need to prove that the world under consideration has enough steps to ensure size-bound termination of the argument of the function. This fact follows from the additional premise in the value relation for function types.

Finally, in the compatibility lemma for application, we have to fulfil this additional premise for function types and show that the $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ function argument size-bound terminates. We get this fact by unfolding a few definitions: from the definition of logical relation and term relation, in the lemma we have to prove that for any related context, the functions applied to the values are in the observation relation. From the observation relation for $\gtrsim$ we obtain the assumption that the $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ function applied to its value size-bound terminates. From this fact we obtain that just the value size-bound terminates.

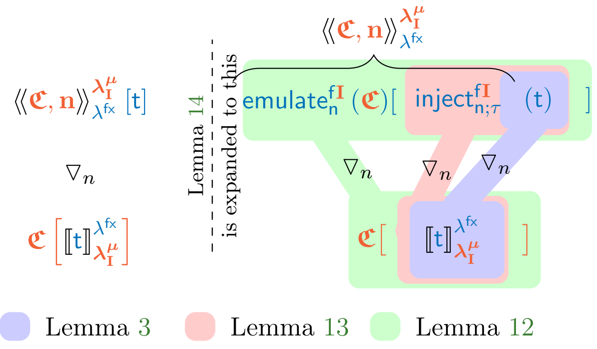
4.2.3. *The Backtranslations.* The backtranslation of a target context based on its type derivation is defined as follows by relying on both $\mathsf{emulate}^{\mathsf{f}\mathbf{I}}(\cdot)$ and $\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\cdot}$. All three backtranslations follow exactly the same pattern and enjoy the same properties. As already shown, the only interesting changes are in the sub-parts of the backtranslation (e.g., in the different definitions of inject/extract). Thus, we only show the backtranslation from $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ to λ^{fx} and we state properties only for this one.

Definition 11 (Approximate backtranslation for $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ contexts into λ^{fx})**.**

$$\langle\!\langle \mathfrak{C}, \mathbf{n} \rangle\!\rangle^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}} \overset{\mathsf{def}}{=} \mathsf{emulate}^{\mathsf{f}\mathbf{I}}_{\mathsf{n}}\left(\vdash \mathfrak{C} : \boldsymbol{\Gamma}, [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}} \to \boldsymbol{\Gamma}', \boldsymbol{\tau}'\right)\left[\mathsf{inject}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\tau}\ \cdot\right] \text{ (provided } \vdash \mathfrak{C} : \boldsymbol{\Gamma}, [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}} \to \boldsymbol{\Gamma}', \boldsymbol{\tau}')$$

As for the compiler from λ^{fx} to λ^{μ}_{E}, we can derive the backtranslation from λ^{μ}_{E} to λ^{fx} by composing the backtranslations through $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$. Thus, $\langle\!\langle t \rangle\!\rangle^{\lambda^{\mu}_{E}}_{\lambda^{\mathsf{fx}}} = \left\langle\!\!\left\langle \langle\!\langle t \rangle\!\rangle^{\lambda^{\mu}_{E}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}} \right\rangle\!\!\right\rangle^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$. Interestingly, this means that the type of λ^{μ}_{E} terms backtranslated into λ^{fx} is the same as the one for λ^{μ}_{E} terms backtranslated into $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$, i.e., the case for $\mathsf{BtT}^{\mathsf{f}E}$ for $\mu\alpha.\,\tau$ should not lose precision (as shown in Figure 5). Notice that the first backtranslation ($\langle\!\langle \cdot \rangle\!\rangle^{\lambda^{\mu}_{E}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}$) directs this, since $\mathbf{BtT}^{\mathbf{I}E}$ is simply a collection of $\hat{\boldsymbol{\tau}} \uplus \hat{\boldsymbol{\tau}}'$ pseudotypes, the second backtranslation ($\langle\!\langle \cdot \rangle\!\rangle^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$) simply relies on the case for $\mathsf{BtT}^{\mathsf{f}\mathbf{I}}_{\mathsf{n};\boldsymbol{\tau}\uplus\boldsymbol{\tau}'}$.

Using the same approach for the correctness of emulate, we can state that the backtranslations are correct. For simplicity, we provide a visual representation of this proof in Figure 15 (adapted from the work of Devriese et al. [2016] to our setting). All of the infrastructure used by the backtranslation (i.e., $\mathsf{inject}^{\mathbf{fI}}$ /$\mathsf{extract}^{\mathbf{fI}}$ and the $\mathsf{BtT}^{\mathbf{fI}}$ helpers) have correctness lemmas that follow the same structure of the one for $\mathsf{emulate}^{\mathbf{fI}}(\cdot)$. Specifically, they relate terms at $\mathsf{EmulT}^{\mathbf{fI}}$, they transform target environments into source ones via function $\texttt{toEmul}(\cdot)$ and they have a condition on the different directions of the approximation (the first line in Lemmas 9 to 12).



$\langle\!\langle \mathfrak{C}, \mathbf{n}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$ [t]
∇_n
$\mathfrak{C}\left[[\![\mathsf{t}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]$
Lemma 14
is expanded to this
$\langle\!\langle \mathfrak{C}, \mathbf{n}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$
$\mathsf{emulate}^{\mathbf{fI}}_{\mathsf{n}}(\mathfrak{C})[\ \mathsf{inject}^{\mathbf{fI}}_{\mathsf{n};\tau}\ (\mathsf{t})\ ]$
∇_n
∇_n
∇_n
$\mathfrak{C}[\ [\![\mathsf{t}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\ ]$
🟦 Lemma 3 🟥 Lemma 13 🟩 Lemma 12

FIGURE 15. Diagram representing the relatedness between different bits of the backtranslation and of the compiler.

Lemma 14 (Correctness of $\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$)**.**

If $(m \geq n$ and $p = \texttt{precise})$ or $(\nabla = \lesssim$ and $p = \texttt{imprecise})$

then if $\vdash \mathfrak{C} : \emptyset, [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} \rightarrow \emptyset, \boldsymbol{\tau}$ and $\emptyset \vdash \mathsf{t}\ \nabla_n\ \mathbf{t} : \tau$ then $\emptyset \vdash \langle\!\langle \mathfrak{C}, \mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}[\mathsf{t}]\ \nabla_n\ \mathfrak{C}[\mathsf{t}] : \mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{m;p;}\boldsymbol{\tau}}$

With correctness of the backtranslation we can prove the preservation direction of fully-abstract compilation for all compilers, following the proof structure of Figure 4.

Theorem 6 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}$ preserves equivalence)**.**

$$\text{If } \emptyset \vdash \mathsf{t}_1 \simeq_{\mathsf{ctx}} \mathsf{t}_2 : \tau \text{ then } \emptyset \vdash [\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} \simeq_{\mathbf{ctx}} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}$$

Proof. Take $\mathfrak{C}$ such that $\vdash \mathfrak{C} : \emptyset, [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} \rightarrow \emptyset, \boldsymbol{\tau}$. We need to prove that $\mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]\Downarrow \iff \mathfrak{C}\left[[\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]\Downarrow$. By symmetry, we prove only that if $\mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]\Downarrow$ then $\mathfrak{C}\left[[\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]\Downarrow$ (HPTT). Take n strictly larger than the steps needed for $\mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]\Downarrow$. By Lemma 3 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}$ is semantics preserving) we have $\emptyset \vdash \mathsf{t}_1\ \nabla_n\ [\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} : \tau$. Take $m = n$, so we have $(m \geq n$ and $p = \texttt{precise})$ and therefore $(\nabla = \gtrsim)$. By Lemma 14 (Correctness of $\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$) we have $\emptyset \vdash \langle\!\langle \mathfrak{C}, \mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}[\mathsf{t}_1] \gtrsim_n \mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right] : \mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{m;p;}\boldsymbol{\tau}}$. By Theorem 1 (Relation between Termination and Size-Bound Termination) with HPTT we have: $\mathfrak{C}\left[[\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right]\not\Downarrow_{\_}$ (HPTS). By Lemma 1 (Adequacy for $\approx$ for $LR^{\mathsf{fx}}_{\boldsymbol{\mu}\mathbf{I}}$) for $\gtrsim$ and HPTS we have: $\langle\!\langle \mathfrak{C}, \mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}[\mathsf{t}_1]\Downarrow$, which by source contextual equivalence gives us $\langle\!\langle \mathfrak{C}, \mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}[\mathsf{t}_2]\Downarrow$ (HPTS2). Given n' the number of steps for HPTS2, by Lemma 3 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}$ is semantics preserving) we have: $\emptyset \vdash \mathsf{t}_2\ \nabla_{n'}\ [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} : \tau$. So by definition: $\emptyset \vdash \mathsf{t}_2 \lesssim_{n'} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}} : \tau$. By Lemma 14 (Correctness of $\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$) (with $n = n'$, $p = \texttt{imprecise}$ and $\nabla = \lesssim$) we can conclude $\emptyset \vdash \langle\!\langle \mathfrak{C}, \mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}[\mathsf{t}_2] \lesssim_n \mathfrak{C}\left[[\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathbf{I}}}\right] : \mathsf{EmulT}^{\mathbf{fI}}_{\mathsf{m;p;}\boldsymbol{\tau}}$.

By Theorem 1 (Relation between Termination and Size-Bound Termination) with HPTS2 we have: $\langle\!\langle \mathfrak{C}, \mathbf{m} \rangle\!\rangle^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}} [\mathsf{t_2}] \not\downarrow_{\_}$ (HPTT2). By Lemma 1 (Adequacy for $\approx$ for $LR^{\mathsf{fx}}_{\mu\mathbf{I}}$) for $\lesssim$ with HPTT2 we conclude the thesis. □

Theorem 7 ($[\![\cdot]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$ preserves equivalence)**.**

$$\text{If } \emptyset \vdash \mathbf{t_1} \simeq_{\mathbf{ctx}} \mathbf{t_2} : \boldsymbol{\tau} \text{ then } \emptyset \vdash [\![\mathbf{t_1}]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} \simeq_{\text{ctx}} [\![\mathbf{t_2}]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} : [\![\boldsymbol{\tau}]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$$

Theorem 8 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ preserves equivalence)**.**

$$\text{If } \emptyset \vdash \mathsf{t_1} \simeq_{\mathsf{ctx}} \mathsf{t_2} : \tau \text{ then } \emptyset \vdash [\![\mathsf{t_1}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \simeq_{\text{ctx}} [\![\mathsf{t_2}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$$

4.3. **Full Abstraction for the Three Compilers.** With the two directions of fully-abstract compilation already proved, we can easily show that all three compilers are fully abstract. As before, full abstraction of $[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ trivially follows from composing full abstraction for the other two compilers.

Theorem 9 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}$ is fully abstract)**.**

$$\emptyset \vdash \mathsf{t_1} \simeq_{\mathsf{ctx}} \mathsf{t_2} : \tau \iff \emptyset \vdash [\![\mathsf{t_1}]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}} \simeq_{\mathbf{ctx}} [\![\mathsf{t_2}]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}$$

Theorem 10 ($[\![\cdot]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$ ms fully abstract)**.**

$$\emptyset \vdash \mathbf{t_1} \simeq_{\mathbf{ctx}} \mathbf{t_2} : \boldsymbol{\tau} \iff \emptyset \vdash [\![\mathbf{t_1}]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} \simeq_{\text{ctx}} [\![\mathbf{t_2}]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} : [\![\boldsymbol{\tau}]\!]^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$$

Theorem 11 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ is fully abstract)**.**

$$\emptyset \vdash \mathsf{t_1} \simeq_{\mathsf{ctx}} \mathsf{t_2} : \tau \iff \emptyset \vdash [\![\mathsf{t_1}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \simeq_{\text{ctx}} [\![\mathsf{t_2}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$$

5. Mechanisation of the Results

A full mechanization of all results in this paper in the Coq proof assistant is available at the following url:

https://github.com/dominiquedevriese/fixismu-coq

As the results of this paper are based on the earlier results of Devriese et al. [2016, 2017], the mechanization is based on the one of Devriese et al. [2017]. It was this mechanization effort which made us notice the errors in our earlier paper-only proofs [Patrignani et al., 2021] and it is the mechanization which makes us confident in our current solution based on Size-Bound Termination. In fact, Size-Bound Termination was first used in the Coq mechanization and subsequently backtranslated - *cough* - to the paper proofs.

The mechanized proof corresponds quite closely to the proofs detailed in this paper, including the addition of Size-Bound Termination. The main technical challenge is that Coq requires us to be more specific about certain aspects that we gloss over informally on paper. This includes specifically the fact that all types in $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ and λ^{μ}_{E} are closed and that all recursive types must be contractive. Interestingly, this contractiveness requirement is necessary for our backtranslation from $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ to λ^{fx} to work, but not essential for the meta-theory of $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ itself, so we had initially not included the requirement in the definition of the language but treated it only as a precondition of the back-translation. This broke down because the meta-theory of λ^{μ}_{E} does not make sense without the contractiveness requirement and embedding potentially uncontractive $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ terms into contractive λ^{μ}_{E} terms does not work, so we ended up including the requirement in the definition of $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ as well.

6. Discussion

At this point, it is useful to take a step back, and reflect on the meaning of our results. As we have explained, our results demonstrate that iso- and equi-recursive types do not fundamentally alter the expressiveness of the simply typed lambda calculus with term-level recursion. This result can appear contradictory, since recursive types certainly make it possible to define types and programs that do not exist in the unmodified simply typed lambda calculus. A simple example is the type of boolean lists $\mathit{BoolList} \overset{\mathsf{def}}{=} \mu\mathbf{X}.\,\mathbf{Unit} \uplus (\mathbf{Bool} \times \mathbf{X})$. This type is inexpressible in the simply typed lambda calculus, as is, in fact, any type that can contain values of an a priori unbounded size. Clearly, the ability to define such types and algorithms that work with it, is useful in a programming language. But what then does it mean that recursive types do not increase the expressiveness of the language?

To understand this well, it is important to reflect on the meaning of programming language expressiveness. As we have explained, we use a fully abstract embedding to express equi-expressiveness between the two languages. Let us investigate the statement of, for example, Theorem 9 again, to reflect upon what it means:

$$\emptyset \vdash \mathsf{t}_1 \simeq_{\mathrm{ctx}} \mathsf{t}_2 : \tau \iff \emptyset \vdash [\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}} \simeq_{\mathrm{ctx}} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}} : [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}$$

The property states that if two terms t_1 and t_2 are contextually equivalent in $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$, then they remain contextually equivalent in λ^{μ}_{E}. To understand what this means for the relative expressiveness of $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E}, one should regard the contextual equivalence $\mathsf{t}_1 \simeq_{\mathrm{ctx}} \mathsf{t}_2$ as an expressiveness challenge for $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ contexts. The property implies that no $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ context is sufficiently expressive to distinguish the two terms t_1 and t_2. The fully abstract embedding of Theorem 10 then, implies that if such a challenge is unsolvable by $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ contexts, then it is also unsolvable by λ^{μ}_{E} contexts.

It is not difficult to see that other language extensions of $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ do change the set of contextual equivalences. For example, adding some form of mutable state would make it easy to distinguish $\lambda\mathsf{f} : \mathsf{Unit} \to \mathsf{Unit}.\,\mathsf{f}\ (\mathsf{f}\ \mathsf{unit})$ from $\lambda\mathsf{f} : \mathsf{Unit} \to \mathsf{Unit}.\,\mathsf{f}\ \mathsf{unit}$. Our results imply that no such expressiveness differences exist between λ^{fx}, $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E}.

Essentially, our proof is based on considering how a λ^{μ}_{E} or $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ context solves one of the expressiveness challenges we consider. Specifically, when a $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ or λ^{μ}_{E} context distinguishes two terms by terminating for one but diverging for another, we cannot simply replicate its behaviour in λ^{fx} because it may have used values of types that are unrepresentable in λ^{fx}. However, the terminating execution will have taken only a finite amount of steps and in this finite amount of steps, it can only have inspected $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ or λ^{μ}_{E} values up to a finite depth. Because of this, we can replicate the context's behaviour in λ^{fx} using only finite types by approximating potentially infinite recursive types up to a sufficiently large but finite depth. It is precisely this approximation of infinite types that we define in our back-translation.

The usage of contextual equivalences as a challenge of expressiveness for program contexts allows us to (1) clarify how our fully abstract embeddings imply a form of equi-expressiveness and (2) understand the limitations of the presented results. Particularly, the results are crucially based on the observation that the challenge only requires accurately emulating the behaviour of a two particular executions and only up to the point that one terminates while the other doesn't. We could, for example, consider expressiveness challenges that involve not two programs, but an infinite sequence of programs, in which case, it might not be possible to determine a finite depth of emulation for the back-translation to work.

A well-known infinitary expressiveness challenge, for example, is to take the set of all Turing machines, encoded as integers, and require the context to terminate iff the corresponding Turing machine terminates. Since λ^{fx} types can only represent finite data types (note the absence of an unbounded integer type), it is not obvious that such a context exists, as Turing machines may use unbounded amounts of memory. Then again, in the absence of infinite types, it is also impossible to encode the infinite set of Turing machines. If we did have a type of unbounded naturals or integers, there would automatically be ways to represent infinite memory, for example, as functions of type $\mathbb{N} \to \mathbb{N}$. As such, it is natural to suspect that such a version of λ^{fx} would be able to semi-decide Turing machine termination, like $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E}.

The expressiveness comparison might also yield different results in versions of λ^{fx}, $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and λ^{μ}_{E} with external effects. In such a setting, the observable behaviour of an expression might consist of a potentially infinite trace of events rather than termination after a finite amount of steps. The infinite nature of observable behaviour in such a setting might also make it impossible to determine a bound on the required back-translation depth. In such a setting, one could imagine an expressiveness challenge that requires the context to produce effectful behaviour that requires an unbounded amount of memory. For example, we might consider a set of programs that invoke a function in the context, where the context needs to respond to each invocation by printing the full list of values received so far. If there is an infinite amount of programs without a bound on the amount of values, then the finite memory of λ^{fx} contexts might not allow them to remember all booleans received, unlike $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ or λ^{μ}_{E} contexts. In such an effectful setting, an infinitary expressiveness challenge might indeed demonstrate a way that recursive types increase the expressiveness of the language.

In this paper, it is not our goal to investigate in detail such other notions of expressiveness, defined by infinitary expressiveness challenges and/or potentially infinite external effects. However, it is important to understand that our results naturally pertain to forms of language expressiveness that are measured using finitary expressiveness challenges like full abstraction. This corresponds to the intuitive understanding that recursive types allow defining potentially infinite types like lists and algorithms that work with them. We consider it likely that the existence of such types and such algorithms can be detected using appropriately-chosen infinitary expressiveness challenges. As such, the equi-expressiveness of our full abstraction results should not be taken to mean that recursive types are useless, just that they do not increase the ability of contexts to distinguish pairs of expressions.

7. Related Work

Two alternative formulations of equi-recursive types exist: one based on an inductive type equality (which we dub $\lambda^{\mu}_{\mathtt{Ei}}$ in this section) and one based on a weak type equality (which we dub $\lambda^{\mu}_{\texttt{Es}}$).[2] $\lambda^{\mu}_{\texttt{Ei}}$ defines an equality relation on types ($\doteq$) that, unlike ours, is inductively defined [Abadi and Fiore, 1996]. Types are equal if they are the same (Rules Eq-type-Base and Eq-type-Var), when their subparts are equal (Rules Eq-type-Bi and Eq-type-Mu) or when one is the unfolding of the other (Rule Eq-type-Unfold). To keep track of type variables, typing equality is defined with respect to an environment $\Delta ::= \emptyset \mid \Delta; \alpha$.

$\boxed{\tau \doteq \sigma}$

$$\text{(Eq-type-Symmetric)}\ \frac{\Delta \vdash \tau' \doteq \tau}{\Delta \vdash \tau \doteq \tau'} \quad\text{and}\quad \text{(Eq-type-Transitive)}\ \frac{\Delta \vdash \tau \doteq \tau'' \quad \Delta \vdash \tau'' \doteq \tau'}{\Delta \vdash \tau \doteq \tau'} \quad \text{(Eq-type-Bi)}\ \frac{\star \in \{\to, \times, \uplus\} \quad \Delta \vdash \tau_1 \doteq \tau_1' \quad \Delta \vdash \tau_2 \doteq \tau_2'}{\Delta \vdash \tau_1 \star \tau_2 \doteq \tau_1' \star \tau_2'} \quad\text{and}\quad \text{(Eq-type-Base)}\ \frac{\iota = \texttt{Unit} \vee \iota = \texttt{Bool}}{\Delta \vdash \iota \doteq \iota}$$

$$\text{and}\quad \text{(Eq-type-Var)}\ \frac{\alpha \in \Delta}{\Delta \vdash \alpha \doteq \alpha} \quad\text{and}\quad \text{(Eq-type-Mu)}\ \frac{\Delta, \alpha \vdash \tau \doteq \tau'}{\Delta \vdash \mu\alpha.\,\tau \doteq \mu\alpha.\,\tau'} \quad\text{and}\quad \text{(Eq-type-Unfold)}\ \frac{\Delta \vdash \tau[\mu\alpha.\,\tau/\alpha] \doteq \tau'}{\Delta \vdash \mu\alpha.\,\tau \doteq \tau'}$$

Cai et al. [2016] explain that this notion of type equality is strictly weaker than the coinductive one we have used. For example, they mention two type equalities that do not hold in $\lambda^{\mu}_{\texttt{Ei}}$:

$$\emptyset \vdash \mu\alpha.\,\alpha \to \texttt{Unit} \not\doteq \mu\alpha.\,(\alpha \to \texttt{Unit}) \to \texttt{Unit} \qquad \emptyset \vdash \mu\alpha.\,\mu\beta.\,\alpha \to \beta \not\doteq \mu\alpha.\,\alpha \to \alpha$$

To understand why these equalities do not hold in the inductive formulation, consider that no amount of unfolding of a recursive type μs will ever produce recursive types with a different body.

$\lambda^{\mu}_{\texttt{Es}}$ instead enforces that just a recursive type and its unfolding are equivalent [Ahmed, 2004; Appel and McAllester, 2001; MacQueen et al., 1984; Urzyczyn, 1995]. This leads to more compact typing rules and it does not require a type equivalence relation, effectively this is like $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ but without **fold**/**unfold** annotations.

$$\text{(Type-}\lambda^{\mu}_{\texttt{Es}}\text{-fold)}\ \frac{\Gamma \vdash \texttt{t} : \tau[\mu\alpha.\,\tau/\alpha]}{\Gamma \vdash \texttt{t} : \mu\alpha.\,\tau} \qquad \text{(Type-}\lambda^{\mu}_{\texttt{Es}}\text{-unfold)}\ \frac{\Gamma \vdash \texttt{t} : \mu\alpha.\,\tau}{\Gamma \vdash \texttt{t} : \tau[\mu\alpha.\,\tau/\alpha]}$$

The main difference is that in this last variant, unfoldings can only happen at the top-level of a type of a term (i.e., when terms are of a recursive type themselves). In both $\lambda^{\mu}_{\texttt{Ei}}$ and in our coinductive variant λ^{μ}_{E}, unfoldings can also happen inside the types. For example, types such as $(\mu\alpha.\,B \uplus \alpha) \to B$ and $(B \uplus (\mu\alpha.\,B \uplus \alpha)) \to B$ are not equivalent in this last variant, because we can unfold $\mu\alpha.\,B \uplus \alpha$ to $(B \uplus (\mu\alpha.\,B \uplus \alpha))$ inside the domain of the function type. These types are however equivalent in $\lambda^{\mu}_{\texttt{Ei}}$ and in λ^{μ}_{E}.

Since terms of $\lambda^{\mu}_{\texttt{Ei}}$ (or $\lambda^{\mu}_{\texttt{Es}}$) can be typed in λ^{μ}_{E} and their semantics do not vary, our results show that all these different formulations of equi-recursive types are equally expressive. Since the approximate backtranslation is needed to deal with the coinductive derivations of λ^{μ}_{E}, we believe that a precise backtranslation akin to that of New et al. [2016] can be used to prove full abstraction for the compiler from $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ to $\lambda^{\mu}_{\texttt{Ei}}$. We leave investigating this for future work.

As mentioned in Section 1, the closest work to ours is that of Abadi and Fiore [1996]. Like us, they study the relation between iso- and equi-recursive types and prove that any term typed $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ can be typed in $\lambda^{\mu}_{\texttt{Ei}}$ and vice versa. For the backward direction, they insert cast functions which appropriately insert **fold** and **unfold** annotations to make terms typecheck. Additionally, they use a logic to prove that the terms with the casts are equivalent to the original, but the logic does not come with a soundness proof. Abadi and Fiore do not connect

[2]We typeset these languages in a `green`, `verbatim` font, though they appear in this section only.

their results to the operational semantics in any way, unlike ours, and their results cannot be used to derive fully-abstract compilation, as they relate one term and its compilation, not two terms and their compilation. Finally, it is not clear if Abadi and Fiore's Theorem 6.8 can be interpreted to imply any form of equi-expressiveness of the two languages. In fact, what Abadi and Fiore prove is that an equi-recursive term is equal to a back-translated term under a certain equality that is (conjectured to be) almost (but not entirely) sound for observational equivalence in equi-recursive contexts. On the other hand, in our setting, the interaction of the same programs with arbitrary contexts provides a measure on the relative expressiveness of those contexts when interacting with the given programs. This difference is key to make claims about the relative expressive power of languages, as we make.

Fully-abstract compilation derived from fully-abstract semantics models [Milner, 1977], and it has been initially devised to study the relative expressive power of programming languages [Felleisen, 1991; Gorla and Nestmann, 2016; Mitchell, 1993].[3] Fully-abstract compilation has been widely used to compare process algebras and their relative expressiveness, as surveyed by Parrow [2008]. Additionally, researchers have argued that fully-abstract compilation is a feasible criterion for secure compilation [Abadi, 1998; Kennedy, 2006], as surveyed by Patrignani et al. [2019].

Proofs of fully-abstract compilation are notoriously complex and thus a large amount of work exists in devising proof techniques for it. Most of these proof techniques require a form of backtranslation [Ahmed and Blume, 2008, 2011; Bowman and Ahmed, 2015]. Precise backtranslations generate source contexts that reproduce the behaviour of the target context faithfully, without any approximation [New et al., 2016; Van Strydonck et al., 2019]. Approximate backtranslations, instead, generate source contexts that reproduce that behaviour up to a certain number of steps. The approximate backtranslation proof technique we use was conjectured by Schmidt-Schauß et al. [2015] and was used by Devriese et al. [2017] to prove full abstraction for a compiler from λ^{fx} to the untyped lambda calculus (λ^u). Unlike these works, we deal with a family of backtranslation types that is indexed by target types. Additionally, our compilers do not perform dynamic typechecks; they are simply the canonical translation of a term in the source language into the target. Finally, we remark that our results cannot be derived from Devriese et al. [2016] since the languages in that paper have no recursive types.

Interestingly, our current result can be seen as factoring out the first phase of Devriese et al. [2016]'s compiler; their result could be seen as composing one of our current results with a second fully abstract compiler from $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ to λ^u, which takes care of dynamic type enforcement. The full abstraction proof for this second compiler could be a lot simpler with recursive types in the source language, as it would no longer require an approximate backtranslation. In fact, we believe that reusable sub-results could be factored out from other full abstraction results in the literature too. For example, we conjecture that one could separate closure conversion from purity enforcement in New et al. [2016]'s compiler, or separate contract enforcement from universal contract erasure in Van Strydonck et al. [2019]'s compiler. We hope our experience can inspire other researchers to pay more attention to such factoring opportunities and strive to minimize compiler phases. In other words, we believe the community could benefit from using a nanopass secure compilation mindset, in the spirit of *nanopass* compilation [Sarkar et al., 2004]. Even computationally-trivial nanopasses like

[3]Not all these works use the term "fully-abstract compilation" but their intuition is the same.

ours can be useful as they enrich the power of contexts and simplify secure compilation proofs further downstream.

8. Conclusion

This paper demonstrates that the simply typed lambda calculus with iso- and equi-recursive types has the same expressive power. To do so, it presented three fully-abstract compilers in order to reason about iso- and equi-recursively typed terms interacting over a simply-typed interface and a recursively-typed one. The first compiler translates from a simply-typed lambda calculus with a fixpoint operator (λ^{fx}) to a simply-typed lambda calculus with iso-recursive types ($\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$). The second compiler translates from λ^{fx} to a simply-typed lambda calculus with coinductive equi-recursive types (λ^{μ}_{E}). These two compilers demonstrate the same expressive power of iso- and equi-recursive types on a simply-typed interface. The third compiler translates from $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ to λ^{μ}_{E}, demonstrating equal expressiveness of iso- and equi-recursive types on a recursively-typed interface. All fully-abstract compilation proofs rely on a novel adaptation of the approximate backtranslation proof technique that works with families of target types-indexed backtranslation type.

Acknowledgements

The authors thank the anonymous reviewers for detailed feedback on an earlier draft as well as Phil Wadler for interesting comments and suggestions and Steven Keuchel for Coq hints. This work was partially supported: by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019); by the Air Force Office of Scientific Research under award number FA9550-21-1-0054, and by the Fund for Scientific Research - Flanders (FWO).

References

M. Abadi. Protection in programming-language translations. In *ICALP'98*, pages 868–883, 1998.

M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 242–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7463-6.

A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming*, pages 157–168. ACM, 2008. ISBN 978-1-59593-919-7.

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. ACM, 2011. ISBN 978-1-4503-0865-6.

A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, Sept. 2001. ISSN 0164-0925.

N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. volume 44, pages 97–108, 2009.

W. J. Bowman and A. Ahmed. Noninterference for free. In *ICFP*. ACM, 2015.

Y. Cai, P. G. Giarrusso, and K. Ostermann. System f-omega with equirecursive types for datatype-generic programming. *SIGPLAN Not.*, 51(1):30–43, Jan. 2016. ISSN 0362-1340.

D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. In *Principles of Programming Languages*, pages 164–177, 2016.

D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science*, Volume 13, Issue 4, Oct. 2017.

M. Felleisen. On the expressive power of programming languages. In *Selected Papers from the Symposium on 3rd European Symposium on Programming*, ESOP '90, pages 35–75, New York, NY, USA, 1991. Elsevier North-Holland, Inc.

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

M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg, 1979. ISBN 978-3-540-09724-2. doi: 10.1007/3-540-09724-4.

D. Gorla and U. Nestmann. Full abstraction for expressiveness: history, myths and facts. *Mathematical Structures in Computer Science*, 26(4):639–654, 2016.

R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, Apr. 1993. ISSN 0164-0925. doi: 10.1145/169701.169696.

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

H. Im, K. Nakata, and S. Park. Contractive signatures with recursive types, type parameters, and abstract types. In F. V. Fomin, R. Freivalds, M. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming*, pages 299–311, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

A. Kennedy. Securing the .NET Programming Model. *Theoretical Computer Science*, 364: 311–317, 2006. ISSN 0304-3975.

D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, page 165–174, New York, NY, USA, 1984. Association for Computing Machinery. ISBN 0897911253. doi: 10.1145/800017.800528. URL https://doi.org/10.1145/800017.800528.

R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1 – 22, 1977. ISSN 0304-3975.

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

J. H. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1968.

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

J. Parrow. Expressiveness of process algebras. *Elec. Not. Theo. Comp. Sci.*, 209(0):173 – 186, 2008.

M. Patrignani. Why should anyone use colours? or, syntax highlighting beyond code snippets. CoRR abs/2001.11334, 2020.

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

M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation a survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, Jan. 2019.

M. Patrignani, E. M. Martin, and D. Devriese. On the semantic expressiveness of recursive types. *Proc. ACM Program. Lang.*, 5(POPL), Jan. 2021. doi: 10.1145/3434302. URL https://doi.org/10.1145/3434302.

B. Pierce. *Types and Programming Languages.* MIT Press, 2002.

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. *ACM SIGPLAN Notices*, 39(9):201–212, Sept. 2004. ISSN 0362-1340. doi: 10.1145/1016848.1016878.

M. Schmidt-Schauß, D. Sabel, J. Niehren, and J. Schwinghammer. Observational program calculi and the correctness of translations. *Theoretical Computer Science*, 577:98 – 124, 2015. ISSN 0304-3975.

L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.*, 3 (POPL):19:1–19:28, Jan. 2019. ISSN 2475-1421.

P. Urzyczyn. Positive recursive type assignment. In *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science*, MFCS '95, pages 382–391, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3-540-60246-1.

T. Van Strydonck, F. Piessens, and D. Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, ICFP, 2019.