On the Semantic Expressiveness of Recursive Types

MARCO PATRIGNANI, Stanford University, USA and CISPA Helmholtz Center for Information Security, Germany
ERIC MARK MARTIN, Stanford University, USA
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium

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.

CCS Concepts: • **Theory of computation → Lambda calculus**; *Type theory*; • **Software and its engineering → Recursion**.

Additional Key Words and Phrases: Fully-abstract compilation, Lambda Calculus, Recursive types, Iso-recursive types, Coinductive Equi-recursive types, Backtranslation

ACM Reference Format:
Marco Patrignani, Eric Mark Martin, and Dominique Devriese. 2021. On the Semantic Expressiveness of Recursive Types. *Proc. ACM Program. Lang.* 5, POPL, Article 21 (January 2021), 29 pages. https://doi.org/10.1145/3434302

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.[1]

[1] 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).

Authors' addresses: Marco Patrignani, Computer Science, Stanford University, USA , CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, mp@cs.stanford.edu; Eric Mark Martin, Computer Science, Stanford University, USA, ericmarkmartin@cs.stanford.edu; Dominique Devriese, Computer Science, Vrije Universiteit Brussel, Brussels, Belgium, dominique.devriese@vub.be;

2475-1421/2021/1-ART21
https://doi.org/10.1145/3434302

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: $\mathbf{unfold}_{\boldsymbol{\mu\alpha.\tau}}\ (\mathbf{fold}_{\boldsymbol{\mu\alpha.\tau}}\ \mathbf{v}) \hookrightarrow \mathbf{v}$. In this paper, we work with a standard iso-recursive calculus $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$.

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 even 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.

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.

The question of 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 calculus $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ or λ^{μ}_{E}. A much more interesting question is whether there are ways in which λ^{μ}_{E} contexts (i.e., larger programs) can interact with t that contexts in $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ 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}.

Concretely, in this paper, we study the expressive power of $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ and λ^{μ}_{E} when interacting over two kinds of 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), and embed them canonically into both $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ and λ^{μ}_{E}. We show that if two λ^{fx} terms cannot be distinguished by λ^{fx} contexts, then the same is true for both $\boldsymbol{\lambda^{\mu}_{\mathbf{I}}}$ and λ^{μ}_{E} contexts. Additionally, we consider STLC types that contain recursive types themselves as interfaces. We take implementations of them in $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ and a canonical compiler into λ^{μ}_{E}. We show that this compiler is also fully abstract. These three fully-abstract compilation results establish the equi-expressiveness of $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$, λ^{μ}_{E}, and λ^{fx} contexts, interacting over simply-typed interfaces with and without recursive types.

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.

Proving full abstraction for a compiler is notoriously hard, particularly 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 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}.

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

Fig. 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.

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

- adapting the approximate backtranslation proof technique to operate on families of backtranslation types that are type-indexed on target types and compilers that do not rely on dynamic typechecks to attain fully-abstract compilation;
- proving that the compiler from λ^{fx} to $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ is fully abstract with an approximate backtranslation;
- proving that the compiler from λ^{fx} to λ^{μ}_{E} is fully abstract with an approximate backtranslation;
- proving that the compiler from $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ to λ^{μ}_{E} is fully abstract with an approximate backtranslation.

Note that technically, we can derive the compiler and backtranslation between λ^{fx} and λ^{μ}_{E} by composing the compilers and backtranslations through $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$. We present this result directly 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}, $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ 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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$, from λ^{fx} to λ^{μ}_{E} and from $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ to λ^{μ}_{E}) and prove that they are fully abstract using three approximate backtranslations (Section 4). Finally, we discuss related work and conclude (Sections 5 and 6).

For space constraints we omit some formalisation, auxiliary lemmas and proofs, which can be found in the online appendix [Patrignani et al. 2020].

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 ($\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$) 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 2.4). 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 $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ has **fold** and **unfold** annotations; λ^{μ}_{E} requires no additional syntactic construct. Regarding types, both $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ and λ^{μ}_{E} add recursive types according to the same syntax. In $\boldsymbol{\lambda^{\mu}_{\mathrm{I}}}$ 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 → or × [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 (Lemma 1). 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.

Lemma 1 (No value has a non-contractive type). *if τ is non-contractive then $\nexists v.\ \varnothing \vdash v : \tau$.*

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}
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} \qquad \Gamma ::= \varnothing \mid \Gamma, x : \tau\\
\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 \qquad \tau^{s} ::= \boldsymbol{\alpha} \mid \alpha \mid \tau\\
t &::= \mathit{unit} \mid \mathit{true} \mid \mathit{false} \mid \lambda x : \tau.\, t \mid x \mid t\ t \mid t.1 \mid t.2 \mid \langle t, t\rangle \mid \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t \mid \mathit{inr}\ x_2 \mapsto t\\
&\quad \mid \mathit{inl}\ t \mid \mathit{inr}\ t \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \mid t; t \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}.1 \mid \mathbb{E}.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}.1 \mid \mathfrak{C}.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} \mid \mathit{inl}\ \mathfrak{C}\\
&\quad \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 \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ \mathfrak{C}\\
&\quad \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}$$

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 5. 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).

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

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

$$\text{(Type-lam)}\ \frac{\Gamma, x : \tau \vdash t : \tau' \qquad \mathtt{ftv}(\tau) = \varnothing}{\Gamma \vdash \lambda x : \tau.\, t : \tau \to \tau'} \qquad \text{(Type-pair)}\ \frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash t' : \tau'}{\Gamma \vdash \langle t, t'\rangle : \tau \times \tau'} \qquad \text{(Type-inl)}\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathit{inl}\ t : \tau \uplus \tau'}$$

$$\overset{\text{(Type-inr)}}{\begin{array}{c}\Gamma \vdash t : \tau' \\ \hline \Gamma \vdash \mathit{inr}\ t : \tau \uplus \tau'\end{array}} \qquad \overset{\text{(Type-app)}}{\begin{array}{c}\Gamma \vdash t : \tau' \to \tau \quad \Gamma \vdash t' : \tau' \\ \hline \Gamma \vdash t\ t' : \tau\end{array}} \qquad \overset{\text{(Type-p1)}}{\begin{array}{c}\Gamma \vdash t : \tau \times \tau' \\ \hline \Gamma \vdash t.\mathit{1} : \tau\end{array}} \qquad \overset{\text{(Type-p2)}}{\begin{array}{c}\Gamma \vdash t : \tau' \times \tau \\ \hline \Gamma \vdash t.\mathit{2} : \tau\end{array}}$$

$$\overset{\text{(Type-case)}}{\begin{array}{c}\Gamma \vdash t : \tau' \uplus \tau'' \quad \Gamma, x_1 : \tau' \vdash t' : \tau \quad \Gamma, x_2 : \tau'' \vdash t'' : \tau \\ \hline \Gamma \vdash \mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto t' \mid \mathit{inr}\ x_2 \mapsto t'' : \tau\end{array}} \qquad \overset{\text{(Type-seq)}}{\begin{array}{c}\Gamma \vdash t : \mathit{Unit} \quad \Gamma \vdash t' : \tau \\ \hline \Gamma \vdash t; t' : \tau\end{array}}$$

$$\overset{\text{(Type-if)}}{\begin{array}{c}\Gamma \vdash t : \mathit{Bool} \quad \Gamma \vdash t' : \tau \quad \Gamma \vdash t'' : \tau \\ \hline \Gamma \vdash \mathit{if}\ t\ \mathit{then}\ t'\ \mathit{else}\ t'' : \tau\end{array}} \qquad \overset{(\lambda^{\mathsf{fx}}\text{-Type-fix})}{\begin{array}{c}\Gamma \vdash \mathsf{t} : (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2 \\ \hline \Gamma \vdash \mathsf{fix}_{\tau_1 \to \tau_2}\ \mathsf{t} : \tau_1 \to \tau_2\end{array}}$$

$$\overset{(\lambda^{\mu}_{\mathbf{I}}\text{-Type-fold})}{\begin{array}{c}\Gamma \vdash \mathbf{t} : \boldsymbol{\tau}[\boldsymbol{\mu\alpha.\tau}/\boldsymbol{\alpha}] \\ \hline \Gamma \vdash \mathbf{fold}_{\boldsymbol{\mu\alpha.\tau}}\ \mathbf{t} : \boldsymbol{\mu\alpha.\tau}\end{array}} \qquad \overset{(\lambda^{\mu}_{\mathbf{I}}\text{-Type-unfold})}{\begin{array}{c}\Gamma \vdash \mathbf{t} : \boldsymbol{\mu\alpha.\tau} \\ \hline \Gamma \vdash \mathbf{unfold}_{\boldsymbol{\mu\alpha.\tau}}\ \mathbf{t} : \boldsymbol{\tau}[\boldsymbol{\mu\alpha.\tau}/\boldsymbol{\alpha}]\end{array}} \qquad \overset{(\lambda^{\mu}_{E}\text{-Type-eq})}{\begin{array}{c}\Gamma \vdash t : \mu\alpha.\tau \quad \mu\alpha.\tau \doteq \sigma \\ \hline \Gamma \vdash t : \sigma\end{array}}$$

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'. These typing rules are unsurprising, so we omit them for space constraints.

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 (Rules $\doteq$-refl to $\doteq$-trans) whose derivations we have proved from the other rules.

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

$$\overset{(\doteq\text{-prim})}{\begin{array}{c}\iota = \mathit{Unit} \ \vee\ \iota = \mathit{Bool} \\ \hline\hline \iota \doteq \iota\end{array}} \qquad \overset{(\doteq\text{-var})}{\begin{array}{c} \\ \hline\hline \alpha \doteq \alpha\end{array}} \qquad \overset{(\doteq\text{-bin})}{\begin{array}{c}\star \in \{\to, \times, \uplus\} \quad \tau_1 \doteq \sigma_1 \quad \tau_2 \doteq \sigma_2 \\ \hline\hline \tau_1 \star \tau_2 \doteq \sigma_1 \star \sigma_2\end{array}} \qquad \overset{(\doteq\text{-}\mu_l)}{\begin{array}{c}\tau[\mu\alpha.\tau/\alpha] \doteq \sigma \\ \tau \text{ contractive in } \alpha \\ \hline\hline \mu\alpha.\tau \doteq \sigma\end{array}}$$

$$\overset{(\doteq\text{-}\mu_r)}{\begin{array}{c}\mathtt{lmc}\,(\tau) = 0 \quad \tau \doteq \sigma[\mu\alpha.\sigma/\alpha] \quad \sigma \text{ contractive in } \alpha \\ \hline\hline \tau \doteq \mu\alpha.\sigma\end{array}} \qquad \overset{(\doteq\text{-refl})}{\begin{array}{c} \\ \hline\hline \tau \doteq \tau\end{array}} \qquad \overset{(\doteq\text{-symm})}{\begin{array}{c}\sigma \doteq \tau \\ \hline\hline \tau \doteq \sigma\end{array}} \qquad \overset{(\doteq\text{-trans})}{\begin{array}{c}\tau \doteq \sigma \quad \sigma \doteq \tau' \\ \hline\hline \tau \doteq \tau'\end{array}}$$

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. $\mathtt{lmc}\,(\mathit{Unit}) = 0$, $\mathtt{lmc}\,(\mu\alpha.\,\alpha \uplus \mathit{Unit}) = 1$). 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 $\mathtt{lmc}\,(\tau)$ times. This restriction is harmless, since non-contractive recursive types are not inhabited by any value (Lemma 1).

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

2.3 Dynamic Semantics

All our languages are given a 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.

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

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

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

$$\text{(Eval-if)}\ \frac{v = true \lor false}{if\ v\ then\ t_{true}\ else\ t_{false} \hookrightarrow_p t_v} \qquad (\lambda^{\mathsf{fx}}\text{-Eval-fix})\ \frac{}{\mathsf{fix}_{\tau\to\tau}\ (\lambda \mathsf{x} : \tau.\, \mathsf{t}) \hookrightarrow_p \mathsf{t}\,[\mathsf{fix}_{\tau\to\tau}\ \lambda \mathsf{x} : \tau.\, \mathsf{t}/\mathsf{x}]}$$

$$(\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}\text{-Eval-fold})\ \frac{}{\mathbf{unfold}_{\mu\alpha.\tau}\ (\mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v}) \hookrightarrow_{\mathbf{p}} \mathbf{v}}$$

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

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.

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 $\triangleright$ 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. Both these relation use notation $t \hookrightarrow^{n} v$, which indicates that term t reduces to value v in n steps or less. The former defines that a source term approximates a target term if termination 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 $\gtrsim$.

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

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

$$O(W)_{\lesssim} \stackrel{\text{def}}{=} \left\{(\mathsf{t}, \mathbf{t}) \;\middle|\; \text{if } lev(W) > n \text{ and } \mathsf{t} \hookrightarrow^{\mathsf{n}} \mathsf{v} \text{ then } \exists \mathbf{k}.\ \mathbf{t} \hookrightarrow^{\mathbf{k}} \mathbf{v}\right\}$$

$$O(W)_{\gtrsim} \stackrel{\text{def}}{=} \left\{(\mathsf{t}, \mathbf{t}) \;\middle|\; \text{if } lev(W) > n \text{ and } \mathbf{t} \hookrightarrow^{\mathbf{n}} \mathbf{v} \text{ then } \exists \mathsf{k}.\ \mathsf{t} \hookrightarrow^{\mathsf{k}} \mathsf{v}\right\}$$

$$O(W)_{\approx} \stackrel{\text{def}}{=} O(W)_{\lesssim} \cap O(W)_{\gtrsim}$$

Fig. 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.

Note that our logical relations (Figure 3) 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 $\mathtt{repEmul}^{\mathtt{fI}}\,(\cdot)$ converts a pseudo-type to an actual source type by replacing all occurrences of $EmulT$ with a concrete source type.[2] 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 a superset of the former. Function $\mathtt{fxToIs}\,(\cdot)$ converts a source pseudo-type into its target-level 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). Finally, function $\mathtt{oftype}^{\mathtt{fI}}\,(\cdot)$ checks that terms have the correct form according to the rules of syntactic typing (Section 2.2). Function $\mathtt{oftype}^{\mathtt{IE}}\,(\cdot)$ does the analogous syntactic typecheck but for terms of $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ and λ^{μ}_{E}.

The value relation $\mathcal{V}\,[\![\hat{\tau}]\!]_{\triangledown}$ is defined inductively on source pseudo-types and it is quite standard. $Unit$ 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. 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 $\rightarrow$ and the relation for these constructors recurses only at strictly-future worlds.

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 $\mathcal{E}\,[\![\hat{\tau}]\!]_{\triangledown}$ if plugging the terms in related evaluation contexts yields terms related according to the observation relation of the world. Relation $\mathcal{G}\,[\![\hat{\Gamma}]\!]_{\triangledown}$ 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:

$$LR^{\mathsf{fx}}_{\boldsymbol{\mu}\mathbf{I}} : \hat{\Gamma} \vdash \mathsf{t} \triangledown_n \mathbf{t} : \hat{\tau} \qquad\qquad LR^{\mathsf{fx}}_{\mu E} : \hat{\Gamma} \vdash \mathsf{t} \triangledown_n t : \hat{\tau} \qquad\qquad LR^{\boldsymbol{\mu}\mathbf{I}}_{\mu E} : \hat{\boldsymbol{\Gamma}} \vdash \mathbf{t} \triangledown_n t : \hat{\boldsymbol{\tau}}$$

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,

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

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

$$\mathtt{oftype}^{\mathtt{fI}}(\hat{\tau}) \overset{\text{def}}{=} \{(\mathsf{v}, \mathbf{v}) \mid \mathsf{v} \in \mathsf{oftype}(\hat{\tau}) \text{ and } \mathbf{v} \in \mathbf{oftype}(\mathtt{fxToIs}(\hat{\tau}))\}$$

$$\mathsf{oftype}(\hat{\tau}) \overset{\text{def}}{=} \left\{\mathsf{v} \;\middle|\; \varnothing \vdash \mathsf{v} : \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau})\right\} \quad \mathbf{oftype}(\boldsymbol{\tau}) \overset{\text{def}}{=} \{\mathbf{v} \mid \boldsymbol{\varnothing} \vdash \mathbf{v} : \boldsymbol{\tau}\}$$

$$\mathtt{repEmul}^{\mathtt{fI}}(\cdot) : \hat{\tau} \to \tau \text{ (see Section 3.3)} \qquad \mathtt{fxToIs}(\cdot) : \hat{\tau} \to \boldsymbol{\tau} \text{ (see Section 3.3)}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

$$LR^{\mathsf{fx}}_{\boldsymbol{\mu}\mathbf{I}} : \vdash \mathfrak{C} \triangledown \boldsymbol{\mathfrak{C}} : \hat{\Gamma}, \hat{\tau} \to \hat{\Gamma}', \hat{\tau}' \quad LR^{\mathsf{fx}}_{\mu E} : \vdash \mathfrak{C} \triangledown \mathfrak{C} : \hat{\Gamma}, \hat{\tau} \to \hat{\Gamma}', \hat{\tau}' \quad LR^{\boldsymbol{\mu}\mathbf{I}}_{\mu E} : \vdash \boldsymbol{\mathfrak{C}} \triangledown \mathfrak{C} : \hat{\boldsymbol{\Gamma}}, \hat{\boldsymbol{\tau}} \to \hat{\boldsymbol{\Gamma}}', \hat{\boldsymbol{\tau}}'$$

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 ($\hat{\tau}'$).[3]

[3] The interested reader will find the formalisation of these definitions in the online appendix [Patrignani et al. 2020].

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 2). We define termination of a term t as reduction to a value in some steps: $t\Downarrow \overset{\text{def}}{=} \exists n, v.\ t \hookrightarrow^{n} v$.

Lemma 2 (Adequacy for $\approx$).

if $\varnothing \vdash \mathsf{t} \lesssim_n \mathsf{t} : \tau$ *and* $\mathsf{t} \hookrightarrow^{\mathsf{m}} \mathsf{v}$ *with* $n \geq m$ *then* $\mathsf{t}\Downarrow$ *if* $\varnothing \vdash \mathsf{t} \gtrsim_n \mathsf{t} : \tau$ *and* $\mathsf{t} \hookrightarrow^{\mathsf{m}} \mathsf{v}$ *with* $n \geq m$ *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 1 (Contextual Equivalence)**.**

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

Quantifying over all contexts in Definition 1 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 2 (Fully-abstract compilation)**.**

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

For simplicity, we instantiate Definition 2 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].

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 program. 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 others (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 2), 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.

$$\begin{tikzcd}
 & & t_1 \overset{?}{\simeq}_{\mathrm{ctx}} t_2 & & \\
\arrow[dddd, Leftarrow, "\text{reflection direction}"] & \mathfrak{C}\,[t_1]\Downarrow \arrow[rr, Rightarrow, "?"] \arrow[d, Rightarrow, "(1)"', "\begin{array}{l}\mathfrak{C} \approx \llbracket\mathfrak{C}\rrbracket\\ t_1 \approx \llbracket t_1\rrbracket\end{array}"] & & \mathfrak{C}\,[t_2]\Downarrow & \\
 & \llbracket\mathfrak{C}\rrbracket\,[\llbracket t_1\rrbracket]\Downarrow \arrow[rr, Rightarrow, "(2)"] & & \llbracket\mathfrak{C}\rrbracket\,[\llbracket t_2\rrbracket]\Downarrow \arrow[u, Rightarrow, "(3)", "\begin{array}{l}\mathfrak{C} \approx \llbracket\mathfrak{C}\rrbracket\\ t_2 \approx \llbracket t_2\rrbracket\end{array}"'] & \\
 & & \llbracket t_1\rrbracket \simeq_{\mathrm{ctx}} \llbracket t_2\rrbracket & & \\
 & & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
\arrow[dddd, Rightarrow, "\text{preservation direction}"] & & t_1 \simeq_{\mathrm{ctx}} t_2 & & \\
 & \langle\!\langle\mathfrak{C}\rangle\!\rangle_n\,[t_1]\Downarrow_{\_} \arrow[rr, Rightarrow, "(2)"'] & & \langle\!\langle\mathfrak{C}\rangle\!\rangle_n\,[t_2]\Downarrow_{\_} \arrow[d, Rightarrow, "(3)"', "\begin{array}{l}\langle\!\langle\mathfrak{C}\rangle\!\rangle_n \lesssim_{\_} \mathfrak{C}\\ t_2 \lesssim_{\_} \llbracket t_2\rrbracket\end{array}"] & \\
 & \mathfrak{C}\,[\llbracket t_1\rrbracket]\Downarrow_n \arrow[u, Rightarrow, "(1)", "\begin{array}{l}\langle\!\langle\mathfrak{C}\rangle\!\rangle_n \gtrsim_n \mathfrak{C}\\ t_1 \gtrsim_{\_} \llbracket t_1\rrbracket\end{array}"'] \arrow[rr, Rightarrow, "?"] & & \mathfrak{C}\,[\llbracket t_2\rrbracket]\Downarrow_{\_} & \\
 & & \llbracket t_1\rrbracket \overset{?}{\simeq}_{\mathrm{ctx}} \llbracket t_2\rrbracket & & \\
 & & & &
\end{tikzcd}$$

Fig. 4. Diagram breakdown of the reflection (left) and preservation (right) proofs 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 $t \gtrsim_n t$ is used to know that if t terminates in n steps in the target, then t also terminates (in arbitrary steps) in the source. The converse, $t \lesssim_n t$ is used to know that if t terminates in n steps in the source, then t also terminates (again in arbitrary steps) in the target. We start with source term t approximating (in both directions) its compilation $\llbracket t\rrbracket$. Then, to prove target contextual equivalence (the ?-decorated equivalence), we start by assuming that a target context $\mathfrak{C}$ linked with $\llbracket t_1\rrbracket$ terminates in some steps ($\Downarrow_n$). Eventually, we need to show that the same target context linked with $\llbracket t_2\rrbracket$ 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. 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 $\llbracket t_2\rrbracket$ 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, we do not know at runtime the level of unfolding we are dealing with, i.e., we cannot inspect n at runtime. Thus, we need a way to model failure (as a sort of catchable exception), or, having reached more than n unfoldings, because in that case we need to diverge.[4] 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.

We make these intuitions concrete and formalise the type for $\lambda^{\mu}_{\mathbf{I}}$ values backtranslated into λ^{fx} as $\mathsf{BtT}^{\mathsf{fI}}_{\mathsf{n};\tau}$ in Figure 5 (for Backtranslation Type; the superscript indicates the languages involved, the subscripts are effectively parameters of this type). Type $\mathsf{BtT}^{\mathsf{fI}}_{\mathsf{n};\tau}$ is defined inductively on n and it backtranslates the structure of τ in the source type it creates. At no steps (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.

$$\mathsf{BtT}^{\mathsf{fI}}_{0;\tau} \overset{\text{def}}{=} \mathsf{Unit}$$

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

$$\mathsf{BtT}^{\mathsf{fE}}_{0;\tau} \overset{\text{def}}{=} \mathsf{Unit}$$

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

$$\mathbf{BtT}^{\mathbf{IE}}_{\mathbf{n};\tau} \overset{\text{def}}{=} \text{as } \mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\tau}$$

Fig. 5. The type of backtranslated terms (excerpts).

The type of λ^{μ}_{E} terms backtranslated in λ^{fx} ($\mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\tau}$, 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}: $\mu\alpha.\ \mathit{Unit} \uplus (\mathit{Bool} \times \alpha)$ (which we dub List_B) and its first unfolding $\mathit{Unit} \uplus (\mathit{Bool} \times \mathit{List}_B)$ (which we dub List^1_B). The backtranslation (for $n = 3$) for this type is the following: $\mathsf{BtT}^{\mathsf{fE}}_{3;\mathit{List}_B} = \mathsf{BtT}^{\mathsf{fE}}_{3;\mathit{Unit}\uplus(\mathit{Bool}\times\mathit{List}_B)} = \cdots = ((\mathit{Unit} \uplus \mathit{Unit}) \uplus (((\mathit{Bool} \uplus \mathit{Unit}) \times \mathsf{BtT}^{\mathsf{fE}}_{1;\mathit{List}_B}) \uplus \mathit{Unit})) \uplus \mathit{Unit}$.[5] Formally, the measure that ensures that this type is well founded is the precision n together with $\mathtt{lmc}$ ($\mu\alpha.\,\tau$) 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 $\lambda^{\mu}_{\mathbf{I}}$ ($\mathbf{BtT}^{\mathbf{IE}}_{\mathbf{n};\tau}$) is the same as the one just presented ($\mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\tau}$). Intuitively, this is because the n-level deep unfolding of τ in the backtranslation type does not rely on recursive types in $\lambda^{\mu}_{\mathbf{I}}$.

3.2.1 Working with the Backtranslation Type.

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

[4] Recall that one of the two terms ($[\![\mathsf{t}_1]\!]$ and $[\![\mathsf{t}_2]\!]$) is guaranteed to terminate within n steps, so if that does not happen, the other term needs to diverge. This ensures that contextually-equivalent terms remain equivalent, i.e., they equi-terminate.

[5] Where the first $\mathit{Unit} \uplus \mathit{Unit}$ is the result of $\mathsf{BtT}^{\mathsf{fE}}_{2;\mathit{Unit}}$ and the $\mathit{Bool} \uplus \mathit{Unit}$ is the result of $\mathsf{BtT}^{\mathsf{fE}}_{1;\mathit{Bool}}$.

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

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

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

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

$$\mathsf{case}^{\mathsf{fE}}_{\mathsf{n};\tau} \ = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n+1};\tau}.\, \mathsf{case\ x\ of\ inl\ x_1} \mapsto \mathsf{x_1} \mid \mathsf{inr\ x_2} \mapsto \mathsf{omega}_{\mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\tau}} \qquad \tau \neq \mu\alpha.\,\tau$$

3.2.2 Increasing and Decreasing the Approximation Level. 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]. Consider $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ term $\boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}.\,\mathbf{inr\ x}$, intuitively its backtranslation (for a sufficiently-large n) is: $\mathsf{inl}\ \lambda\mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n-1};\tau}.\,\mathsf{inl\ inr\ x}$ If we try to typecheck this, though, we see that x has type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n-1};\tau}$ while it is expected to have type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n-2};\tau}$, 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 $n - \mathit{1}$ steps, x needs to only be precise up to $n - \mathit{2}$ steps. Thus, it is safe to throw away one level of precision and *downgrade* x from type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n-1};\tau}$ to $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n-2};\tau}$.

However, downgrading is not sufficient. Consider how we can downgrade a value of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\tau\to\tau'}$ to one of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau\to\tau'}$. We need to convert a function of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau} \to \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau'}$ into one of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\tau} \to \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\tau'}$. To do this, we need to upgrade the argument value of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}$ into one of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\tau}$. Fortunately, this does not mean we need to magically improve the approximation precision of the value concerned. Type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}$ 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 $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}$.

Finally, another reason we need to upgrade and downgrade a value is that type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}$ 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 $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}$ will be downgraded to type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{m};\tau}$. Dually, there will be cases where some values need to be upgraded.

$$\boxed{\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau} : \mathsf{BtT}^{\mathsf{fl}}_{n;\tau} \to \mathsf{BtT}^{\mathsf{fl}}_{n+1;\tau} \quad \text{and} \quad \mathsf{downgrade}^{\mathsf{fl}}_{n;\tau} : \mathsf{BtT}^{\mathsf{fl}}_{n+1;\tau} \to \mathsf{BtT}^{\mathsf{fl}}_{n;\tau}}$$

$$\begin{aligned}
\mathsf{upgrade}^{\mathsf{fl}}_{0;\tau} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{0;\tau}.\,\mathsf{unk}\\
\mathsf{upgrade}^{\mathsf{fl}}_{n+1;\mathbf{Unit}} &= \lambda \mathsf{x} : \mathsf{Unit} \uplus \mathsf{Unit}.\,\mathsf{x}\\
\mathsf{upgrade}^{\mathsf{fl}}_{n+1;\tau\times\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{n+1;\tau\times\tau'}.\\
&\quad \mathsf{case}\ \mathsf{x}\ \mathsf{of}\\
&\quad\left|\begin{array}{l}\mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl} \left\langle \begin{array}{l}\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau}\ x_1.1,\\ \mathsf{upgrade}^{\mathsf{fl}}_{n;\tau'}\ x_1.2\end{array}\right\rangle\\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2\end{array}\right.\\
\mathsf{upgrade}^{\mathsf{fl}}_{n+1;\tau\to\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{n+1;\tau\to\tau'}.\\
&\quad \mathsf{case}\ \mathsf{x}\ \mathsf{of}\\
&\quad\left|\begin{array}{l}\mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl} \begin{array}{l}\lambda \mathsf{z} : \mathsf{BtT}^{\mathsf{fl}}_{n+1;\tau}.\,\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau'}\\ (\mathsf{x}_1\ (\mathsf{downgrade}^{\mathsf{fl}}_{n;\tau}\ \mathsf{z}))\end{array}\\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2\end{array}\right.\\
\mathsf{upgrade}^{\mathsf{fl}}_{n+1;\mathsf{d}\mu\alpha.\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{n+1;\mu\alpha.\tau'}.\\
&\quad \mathsf{case}\ \mathsf{x}\ \mathsf{of}\\
&\quad\left|\begin{array}{l}\mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau'[\mu\alpha.\tau'/\alpha]}\ \mathsf{x}_1)\\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2\end{array}\right.
\end{aligned}$$

$$\mathsf{unk} = \mathsf{inr}\ \mathsf{unit}$$

$$\begin{aligned}
\mathsf{downgrade}^{\mathsf{fl}}_{0;\tau} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{0;\tau}.\,\mathsf{unit}\\
\mathsf{downgrade}^{\mathsf{fl}}_{n+1;\mathbf{Unit}} &= \lambda \mathsf{x} : \mathsf{Unit} \uplus \mathsf{Unit}.\,\mathsf{x}\\
\mathsf{downgrade}^{\mathsf{fl}}_{n+1;\tau\to\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{n+2;\tau\to\tau'}.\\
&\quad \mathsf{case}\ \mathsf{x}\ \mathsf{of}\\
&\quad\left|\begin{array}{l}\mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl} \begin{array}{l}\lambda z : \mathsf{BtT}^{\mathsf{fl}}_{n;\tau}.\,\mathsf{downgrade}^{\mathsf{fl}}_{n;\tau'}\\ (x_1\ (\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau}\ z))\end{array}\\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2\end{array}\right.\\
\mathsf{downgrade}^{\mathsf{fl}}_{n+1;\mu\alpha.\tau'} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{n+2;\mu\alpha.\tau'}.\\
&\quad \mathsf{case}\ \mathsf{x}\ \mathsf{of}\\
&\quad\left|\begin{array}{l}\mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{downgrade}^{\mathsf{fl}}_{n;\tau'[\mu\alpha.\tau'/\alpha]}\ \mathsf{x}_1)\\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{x}_2\end{array}\right.
\end{aligned}$$

$$\begin{aligned}
\mathsf{upgrade}^{\mathsf{fE}}_{n+1;\mu\alpha.\tau} &= \mathsf{upgrade}^{\mathsf{fE}}_{n+1;\tau[\mu\alpha.\tau/\alpha]} & \mathsf{downgrade}^{\mathsf{fE}}_{n+1;\mu\alpha.\tau} &= \mathsf{downgrade}^{\mathsf{fE}}_{n+1;\tau[\mu\alpha.\tau/\alpha]}\\
\mathsf{upgrade}^{\mathsf{fE}}_{n;\tau} &= \text{as above} & \mathsf{downgrade}^{\mathsf{fE}}_{n;\tau} &= \text{as above}
\end{aligned}$$

$$\begin{aligned}
\mathbf{upgrade}^{\mathbf{IE}}_{\mathbf{n};\tau} &= \text{as } \mathsf{upgrade}^{\mathsf{fE}}_{n;\tau} & \mathbf{downgrade}^{\mathbf{IE}}_{\mathbf{n};\tau} &= \text{as } \mathsf{downgrade}^{\mathsf{fE}}_{n;\tau}
\end{aligned}$$

Fig. 6. Definition of the upgrade and downgrade functions (excerpts).

Functions $\mathsf{upgrade}^{\mathsf{fl}}_{\cdot}$ and $\mathsf{downgrade}^{\mathsf{fl}}_{\cdot}$ perform what we just discussed; their types and formalisation is presented in Figure 6 (partially for space constraints). The cases for Unit and Bool are optimised based on the fact that $\mathsf{BtT}^{\mathsf{fl}}_{n;\mathbf{Unit}} = \mathsf{BtT}^{\mathsf{fl}}_{m;\mathbf{Unit}}$ (resp. $\mathsf{BtT}^{\mathsf{fl}}_{n;\mathbf{Bool}} = \mathsf{BtT}^{\mathsf{fl}}_{m;\mathbf{Bool}}$) 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 $\mathsf{downgrade}^{\mathsf{fl}}_{n;\tau}$ $(\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau}\ \mathsf{t})$ reduces to t, term $\mathsf{upgrade}^{\mathsf{fl}}_{n;\tau}$ $(\mathsf{downgrade}^{\mathsf{fl}}_{n;\tau}\ \mathsf{t})$ does not reduce to t because information was lost (Example 1).

Example 1 (Upgrading after downgrading forgets information)**.** Consider the following term: $\mathsf{downgrade}^{\mathsf{fl}}_{0;\mathbf{Bool}}$ inl true, which reduces to unit. If we apply $\mathsf{upgrade}^{\mathsf{fl}}_{0;\mathbf{Bool}}$ 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 $\mathsf{BtT}^{\mathsf{fE}}$ and $\mathbf{BtT}^{\mathbf{IE}}$. As mentioned, the main difference between these last two backtranslation types and $\mathsf{BtT}^{\mathsf{fl}}$ 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.

$$\boxed{\text{in-dn}^{\text{fl}}_{\text{n};\tau} \quad \text{and} \quad \text{case-up}^{\text{fl}}_{\text{n};\tau}}$$

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

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

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

$$\text{case-up}^{\text{fl}}_{\text{n};\tau} = \lambda \text{x} : \text{BtT}^{\text{fl}}_{\text{n};\tau}.\, \text{case}^{\text{fl}}_{\text{n};\tau}\left(\text{upgrade}^{\text{fl}}_{\text{n};\tau}\ (\text{x})\right)$$

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

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

Fig. 7. Compacted functions used to manipulate backtranslated values.

In the backtranslation, we generally use creation of a backtranslated value together with a $\text{downgrade}^{\text{fl}}$, while we use destruction of backtranslated values together with an $\text{upgrade}^{\text{fl}}$. Thus, we provide compacted functions that do exactly this, $\text{in-dn}^{\text{fl}}_{\text{n};\tau}$ and $\text{case-up}^{\text{fl}}_{\text{n};\tau}$ (Figure 7). 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 $\text{BtT}^{\text{fl}}_{\text{n};\mathbf{Unit}}$. As for the previous helpers, the compacted versions that operate on terms of type $\text{BtT}^{\text{fE}}_{\text{n};\mu\alpha.\tau}$ (and $\mathbf{BtT}^{\mathbf{IE}}_{\mathbf{n};\mu\alpha.\tau}$) are different. Since there is no destructor for $\text{BtT}^{\text{fE}}_{\text{n};\mu\alpha.\tau}$, 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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ type **Unit**, that is backtranslated (at any approximation $n > 0$) into $\text{BtT}^{\text{fl}}_{\text{n};\mathbf{Unit}}$, i.e., $\text{Unit} \uplus \text{Unit}$. Value **unit** should normally be backtranslated to inl unit. Following the value relation in $LR^{\text{fx}}_{\boldsymbol{\mu}\mathbf{I}}$ 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* .

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 $\text{EmulT}^{\text{fl}}_{\text{n};\text{p};\tau}$, the type of backtranslated $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ terms into λ^{fx} (top of Figure 8). $\text{EmulT}^{\text{fl}}_{\text{n};\text{p};\tau}$ 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., inr unit)

$$\mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fl}}_{0;\mathtt{imprecise};\boldsymbol{\tau}} \rrbracket_{\triangledown} \overset{\text{def}}{=} \{(W,\mathsf{v},\mathbf{v}) \mid \mathsf{v} = \mathsf{unit}\} \qquad \mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fl}}_{0;\mathtt{precise};\boldsymbol{\tau}} \rrbracket_{\triangledown} \overset{\text{def}}{=} \varnothing$$

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

$$\mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fE}}_{0;\mathtt{imprecise};\tau} \rrbracket_{\triangledown} \overset{\text{def}}{=} \{(W,\mathsf{v},v) \mid \mathsf{v} = \mathsf{unit}\} \qquad \mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fE}}_{0;\mathtt{precise};\tau} \rrbracket_{\triangledown} \overset{\text{def}}{=} \varnothing$$

$$\mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fE}}_{n+1;p;\tau} \rrbracket_{\triangledown} \overset{\text{def}}{=} \left\{ (W,\mathsf{v},v) \mid \mathsf{v} \in \mathsf{oftype}\left(\mathsf{EmulT}^{\mathsf{fE}}_{n+1;p;\tau}\right) \text{ and } v \in \mathit{oftype}(\tau) \text{ and } \begin{array}{l} \text{either } \cdot\ \mathsf{v} = \mathsf{inr\ unit} \text{ and } \mathsf{p} = \mathsf{imprecise} \\ \text{or } \cdot \begin{cases} \cdot\ \text{omitted parts are as above} \\ \cdot\ \tau = \mu\alpha.\,\tau \text{ and } \tau \text{ contractive in } \alpha \text{ and } (W,\mathsf{v},v) \in \mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fl}}_{n+1;p;\tau[\mu\alpha.\tau/\alpha]} \rrbracket_{\triangledown} \end{cases} \end{array} \right\}$$

$$\mathcal{V}\llbracket \mathbf{EmulT}^{\mathbf{IE}}_{\mathbf{n;p};\tau} \rrbracket_{\triangledown} \text{ is defined analogously to } \mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fE}}_{n;p;\tau} \rrbracket_{\triangledown}$$

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{repEmul}^{\mathtt{fI}}\left(\mathsf{EmulT}^{\mathsf{fl}}_{n;p;\boldsymbol{\tau}}\right) &=& \mathsf{BtT}^{\mathsf{fl}}_{n;\boldsymbol{\tau}} & \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_1} \to \hat{\tau_2}) &=& \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_1}) \to \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_2}) \\ \mathtt{repEmul}^{\mathtt{fI}}(\mathsf{Bool}) &=& \mathsf{Bool} & \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_1} \times \hat{\tau_2}) &=& \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_1}) \times \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_2}) \\ \mathtt{repEmul}^{\mathtt{fI}}(\mathsf{Unit}) &=& \mathsf{Unit} & \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_1} \uplus \hat{\tau_2}) &=& \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_1}) \uplus \mathtt{repEmul}^{\mathtt{fI}}(\hat{\tau_2}) \\ \mathtt{fxToIs}\left(\mathsf{EmulT}^{\mathsf{fl}}_{n;p;\boldsymbol{\tau}}\right) &=& \boldsymbol{\tau} & \mathtt{fxToIs}(\hat{\tau_1} \to \hat{\tau_2}) &=& \mathtt{fxToIs}(\hat{\tau_1}) \boldsymbol{\to} \mathtt{fxToIs}(\hat{\tau_2}) \\ \mathtt{fxToIs}(\mathsf{Unit}) &=& \mathbf{Unit} & \mathtt{fxToIs}(\hat{\tau_1} \times \hat{\tau_2}) &=& \mathtt{fxToIs}(\hat{\tau_1}) \boldsymbol{\times} \mathtt{fxToIs}(\hat{\tau_2}) \\ \mathtt{fxToIs}(\mathsf{Bool}) &=& \mathbf{Bool} & \mathtt{fxToIs}(\hat{\tau_1} \uplus \hat{\tau_2}) &=& \mathtt{fxToIs}(\hat{\tau_1}) \boldsymbol{\uplus} \mathtt{fxToIs}(\hat{\tau_2}) \end{array}$$

$$\mathtt{repEmul}^{\mathtt{fE}}(\cdot) : \hat{\tau} \to \tau \qquad \mathtt{repEmul}^{\mathtt{IE}}(\cdot) : \hat{\boldsymbol{\tau}} \to \tau \qquad \mathtt{fxToEq}(\cdot) : \hat{\tau} \to \tau \qquad \mathtt{isToEq}(\cdot) : \hat{\boldsymbol{\tau}} \to \tau$$

Fig. 8. Missing bits of the logical relation: value relation for backtranslation type (excerpts). Note that p can be either $\mathtt{precise}$ or $\mathtt{imprecise}$ in the second clause (the 'or') of the $\mathsf{n}+1$ case.

in the backtranslation Thus, $\mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fl}}_{n;p;\boldsymbol{\tau}} \rrbracket_{\triangledown}$ regulates how these 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, $\mathcal{V}\llbracket \mathsf{EmulT}^{\mathsf{fl}}_{n;p;\boldsymbol{\tau}} \rrbracket_{\triangledown}$ 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 $\mathcal{V}\ \llbracket \mathsf{EmulT}^{\mathsf{fI}}_{\mathsf{n;p;}\boldsymbol{\tau}} \rrbracket_{\triangledown}$ within depth n, i.e., we cannot reach unk in the steps of the world.

The pseudotype for the λ^{μ}_{E} to λ^{fx} backtranslation ($\mathsf{EmulT}^{\mathsf{fE}}_{\cdot}$) follows the same pattern as $\mathsf{BtT}^{\mathsf{fE}}_{\cdot}$: it does not lose a step in the $\mu\alpha.\,\tau$ case (Figure 8). 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 ($\mathbf{EmulT}^{\mathbf{IE}}_{\cdot}$) follows the same approach as $\mathsf{EmulT}^{\mathsf{fE}}_{\cdot}$.

Finally, we can define function $\mathtt{repEmul}^{\mathtt{fI}}\,(\cdot)$ that translate from source pseudo-types into plain source types and function $\mathtt{fxToIs}\,(\cdot)$, that translates source pseudotypes into target types. We present the formalisation for the case for source types being λ^{fx} types and target types being $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ types. As expected, these functions exists for all backtranslations and they follow the same pattern presented here; for completeness 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 9) 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 $\mathsf{fix}_{\cdot}$ 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 4 to 6 below) is proven via a series of standard compatibility lemmas (Lemma 3, 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 3 (Compatibility for λ). *if* $\Gamma, \mathsf{x} : \tau' \vdash \mathsf{t} \mathrel{\triangledown_n} \mathbf{t} : \tau$ *then* $\Gamma \vdash \lambda\mathsf{x} : \tau'.\,\mathsf{t} \mathrel{\triangledown_n} \boldsymbol{\lambda}\mathbf{x} : \boldsymbol{\tau}'.\,\mathbf{t} : \tau' \to \tau$

Lemma 4 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}$ is semantics preserving). *if* $\Gamma \vdash \mathsf{t} : \tau$ *then* $\Gamma \vdash \mathsf{t} \mathrel{\triangledown_n} \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}} : \tau$

Lemma 5 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ is semantics preserving). *if* $\Gamma \vdash \mathsf{t} : \tau$ *then* $\Gamma \vdash \mathsf{t} \mathrel{\triangledown_n} \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : \tau$

Lemma 6 ($\llbracket\cdot\rrbracket^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mu}_{E}}$ is semantics preserving). *if* $\boldsymbol{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau}$ *then* $\boldsymbol{\Gamma} \vdash \mathbf{t} \mathrel{\triangledown_n} \llbracket\mathbf{t}\rrbracket^{\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}}_{\lambda^{\mu}_{E}} : \boldsymbol{\tau}$

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 (Lemma 7).

$$\boxed{\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} : \mathsf{t} \to \mathbf{t} \quad \text{and} \quad \llbracket\cdot\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}} : \mathbf{t} \to t \quad \text{and} \quad \llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : \mathsf{t} \to t}$$

$$\llbracket\mathsf{unit}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \mathbf{unit} \qquad \llbracket\lambda\mathsf{x} : \tau.\,\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \boldsymbol{\lambda}\mathbf{x} : \llbracket\tau\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}.\,\llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} \qquad \llbracket\mathsf{t}.1\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}.\mathbf{1} \qquad \llbracket\mathsf{x}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \mathbf{x}$$

$$\llbracket\mathsf{true}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \mathbf{true} \qquad \llbracket\mathsf{t}\ \mathsf{t}'\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\ \llbracket\mathsf{t}'\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} \qquad \llbracket\mathsf{t}.2\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}.\mathbf{2} \qquad \llbracket\mathsf{inl}\ \mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \mathbf{inl}\ \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$$

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

$$\llbracket\mathsf{case}\ \mathsf{t}\ \mathsf{of}\ \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{t}' \mid \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{t}''\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \mathbf{case}\ \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\ \mathbf{of}\ \mathbf{inl}\ \mathbf{x_1} \mapsto \llbracket\mathsf{t}'\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} \mid \mathbf{inr}\ \mathbf{x_2} \mapsto \llbracket\mathsf{t}''\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$$

$$\llbracket\mathsf{if}\ \mathsf{t}\ \mathsf{then}\ \mathsf{t}'\ \mathsf{else}\ \mathsf{t}''\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} = \mathbf{if}\ \llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\ \mathbf{then}\ \llbracket\mathsf{t}'\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\ \mathbf{else}\ \llbracket\mathsf{t}''\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$$

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

$$\text{omitted rules are as above} \qquad \llbracket\mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{t}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}} = \llbracket\mathbf{t}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}} \qquad \llbracket\mathbf{unfold}_{\mu\alpha.\tau}\ \mathbf{t}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}} = \llbracket\mathbf{t}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}}$$

$$\llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} = \left\llbracket\llbracket\mathsf{t}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}}, \text{ i.e., as above, without } \mathbf{fold}/\mathbf{unfold} \text{ annotations in the compilation of } \mathsf{fix}$$

Fig. 9. Definition of our compilers (excerpts).

LEMMA 7 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$ IS SEMANTICS PRESERVING FOR CONTEXTS).

if $\vdash \mathfrak{C} : \Gamma, \tau \to \Gamma', \tau'$ *then* $\vdash \mathfrak{C} \mathrel{\nabla_n} \llbracket\mathfrak{C}\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} : \Gamma, \tau \to \Gamma', \tau'$

With these results, we can already prove the reflection direction of fully-abstract compilation (Theorems 8 to 10). The proof follows the structure depicted in the left part of Figure 4.

Theorem 8 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$ reflects equivalence). If $\boldsymbol{\varnothing} \vdash \llbracket\mathsf{t}_1\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} \boldsymbol{\simeq_{\mathrm{ctx}}} \llbracket\mathsf{t}_2\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} : \llbracket\tau\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$ then $\varnothing \vdash \mathsf{t}_1 \simeq_{\mathrm{ctx}} \mathsf{t}_2 : \tau$

Theorem 9 ($\llbracket\cdot\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}}$ reflects equivalence). If $\varnothing \vdash \llbracket\mathbf{t_1}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} \llbracket\mathbf{t_2}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}} : \llbracket\boldsymbol{\tau}\rrbracket^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mu}_{E}}$ then $\boldsymbol{\varnothing} \vdash \mathbf{t_1} \boldsymbol{\simeq_{\mathrm{ctx}}} \mathbf{t_2} : \boldsymbol{\tau}$

Theorem 10 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ reflects equivalence). If $\varnothing \vdash \llbracket\mathsf{t}_1\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} \llbracket\mathsf{t}_2\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : \llbracket\tau\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ then $\varnothing \vdash \mathsf{t}_1 \simeq_{\mathrm{ctx}} \mathsf{t}_2 : \tau$

Since this last compiler is the composition of the other two, the proof of Theorem 10 trivially follows from composing the proofs of the other two compilers.

4.2 Backtranslations and Preservation of Fully-Abstract Compilation

Function $\mathsf{emulate}^{\mathsf{fl}}(\cdot)$ is responsible for translating a target term of type $\boldsymbol{\tau}$ into a source one of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\boldsymbol{\tau}}$ (Section 4.2.1) by relying on the machinery needed for working with $\mathsf{BtT}^{\mathsf{fl}}$ terms from Section 3.2. This function is easily extended to work with program contexts, producing contexts with hole of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\boldsymbol{\tau}}$. 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 $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\boldsymbol{\tau}}$ 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 $\mathsf{BtT}^{\mathsf{fl}}$ values, we need a function that wraps terms of an arbitrary type τ into a value of type $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\boldsymbol{\tau}}$. This function is called $\mathsf{inject}^{\mathsf{fl}}$ (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 $\mathsf{BtT}^{\mathsf{fl}}$).

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

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.

Function $\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}(\cdot)$ (Figures 10 and 11) is indexed by the approximation index n in order to know which $\mathsf{BtT}^{\mathsf{fl}}$-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 $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\boldsymbol{\tau}}$ and, in order to be well-typed, we $\mathsf{downgrade}^{\mathsf{fl}}$ that value by 1. Dually, emulating destructors for terms of type $\boldsymbol{\tau}$ requires upgrading the term for 1 level of precision because they are then destructed to access the underlying type. When emulating $\boldsymbol{\lambda}^{\boldsymbol{\mu}}_{\mathbf{I}}$ derivations into λ^{fx}, we need to consider the case when $\mathbf{fold}_{\boldsymbol{\mu\alpha.\tau}}$ and $\mathbf{unfold}_{\boldsymbol{\mu\alpha.\tau}}$ annotations are encountered. There, we know that the backtranslation will work with terms typed at the unfolding of $\boldsymbol{\mu\alpha.\tau}$, so we simply perform the recursive call and insert the appropriate helper function to ensure the resulting term is well-typed.

When emulating λ^{μ}_{E} derivations (in the other two emulates in Figure 10), we need to consider the case when term t is given type τ knowing it had type σ and that $\sigma\doteq\tau$ (Rule λ^{μ}_{E}-Type-eq). Here we rely on a crucial observation: given two equivalent types, their backtranslation types are *the same* (Theorem 11). To understand why this is the case, consider how the definition of $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}$ 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 $\tau\doteq\sigma$. With this fact, we can get away with just performing the recursive call on the sub-derivation for t at type σ.

Theorem 11 (Equivalent types are backtranslated to the same type). If $\tau\doteq\sigma$ then $\mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\tau}=\mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\sigma}$

Finally, consider $\mathbf{emulate}^{\mathbf{IE}}_{\cdot}(\cdot)$, 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 $\mathbf{BtT}^{\mathbf{IE}}_{\mathbf{n};\mu\alpha.\tau}$, which is $\mathbf{BtT}^{\mathbf{IE}}_{\mathbf{n};\tau[\mu\alpha.\tau/\alpha]}$ 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.

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

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

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

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

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

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

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

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

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\begin{array}{l} \mathbf{\Gamma} \vdash \mathbf{case\ t\ of} \\ \left|\begin{array}{l} \mathbf{inl\ x_1} \mapsto \mathbf{t}' \\ \mathbf{inr\ x_2} \mapsto \mathbf{t}'' \end{array}\right. \mathbf{:}\ \boldsymbol{\tau} \end{array}\right) \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{case}\ \left(\mathsf{case\text{-}up}^{\mathsf{fl}}_{\mathsf{n};\boldsymbol{\tau_1}\uplus\boldsymbol{\tau_2}}\ \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\tau_1} \uplus \boldsymbol{\tau_2}\right)\right) \\ \mathsf{of} \left|\begin{array}{l} \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\mathbf{\Gamma}, (\mathbf{x_1} : \boldsymbol{\tau_1}) \vdash \mathbf{t}' : \boldsymbol{\tau}\right) \\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\mathbf{\Gamma}, (\mathbf{x_2} : \boldsymbol{\tau_2}) \vdash \mathbf{t}'' : \boldsymbol{\tau}\right) \end{array}\right. \end{array}$$

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

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

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

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

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

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

$$\mathsf{emulate}^{\mathsf{fE}}_{\mathsf{n}}\left(\frac{\Gamma \vdash t : \tau \qquad \tau \mathrel{\dot{\triangleq}} \sigma}{\Gamma \vdash t : \sigma}\right) \stackrel{\text{def}}{=} \mathsf{emulate}^{\mathsf{fE}}_{\mathsf{n}}\left(\ \Gamma \vdash t : \tau\ \right) \qquad \mathsf{emulate}^{\mathsf{fE}}_{\mathsf{n}}\left(\cdots\right) \stackrel{\text{def}}{=} \begin{array}{l}\text{other cases} \\ \text{are as above}\end{array}$$

$$\mathbf{emulate}^{\mathbf{IE}}_{\mathbf{n}}\left(\cdots\right) \stackrel{\text{def}}{=} \text{as } \mathsf{emulate}^{\mathsf{fE}}_{\mathsf{n}}\left(\cdots\right)$$

Fig. 10. Emulation of target terms into source ones.

In order to state that $\mathsf{emulate}^{\mathsf{fl}}\left(\cdot\right)$ is correct, we rely on compatibility lemmas akin to those used for compiler correctness (recall Lemma 3). 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 $\mathsf{EmulT}^{\mathsf{fl}}$, 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 $\texttt{toEmul}\left(\cdot\right)$. 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

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

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\,([\cdot]) \overset{\text{def}}{=} [\cdot]$$

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\begin{pmatrix}\vdash \lambda \mathrm{x} : \tau' .\, \mathfrak{C} : \\ \Gamma'', \tau'' \to \Gamma, \tau' \to \tau\end{pmatrix} \overset{\text{def}}{=} \mathsf{in\text{-}dn}^{\mathsf{fl}}_{\mathsf{n};\tau\to\tau'} \ \left(\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau}.\, \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathfrak{C} : \Gamma'', \tau'' \to \Gamma, \mathrm{x} : \tau', \tau\right)\right)$$

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathfrak{C}\ \mathrm{t}_2 : \Gamma', \tau' \to \Gamma, \tau_2\right) \overset{\text{def}}{=} \begin{array}{l}\left(\mathsf{case\text{-}up}^{\mathsf{fl}}_{\mathsf{n};\tau'\to\tau}\ \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1 \to \tau_2\right)\right) \\ \left(\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\,(\Gamma \vdash \mathrm{t}_2 : \tau_1)\right)\end{array}$$

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathrm{t}_1\ \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_2\right) \overset{\text{def}}{=} \begin{array}{l}\left(\mathsf{case\text{-}up}^{\mathsf{fl}}_{\mathsf{n};\tau'\to\tau}\ \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\,(\Gamma \vdash \mathrm{t}_1 \colon \tau_1 \to \tau_2)\right) \\ \left(\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1\right)\right)\end{array}$$

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

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

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\begin{pmatrix}\vdash \langle \mathfrak{C}, \mathrm{t}_2\rangle : \\ \Gamma', \tau' \to \Gamma, \tau_1 \times \tau_2\end{pmatrix} \overset{\text{def}}{=} \mathsf{in\text{-}dn}^{\mathsf{fl}}_{\mathsf{n};\tau_1\times\tau_2} \ \left\langle \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_1\right), \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\,(\Gamma \vdash \mathrm{t}_2 : \tau_2)\right\rangle$$

$$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\begin{pmatrix}\vdash \langle \mathrm{t}_1, \mathfrak{C}\rangle : \\ \Gamma', \tau' \to \Gamma, \tau_1 \times \tau_2\end{pmatrix} \overset{\text{def}}{=} \mathsf{in\text{-}dn}^{\mathsf{fl}}_{\mathsf{n};\tau_1\times\tau_2} \ \left\langle \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\,(\Gamma \vdash \mathrm{t}_1 : \tau_1), \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\left(\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau_2\right)\right\rangle$$

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

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

Fig. 11. Emulation of target contexts into source ones (excerpts).

$lev(W) \leq n < m$) and $p = \mathtt{precise}$ or $\triangledown = \lesssim$ and $p = \mathtt{imprecise}$, for m being the approximation level of interest. Thus, a typical compatibility lemma for emulate looks like Lemma 12.

Lemma 12 (Compatibility for λ Emulation).

$$\begin{array}{ll} & \textit{if}\ (m > n\ \textit{and}\ p = \mathtt{precise})\ \textit{or}\ (\triangledown = \lesssim\ \textit{and}\ p = \mathtt{imprecise}) \\ \textit{then} & \textit{if}\ \mathtt{toEmul}_{\mathsf{m;p}}\,(\Gamma, \mathrm{x} : \tau) \vdash \mathsf{t} \ \triangledown_n\ \mathrm{t} : \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m;p};\tau'} \\ & \quad \textit{then}\ \mathtt{toEmul}_{\mathsf{m;p}}\,(\Gamma) \vdash \mathsf{in\text{-}dn}^{\mathsf{fl}}_{\mathsf{m};\tau\to\tau'}\ \left(\lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{m};\tau}.\,\mathsf{t}\right) \triangledown_n\ \lambda \mathrm{x} : \tau.\,\mathrm{t} : \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m;p};\tau\to\tau'} \end{array}$$

The compatibility lemma for terms typed using type equality (Lemma 13) 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 11, which proves that equivalent types have not only the same backtranslation type but also the same term relation.

Lemma 13 (Compatibility lemma for emulation of type equality).

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

$$\textit{then}\ \ \textit{if}\ \mathtt{toEmul}^{\mathsf{fE}}_{\mathsf{m;p}}\,(\Gamma) \vdash \mathsf{t}\ \triangledown_n\ t : \mathsf{EmulT}^{\mathsf{fE}}_{\mathsf{m;p};\tau}\ \ \textit{and}\ \tau \doteq \sigma\ \ \textit{then}\ \mathtt{toEmul}^{\mathsf{fE}}_{\mathsf{m;p}}\,(\Gamma) \vdash \mathsf{t}\ \triangledown_n\ t : \mathsf{EmulT}^{\mathsf{fE}}_{\mathsf{m;p};\sigma}$$

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

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

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

Lemma 14 (Emulate is semantics-preserving).

$$\begin{aligned}&\textit{if}\ \ (m > n\ \textit{and}\ p = \texttt{precise})\ \textit{or}\ (\nabla = \lesssim\ \ \textit{and}\ p = \texttt{imprecise})\ \textit{and}\ \Gamma \vdash \mathbf{t} : \boldsymbol{\tau}\\ \textit{then}\ \ &\texttt{toEmul}_{m;p}\,(\Gamma) \vdash \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{m}}\,(\Gamma \vdash \mathbf{t} : \boldsymbol{\tau})\ \nabla_n\ \mathbf{t} : \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m;p};\boldsymbol{\tau}}\end{aligned}$$

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 15 (Emulate is semantics preserving for contexts).

$$\begin{aligned}&\textit{if}\ \ (m > n\ \textit{and}\ p = \texttt{precise})\ \textit{or}\ (\nabla = \lesssim\ \ \textit{and}\ p = \texttt{imprecise})\ \textit{and}\ \vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau\\ \textit{then}\ \ &\vdash \mathsf{emulate}^{\mathsf{fl}}_{\mathsf{m}}\left(\vdash \mathfrak{C} : \Gamma', \tau' \to \Gamma, \tau\right)\ \nabla_n\ \mathfrak{C} : \texttt{toEmul}_{m;p}\,(\Gamma'), \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m;p};\tau'} \to \texttt{toEmul}_{m;p}\,(\Gamma), \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m;p};\tau}\end{aligned}$$

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 $\mathsf{BtT}^{\mathsf{fE}}_{\mathsf{n};\tau}$, so we need to convert values of source type into values of backtranslation type (and back). To do this conversion we rely on functions $\mathsf{inject}^{\mathsf{fl}}$ and $\mathsf{extract}^{\mathsf{fl}}$ whose types and definitions are in Figure 12. Function $\mathsf{inject}^{\mathsf{fl}}$ 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 $\texttt{fxToIs}\,(\cdot)$ to generate the target type related to τ. Function $\mathsf{extract}^{\mathsf{fl}}$ 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.

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 $\mathsf{inject}^{\mathsf{fl}}$ to return unit and for $\mathsf{extract}^{\mathsf{fl}}$ to just diverge.

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: $\boldsymbol{\mu\alpha.\,\tau}$. Thus, for the third backtranslation, we have a different, extended version of $\mathbf{inject}^{\mathbf{IE}}$ /$\mathbf{extract}^{\mathbf{IE}}$ 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 $\mathsf{inject}^{\mathsf{fl}}$ while terms that are related at backtranslation type are related at source type after an $\mathsf{extract}^{\mathsf{fl}}$.

Lemma 16 (Inject and extract are semantics preserving).

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

$$\boxed{\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau} \ : \tau \to \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\mathtt{fxToIs}(\tau)} \quad \text{and} \quad \mathsf{extract}^{\mathsf{fl}}_{\mathsf{n};\tau} \ : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\mathtt{fxToIs}(\tau)} \to \tau}$$

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

$$\begin{aligned}
\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n+1};\tau\to\tau'} &= \lambda \mathsf{x} : \tau \to \tau'.\mathsf{inl}\ \lambda \mathsf{y} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\mathtt{fxToIs}(\tau)}.\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau'} \left(\mathsf{x}\ (\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n};\tau}\ \mathsf{y})\right)\\
\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n+1};\tau\times\tau'} &= \lambda \mathsf{x} : \tau \times \tau'.\mathsf{inl}\ \left\langle \mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau}\ (\mathsf{x}.1), \mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau'}\ (\mathsf{x}.2)\right\rangle\\
\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n+1};\tau\uplus\tau'} &= \lambda \mathsf{x} : \tau \uplus \tau'.\mathsf{inl}\ \mathsf{case}\ \mathsf{x}\ \mathsf{of}\ \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ (\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau}\ \mathsf{x}_1) \mid \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ (\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau'}\ \mathsf{x}_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{extract}^{\mathsf{fl}}_{0;\tau} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\mathtt{fxToIs}(\tau)}.\, \mathsf{omega}_{\tau}\\
\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n+1;Unit}} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\mathbf{Unit}}.\, \mathsf{case}^{\mathsf{fl}}_{\mathsf{n+1};\mathbf{Unit}}\ \mathsf{x}\\
\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n+1;Bool}} &= \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\mathbf{Bool}}.\, \mathsf{case}^{\mathsf{fl}}_{\mathsf{n+1};\mathbf{Bool}}\ \mathsf{x}
\end{aligned}$$

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

$$\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n+1};\tau\times\tau'} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\mathtt{fxToIs}(\tau\times\tau')}.\left\langle \begin{aligned} &\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n};\tau} \left(\mathsf{case}^{\mathsf{fl}}_{\mathsf{n+1};\mathtt{fxToIs}(\tau)}\ \mathsf{x}.1\right),\\ &\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n};\tau'} \left(\mathsf{case}^{\mathsf{fl}}_{\mathsf{n+1};\mathtt{fxToIs}(\tau')}\ \mathsf{x}.2\right) \end{aligned} \right\rangle$$

$$\mathsf{extract}^{\mathsf{fl}}_{\mathsf{n+1};\tau\uplus\tau'} = \lambda \mathsf{x} : \mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n+1};\mathtt{fxToIs}(\tau\uplus\tau')}.\ \begin{aligned} &\mathsf{case}\ \left(\mathsf{case}^{\mathsf{fl}}_{\mathsf{n+1};\mathtt{fxToIs}(\tau\uplus\tau')}\ \mathsf{x}\right)\ \mathsf{of}\\ &\left|\ \mathsf{inl}\ \mathsf{x}_1 \mapsto \mathsf{inl}\ \mathsf{extract}^{\mathsf{fl}}_{\mathsf{n};\mathtt{fxToIs}(\tau)}\ \mathsf{x}_1\right.\\ &\left|\ \mathsf{inr}\ \mathsf{x}_2 \mapsto \mathsf{inr}\ \mathsf{extract}^{\mathsf{fl}}_{\mathsf{n};\mathtt{fxToIs}(\tau')}\ \mathsf{x}_2\right. \end{aligned}$$

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

$$\begin{aligned}
\mathbf{inject}^{\mathbf{IE}}_{\mathbf{n+1};\mu\alpha.\tau} &= \boldsymbol{\lambda}\mathbf{x} : \mu\alpha.\,\tau.\, \mathbf{inject}^{\mathbf{IE}}_{\mathbf{n+1};\tau[\mu\alpha.\tau/\alpha]}\ (\mathbf{unfold}_{\mu\alpha.\tau}\ \mathbf{x})\\
\mathbf{extract}^{\mathbf{IE}}_{\mathbf{n+1};\mu\alpha.\tau} &= \boldsymbol{\lambda}\mathbf{x} : \mathbf{BtT}^{\mathbf{IE}}_{\mathbf{n+1};\mathtt{isToEq}(\mu\alpha.\tau)}.\, \mathbf{extract}^{\mathbf{IE}}_{\mathbf{n+1};\mu\alpha.\tau}\ \mathbf{fold}_{\mu\alpha.\tau}\ (\mathbf{case}^{\mathbf{IE}}_{\mathbf{n+1};\mathtt{isToEq}(\mu\alpha.\tau)}\ \ \mathbf{x})\\
&\quad \text{omitted cases are as above}
\end{aligned}$$

Fig. 12. Definition of the inject and extract functions.

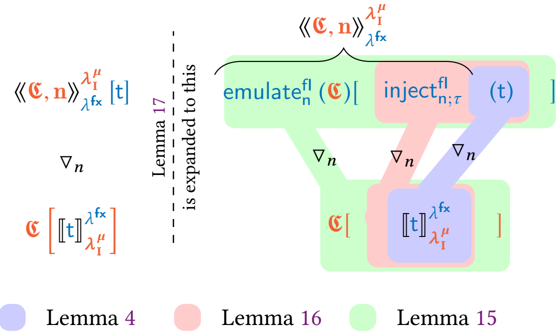
4.2.3 The Backtranslations. The backtranslation of a target context based on its type derivation is defined as follows by relying on both $\mathsf{emulate}^{\mathsf{fl}}\ (\cdot)$ and $\mathsf{inject}^{\mathsf{fl}}$. All three backtranslations follow exactly the same pattern and enjoy the same properties. 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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ to λ^{fx} and we state properties only for this one.

Definition 3 (Approximate backtranslation for $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ contexts into λ^{fx}).

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

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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$. Thus, $\langle\!\langle t \rangle\!\rangle^{\lambda^{\mu}_{E}}_{\lambda^{\mathsf{fx}}} = \left\langle\!\!\left\langle \langle\!\langle t \rangle\!\rangle^{\lambda^{\mu}_{E}}_{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}} \right\rangle\!\!\right\rangle^{\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}}_{\lambda^{\mathsf{fx}}}$. 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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$, i.e., the case for $\mathsf{BtT}^{\mathsf{fE}}$ for $\mu\alpha.\,\tau$ should not lose precision (as shown in Figure 5). Notice that the first backtranslation ($\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{E}}_{\lambda^{\mu}_{\mathrm{I}}}$) directs this, since $\mathbf{BtT}^{\mathbf{IE}}$ is simply a collection of $\hat{\tau} \uplus \hat{\tau}'$ pseudotypes, the second backtranslation ($\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}$) simply relies on the case for $\mathsf{BtT}^{\mathsf{fl}}_{\mathsf{n};\tau\uplus\tau'}$.

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 13 (adapted from the work of Devriese et al. [2016] to our setting). All of the infrastructure used by the backtranslation (i.e., $\mathsf{inject}^{\mathsf{fl}}$ /$\mathsf{extract}^{\mathsf{fl}}$ and the $\mathsf{BtT}^{\mathsf{fl}}$ helpers) have correctness lemmas that follow the same structure of the one for $\mathsf{emulate}^{\mathsf{fl}}\ (\cdot)$. Specifically, they relate terms at $\mathsf{EmulT}^{\mathsf{fl}}$, they transform target environments into source ones via function $\mathtt{toEmul}\ (\cdot)$ and they have a condition on the different directions of the approximation (the first line in Lemmas 12 to 15).



$\langle\!\langle\mathfrak{C},\mathbf{n}\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}\ [\mathsf{t}]$
∇_n
$\mathfrak{C}\left[[\![\mathsf{t}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right]$
Lemma 17
is expanded to this
$\langle\!\langle\mathfrak{C},\mathbf{n}\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}$
$\mathsf{emulate}^{\mathsf{fl}}_{\mathsf{n}}\ (\mathfrak{C})[$
$\mathsf{inject}^{\mathsf{fl}}_{\mathsf{n};\tau}$
(t)
$]$
∇_n
∇_n
∇_n
$\mathfrak{C}[$
$[\![\mathsf{t}]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$
$]$
Lemma 4
Lemma 16
Lemma 15

Fig. 13. Diagram representing the relatedness between different bits of the backtranslation and of the compiler.

Lemma 17 (Correctness of $\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}$).

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

then *if* $\vdash \mathfrak{C} : \varnothing, [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} \to \varnothing, \tau$ *and* $\varnothing \vdash \mathsf{t} \mathrel{\nabla_n} \mathsf{t} : \tau$ *then* $\varnothing \vdash \langle\!\langle\mathfrak{C},\mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}\ [\mathsf{t}] \mathrel{\nabla_n} \mathfrak{C}[\mathsf{t}] : \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m};\mathsf{p};\tau}$

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 18 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$ preserves equivalence).

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

Proof. Take $\mathfrak{C}$ such that $\vdash \mathfrak{C} : \varnothing, [\![\tau]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} \to \varnothing, \tau$. We need to prove that $\mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right]\Downarrow \iff \mathfrak{C}\left[[\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right]\Downarrow$. By symmetry, we prove only that if $\mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right]\Downarrow$ then $\mathfrak{C}\left[[\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right]\Downarrow$ (HPTT). Take n strictly larger than the steps needed for $\mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right]\Downarrow$. By Lemma 4 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$ is semantics preserving) we have $\varnothing \vdash \mathsf{t}_1 \mathrel{\nabla_n} [\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} : \tau$. Take $m = n$, so we have ($m \geq n$ and $p = \mathtt{precise}$) and therefore ($\nabla = \gtrsim$). By Lemma 17 (Correctness of $\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}$) we have $\varnothing \vdash \langle\!\langle\mathfrak{C},\mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}\ [\mathsf{t}_1] \gtrsim_n \mathfrak{C}\left[[\![\mathsf{t}_1]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}\right] : \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m};\mathsf{p};\tau}$. By Lemma 2 (Adequacy for $\approx$) for $\gtrsim$ and HPTT we have: $\langle\!\langle\mathfrak{C},\mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}\ [\mathsf{t}_1]\Downarrow$, which by source contextual equivalence gives us $\langle\!\langle\mathfrak{C},\mathbf{m}\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}\ [\mathsf{t}_2]\Downarrow$ (HPTS2). Given n' the number of steps for HPTS2, by Lemma 4 ($[\![\cdot]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}}$ is semantics preserving) we have: $\varnothing \vdash \mathsf{t}_2 \mathrel{\nabla_{n'}} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} : \tau$. So by definition: $\varnothing \vdash \mathsf{t}_2 \lesssim_{n'} [\![\mathsf{t}_2]\!]^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathrm{I}}} : \tau$. By Lemma 17 (Correctness of $\langle\!\langle\cdot\rangle\!\rangle^{\lambda^{\mu}_{\mathrm{I}}}_{\lambda^{\mathsf{fx}}}$) (with $n = n'$, $p = \mathtt{imprecise}$ and

$\triangledown = \lesssim$) we can conclude $\varnothing \vdash \langle\!\langle \mathfrak{C}, \mathrm{m} \rangle\!\rangle^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mathsf{fx}}} [\mathsf{t}_2] \lesssim_n \mathfrak{C}\left[\llbracket \mathsf{t}_2 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathtt{I}}} \right] : \mathsf{EmulT}^{\mathsf{fl}}_{\mathsf{m;p;}\tau}$. By Lemma 2 (Adequacy for $\approx$) for $\lesssim$ with HPTS2 we conclude the thesis. □

Theorem 19 ($\llbracket\cdot\rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}}$ preserves equivalence).

If $\varnothing \vdash \mathbf{t}_1 \simeq_{\mathbf{ctx}} \mathbf{t}_2 : \boldsymbol{\tau}$ then $\varnothing \vdash \llbracket \mathbf{t}_1 \rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} \llbracket \mathbf{t}_2 \rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}} : \llbracket \tau \rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}}$

Theorem 20 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ preserves equivalence).

If $\varnothing \vdash \mathsf{t}_1 \simeq_{\mathsf{ctx}} \mathsf{t}_2 : \tau$ then $\varnothing \vdash \llbracket \mathsf{t}_1 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} \llbracket \mathsf{t}_2 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : \llbracket \tau \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$

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 $\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ trivially follows from composing full abstraction for the other two compilers.

Theorem 21 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathtt{I}}}$ is fully abstract). $\varnothing \vdash \mathsf{t}_1 \simeq_{\mathsf{ctx}} \mathsf{t}_2 : \tau \iff \varnothing \vdash \llbracket \mathsf{t}_1 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathtt{I}}} \simeq_{\mathbf{ctx}} \llbracket \mathsf{t}_2 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathtt{I}}} : \llbracket \tau \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{\mathtt{I}}}$

Theorem 22 ($\llbracket\cdot\rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}}$ is fully abstract). $\varnothing \vdash \mathbf{t}_1 \simeq_{\mathbf{ctx}} \mathbf{t}_2 : \boldsymbol{\tau} \iff \varnothing \vdash \llbracket \mathbf{t}_1 \rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} \llbracket \mathbf{t}_2 \rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}} : \llbracket \boldsymbol{\tau} \rrbracket^{\lambda^{\mu}_{\mathtt{I}}}_{\lambda^{\mu}_{E}}$

Theorem 23 ($\llbracket\cdot\rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$ is fully abstract). $\varnothing \vdash \mathsf{t}_1 \simeq_{\mathsf{ctx}} \mathsf{t}_2 : \tau \iff \varnothing \vdash \llbracket \mathsf{t}_1 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} \simeq_{\mathrm{ctx}} \llbracket \mathsf{t}_2 \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}} : \llbracket \tau \rrbracket^{\lambda^{\mathsf{fx}}}_{\lambda^{\mu}_{E}}$

5 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 $\lambda^{\mu}_{\mathtt{Es}}$).[6] $\lambda^{\mu}_{\mathtt{Ei}}$ defines an equality relation on types ($\bumpeq$) 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 $\Delta ::= \varnothing \mid \Delta; \alpha$.

$$\boxed{\tau \bumpeq \sigma}$$

$$\text{(Eq-type-Symmetric)}\ \frac{\Delta \vdash \tau' \bumpeq \tau}{\Delta \vdash \tau \bumpeq \tau'} \qquad \text{(Eq-type-Transitive)}\ \frac{\Delta \vdash \tau \bumpeq \tau'' \quad \Delta \vdash \tau'' \bumpeq \tau'}{\Delta \vdash \tau \bumpeq \tau'}$$

$$\text{(Eq-type-Bi)}\ \frac{\star \in \{\to, \times, \uplus\} \quad \Delta \vdash \tau_1 \bumpeq \tau_1' \quad \Delta \vdash \tau_2 \bumpeq \tau_2'}{\Delta \vdash \tau_1 \star \tau_2 \bumpeq \tau_1' \star \tau_2'} \qquad \text{(Eq-type-Base)}\ \frac{\iota = \mathtt{Unit} \vee \iota = \mathtt{Bool}}{\Delta \vdash \iota \bumpeq \iota}$$

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

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 $\lambda^{\mu}_{\mathtt{Ei}}$:

$$\varnothing \vdash \mu\alpha.\,\alpha \to \mathtt{Unit} \not\bumpeq \mu\alpha.\,(\alpha \to \mathtt{Unit}) \to \mathtt{Unit} \qquad \varnothing \vdash \mu\alpha.\,\mu\beta.\,\alpha \to \beta \not\bumpeq \mu\alpha.\,\alpha \to \alpha$$

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.

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

$\lambda^{\mu}_{\texttt{Es}}$ instead enforces that just a recursive type and its unfolding are equivalent [Ahmed 2004; Appel and McAllester 2001; MacQueen et al. 1986; 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.

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

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 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].[7] 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

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

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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ 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 ours can be useful as they enrich the power of contexts and simplify secure compilation proofs further downstream.

6 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 ($\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$). 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 $\boldsymbol{\lambda}^{\mu}_{\mathbf{I}}$ 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.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for detailed feedback on an earlier draft as well as Phil Wadler for interesting comments and suggestions. This work was partially supported: by the Office of Naval Research for support through grant N00014-18-1-2620, Accountable Protocol Customization, by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), 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

Martín Abadi. 1998. Protection in Programming-Language Translations. In *ICALP'98*. 868–883.

Martin Abadi and Marcelo P. Fiore. 1996. Syntactic Considerations on Recursive Types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*. IEEE Computer Society, Washington, DC, USA, 242–.

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming*. ACM, 157–168.

Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. ACM, 431–444.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.* 44, 97–108.

William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *ICFP*. ACM.

Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. 2016. System F-omega with Equirecursive Types for Datatype-generic Programming. *SIGPLAN Not.* 51, 1 (Jan. 2016), 30–43.

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract Compilation by Approximate Back-translation. In *Principles of Programming Languages*. 164–177.

Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017).

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. In *Selected Papers from the Symposium on 3rd European Symposium on Programming (ESOP '90)*. Elsevier North-Holland, Inc., New York, NY, USA, 35–75.

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Principles of Programming Languages*. ACM, 371–384.

M. Gordon, R. Milner, and C. P. Wadsworth. 1979. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, Berlin Heidelberg. https://doi.org/10.1007/3-540-09724-4

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

Robert Harper and John C. Mitchell. 1993. On the Type Structure of Standard ML. *ACM Transactions on Programming Languages and Systems* 15, 2 (April 1993), 211–252. https://doi.org/10.1145/169701.169696

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *Principles of Programming Languages*. 133–146.

Hyeonseung Im, Keiko Nakata, and Sungwoo Park. 2013. Contractive Signatures with Recursive Types, Type Parameters, and Abstract Types. In *Automata, Languages, and Programming*, Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–311.

Andrew Kennedy. 2006. Securing the .NET Programming Model. *Theoretical Computer Science* 364 (2006), 311–317.

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1984. An Ideal Model for Recursive Polymorphic Types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/800017.800528

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1986. An ideal model for recursive polymorphic types. *Information and Control* 71, 1 (1986), 95 – 130.

Robin Milner. 1977. Fully abstract models of typed λ-calculi. *Theoretical Computer Science* 4, 1 (1977), 1 – 22.

John C. Mitchell. 1993. On abstraction and the expressive power of programming languages. *Science of Computer Programming* 21, 2 (1993), 141 – 163.

James H. Morris. 1968. *Lambda-Calculus Models of Programming Languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *International Conference on Functional Programming*. ACM, 103–116.

Joachim Parrow. 2008. Expressiveness of Process Algebras. *Elec. Not. Theo. Comp. Sci.* 209, 0 (2008), 173 – 186.

Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. CoRR abs/2001.11334.

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, Article 6 (April 2015), 6:1–6:50 pages.

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Jan. 2019), 36 pages.

Marco Patrignani, Eric Mark Martin, and Dominique Devriese. 2020. On the Semantic Expressiveness of Recursive Types. arXiv:2010.10859 [cs.PL]

Benjamin Pierce. 2002. *Types and Programming Languages*. MIT Press.

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science* 5 (1977), 223–255.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A Nanopass Infrastructure for Compiler Education. *ACM SIGPLAN Notices* 39, 9 (Sept. 2004), 201–212. https://doi.org/10.1145/1016848.1016878

Manfred Schmidt-Schauß, David Sabel, Joachim Niehren, and Jan Schwinghammer. 2015. Observational program calculi and the correctness of translations. *Theoretical Computer Science* 577 (2015), 98 – 124.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 19:1–19:28.

Pawel Urzyczyn. 1995. Positive Recursive Type Assignment. In *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science (MFCS '95)*. Springer-Verlag, Berlin, Heidelberg, 382–391.

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code. *Proc. ACM Program. Lang.* ICFP (2019).