Fully Abstract from Static to Gradual

KOEN JACOBS, imec-DistriNet, KU Leuven, Belgium
AMIN TIMANY, Aarhus University, Denmark
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium

What is a good gradual language? Siek et al. have previously proposed the refined criteria, a set of formal ideas that characterize a range of guarantees typically expected from a gradual language. While these go a long way, they are mostly focused on syntactic and type safety properties and fail to characterize how richer semantic properties and reasoning principles that hold in the static language, like non-interference or parametricity for instance, should be upheld in the gradualization.

In this paper, we investigate and argue for a new criterion previously hinted at by Devriese et al.: the embedding from the static to the gradual language should be fully abstract. Rather than preserving an arbitrarily chosen interpretation of source language types, this criterion requires that *all* source language equivalences are preserved. We demonstrate that the criterion weeds out erroneous gradualizations that nevertheless satisfy the refined criteria. At the same time, we demonstrate that the criterion is realistic by reporting on a mechanized proof that the property holds for a standard example: GTLC$_\mu$, the natural gradualization of STLC$_\mu$, the simply typed lambda-calculus with equirecursive types. We argue thus that the criterion is useful for understanding, evaluating, and guiding the design of gradual languages, particularly those which are intended to preserve source language guarantees in a rich way.

CCS Concepts: • **Theory of computation** → **Program reasoning**; **Type structures**; **Type theory**; *Modal and temporal logics*; *Logic and verification*; • **Software and its engineering** → *General programming languages*; *Compilers*.

Additional Key Words and Phrases: gradual typing, fully abstract compilation, fully abstract embedding

ACM Reference Format:
Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL, Article 7 (January 2021), 30 pages. https://doi.org/10.1145/3434288

1 INTRODUCTION

The Aim of Gradual Typing. The aim of gradual typing is to enable a "programmer-controlled migration between dynamic and static typing" [Siek and Taha 2006]. That is, given a static type system, the goal is to construct a gradual type system in which existing untyped codebases can be gradually migrated to follow the static typing discipline. To live up to this ambition, a gradual language should be well designed:

(1) It should directly support existing untyped and typed programs and not break their well-formedness or well-typedness, or modify their semantics.

Authors' addresses: Koen Jacobs, imec-DistriNet, KU Leuven, Leuven, Belgium, koen.jacobs@kuleuven.be; Amin Timany, Aarhus University, Aarhus, Denmark, timany@cs.au.dk; Dominique Devriese, Vrije Universiteit Brussel, Belgium, dominique.devriese@vub.be.

2475-1421/2021/1-ART7
https://doi.org/10.1145/3434288

(2) When the programmer adds type annotations to a part of the program, this should only modify the semantics when necessary: it is okay to throw type errors for wrong programs, but *accurate* type annotations should not change a program's semantics.
(3) Finally, a programmer should not have to migrate an entire large codebase before enjoying the benefits of the type system. The type system's benefits (performance, type safety or others) should already apply *locally* in the fully typed parts of the code, no matter how dynamic the surrounding code may be.

If a gradual language provides these guarantees, it offers a viable migration path for large existing dynamically typed codebases. Programmers can flexibly spread the migration over time or even *economically* apply the static typing discipline where it is of most importance, without necessarily migrating the entire codebase.

As an aside, we briefly mention that the dynamic part of a gradual language need not be fully untyped. Given, for instance, a STLC-like language with security types as the static language, one can aim to gradualize only the security aspect of this typing discipline such that the dynamic language corresponds to the STLC [Garcia and Tanter 2015].

Refined Criteria. To formally capture these goals, Siek et al. [2015] have put forward a set of refined criteria for a satisfactory gradual language. They formally define when a gradual type is more precise than another and formalize the above properties for a specific gradual language (the GTLC, see below). They can be summarized in general as follows:

(1) **Superset criteria** The gradual language must completely contain the static and dynamic language under consideration. Static/dynamic terms must be appropriately typed in the gradual language and their operational semantics must coincide with those in the static/dynamic language.
(2) **The Gradual Guarantee** Weakening type annotations (e.g. by removing some of the annotations) of a *well-typed* term preserves its well-typedness and does not change its operational semantics. Note that starting from a more precise term and weakening its type annotations is a way of characterizing how *correct* type annotations can be added to the less precise resulting term.
(3) **Blame theorem** The gradual language satisfies a blame theorem [Wadler and Findler 2009], which *implies* that fully-annotated pieces of code are never to blame for a cast error.[1]

While the two former criteria seem to satisfactorily capture the intuitive properties mentioned, the final criterion does so only partially.

The Refined Criteria are Not Enough. The Blame theorem implies that the gradual language should preserve some guarantees of the static type system for fully annotated code. However, Garcia and Tanter [2020] have recently pointed out that it does so only for those concerning type safety. Note that in this text, we use the words type safety to capture syntactic type safety properties, typically embodied by progress and preservation theorems in the style of Wright and Felleisen [1994]. For some type systems this is sufficient, but type systems often provide additional guarantees.

Garcia and Tanter [2020] propose that a well-designed gradual language should not just enforce type safety, but type soundness. Type soundness refers to a semantic interpretation of typing judgments $\Gamma \vDash t : \tau$, a property which depends on the type system at hand and can to some extent be chosen by the gradual language designer. For example, they explain how the meaning of STLC types can be taken to include termination or just type safety. For gradualizations of polymorphic types, they explain how type soundness can be taken to include parametricity or not, and in fact,

[1] A blame theorem is typically a lot stronger, but as we are not currently interested about the practical implementation of error documentation, this is the crux of it as a language criterion.

Devriese et al. [2017a, 2020] have explained that existing gradual parametric languages only satisfy a standard form of parametricity that is strictly weaker than the one that Reynolds originally formulated for System F types [Reynolds 1983]. For gradualizations of security type systems, Garcia and Tanter [2020] explain how the security property of non-interference can be taken as part of the meaning of types.

Choosing the Meaning of Types? Essentially, what Garcia and Tanter [2020] propose is that a gradual language should preserve the meaning of types, but leave that meaning open to interpretation. Letting gradual language designers select a meaning of types allows them to choose one they can actually enforce in the gradual language. In the context of gradual parametricity, for example, most people agree that the gradual language should enforce parametricity, an important reasoning principle which holds in static languages like System F. However, parametricity can be defined to mean very different things and it has taken several iterations to find a form of parametricity that gradual parametric languages can actually guarantee [Ahmed et al. 2011a, 2017, 2011b; Matthews and Ahmed 2008; Toro et al. 2019]. Because of the existence of a universal type, the parametricity offered by these calculi (if any) is defined by a type-world logical relation [Devriese et al. 2017a, 2020] that is strictly weaker than traditional Reynolds-style parametricity that one may be accustomed to in System F [Reynolds 1983]. In the case of security types, the reasoning principle of non-interference [Toro et al. 2018] can also be defined in different ways (e.g. termination-sensitive or not) and it is not a priori clear which one is right.

As such, Garcia and Tanter's approach lets gradual language designers select a meaning of types that they can enforce and thus offers flexibility. However, this raises the question if it is really enough to enforce only a carefully selected meaning of types, which is strictly weaker than the meaning of types in the static language. For example, can we really claim to have gradualized System F if we only enforce part of the reasoning principles that hold in System F? For programmers, this would mean that embedding existing static programs in the gradual language requires one to ascertain that their correctness, safety or security only relies on those reasoning principles that are preserved in the gradual language. It also means that in the static language, only those optimizations and refactorings should be applied for which the validity follows just from the reasoning principles that are preserved in the gradual language. Restricting the meaning of types to an enforceable subset like this may be unavoidable for some static languages, but this paper explores a more ambitious approach.

Preserving Reasoning More Generally? What if we were building a gradual language that aims to preserve arbitrary reasoning about static programs? Devriese et al. [2017a, 2020] have recently proposed a criterion that takes this perspective, at least for reasoning about program equivalences: the fully abstract embedding (FAE) property. The idea is to consider the embedding function from the static language into the gradual language as a translation function and consider whether it satisfies full abstraction, a property proposed by Abadi [1998, 1999] in the context of secure compilation [Patrignani et al. 2019a]. This criterion implies that the gradual language should preserve contextual equivalences in the static language. Not a suitably selected set of contextual equivalences, but simply all of them.

If two static terms always behave the same in the static language (i.e. no surrounding static code can distinguish the two), then they should always behave the same when embedded in the gradual language (i.e. *gradual* code should not distinguish the two either). In short, equivalent terms in the static language should be equivalent (as fully annotated terms) in the gradual language.

Another way of interpreting this criterion is that contexts in the gradual language should not have more distinguishing power over static terms than static contexts do and that abstractions and reasoning principles from the static language should remain valid in the gradual language.

Practically, it means for example that refactorings and compiler optimizations which are semantics-preserving in the static language are also sound in the gradual extension.

What is in this Paper? After providing some background in §2, we demonstrate the value of the FAE criterion in §3 by showing how it successfully detects unsatisfactory gradualizations. In other words, we study three gradual languages where the FAE fails.

Next, in §4, we show that the FAE is also realistic, by proving the property for a simple but representative example; the GTLC$_\mu$. Taking inspiration from similar proofs in the context of secure compilation [Devriese et al. 2016, 2017b; New et al. 2016], we prove that the embedding of STLC$_\mu$ into GTLC$_\mu$ is fully abstract. This proof has been mechanically verified using Coq and Iris [Jung et al. 2018], and can be checked out at https://github.com/scaup/fae-gtlc-mu[2].

In §5, we elaborate on some of the choices made in earlier sections and lay out some interesting future work. We discuss related work in §6 and conclude in §7.

Contributions. Our central contribution is the study of FAE as a general formal criterion for satisfactory gradual languages, supplementing Siek et al.'s refined criteria. It captures the intuition that a gradual language should preserve the static language's benefits also when it comes to reasoning. Specifically, we demonstrate the following

- We explain a number of interesting gradual languages where FAE fails, i.e. where the criterion adequately identifies unsatisfactory gradualizations.
- We show that the criterion is realistic, by proving it for the GTLC$_\mu$, with respect to STLC$_\mu$. This proof is mechanically verified using Coq and builds on Iris machinery like guarded recursion for convenient reasoning. This proof is only the second[3] mechanized full abstraction proof in the literature (the first using Iris) and (perhaps surprisingly) one of the first mechanized results about a gradual language.

2 SOME BACKGROUND

Before we continue, let us quickly refresh some material from earlier work, specifically the Gradually Typed Lambda Calculus (GTLC) with recursive types (§2.1) and fully abstract translations (§2.2).

2.1 Gradualizing STLC$_\mu$

First, we introduce GTLC$_\mu$: the gradualization of the STLC$_\mu$. We follow the standard literature in doing so [Siek et al. 2015], remarking where we slightly deviate.

The Statics. Gradualizing the STLC$_\mu$ — the simply-typed lambda calculus with sum, product, and (iso-)recursive types — we want to obtain a language in which we can freely navigate the landscape between fully annotated code and fully dynamic code. To this end, we extent the grammar of our types as follows.

$$\tau ::= B \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X.\, \tau \mid \ ?$$

Aside from *some* base type[4], arrow types, sum types, product types, and recursive types, we now have the additional *unknown* type, ? and of course all the types containing it.[5]

[2] A VM-image preinstalled with the required libraries and the right version of Coq can be downloaded at https://doi.org/10.5281/zenodo.4071674

[3] The first mechanized proof is reported on by Devriese et al. [2017b].

[4] To keep the formalities in §4 from being annoyingly verbose, we assume only one base type here (in the Coq development, this is the unit type). When giving examples, we will sometimes assume multiple concrete base types like $\mathbb{N}$ or $\mathbb{B}$ for instance, as it is entirely moral to do so.

[5] We will always consider *closed* types (i.e. types that do not contain any free variables); in STLC$_\mu$, type variables are only meaningful to the extent that they are bound to define a recursive type.

$$? \sim \tau \qquad \tau \sim ? \qquad B \sim B \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 + \tau_2 \sim \tau_1' + \tau_2'} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'}$$

$$\frac{\tau \sim \tau'}{\mu X.\, \tau \sim \mu X.\, \tau'} \qquad X \sim X$$

Fig. 1. The consistency relation between types.

Semantically, a value of type $?$ can be of *any* static type. Accordingly, we can make the type-annotations in our code as precise or as vague as we desire, navigating toward the static/dynamic end of the spectrum respectively.

In the presence of potentially imprecisely annotated code, the job of type checking in the gradual language is twofold. While it should allow for untyped code (possibly failing upon execution), it should do its best to reject code for which it can *statically* be deduced that it is up to no good. Consequently, type-checked code should retain the safety guarantee as much as is statically possible.

For instance, the term $(\lambda x : \mathbb{N} \times \mathbb{B}.\, \neg(\pi_2\ x))\ 6$ should not type-check, as we statically see that an integer is not of type $\mathbb{N} \times \mathbb{B}$. Type-checking should only allow the application of a function to an argument if the advertised input type of the function is consistent with the actual type of the argument, i.e. if they are *conceivably* compatible. The type $\mathbb{N} \times \mathbb{B}$ is obviously not consistent with $\mathbb{N}$, nor, for instance, with $? + ?$. It is however consistent with $\mathbb{N} \times \mathbb{B}$, and more interestingly, with $\mathbb{N} \times ?$, $? \times \mathbb{B}$, $? \times ?$, and $?$.

Formally, we have the following typing rule for application.

$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash e' : \tau_1' \qquad \tau_1 \sim \tau_1'}{\Gamma \vdash e\ e' : \tau_2}$$

Consistency between types, $\tau \sim \tau'$, is defined in fig. 1. Note that we assume that two types are consistent if they are after possible α-conversion.

The Dynamics. The dynamics of the gradual language are defined by translating its terms to a different language, the *cast calculus*. This calculus contains an explicit cast construct, $e : \tau \Rightarrow \tau'$, to cast an expression, e, from one type τ to another (consistent) type τ'. Through the translation of a well-typed gradual term, the implicit applications of consistency are replaced by explicit casts. Translating for instance, $(\lambda x : \mathbb{N}.\, x)\ ((\lambda x : ?.\, x)\ \mathit{True})$, we obtain the following:

$$(\lambda x : \mathbb{N}.\, x)\ (((\lambda x : ?.\, x)\ (\mathit{True} : \mathbb{B} \Rightarrow ?)) : ? \Rightarrow \mathbb{N})$$

We summarize the cast calculus in fig. 2. Unsurprisingly, the dynamics of the cast calculus mostly coincide with STLC_μ; looking at the expressions, we have constants k; variables x; lambda-abstractions $\lambda x : \tau.\, e$ and applications $e\ e$; pairs (e, e) and projections $\pi_{1,2}\ e$ as constructor/destructors of product types; injections $\mathtt{inj}_{1,2}\ e$ and case matches $\mathtt{case}\ e\ \mathtt{of}\ (e \mid e)$ as constructors/destructor of sum types; and $\mathtt{fold}$ and $\mathtt{unfold}$ as constructor/destructor of recursive types. The only thing of interest here is the cast construct and the evaluation rules concerning it.

We define a collection of ground types, G, through which we define the dynamics of the casts (to be explained later). Aside from B and $? \to ?$, we also include $? + ?$ and $? \times ?$, following Siek [2019]. Lastly, we also add $\mu X.?$ whose values are exactly those that we can unfold at least one time.

Casting a value from a ground type to the unknown forms a value $v : G \Rightarrow ?$, corresponding to the idea that such a cast will always go through, as we are merely losing type information. The same is true for casting a value from one function type to another, $v : \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2$, as such a cast is suspended (*Factor-App*).

$$
\begin{array}{rcl}
G & ::= & B \mid ? \to ? \mid ? + ? \mid ? \times ? \mid \mu X.?\\
e & ::= & k \mid x \mid \lambda x : \tau. e \mid e\ e \mid (e, e) \mid \pi_1\ e \mid \pi_2\ e \mid \mathtt{inj}_1\ e \mid \mathtt{inj}_2\ e \mid \mathtt{case}\ e\ \mathtt{of}\ (e \mid e) \mid \mathtt{fold}\ e \mid \mathtt{unfold}\ e \mid \\
& & e : \tau \Rightarrow \tau \mid \mathtt{CastError}\\
v & ::= & k \mid \lambda x : \tau. e \mid \mathtt{fold}\ v \mid (v, v) \mid \mathtt{inj}_{1,2}\ v \mid v : G \Rightarrow ? \mid v : \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'\\
K & ::= & [\cdot] \mid K\ e \mid v\ K \mid \pi_1\ K \mid \pi_2\ K \mid \mathtt{inj}_1\ K \mid \mathtt{inj}_2\ K \mid (K, e) \mid (v, K) \mid \mathtt{case}\ K\ \mathtt{of}\ (e \mid e) \mid \mathtt{fold}\ K \mid \mathtt{unfold}\ K \mid \\
& & K : \tau \Rightarrow \tau'
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S} : \mathit{Type} \rightharpoonup \mathit{Type}\\
\mathcal{S}(\tau_1 \times \tau_2) \triangleq ? \times ?\\
\mathcal{S}(\tau_1 \to \tau_2) \triangleq ? \to ?\\
\mathcal{S}(\tau_1 + \tau_2) \triangleq ? + ?\\
\mathcal{S}(\mu.\, \tau) \triangleq \mu X.?
\end{array}
$$

$$
\begin{array}{rcll}
\ldots & & & \\
v : B \Rightarrow B & \to_{\mathsf{h}} & v & \textit{Base-Id}\\
v : ? \Rightarrow ? & \to_{\mathsf{h}} & v & \textit{Unknown-Id}\\
v : \tau \Rightarrow ? & \to_{\mathsf{h}} & v : \tau \Rightarrow G \Rightarrow ? \quad \text{if } \mathcal{S}(\tau) = G \neq \tau & \textit{Factor-Up}\\
v : ? \Rightarrow \tau & \to_{\mathsf{h}} & v : ? \Rightarrow G \Rightarrow \tau \quad \text{if } \mathcal{S}(\tau) = G \neq \tau & \textit{Factor-Down}\\
v : G \Rightarrow ? \Rightarrow G & \to_{\mathsf{h}} & v & \textit{Ground-Id}\\
v : G_1 \Rightarrow ? \Rightarrow G_2 & \to_{\mathsf{h}} & \mathtt{CastError} \quad \text{if } G_1 \neq G_2 & \textit{Ground-Fail}\\
(v : \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2')\ w & \to_{\mathsf{h}} & (v\ (w : \tau_1' \Rightarrow \tau_1)) : \tau_2 \Rightarrow \tau_2' & \textit{Factor-App}\\
(\mathtt{inj}_i\ v) : \tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2' & \to_{\mathsf{h}} & \mathtt{inj}_i\ (v_i : \tau_i \Rightarrow \tau_i') \quad \text{for } i = 1, 2 & \textit{Factor-Sum}\\
(v_1, v_2) : \tau_1 \times \tau_2 \Rightarrow \tau_1' \times \tau_2' & \to_{\mathsf{h}} & (v_1 : \tau_1 \Rightarrow \tau_1',\ v_2 : \tau_2 \Rightarrow \tau_2') & \textit{Factor-Prod}\\
(\mathtt{fold}\ v) : \mu X.\tau \Rightarrow \mu Y.\tau' & \to_{\mathsf{h}} & \mathtt{fold}\ (v : \tau[X \mapsto \mu X.\tau] \Rightarrow \tau'[Y \mapsto \mu Y.\tau']) & \textit{Unfold-Rec}\\
K[e] & \to & K[e'] \quad \text{if } e \to_{\mathsf{h}} e' & \textit{Congruence}\\
K[\mathtt{CastError}] & \to & \mathtt{CastError} \quad \text{if } K \neq [\cdot] & \textit{Report-CastError}
\end{array}
$$

$$
\ldots \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \sim \tau'}{\Gamma \vdash (e : \tau \Rightarrow \tau') : \tau'} \qquad\qquad \Gamma \vdash \mathtt{CastError} : \tau
$$

Fig. 2. Cast calculus of GTLC_μ

Then, there are a lot of casts which are just decomposed into other casts. When trying to cast down a value of the unknown type to a (non-ground) type τ, we factor out the cast through the corresponding ground type (*Factor-Down*). Conversely, we have (*Factor-Up*). A cast between function types is reduced — upon application of the function to an argument — to the *contravariant* cast on the argument, and a *covariant* one on the result (*Factor-App*). Following Siek [2019], a cast between sum types is delegated to the appropriate cast of the two components (*Factor-Sum*), and a cast between product types is delegated to the appropriate casts on each component (*Factor-Prod*). Lastly, a cast of a value between two recursive types recursively goes through its body. Casts between (iso)-recursive types behave thus similar to casts between product and sum types; they are all evaluated eagerly. This is in contrast to casts between function types, which are suspended until application. The latter is unavoidable however; *strictly* evaluating a cast between two function types requires checking if the final cast is respected for *any* well-typed input.

In the remaining evaluation rules, casts actually fail or succeed. In *Base-Id*, *Unknown-Id* and *Ground-Id*, a cast trivially goes through. It must fail in *Ground-Fail* however, as one tries to cast a value from one ground type to a different one.

In contrast to Siek et al. [2015], we do not implement blame, as we do not need it for our purposes (see §5). Instead, a failing cast — following Siek and Taha [2006] — evaluates to the stuck term `CastError` (*Ground-Fail*, *Report-CastError*).

2.2 Fully Abstract Compilation

Generally, fully abstract compilation [Abadi 1998, 1999; Patrignani et al. 2019a] is formally defined as follows.[6]

Definition 2.1 (Fully abstract compilation). Given a compiler, say $[\![\_]\!] : \mathcal{L}_s \to \mathcal{L}_t$, and equivalence relations in the source and target language, say $\simeq$ and $\mathrel{\tilde{\simeq}}$, we have fully abstract compilation if the compiler both preserves and reflects said equivalences. That is, if the following holds.

$$\forall e_1, e_2, e_1 \simeq e_2 \text{ if and only if } [\![e_1]\!] \mathrel{\tilde{\simeq}} [\![e_2]\!]$$

The equivalence relations at stake here relate terms that behave similarly under any possible interaction. Take, for instance the well-typed terms $\cdot \vdash 3 + 3 : \mathbb{N}$ and $\cdot \vdash \pi_2\ (\text{True}, 6) : \mathbb{N}$ in the STLC. These two terms are equivalent, as no well-typed surrounding program can ever distinguish the two. This is formalized as *contextual equivalence*.

Definition 2.2 (Contextual equivalence in STLC_μ*).* Two terms, e_1 and e_2, are contextually equivalent, written $e_1 \simeq_{ctx} e_2$, iff $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$ and for every context, $C : (\Gamma; \tau) \Rightarrow (\cdot\,; \mathbb{1})$, we have that the following holds.[7]

$$C[e_1] \Downarrow \text{ if and only if } C[e_2] \Downarrow$$

A surrounding program is formalized by a context; a programs with a hole. Any context C can be plugged in with a term e such that the program $C[e]$ is obtained. A context C has type $(\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$ if when plugged in with a well-typed term $\Gamma \vdash e : \tau$, we obtain the well-typed term $\Gamma' \vdash C[e] : \tau'$.

Notice that the definition is stronger than it appears. For example, if e_1 and e_2 are of boolean type, contextual equivalence between the two implies that if one of them reduces to a boolean value, the other one must reduce to the *same* value. This can be shown using the following context in which $\Omega_{\mathbb{1}}$ a diverging term of type $\mathbb{1}$.

$$\mathit{if}\ [\_]\ \mathit{then}\ \text{unit}\ \mathit{else}\ \Omega_{\mathbb{1}}$$

To formalize contextual equivalence in GTLC_μ, we chose an analogous definition by equi-termination.

Definition 2.3 (Contextual equivalence in Gradual Language). Two terms, e_1 and e_2, in GTLC_μ are equivalent, written $e_1 \mathrel{\tilde{\simeq}}_{ctx} e_2$, iff $\Gamma \vdash e_1, e_2 : \tau$ and for all gradual contexts $C : (\Gamma; \tau) \Rightarrow (\cdot\,; \mathbb{1})$, we have $C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$.

Notice that this definition does not distinguish between divergence and runtime errors. In §5.1, we motivate this choice and discuss the alternatives in detail.

[6] Expressions, types, and relations with respect to static/source languages will be formatted in blue; those related with target/gradual languages will be formatted in red.

[7] Here, and throughout the rest of this paper, we denote $e \Downarrow$ to mean, $\exists v.\, e \to^* v$.

3 GRADUAL LANGUAGES WITHOUT AN FAE

In this section, we touch upon three gradualizations from the literature that fail the FAE criterion. Two of these are the result of a straightforward but naive application of the AGT framework [Garcia et al. 2016]. AGT (Abstracting Gradual Typing) is a general framework that uses abstract interpretation of types to gradualize an arbitrary static language. It requires only a static language, a syntax for the gradual types, and a Galois connection between sets of static types (ordered by the subset relation) and gradual types (ordered by the precision relation). This connection defines the set of static types that represents a gradual type and the most precise gradual type that represents a given set of static types. Knowledge of AGT is not a prerequisite for this section however.

The AGT framework ensures that any resulting gradual language satisfies — by construction — the refined criteria. Unfortunately, the derived dynamics of the gradual language are only designed to preserve type safety. As a result, the gradualization often fails to preserve the guarantees of the static typing discipline that go beyond mere type safety [Garcia and Tanter 2020].

So far, problems like these have only been described in terms of the failure of properties that are specific to the static language at hand. In this section, we demonstrate that they are detected through failure of the general FAE criterion.

3.1 Gradualising a Parametric Language, Naively

Toro et al. [2019] define their static language SF, which is essentially a representation of System F that makes it more amenable to applying AGT-recipe. Because of parametricity, we have, for instance, the following contextual equivalence in this language:

$$\lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 0 \simeq_{ctx} \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 5$$

Indeed, from parametricity [Reynolds 1983] we know that instantiating a function f of type $\forall X.\, X \to \mathbb{Z}$ with the type $\mathbb{Z}$ (or any type for that matter) and applying it to a value, will always return the same integer. Intuitively, this is because the language does not allow f to detect which type it is being applied to and therefore, it must behave the same for any type.

If the FAE criterion holds for a gradualization of SF, the same equivalence must hold after embedding:

$$\lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 0 \rceil\rceil \overset{?}{\simeq}_{ctx} \lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 5 \rceil\rceil$$

When Toro et al. [2019] first applied AGT to SF naively, they used a standard representation of evidence (a concept introduced in AGT with which the dynamics of the gradualization is derived) as just a pair of gradual types. However, the resulting gradualization broke parametricity and the above equivalence. In their calculus, they can construct a function of type $\forall X.\, X \to \mathbb{Z}$ that acts as the successor function when instantiated with $\mathbb{Z}$, thereby demonstrating the lack of parametricity [Toro et al. 2019, §7.1]:

$$\Lambda X.\, \lambda x : X.\, (\lambda y : ?.\, (\lambda z : ?.\, z + 1)\ y)\ x$$

This function can be used to discriminate the two terms we saw before:

$$\lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 0 \rceil\rceil\ (\Lambda X.\, \lambda x : X.\, (\lambda y : ?.\, (\lambda z : ?.\, z + 1)\ y)\ x) \to^* 1$$
$$\lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 5 \rceil\rceil\ (\Lambda X.\, \lambda x : X.\, (\lambda y : ?.\, (\lambda z : ?.\, z + 1)\ y)\ x) \to^* 6$$

In other words, although the FAE criterion is not in any way tied to System F, the criterion accurately detects the failure of parametricity in this naive gradualization.[8]

[8] FAE also fails here because of partiality (divergence/errors) present in the gradual language, but this is orthogonal to the issue here.

3.2 Gradualizing a Parametric Language, Non-naively

In the last paragraph, we have examined an obviously naive gradualization of a parametric language. However, even less naive gradual languages are not without problems. Devriese et al. [2017a, 2020] have pointed out that even gradual polymorphic languages that come with a proof of parametricity [Ahmed et al. 2017; New et al. 2019; Toro et al. 2019] break certain equivalences that hold in System F.

The problem is demonstrated using the type Univ:

$$\text{Univ} = \exists Y.\, \forall X.\, (X \to Y) \times (Y \to X)$$

In a sense, this type expresses the existence of a universal type: a single type Y which every other type X can be embedded into and extracted from. Interestingly, Devriese et al. [2020] demonstrate that standard Reynolds parametricity implies degeneracy of this type: instantiating X with an arbitrary type, embedding a value of X into Y and then extracting it again, must necessarily diverge.

This property can be used to show contextual equivalence of the following two terms [Devriese et al. 2020]:

$$\begin{aligned} e_1 &= \lambda x : \text{Univ}.\ \text{unpack } x \text{ as } Y, x' \text{ in} \\ &\qquad \text{let } x'' : (\mathbb{1} \to Y) \times (Y \to \mathbb{1}) = x'\ [\mathbb{1}] \text{ in } (\pi_2\ x'')\ ((\pi_1\ x'')\ \text{unit}) \\ e_2 &= \lambda x : \text{Univ}.\ \Omega_{\mathbb{1}} \quad \textit{(where } \Omega_{\mathbb{1}} \textit{ a diverging term of type } \mathbb{1}\textit{)} \end{aligned}$$

Of course, gradual parametric languages contain the gradual type ? and indeed, the equivalence of $\lceil\lceil e_1 \rceil\rceil$ and $\lceil\lceil e_2 \rceil\rceil$ no longer holds because of this. It is not that hard to construct a context that uses ? to construct a non-degenerate value of type $\lceil\lceil \text{Univ} \rceil\rceil$ and use it to break the equivalence above:

$$[\cdot]\ \ (\text{pack } ?, \Lambda X.\, \langle \lambda x : X.\, x : X \Rightarrow ?, \lambda x : ?.\, x : ? \Rightarrow X \rangle \text{ as Univ})$$

What this means is that the parametricity offered by gradual parametric calculi is a weaker form of parametricity than the one that holds in System F. Because of this difference, not all equivalences that hold in System F are preserved in these languages. The FAE criterion picks up this difference in the kind of parametricity that is offered by the language. The fact that not all of System F's equivalences continue to hold in a gradual parametric calculus may be important for users to be aware of. As such, we think one should be careful to claim that these calculi are gradualizations of System F, but instead regard them as gradualizations of a variant of System F with a universal type.

3.3 Gradualizing an Information-flow Language with References

Toro et al. [2018] set out to gradualize the security aspect in the typing discipline of a non-trivial information-flow language with references. They first define their static language SSL_{ref}, a statically-typed language with references in which types and terms are indexed by a secrecy label in a lattice. A value of type τ_ℓ corresponds to a value of type τ that is indexed by a particular secrecy level ℓ. The static typing discipline then ensures that in a well-typed program, computations of some value at a certain secrecy level are never influenced by data coming from a higher secrecy level. In other words, confidential data at a level ℓ can never leak — directly or indirectly — to a level $\ell' \preccurlyeq \ell$. This general property is referred to as "non-interference".

We have (given $L \preccurlyeq H$) the following contextual equivalence in this language.

$$\lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L.\, \lambda^H b : \mathbb{B}_H.\, (f\ b) \simeq_{ctx} \lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L.\, \lambda^H b : \mathbb{B}_H.\, (f\ \neg b)$$

Indeed, by non-interference, a function of type $\mathbb{B}_H \xrightarrow{H} \mathbb{B}_L$ cannot actually depend upon its argument. Therefore, we can safely negate its argument for instance.

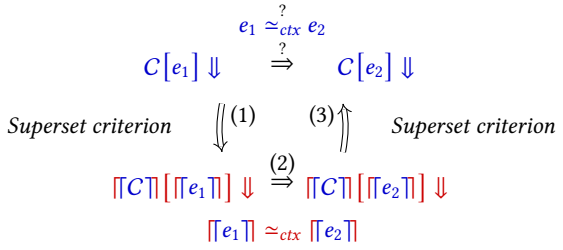

Fig. 3. Reflection of equivalences: overview proof. Structure of diagram taken from Devriese et al. [2016].

Like before, in §3.1, naively applying AGT (with evidence represented by pairs of labels) gives us a language in which this is not the case anymore. Toro et al. [2018] show that this language cannot hope to satisfy non-interference by giving the following example:

$$\texttt{true}_? :: \mathbb{B}_H :: \mathbb{B}_? :: \mathbb{B}_L \longrightarrow^* \texttt{true}_L$$

From this counterexample, it is again not hard to construct a gradual context C demonstrating that the aforementioned equivalence is not preserved:

$$C \triangleq [\cdot]\ g\ t \text{ where}$$
$$g \triangleq \lambda^H b : \mathbb{B}_H.\, b :: \mathbb{B}_? :: \mathbb{B}_L$$
$$t \triangleq \texttt{true}_? :: \mathbb{B}_H$$

Indeed, we now have the following:

$$\lceil \lceil \lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L.\, \lambda^H b : \mathbb{B}_H.\, (f\ b) \rceil \rceil\ g\ t \longrightarrow^* \texttt{true}_L$$
$$\lceil \lceil \lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L.\, \lambda^H b : \mathbb{B}_H.\, (f\ \neg b) \rceil \rceil\ g\ t \longrightarrow^* \texttt{false}_L$$

4 FAE FOR THE GTLC$_\mu$

Now that we have demonstrated that the FAE criterion can successfully detect gradual languages that are unsatisfactory in the sense that they do not preserve static equivalences in the gradualization, the question remains whether the criterion is not too ambitious. Can we really hope for gradual languages that preserve arbitrary static equivalences? In this section, we show that an FAE may not be too much to hope for, by proving the property in at least one simple but representative setting. We report on our computer-verified proof[9] that it is indeed satisfied by the GTLC$_\mu$ for source language STLC$_\mu$, laid out in §2.1.

In §4.1, we present a high-level overview of the proof, introducing the concept of a *backtranslation* and its role in the proof. We present our backtranslation in §4.2, and we prove its validity in §4.3.

4.1 High-level Overview

Proving Reflection of Contextual Equivalences. Proving the reflection of contextual equivalences is the less interesting part of this proof, as it directly follows from the superset criterion. Below, we provide a quick proof.

[9]It can be checked out at https://github.com/scaup/fae-gtlc-mu.

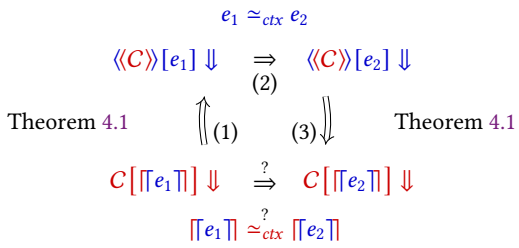

Fig. 4. Preservation of equivalences: overview proof. Structure of diagram taken from Devriese et al. [2016].

Given two arbitrary static terms, say e_1 and e_2, whose embeddings are equivalent in the gradual language, we need to prove that they are equivalent in the static language. Thus given an arbitrary static context, say C, we need to prove that $C\left[e_1\right] \Downarrow$ implies $C\left[e_2\right] \Downarrow$ (the reverse direction is symmetric). As depicted in fig. 3, this follows directly from the superset criterion (1,3) and the given contextual equivalence in the gradual language, instantiated with $\lceil\lceil C \rceil\rceil$ (2).

In §4.3, we briefly mention how the same can be proven without appeal to the superset criterion.

Proving Preservation of Contextual Equivalences. Intuitively, we need to prove somehow that gradual contexts do not possess more distinguishing power over fully annotated terms than static contexts do. One way to do this is to *emulate* the behavior of a *gradual context* by that of a *static context*. Formally, we would have an emulation function (following the literature on fully abstract compilation proofs [Patrignani et al. 2019a], we also refer to this function as a "backtranslation"), $\langle\!\langle \_ \rangle\!\rangle : C \mapsto \langle\!\langle C \rangle\!\rangle$, which maps gradual contexts to their appropriate emulations in the static language. With this in mind, we have the following proof sketch.

Given two equivalent static terms, say $e_1 \simeq_{ctx} e_2$, we need to prove that $\lceil\lceil e_1 \rceil\rceil$ and $\lceil\lceil e_2 \rceil\rceil$ are equivalent in the gradual language. That is, given an arbitrary gradual context, say C, we need to prove that $C\left[\lceil\lceil e_1 \rceil\rceil\right] \Downarrow$ implies $C\left[\lceil\lceil e_2 \rceil\rceil\right] \Downarrow$ (the reverse direction is again symmetric). As depicted in fig. 4, this follows if we can construct a valid emulation function (1,3), together with the given contextual equivalence in the static language, instantiated with $\langle\!\langle C \rangle\!\rangle$ (2).

What it means for our emulation function to be valid, is easily extracted from the requirements of the proof; we have the following specification.

Theorem 4.1 (Specification for the backtranslation). *Given arbitrary well-typed static term, $\Gamma \vdash e : \tau$, and arbitrary target context, $C : (\lceil\lceil \Gamma \rceil\rceil ; \lceil\lceil \tau \rceil\rceil) \Rightarrow (\cdot ; \mathbb{1})$, we must have that*

$$\langle\!\langle C \rangle\!\rangle : (\Gamma ;\ \tau) \Rightarrow (\cdot ; \mathbb{1}) \ \textit{and}\ \langle\!\langle C \rangle\!\rangle [e] \Downarrow \ \textit{iff}\ C\left[\lceil\lceil e \rceil\rceil\right] \Downarrow.$$

Exactly defining our backtranslation and proving that it satisfies theorem 4.1 will be the subject of §4.2 and §4.3 respectively.

4.2 Backtranslation

As seen in §2.1, the dynamics of GTLC$_\mu$ are defined by a translation to the cast calculus, in which non-trivial consistencies are interspersed with explicit casts. We define our backtranslation directly on this cast calculus; this turns out to be much easier, despite the fact that the cast calculus allows for much more contexts. Throughout this section, we define the backtranslation only on *terms*, but it naturally extends to contexts.

$$\langle\!\langle k \rangle\!\rangle \triangleq k \qquad \langle\!\langle (e_1, e_2) \rangle\!\rangle \triangleq (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle)$$
$$\langle\!\langle x \rangle\!\rangle \triangleq x \qquad \langle\!\langle \mathtt{inj}_1\ e \rangle\!\rangle \triangleq \mathtt{inj}_1\ \langle\!\langle e \rangle\!\rangle$$
$$\langle\!\langle \lambda x : \tau.\, e \rangle\!\rangle \triangleq \lambda x : \langle\!\langle \tau \rangle\!\rangle.\, \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathtt{inj}_2\ e \rangle\!\rangle \triangleq \mathtt{inj}_2\ \langle\!\langle e \rangle\!\rangle$$
$$\langle\!\langle e_1\ e_2 \rangle\!\rangle \triangleq \langle\!\langle e_1 \rangle\!\rangle\ \langle\!\langle e_2 \rangle\!\rangle \qquad \langle\!\langle \mathtt{case}\ e\ \mathtt{of}\ (e_1 \mid e_2) \rangle\!\rangle \triangleq \mathtt{case}\ \langle\!\langle e \rangle\!\rangle\ \mathtt{of}\ (\langle\!\langle e_1 \rangle\!\rangle \mid \langle\!\langle e_2 \rangle\!\rangle)$$
$$\langle\!\langle \pi_1\ e \rangle\!\rangle \triangleq \pi_1\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathtt{fold}\ e \rangle\!\rangle \triangleq \mathtt{fold}\ \langle\!\langle e \rangle\!\rangle$$
$$\langle\!\langle \pi_2\ e \rangle\!\rangle \triangleq \pi_2\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathtt{unfold}\ e \rangle\!\rangle \triangleq \mathtt{unfold}\ \langle\!\langle e \rangle\!\rangle$$

Fig. 5. The uninteresting parts of the backtranslation on expressions.

$$\langle\!\langle B \rangle\!\rangle \triangleq B$$
$$\langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle \triangleq \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_2 \rangle\!\rangle$$
$$\langle\!\langle \tau_1 + \tau_2 \rangle\!\rangle \triangleq \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle$$
$$\langle\!\langle \tau_1 \times \tau_2 \rangle\!\rangle \triangleq \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle$$
$$\langle\!\langle X \rangle\!\rangle \triangleq X$$
$$\langle\!\langle \mu X.\, \tau \rangle\!\rangle \triangleq \mu X.\, \langle\!\langle \tau \rangle\!\rangle$$

Fig. 6. The uninteresting parts of the backtranslation on types.

The Easy Part. Structurally, the cast calculus is just the static language extended with the cast construct. Much of the backtranslation is therefore quite uninteresting, as can be seen in fig. 5.

Naturally, these parts of the backtranslating preserve well-typedness; a fully annotated term (i.e. a term with no casts) of type τ is backtranslated to a term of type $\langle\!\langle \tau \rangle\!\rangle$, where the uninteresting parts of the backtranslation on types can be seen in fig. 6. We quickly note also that embedding terms from the static language into the gradual one and backtranslating them forms a retraction pair; we have that $\langle\!\langle \lceil\!\lceil \_ \rceil\!\rceil \rangle\!\rangle$ is the identity function.

The Universe. Of course, things get more interesting when we look at non-static gradual expressions. To begin with, we require a static type that can accommodate expressions of, for instance, the unknown type for which — semantically — expressions can have *any* static type. Taking inspiration from Devriese et al. [2016]; New et al. [2016], we will backtranslate expressions of the unknown type to expressions of the following static type.

$$\langle\!\langle ? \rangle\!\rangle \triangleq \mu X.\, (B + (X \to X) + (X + X) + (X \times X) + X) \tag{1}$$

This type is often referred to as a universal type of the STLC$_\mu$, as we can embed every static type in it. Throughout this section, we denote it by $\mathscr{U}$. The above completes the backtranslation on types given in fig. 6.

For convenience, we will use a variant-like syntax with labels (even though technically, we only have binary sum types):

$$\langle (B : B); (\to : (\mathscr{U} \to \mathscr{U})); (+ : (\mathscr{U} + \mathscr{U})); (\times : (\mathscr{U} \times \mathscr{U})); (\mu : \mathscr{U}) \rangle$$

This syntax allows us to use constructors $\mathtt{inj}_B$, $\mathtt{inj}_\to$, $\mathtt{inj}_\times$, and $\mathtt{inj}_\mu$, as well as the following generalized case match.

$$\mathtt{case}\ e\ \mathtt{of}\ (\lambda x : B.\, e_1) \mid (\lambda x : \mathscr{U} + \mathscr{U}.\, e_2) \mid (\lambda x : \mathscr{U} \times \mathscr{U}.\, e_3) \mid (\lambda x : \mathscr{U} \to \mathscr{U}.\, e_4) \mid (\lambda x : \mathscr{U}.\, e_5)$$

$$
\begin{aligned}
\mathit{inject}_B &\triangleq \lambda u : B.\ \mathtt{fold}\ (\mathtt{inj}_B\ u) : B \to \mathscr{U} \\
\mathit{inject}_{?+?} &\triangleq \lambda u : \mathscr{U} + \mathscr{U}.\ \mathtt{fold}\ (\mathtt{inj}_+\ u) : \mathscr{U} + \mathscr{U} \to \mathscr{U} \\
\mathit{inject}_{?\times?} &\triangleq \lambda u : \mathscr{U} \times \mathscr{U}.\ \mathtt{fold}\ (\mathtt{inj}_\times\ u) : \mathscr{U} \times \mathscr{U} \to \mathscr{U} \\
\mathit{inject}_{?\to?} &\triangleq \lambda u : \mathscr{U} \to \mathscr{U}.\ \mathtt{fold}\ (\mathtt{inj}_\to\ u) : (\mathscr{U} \to \mathscr{U}) \to \mathscr{U} \\
\mathit{inject}_{\mu X.?} &\triangleq \lambda u : \mu X.\mathscr{U}.\ \mathtt{fold}\ (\mathtt{inj}_\mu\ (\mathtt{unfold}\ u)) : \mu X.\mathscr{U} \to \mathscr{U}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{extract}_B &\triangleq \lambda u : \mathscr{U}.\ \mathtt{case}\ (\mathtt{unfold}\ u)\ \mathtt{of}\ (\lambda x : B.\, x) \mid (\lambda x : \_.\, \Omega_B) : \mathscr{U} \to B \\
\mathit{extract}_{?+?} &\triangleq \lambda u : \mathscr{U}.\ \mathtt{case}\ (\mathtt{unfold}\ u)\ \mathtt{of}\ (\lambda x : \mathscr{U} + \mathscr{U}.\, x) \mid (\lambda x : \_.\, \Omega_{\mathscr{U}+\mathscr{U}}) : \mathscr{U} \to (\mathscr{U} + \mathscr{U}) \\
\mathit{extract}_{?\times?} &\triangleq \lambda u : \mathscr{U}.\ \mathtt{case}\ (\mathtt{unfold}\ u)\ \mathtt{of}\ (\lambda x : \mathscr{U} \times \mathscr{U}.\, x) \mid (\lambda x : \_.\, \Omega_{\mathscr{U}\times\mathscr{U}}) : \mathscr{U} \to (\mathscr{U} \times \mathscr{U}) \\
\mathit{extract}_{?\to?} &\triangleq \lambda u : \mathscr{U}.\ \mathtt{case}\ (\mathtt{unfold}\ u)\ \mathtt{of}\ (\lambda x : \mathscr{U} \to \mathscr{U}.\, x) \mid (\lambda x : \_.\, \Omega_{\mathscr{U}\to\mathscr{U}}) : \mathscr{U} \to (\mathscr{U} \to \mathscr{U}) \\
\mathit{extract}_{\mu X.?} &\triangleq \lambda u : \mathscr{U}.\ \mathtt{case}\ (\mathtt{unfold}\ u)\ \mathtt{of}\ (\lambda x : \mathscr{U}.\, \mathtt{fold}\ x) \mid (\lambda x : \_.\, \Omega_{\mu X.\mathscr{U}}) : \mathscr{U} \to \mu X.\mathscr{U}
\end{aligned}
$$

Fig. 7. Defining the injection and extraction functions for every ground type

Moreover, we shall further abbreviate this if in all but one of the branches we have the same function body (that is independent of the input argument). The expression below for instance, shall be abbreviated to $\mathtt{case}\ e\ \mathtt{of}\ (\lambda x : \mathscr{U} + \mathscr{U}.\, e'') \mid (\lambda \_ : \_.\, e')$.

$$\mathtt{case}\ e\ \mathtt{of}\ (\lambda \_ : B.\, e') \mid (\lambda x : \mathscr{U} + \mathscr{U}.\, e'') \mid (\lambda \_ : \mathscr{U} \times \mathscr{U}.\, e') \mid (\lambda \_ : \mathscr{U} \to \mathscr{U}.\, e') \mid (\lambda \_ : \mathscr{U}.\, e')$$

Backtranslating Casts to Functions. The only non-static construct in our gradual language is the cast construct. To complete the backtranslation on terms from fig. 5, it remains to define the backtranslation of that construct. We will backtranslate a cast of a term e from τ to τ' to the application of a function, $\mathscr{F}_{\tau\Rightarrow\tau'}$, to be defined later:

$$\langle\!\langle e : \tau \Rightarrow \tau' \rangle\!\rangle \triangleq \mathscr{F}_{\tau\Rightarrow\tau'}\ \langle\!\langle e \rangle\!\rangle \tag{2}$$

To ensure well-typedness of our backtranslation, this function $\mathscr{F}_{\tau\Rightarrow\tau'}$ must be of type $\langle\!\langle \tau \rangle\!\rangle \to \langle\!\langle \tau' \rangle\!\rangle$.

Some Simple Examples. Let us look at some simple examples.

Consider some value of base type, $v : B$. Given $\langle\!\langle v \rangle\!\rangle : B$, we backtranslate $v : B \Rightarrow ?$ by injecting $\langle\!\langle v \rangle\!\rangle$ into the universe:

$$\langle\!\langle v : B \Rightarrow ? \rangle\!\rangle = (\lambda u : B.\ \mathtt{fold}\ (\mathtt{inj}_B\ u))\ \langle\!\langle v \rangle\!\rangle$$

Note that the application of this function always succeeds; it will always terminate to a value of type $\mathscr{U}$, reflecting the fact that $v : B \Rightarrow ?$ is a value of type $?$.

Conversely though, casting *down* a value of the unknown type, say $v : ?$, to base type B should only go through if the unknown value was "of the right form", i.e. $v = (b : B \Rightarrow ?)$. Accordingly, we backtranslate $v : ? \Rightarrow B$ as follows:

$$\langle\!\langle v : ? \Rightarrow B \rangle\!\rangle = (\lambda u : \mathscr{U}.\ \mathtt{case}\ (\mathtt{unfold}\ u)\ \mathtt{of}\ (\lambda x : B.\, x) \mid (\lambda x : \_.\, \Omega_B))\ \langle\!\langle v \rangle\!\rangle$$

If v was of the form $b : B \Rightarrow ?$, i.e. the evaluation of $\langle\!\langle v \rangle\!\rangle$ of the form $\mathtt{fold}\ (\mathtt{inj}_B\ b)$, then the application will evaluate to b. If not, it will evaluate to a diverging term Ω_B of type B, implemented using recursive types.

In general, we have for any ground type G a function $\mathit{inject}_G : \langle\!\langle G \rangle\!\rangle \to \mathscr{U}$ and a function $\mathit{extract}_G : \mathscr{U} \to \langle\!\langle G \rangle\!\rangle$, defined in fig. 7. They all follow — more or less — the same pattern. For the case of the ground type $\mu X.?$, we do have a small caveat as we have to put an extra $\mathtt{unfold}$ / $\mathtt{fold}$ into place.

$$B \rightsquigarrow B \qquad ? \rightsquigarrow ? \qquad ? \rightsquigarrow G \qquad G \rightsquigarrow ?$$

$$\frac{\tau \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad G \rightsquigarrow ?}{\tau \rightsquigarrow ?} \qquad \frac{? \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad G \rightsquigarrow \tau}{? \rightsquigarrow \tau} \qquad \frac{\tau_1' \rightsquigarrow \tau_1 \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \to \tau_2 \rightsquigarrow \tau_1' \to \tau_2'}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \times \tau_2 \rightsquigarrow \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 + \tau_2 \rightsquigarrow \tau_1' + \tau_2'} \qquad \frac{\tau[X \mapsto \mu X.\,\tau] \rightsquigarrow \tau'[Y \mapsto \mu Y.\,\tau']}{\mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'}$$

Fig. 8. Decomposition of casts in terms of each other.

We remark that the extract and inject functions satisfy the specification that arises from the evaluation rule *Ground-Id*, as composing $\mathit{extract}_G$ with inject_G will behave as the identity function.

Composing $\mathit{extract}_{G_1}$ with inject_{G_2} where $G_1 \neq G_2$, will behave as $\lambda x : \langle\!\langle G_2 \rangle\!\rangle.\Omega_{\langle\!\langle G_1 \rangle\!\rangle}$, in accordance to the *Ground-Fail* evaluation rule. Indeed, we are backtranslating a cast error to a diverging term.

$$\langle\!\langle \texttt{CastError} : \tau \rangle\!\rangle \triangleq \Omega_{\langle\!\langle \tau \rangle\!\rangle}$$

Casts that Decompose. Towards a general definition of the backtranslation for the casts, we want to mimic the cast calculus as faithful as possible, as doing so will make it easier to prove that it satisfies theorem 4.1. More precisely, we want our backtranslation to reflect the manner in which casts are decomposed in terms of each other. We summarize this structure (the decompositions of casts) in a binary relation on types, depicted in fig. 8; *derivations* of this relation reflect the way in which casts are decomposed in terms of each other.

As a first step, we now start defining the backtranslation by induction on this relation, after which we will tweak the relation slightly to accommodate the backtranslation of casts between recursive types.

The atomic casts are backtranslated easily enough. We backtranslate $G \Rightarrow ?$ to inject_G and $? \Rightarrow G$ to $\mathit{extract}_G$ from the previous section, reflecting both *Ground-Id* and *Ground-Fail*. The casts $B \Rightarrow B$ and $? \Rightarrow ?$ are trivially backtranslated to $\lambda x : B.\ x$ and $\lambda x : \mathscr{U}.\ x$ respectively, mirroring *Base-Id* and *Unknown-Id*.

We will consider the compositional casts now. Given $f_1 : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_1' \rangle\!\rangle$, $f_2 : \langle\!\langle \tau_2 \rangle\!\rangle \to \langle\!\langle \tau_2' \rangle\!\rangle$, backtranslations for $\tau_1 \Rightarrow \tau_1'$, $\tau_2 \Rightarrow \tau_2'$ respectively, we can backtranslate $\tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2'$, mirroring *Factor-Sum*:

$$\lambda s : \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle.\ \texttt{case}\ s\ \texttt{of}\ \lambda x_1 : \langle\!\langle \tau_1 \rangle\!\rangle.\ \texttt{inj}_1\ (f_1\ x_1)\ \mid\ \lambda x_2 : \langle\!\langle \tau_2 \rangle\!\rangle.\ \texttt{inj}_2\ (f_2\ x_2)$$

Similarly, given $f_1 : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_1' \rangle\!\rangle$, $f_2 : \langle\!\langle \tau_2 \rangle\!\rangle \to \langle\!\langle \tau_2' \rangle\!\rangle$, backtranslations for $\tau_1 \Rightarrow \tau_1'$, $\tau_2 \Rightarrow \tau_2'$ respectively, we backtranslate $\tau_1 \times \tau_2 \Rightarrow \tau_1' \times \tau_2'$, mirroring *Factor-Prod*.

$$\lambda p : \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle.\ (f_1\ (\pi_1\ p),\ f_2\ (\pi_2\ p))$$

Given $f_1 : \langle\!\langle \tau_1' \rangle\!\rangle \to \langle\!\langle \tau_1 \rangle\!\rangle$ and $f_2 : \langle\!\langle \tau_2 \rangle\!\rangle \to \langle\!\langle \tau_2' \rangle\!\rangle$, backtranslations for $\tau_1' \Rightarrow \tau_1$ and $\tau_2 \Rightarrow \tau_2'$ respectively, we mirror the backtranslation of $\tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'$ according to *Factor-App*.

$$\lambda f : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_2 \rangle\!\rangle.\ \lambda x : \langle\!\langle \tau_1' \rangle\!\rangle.\ f_2\ (f\ (f_1\ x))$$

This reflects the suspended nature of the cast as f_1 will only be applied on the argument when it is actually supplied, after which f_2 is applied on the result (after applying f).

Unfortunately, the relation as is does not allow for a valid definition of the backtranslation; delegating a cast between two recursive types, say $\mu X.\,\tau \Rightarrow \mu Y.\,\tau'$ to $\tau[X \mapsto \mu X.\,\tau] \Rightarrow \tau'.[Y \mapsto \mu Y.\,\tau']$, would clearly make the definition cyclic. Naively backtranslating, for instance, the cast $\mu X.\,? \times X \Rightarrow$

$$\textsc{Atomic-Base}\quad F \vdash_s B \rightsquigarrow B$$

$$\textsc{Atomic-Unknown}\quad F \vdash_s ? \rightsquigarrow ?$$

$$\textsc{Atomic-Unknown-G}\quad F \vdash_s ? \rightsquigarrow G$$

$$\textsc{Atomic-G-Unknown}\quad F \vdash_s G \rightsquigarrow ?$$

$$\textsc{FactorUp-G}\quad \frac{F \vdash_s \tau \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad F \vdash_s G \rightsquigarrow ?}{F \vdash_s \tau \rightsquigarrow ?}$$

$$\textsc{FactorDown-G}\quad \frac{F \vdash_s ? \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad F \vdash_s G \rightsquigarrow \tau}{F \vdash_s ? \rightsquigarrow \tau}$$

$$\textsc{ThroughArrow}\quad \frac{F \vdash_s \tau_1' \rightsquigarrow \tau_1 \qquad F \vdash_s \tau_2 \rightsquigarrow \tau_2'}{F \vdash_s \tau_1 \to \tau_2 \rightsquigarrow \tau_1' \to \tau_2'}$$

$$\textsc{ThroughSum}\quad \frac{F \vdash_s \tau_1 \rightsquigarrow \tau_1' \qquad F \vdash_s \tau_2 \rightsquigarrow \tau_2'}{F \vdash_s \tau_1 + \tau_2 \rightsquigarrow \tau_1' + \tau_2'}$$

$$\textsc{ThroughProd}\quad \frac{F \vdash_s \tau_1 \rightsquigarrow \tau_1' \qquad F \vdash_s \tau_2 \rightsquigarrow \tau_2'}{F \vdash_s \tau_1 \times \tau_2 \rightsquigarrow \tau_1' \times \tau_2'}$$

$$\textsc{ExposeRecursiveCall}\quad \frac{(\mu X.\,\tau \rightsquigarrow \mu Y.\,\tau') \notin \mathrm{Im}(F) \qquad F(f)\ \textit{undefined} \qquad F[(f \mapsto \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau')] \vdash_s \tau[X \mapsto \mu X.\,\tau] \rightsquigarrow \tau'[Y \mapsto \mu Y.\,\tau']}{F \vdash_s \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'}$$

$$\textsc{Atomic-UseRecursion}\quad \frac{F(f) = \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'}{F \vdash_s \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'}$$

Fig. 9. Alternative consistency relation for definition of the backtranslation.

$\mu X.\,\mathbb{N} \times X$, we would obtain the following infinite definition:

$$\lambda x : \mu X.\,(\mathscr{U} \times X)\,.\,\mathtt{fold}\,\left((\lambda p : \mathscr{U} \times \mu X.\,(\mathscr{U} \times X)\,.\,(\mathit{extract}_{\mathbb{N}}\ (\pi_1\ p)\,,\ f\ (\pi_2\ p)))\ (\mathtt{unfold}\ x)\right)$$
$$\textit{where}\ \ f \triangleq \lambda x : \mu X.\,(\mathscr{U} \times X).\ldots$$

To solve this, we simply implement a fixpoint operator with recursive types that we then use to expose a recursive call, i, that we make use of accordingly:

$$\lambda y : \mu X.\,(\mathscr{U} \times X).\,\overrightarrow{\mathit{fix}}(\lambda i : \mu X.\,(\mathscr{U} \times X) \to \mu Y.\,(\mathbb{N} \times Y).$$
$$(\lambda x : \mu X.\,(\mathscr{U} \times X).\,\mathtt{fold}\ ((\lambda p : \mathscr{U} \times \mu X.\,(\mathscr{U} \times X).\,(\mathit{extract}_{\mathbb{N}}\ (\pi_1\ p),\ i\ (\pi_2\ p)))\ (\mathtt{unfold}\ x))))\ y$$

The General Definition. To accommodate our last example, we slightly tweak the relation in fig. 8, obtaining the one depicted in fig. 9. It is by induction on this relation that we establish the general definition of the backtranslation on casts. In it, every judgment is now accompanied by a partial function mapping static variables to a pair of gradual types. In the backtranslation, this partial function will encode the variables in scope that correspond to a recursive call. A variable, say f, for which $F(f) = \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'$, corresponds to a recursive call to a function of type $\langle\!\langle \mu X.\,\tau \rangle\!\rangle \to \langle\!\langle \mu Y.\,\tau' \rangle\!\rangle$.

Backtranslating a judgment between two recursive types, say $F \vdash_s \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'$, to a static function (to be denoted by $\mathscr{F}^{F}_{\mu X.\tau \rightsquigarrow \mu Y.\tau'}$), we follow one of the following two paths.

A) If we "previously" exposed a recursive call for this already, i.e. we have $F(f) = \mu X.\,\tau \rightsquigarrow \mu Y.\,\tau'$, then the backtranslation is exactly that variable f.

B) If we have no exposed call for this, i.e. $(\mu X.\,\tau \rightsquigarrow \mu Y.\,\tau') \notin \mathrm{Im}(F)$, then we unfold the bodies and expose an additional recursive call. That is, we now consider the backtranslation of the judgment $F[(f \mapsto \mu X.\,\tau \sim \mu Y.\,\tau')] \vdash_s \tau[X \mapsto \mu X.\,\tau] \rightsquigarrow \tau[Y \mapsto \mu Y.\,\tau']$, where we have extended F to $F[(f \mapsto \mu X.\,\tau \sim \mu Y.\,\tau')]$ for some fresh f.

$$
\begin{aligned}
\mathscr{F}^F_{B\rightsquigarrow B} &\triangleq \lambda x : \langle\langle B\rangle\rangle.\ x \\
\mathscr{F}^F_{?\rightsquigarrow ?} &\triangleq \lambda x : \langle\langle ?\rangle\rangle.\ x \\
\mathscr{F}^F_{G\rightsquigarrow ?} &\triangleq \mathit{inject}_G \\
\mathscr{F}^F_{?\rightsquigarrow G} &\triangleq \mathit{extract}_G \\
\mathscr{F}^F_{\tau\rightsquigarrow ?} &\triangleq \lambda x : \langle\langle \tau\rangle\rangle.\ \mathscr{F}^F_{G\rightsquigarrow ?}\ (\mathscr{F}^F_{\tau\rightsquigarrow G}\ x) \quad \text{if } \mathcal{S}(\tau) = G \neq \tau \\
\mathscr{F}^F_{?\rightsquigarrow \tau} &\triangleq \lambda x : \langle\langle ?\rangle\rangle.\ \mathscr{F}^F_{G\rightsquigarrow \tau}\ (\mathscr{F}^F_{?\rightsquigarrow G}\ x) \quad \text{if } \mathcal{S}(\tau) = G \neq \tau \\
\mathscr{F}^F_{\tau_1\to\tau_2\rightsquigarrow\tau_1'\to\tau_2'} &\triangleq \lambda f : \langle\langle\tau_1\rangle\rangle \to \langle\langle\tau_2\rangle\rangle.\ \lambda x : \langle\langle\tau_1'\rangle\rangle.\ \mathscr{F}^F_{\tau_2\rightsquigarrow\tau_2'}\ (f\ (\mathscr{F}^F_{\tau_1'\rightsquigarrow\tau_1}\ x)) \\
\mathscr{F}^F_{\tau_1+\tau_2\rightsquigarrow\tau_1'+\tau_2'} &\triangleq \lambda s : \langle\langle\tau_1\rangle\rangle + \langle\langle\tau_2\rangle\rangle.\ \mathtt{case}\ s\ \mathtt{of}\ \lambda x_1 : \langle\langle\tau_1\rangle\rangle.\ \mathtt{inj}_1\ (\mathscr{F}^F_{\tau_1\rightsquigarrow\tau_1'}\ x_1)\ \mid\ \lambda x_2 : \langle\langle\tau_2\rangle\rangle.\ \mathtt{inj}_2\ (\mathscr{F}^F_{\tau_2\rightsquigarrow\tau_2'}\ x_2) \\
\mathscr{F}^F_{\tau_1\times\tau_2\rightsquigarrow\tau_1'\times\tau_2'} &\triangleq \lambda p : \langle\langle\tau_1\rangle\rangle \times \langle\langle\tau_2\rangle\rangle.\ (\mathscr{F}^F_{\tau_1\rightsquigarrow\tau_1'}(\pi_1\ p),\ \mathscr{F}^F_{\tau_2\rightsquigarrow\tau_2'}(\pi_2\ p)) \\
\mathscr{F}^F_{\mu X.\tau\rightsquigarrow\mu Y.\tau'} &\triangleq \begin{cases} f & F(f) = \mu X.\tau \rightsquigarrow \mu Y.\tau' \\ \lambda x : \mu X.\ \langle\langle\tau\rangle\rangle.\ \overrightarrow{\mathit{fix}}(\lambda f : \mu X.\ \langle\langle\tau\rangle\rangle \to \mu Y.\ \langle\langle\tau'\rangle\rangle. & \\ \quad \lambda r : \mu X.\ \langle\langle\tau\rangle\rangle.\ \mathtt{fold}\ (\mathscr{F}^{F[(f\mapsto\mu X.\tau\rightsquigarrow\mu Y.\tau')]}_{\tau[X\mapsto\mu X.\tau]\rightsquigarrow\tau'[Y\mapsto\mu Y.\tau']}\ (\mathtt{unfold}\ r))) \ x & (\mu X.\tau \rightsquigarrow \mu Y.\tau') \notin \mathrm{Im}(F) \end{cases}
\end{aligned}
$$

Fig. 10. Defining the function in the backtranslation of casts

The complete backtranslation by induction on the relation in fig. 9 is now depicted in fig. 10; the function $\mathscr{F}_{\tau\Rightarrow\tau'}$ in eq. (2) is now defined as $\mathscr{F}^{\emptyset}_{\tau\rightsquigarrow\tau'}$.

Is this Well-defined? Our backtranslation needs to be defined between *any* two consistent types. Therefore, the validity of this definition depends on the assumption that for each two consistent types, $\tau \sim \tau'$, we can derive $\emptyset \vdash_s \tau \rightsquigarrow \tau'$. Fortunately, this is indeed the case, as we have formally proven it in our Coq development. What follows is an intuitive explanation for why this is so.

Given two arbitrary (closed) consistent types, $\tau \sim \tau'$, one can imagine the following procedure to attempt building up a derivation tree for $\emptyset \vdash_s \tau \rightsquigarrow \tau'$: If $\emptyset \vdash_s \tau \rightsquigarrow \tau'$ is a base case, then great, we are done. Else, we just apply (backward reasoning) the applicable inference-rule (it is easily checked that there is always a *unique* rule that is applicable) and instantiate the procedure among all newly created subbranches.

Of course, this procedure needs to terminate! That is, we need to obtain a finite tree. To guarantee this, something must therefore consistently decrease — in a well-founded ordering — in the application of every inference rule.

This does appear the case for *most* of the constructors in the relation: One of the types in the pair of an argument (a judgment above the line) seems to — eventually — always be strictly smaller (structurally) than one of the types in the pair of the conclusion, while the remaining type in the pair of the argument is structurally smaller or equal than that of the remaining type in the conclusion.[10]

Considering ThroughSum for instance, we have $\tau_1 < \tau_1 + \tau_2$ and $\tau_1' < \tau_1' + \tau_2'$ for the first argument and $\tau_2 < \tau_1 + \tau_2$ and $\tau_2' < \tau_1' + \tau_2'$ for the second. A similar results applies to ThroughProd and ThroughArrow even though the latter is contravariant in one of its arguments. Composing FactorDown-$(? + ?)$ (taking τ to be $\tau_1 + \tau_2$) with ThroughSum, we get $? \leq ?$ and $\tau_1 < \tau_1 + \tau_2$ for the first argument, and $? \leq ?$ and $\tau_2 < \tau_1 + \tau_2$ for the second. Similar results apply to the composition of FactorDown-$(? \times ?)$ with ThroughProd and the composition between FactorDown-$(? \to ?)$ with ThroughArrow.

[10] We will write $<$ for structurally smaller than and $\leq$ for structurally smaller or equal than.

Unfortunately, the same is not true for ExposeRecursiveCall and possibly FactorUp-($\mu X.\,?$) and FactorDown-($\mu X.\,?$). In ExposeRecursiveCall for instance, the types *in the conclusion* are typically structurally smaller than those in the arguments!

Fortunately, this is not a problem however, as it is guaranteed that ExposeRecursiveCall will only be applicable a *finite* amount of times. Indeed, there is only a finite amount of recursive calls to be exposed before having exerted all possibilities, after which the base-case counterpart (Atomic-UseRecursion) is bound to be applicable.

In the previous example, $\mu X.\,? \times X \sim \mu X.\,\mathbb{N} \times X$, we apply ExposeRecursiveCall one time, after which Atomic-UseRecursion becomes applicable. Generally though, some branches will require multiple applications of ExposeRecursiveCall before Atomic-UseRecursion becomes applicable. As an example, one can consider a derivation tree for $\cdot \vdash_s \mu X.\,X + ? + X \to ? + X + X \rightsquigarrow \mu X.\,X + X + ? \to X + ? + X$.

The reader who is interested in the formal side of this argument, is welcome to check out our Coq code.

4.3 Logical Relations and Correctness of the Backtranslation

Of course, the backtranslation is only as useful as it is correct; in order to complete the proof in §4.1, we need to prove that theorem 4.1 holds. To do so, we set up two logical relations models, as is often done in fully abstract compilation proofs [Patrignani et al. 2019a].

We define our logical relations in the Iris program logic [Jung et al. 2018]. Iris features an abstract form of step-indexing, a technique that is usually employed in defining logical relations models for programming languages featuring recursive types [Appel and McAllester 2001]. Using Iris allows us to reason about programs at a high level of abstraction, especially for the use of step indexes. In addition to recursive types, we also use step-indexing for modeling the unknown type in our logical relations. The use of a program logic for expressing logical relations for programming languages featuring recursive types goes back to Dreyer et al. [2009].

In §4.3.1 we present a short Iris primer. Afterwards, in §4.3.2, we specify the properties that our logical relations should satisfy in order to be able to prove the correctness of our backtranslation (see theorem 4.1). We define our logical relations models in §4.3.3. In §4.3.4, we demonstrate how our logical relations models satisfy the properties presented in §4.3.2.

To get the intuition behind the essence of our logical relations, one does not need to possess any knowledge of Iris. Accordingly, hurried readers who feel comfortable to brush aside the technicalities may skip §4.3.1 and go over §4.3.3 by focusing on the logical relations on values in fig. 11 (ignoring the modalities and reading $*$ as $\wedge$) and trusting that the relations are naturally extended to (open) expressions as described informally in the other paragraphs of that section.

4.3.1 An Iris Primer. Iris [Jung et al. 2018] is a modal, separation logic geared toward reasoning about the correctness of programs. While the framework is vast, here, we restrict ourselves to the small subset necessary to understand our logical relations.

Below, we present the grammar of the relevant parts of *iProp*, the universe of Iris propositions.

$$P ::= P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \triangleright P \mid x \mid \mu x.P \mid \boxed{P} \mid \Rightarrow\!\!\!\Vdash P \mid \mathsf{wp}\ e\ \{\Phi\} \mid \ldots$$

Iris is a *separation* logic, hence we have separating conjunction, $P * Q$ (pronounced "P star Q"), and a magic wand operator, $P \mathrel{-\!\!*} Q$ (pronounced "P wand Q"), analogous respectively to ordinary conjunction and ordinary implication. In general, Iris propositions are *ephemeral*; we can only use them once (similar to linear/affine types). That is, once we use an Iris proposition to prove something, we lose it; we have "consumed" it. For instance, if in trying to prove $P * Q \vdash R * S$ we

use P to prove R, then only Q remains in the proof effort of S. Here, $\vdash$ is the entailment relation on Iris propositions.

Besides ephemeral propositions, we have *persistent* ones; propositions that can be freely duplicated. That is, if P is persistent, then we have $P \vdash P * P$. The $\Box$-modality, $\Box P$ (pronounced "persistently P"), asserts that P holds *persistently*. In particular, $\Box P$ is persistent and hence duplicable, and we have $\Box P \vdash P$. As we will discuss, we use the persistently modality to express and enforce that our logical relations on values are persistent.

Iris features a notion of abstract step-indexing. Given an Iris proposition P, $\triangleright P$ (pronounced "later P") asserts that P holds one step later. The abstract step-indexing in Iris allows for the construction of so-called guarded recursive propositions and predicates. The proposition $\mu x.P$ is well-defined if all occurrences of x in P are guarded, i.e., they appear under the later modality. Intuitively, guarded recursive definitions are the unique fixpoint characterized by the equation: $\mu x.P \equiv P[\mu x.P/x]$.

Iris features invariants and resources to facilitate reasoning about programs. An Iris invariant $\boxed{P}$ asserts that P holds invariantly, i.e., at all times. The update modality, $\Rrightarrow$, allows us to update resources (allocation, modification, and deallocation) and to access invariants. The proposition $\Rrightarrow P$ states that P holds *after an appropriate update of the underlying resources and/or accessing invariants.*[11] Iris invariants are persistent and can hence be freely duplicated. The following rule allows us to create invariants:

$$\triangleright P \vdash \Rrightarrow \boxed{P} \qquad \text{(invariant allocation)}$$

As a program logic, Iris is equipped with a weakest precondition calculus to reason about programs. The weakest precondition of a program e with respect to a postcondition $\Phi : \mathit{Val} \to \mathit{iProp}$, written $\mathsf{wp}\ e\ \{\Phi\}$, holds if whenever e reduces to a value v, the postcondition Φ holds for v, i.e., $\Phi(v)$ holds.[12] In some places we write $\mathsf{wp}\ e\ \{v.\ \Phi(v)\}$ instead of $\mathsf{wp}\ e\ \{\Phi\}$. Iris weakest preconditions are defined in terms of the more low-level Iris-constructs in such a way that they tie the aforementioned abstract step indexes to the evaluation steps taken by the program.

Crucially, update modalities can be eliminated when proving that weakest preconditions hold:

$$\frac{P \vdash \mathsf{wp}\ e\ \{\Phi\}}{\Rrightarrow P \vdash \mathsf{wp}\ e\ \{\Phi\}} \qquad \frac{P \vdash \mathsf{wp}\ e\ \{\Phi\}}{P \vdash \mathsf{wp}\ e\ \{v.\ \Rrightarrow \Phi(v)\}} \qquad \text{(WP and update)}$$

4.3.2 Specifying the Logical Relation Models. In order to prove theorem 4.1, we shall set up two logical relations models, $\Gamma \vDash e \preccurlyeq e' : \tau$ and $\Gamma \vDash e \succcurlyeq e' : \tau$, both indexed by *gradual* types, relating static terms to gradual terms. We shall carefully construct these relations so as to have them satisfy lemmas 4.2 to 4.5.

Lemma 4.2. *Given an arbitrary well-typed static term e for which $\Gamma \vdash e : \tau$ holds and an arbitrary context C for which $C : (\lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot\,; \mathbb{1})$ holds, we have*

$$\vdash \cdot \vDash \langle\!\langle C \rangle\!\rangle \left[e\right] \preccurlyeq C\left[\lceil\lceil e\rceil\rceil\right] : \mathbb{1}$$

Lemma 4.3 (Adequacy of $\preccurlyeq$-logical relations).

$$\textit{If} \vdash \cdot \vDash e \preccurlyeq e' : \tau, \textit{ then } e \Downarrow \textit{ implies } e' \Downarrow$$

[11] In Iris, invariants have names, and the constructs interacting with them (e.g., the to-be-discussed weakest preconditions and the update modality) are annotated with masks to track how these invariants are accessed. This is because accessing the same invariant twice in a nested fashion is generally unsound. We elide these details here which are not crucial for following the arguments presented in this paper; see our Coq development for more details.

[12] This actually corresponds to the Iris proposition $\mathsf{wp}_{\mathsf{Stuck}}\ e\ \{\Phi\}$, the *stuck* version of the weakest precondition that does not prove safety.

Lemma 4.4. *Given an arbitrary static term e for which $\Gamma \vdash e : \tau$ holds and an arbitrary context C for which $C : (\lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot\,; \mathbb{1})$ holds, we have*

$$\vdash \cdot \vDash \langle\!\langle C \rangle\!\rangle \left[e\right] \geqslant C\left[\lceil\lceil e \rceil\rceil\right] : \mathbb{1}$$

Lemma 4.5 (Adequacy of $\geqslant$-logical relations).

$$\textit{If} \vdash \cdot \vDash e \geqslant e' : \tau, \textit{ then } e' \Downarrow \textit{ implies } e \Downarrow$$

Given lemmas 4.2 to 4.5, theorem 4.1 follows easily. The challenge is of course to define the logical relation models (§4.3.3) and prove that they do indeed satisfy said lemmas(§4.3.4).

4.3.3 Defining the Logical Relation Models. We define our logical relations models following the standard pattern. We first define the logical relations on closed values and expressions. Afterwards, we canonically extend the relations on closed expressions to the general relations on open terms.

Throughout this section, we focus exclusively on the $\leqslant$-relations; the $\geqslant$-relations are defined analogously.

The Logical Relations on Values. We begin by defining the logical relations on closed values, $\mathcal{V}_{\leqslant}[\![\tau]\!] : \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$, in fig. 11. Such relations are usually defined by induction on the

$$\begin{aligned}
\mathcal{V}_{\leqslant} &\triangleq \mu\Psi.\ \mathbf{V}_{\leqslant}(\Psi) \\
\mathbf{V}_{\leqslant}[\![B]\!](v, v') &\triangleq \exists b : B.\ v = b * v' = \lceil\lceil b \rceil\rceil \\
\mathbf{V}_{\leqslant}[\![\tau_1 + \tau_2]\!](v, v') &\triangleq \bigvee_{i \in \{1,2\}} \exists v_i, v_i'.\ v = \mathtt{inj}_i\ v_i * v' = \mathtt{inj}_i\ v_i' * \mathbf{V}_{\leqslant}[\![\tau_i]\!](v_i, v_i') \\
\mathbf{V}_{\leqslant}[\![\tau_1 \times \tau_2]\!](v, v') &\triangleq \exists v_1, v_2, v_1', v_2'.\ v = (v_1, v_2) * v' = (v_1', v_2') * \mathbf{V}_{\leqslant}[\![\tau_1]\!](v_1, v_1') * \mathbf{V}_{\leqslant}[\![\tau_2]\!](v_2, v_2') \\
\mathbf{V}_{\leqslant}[\![\tau_1 \to \tau_2]\!](v, v') &\triangleq \Box(\forall w, w'.\ \mathbf{V}_{\leqslant}[\![\tau_1]\!](w, w') \mathrel{-\!\!*} \mathcal{L}_{\leqslant}(\mathbf{V}_{\leqslant}[\![\tau_2]\!])(v\ w, v'\ w')) \qquad (\mathcal{L}_{\leqslant} \textit{ to be defined later}) \\
\mathbf{V}_{\leqslant}[\![\mu X.\, \tau]\!](v, v') &\triangleq \Box(\exists w, w'.\ v = \mathtt{fold}\ w * v' = \mathtt{fold}\ w' * \triangleright\, \Psi[\![\tau[X \mapsto \mu X.\, \tau]]\!](w, w')) \\
\mathbf{V}_{\leqslant}[\![?]\!](v, v') &\triangleq \Box(\exists w, w'.\ (v = \mathtt{fold}\ (\mathtt{inj}_B\ w) * v' = w' : B \Rightarrow ? * \triangleright\, \Psi[\![B]\!](w, w') \\
&\qquad \vee (v = \mathtt{fold}\ (\mathtt{inj}_+\ w) * v' = w' : ? + ? \Rightarrow ? * \triangleright\, \Psi[\![? + ?]\!](w, w')) \\
&\qquad \vee (v = \mathtt{fold}\ (\mathtt{inj}_\times\ w) * v' = w' : ? \times ? \Rightarrow ? * \triangleright\, \Psi[\![? \times ?]\!](w, w')) \\
&\qquad \vee (v = \mathtt{fold}\ (\mathtt{inj}_\to\ w) * v' = w' : ? \to ? \Rightarrow ? * \triangleright\, \Psi[\![? \to ?]\!](w, w')) \\
&\qquad \vee (v = \mathtt{fold}\ (\mathtt{inj}_\mu\ w) * v' = (\mathtt{fold}\ w') : \mu X.\, ? \Rightarrow ? * \triangleright\, \Psi[\![?]\!](w, w')))
\end{aligned}$$

Fig. 11. The logical relations on values

structure of the type, i.e., τ. However, we cannot do this for our type system due to the presence of recursive types, $\mu X.\tau$, and the unknown type, $?$. Instead, we define our relation as a guarded recursive predicate, $\mathcal{V}_{\leqslant} \triangleq \mu\Psi.\ \mathbf{V}_{\leqslant}(\Psi)$. The higher-order predicate $\mathbf{V}_{\leqslant}$ of type $(\mathit{Type} \to \mathit{Val} \to \mathit{Val} \to \mathit{iProp}) \to \mathit{Type} \to \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$ is defined by induction on types. For cases where the type does not become structurally smaller, i.e., in recursive types and the unknown type, we use guarded recursion; note that all such occurrences are guarded by a later modality. Readers familiar with the usual encoding of logical relations in program logics featuring abstract step-indexing may note that our approach differs slightly from the usual practice, e.g., by Timany et al. [2017b], where only the value relation for the recursive type is defined using guarded recursive predicates.

Notice also that we use the persistently modality three times to ensure that the relations are persistent in order for them to be duplicable. This is necessary as values can be used multiple times, e.g., when passed as a function argument.

The goal of this relation is twofold, as can be seen from lemma 4.2. It must relate static values (on the left-hand side) to their embeddings (on the right-hand side), as well as gradual values (on the right-hand side) to their appropriate static values in backtranslation. The latter is due to having the backtranslation of the context on the left-hand side. With this in mind, the value relation at the base type, which intuitively requires that the two values are related if they are "the same", can also be formally expressed as follows ($\dashv\vdash$ denotes the logical equivalence relation on Iris propositions):

$$\mathbf{V}_{\preccurlyeq}[\![B]\!](v, v') \dashv\vdash \exists b : B.\, v = \langle\!\langle b \rangle\!\rangle * v' = b$$

Values at a sum type are related if they are constructed using the same injection and the injected values are related at the appropriate type. Value at a product type, $\tau_1 \times \tau_2$, are related if they are both pairs in which the first components are related at τ_1 and the second components at τ_2.

Intuitively, two functions, say v and v', are related at $\tau_1 \to \tau_2$, if for any arbitrary pair of values w and w' related at τ_1, we have that the two expressions $v\ w$ and $v'\ w'$ are related at τ_2. We define the relation on closed expressions by lifting the relation on closed values. Given any relation on values, the higher-order predicate $\mathcal{L}_{\preccurlyeq}$ of type $(\mathit{Val} \to \mathit{Val} \to \mathit{iProp}) \to \mathit{Expr} \to \mathit{Expr} \to \mathit{iProp}$ lifts it to a relation on expressions. We define the predicate $\mathcal{L}_{\preccurlyeq}$ later on.

Values at a recursive type, $\mu X.\tau$, are related if they are both folded and the original (unfolded) values are, one step later, related at type $\tau[X \mapsto \mu X.\tau]$. Two values, say v and v', are related at the unknown type if for some ground type G (excluding $\mu X.?$ here), we have some w and w' related (one step later) at G for which v is the result of applying inject_G to w and v' is equal to $w' : G \Rightarrow ?$. They are also related if, for some w and w' related (one step later) at $?$, we have that v is the result of applying $\mathit{inject}_{\mu X.?}$ to $\texttt{fold}\ w$ and v' is equal to $\texttt{fold}\ w' : \mu X.? \Rightarrow ?$.

Lifting Value Relations to Expression Relations. Intuitively, what we expect of the expression relation is that if two closed expressions e and e' are related (in the $\preccurlyeq$-relation) at a type τ, then it must follow that $e \to^* v$ implies $e' \to^* v'$ for some v' such that $\mathbf{V}_{\preccurlyeq}[\![\tau]\!](v, v')$. We define this expression relation following the same idea that is usually employed for defining binary logical relations models in Iris, e.g., in Frumin et al. [2018]; Timany and Birkedal [2019]; Timany et al. [2017a,b]. That is, we use weakest preconditions for the left-hand side, and a custom-defined ghost resource and an appropriate invariant to keep track of the execution on right-hand side. Intuitively, in this approach, a binary logic is emulated as a unary logic for the left-hand side term, while the right-hand side term is treated as a specification for how the left term should behave.

Given a relation on values, Ψ, we define the lifting of that relation to closed expressions as follows.

$$\mathcal{L}_{\preccurlyeq}(\Psi)(e, e') \triangleq \forall e_i, K.\, \mathit{Initially}(e_i) * \mathit{Currently}_2(K[e']) \twoheadrightarrow \mathrm{wp}\ e \left\{v.\, \exists v'.\, \mathit{Currently}_2(K[v']) * \Psi(v, v')\right\}$$

Here, we have two ghost resources, $\mathit{Currently}_1$ and $\mathit{Currently}_2$, which are defined such that their expressions are required to always be the same:

$$\mathit{Currently}_1(e_1) * \mathit{Currently}_2(e_2) \vdash e_1 = e_2$$

Consequently, we can only update the expression if we update *both* propositions.

$$\mathit{Currently}_1(e_1) * \mathit{Currently}_2(e_1) \vdash \Rrightarrow \mathit{Currently}_1(e_2) * \mathit{Currently}_2(e_2) \qquad \text{(Currently update)}$$

We can allocate resources to establish $\mathit{Currently}_1(e')$ and $\mathit{Currently}_2(e')$:

$$\vdash \Rrightarrow \mathit{Currently}_1(e') * \mathit{Currently}_2(e') \qquad \text{(Currently allocation)}$$

The proposition $\mathit{Initially}(e_i)$ is an invariant, defined as follows.

$$\mathit{Initially}(e_i) \triangleq \boxed{\exists z. \mathit{Currently}_1(z) * e_i \to^* z}$$

The $Currently_1$ resource is therefore always owned by this invariant. Keep in mind though that this invariant is persistent and needs to be preserved during the entire execution, even when it is not returned in the postcondition. Moreover, maintaining this invariant poses another restriction on how we can update $Currently_1$ and $Currently_2$: As seen in eq. (Currently update), we can only update these ghost-resources by accessing the invariant (in order to obtain $Currently_1$). Note however that because of the invariant $Initially(e_i)$, we can only update $Currently_1$ and $Currently_2$ to an expression e' *if* e' *is such that* $e_i \to^* e'$.

Importantly, we also want to reason modularly about the relatedness of two *sub*-expressions. This idea is captured in the following rule:

$$\frac{\mathcal{L}_{\preccurlyeq}(\Phi)(e, e') \qquad \forall v, v'.\ \Phi(v, v') \implies \mathcal{L}_{\preccurlyeq}(\Psi)(K[v], K'[v'])}{\mathcal{L}_{\preccurlyeq}(\Psi)(K[e], K'[e'])}$$

This rule follows straightforwardly from the definition of the expression relation above together with the bind rule for Iris weakest preconditions:

$$\frac{\mathsf{wp}\ e\ \{\Phi\} \qquad \forall v.\ \Phi(v) \implies \mathsf{wp}\ K[v]\ \{\Psi\}}{\mathsf{wp}\ K[e]\ \{\Psi\}}$$

Having defined $\mathcal{L}_{\preccurlyeq}$, we have completed the relation on values, and moreover the relation on closed expressions, $\mathcal{E}_{\preccurlyeq}[\![\tau]\!] : Expr \to Expr \to iProp$, follows easily.

$$\mathcal{E}_{\preccurlyeq}[\![\tau]\!] = \mathcal{L}_{\preccurlyeq}(\mathcal{V}_{\preccurlyeq}[\![\tau]\!])$$

The Relations on Open Expressions. Following the standard practice, we now extend our logical relations (to open terms) based on our relations on closed expressions and values. Open terms are related if they are related as closed terms under any possible substitution of related values at the appropriate types. To formalize this, we first extend our value relations for gradual types to relations on *vectors* of values for a gradual *typing contexts*, $\mathcal{G}_{\preccurlyeq}$:

$$\begin{aligned} \mathcal{G}_{\preccurlyeq}[\![\cdot]\!](\varepsilon, \varepsilon) &\triangleq \mathsf{True} \\ \mathcal{G}_{\preccurlyeq}[\![x : \tau, \Gamma]\!]((v; \vec{w}), (v'; \vec{w}')) &\triangleq \mathcal{V}_{\preccurlyeq}[\![\tau]\!](v, v') * \mathcal{G}_{\preccurlyeq}[\![\Gamma]\!](\vec{w}, \vec{w}') \end{aligned} \tag{3}$$

Here, ε is the empty vector. We now define our logical relations:

$$\Gamma \vDash e \preccurlyeq e' : \tau \triangleq \forall \vec{v}, \vec{v}'. \mathcal{G}_{\preccurlyeq}[\![\Gamma]\!](\vec{v}, \vec{v}') \twoheadrightarrow \mathcal{E}_{\preccurlyeq}[\![\tau]\!](e[\vec{x} \mapsto \vec{v}], e'[\vec{x} \mapsto \vec{v}']) \tag{4}$$

4.3.4 The Logical Relations Satisfy the Specifications. Here, we give proof sketches for why lemmas 4.2 and 4.3 hold; given the analogously-defined $\geqslant$-relations, the arguments for lemmas 4.4 and 4.5 are similar.

Proof Sketch for Lemma 4.3. Let e and e' be two closed expressions for which we can derive in Iris that they are related at type τ, i.e., the following holds:

$$\vdash \cdot \vDash e \preccurlyeq e' : \tau$$

We show that $e \Downarrow$ implies $e' \Downarrow$. By definition $\cdot \vDash e \preccurlyeq e' : \tau$ is equivalent to $\mathcal{E}_{\preccurlyeq}[\![\tau]\!](e, e')$, and hence to the following:

$$\forall e_i, K.\ Initially(e_i) * Currently_2(K[e']) \twoheadrightarrow \mathsf{wp}\ e\ \{v.\ \exists v'.Currently_2(K[v']) * \mathcal{V}_{\preccurlyeq}[\![\tau]\!](v, v')\}$$

We use eq. (Currently allocation) to create the pair of propositions, $Currently_1(e')$ and $Currently_2(e')$, and use the former together with eq. (invariant allocation) to establish $Initially(e')$; note that

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vDash x \preccurlyeq x : \tau} \qquad \frac{\Gamma \vDash e_1 \preccurlyeq e_1' : \tau_1 \qquad \Gamma \vDash e_2 \preccurlyeq e_2' : \tau_2}{\Gamma \vDash (e_1, e_2) \preccurlyeq (e_1', e_2') : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vDash e \preccurlyeq e' : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Gamma \vDash \pi_i\ e \preccurlyeq \pi_i\ e' : \tau_i}$$

$$\frac{\Gamma \vdash b : B}{\Gamma \vDash b \preccurlyeq b : B} \qquad \frac{\Gamma \vDash e \preccurlyeq e' : \tau_i \qquad i \in \{1,2\}}{\Gamma \vDash \mathtt{inj}_i\ e \preccurlyeq \mathtt{inj}_i\ e' : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vDash e \preccurlyeq e' : \tau_1 + \tau_2 \qquad \Gamma \vDash e_1 \preccurlyeq e_1' : \tau_1 \to \tau \qquad \Gamma \vDash e_2 \preccurlyeq e_2' : \tau_2 \to \tau}{\Gamma \vDash \mathtt{case}\ e\ \mathtt{of}\ (e_1 \mid e_2) \preccurlyeq \mathtt{case}\ e'\ \mathtt{of}\ (e_1' \mid e_2') : \tau}$$

$$\frac{\Gamma \vDash e \preccurlyeq e' : \tau_1 \to \tau_2 \qquad \Gamma \vDash e_1 \preccurlyeq e_1' : \tau_1}{\Gamma \vDash e\ e_1 \preccurlyeq e'\ e_1' : \tau_2} \qquad \frac{x : \tau_1, \Gamma \vDash e \preccurlyeq e' : \tau_2}{\Gamma \vDash \lambda x : \langle\!\langle \tau_2 \rangle\!\rangle.\ e \preccurlyeq \lambda x : \tau_2.\ e' : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vDash e \preccurlyeq e' : \tau[X \mapsto \mu X.\tau]}{\Gamma \vDash \mathtt{fold}\ e \preccurlyeq \mathtt{fold}\ e' : \mu X.\tau} \qquad \frac{\Gamma \vDash e \preccurlyeq e' : \mu X.\tau}{\Gamma \vDash \mathtt{unfold}\ e \preccurlyeq \mathtt{unfold}\ e' : \tau[X \mapsto \mu X.\tau]}$$

Fig. 12. Unexciting compatibility lemmas

$e' \to^* e'$ holds trivially.[13] Now by picking e_i to be e' and K to be the empty evaluation context, $[\cdot]$, we obtain the following:

$$\mathsf{wp}\ e\ \{v.\ \exists v'.\mathit{Currently}_2(v') * \mathcal{V}_{\preccurlyeq}[\![\tau]\!](v, v')\}$$

Using eq. (WP and update) and the invariant $\mathit{Initially}(e')$ we can conclude:

$$\mathsf{wp}\ e\ \{v.\ \exists v'.\ e' \to^* v'\}$$

At this point, we can apply the adequacy theorem of Iris's weakest preconditions. It states that for any meta-level (i.e. outside Iris) postcondition[14] φ, if $\mathsf{wp}\ e\ \{\varphi\}$ holds, then $\varphi(v)$ holds at the meta level for any value v that e reduces to. Here $\varphi(v)$ is taken to be $\exists v'.\ e' \to^* v'$.

Proof Sketch for Lemma 4.2. Lemma 4.2 follows from the compatibility lemmas in fig. 12 and the much more interesting one given below, which plays a central role in our full abstraction argument:

Lemma 4.6 (Compatibility lemma for casts).

$$\frac{\Gamma \vDash e \preccurlyeq e' : \tau_i \qquad \tau_i \sim \tau_f}{\Gamma \vDash (\mathscr{F}^{\emptyset}_{\tau_i \rightsquigarrow \tau_f} e) \preccurlyeq (e' : \tau_i \Rightarrow \tau_f) : \tau_f}$$

Remember that the backtranslation of casts mirrors the structure of the cast calculus quite faithfully as we defined it by induction on the relation in fig. 9. Moreover, we prove this lemma by induction on said relation. To do so though, we generalize it to $\mathscr{F}^{F}_{\tau_i \rightsquigarrow \tau_f}$ for an arbitrary derivation $F \vdash_s \tau_i \rightsquigarrow \tau_f$. Of course, $\mathscr{F}^{F}_{\tau_i \rightsquigarrow \tau_f}$ contains some free variables ($\mathrm{dom}(F)$), so we will want to substitute these with (arbitrary) "appropriate" values. We first formalize this notion of "appropriate" values: a partial function, $\alpha : \mathit{Var} \rightharpoonup \mathit{Val}$, serves as an appropriate substituting function in $\mathscr{F}^{F}_{\tau_i \rightsquigarrow \tau_f}$ if $\mathcal{H}[\![F]\!]_{\preccurlyeq}(\alpha)$ holds:

$$\mathcal{H}[\![F]\!]_{\preccurlyeq}(\alpha) \triangleq \mathop{\Large *}_{f \mapsto (\tau_1 \rightsquigarrow \tau_2) \in F} \square(\forall v, v'.\mathcal{V}_{\preccurlyeq}[\![\tau_1]\!](v, v') \mathrel{-\!\!*} \mathcal{E}_{\preccurlyeq}[\![\tau_2]\!](\alpha(f)\ v, v' : \tau_1 \Rightarrow \tau_2)) \qquad (5)$$

[13] Note that here we are using eq. (WP and update) implicitly.
[14] For our Coq formalization, this means a Coq predicate over values.

That is, for each $F(f) = (\tau_1 \rightsquigarrow \tau_2)$ we must have $\alpha(f) : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_2 \rangle\!\rangle$ defined such that it serves as a successful emulation of a cast from τ_1 to τ_2. Given this definition, we obtain the following generalization of lemma 4.6, where $\mathscr{F}^F_{\tau_i \rightsquigarrow \tau_f}[F \mapsto \alpha]$ denotes the substitution of the free variables in $\mathscr{F}^F_{\tau_i \rightsquigarrow \tau_f}$ with the values defined by α.

Lemma 4.7 (Compatibility lemma for casts – Generalized).

$$\frac{\Gamma \vDash e \preccurlyeq e' : \tau_i \qquad F \vdash_s \tau_i \rightsquigarrow \tau_f}{\forall \alpha. \mathcal{H}[\![F]\!]_{\preccurlyeq}(\alpha) \twoheadrightarrow \Gamma \vDash \mathscr{F}^F_{\tau_i \rightsquigarrow \tau_f}[F \mapsto \alpha]\ e \preccurlyeq e' : \tau_i \Rightarrow \tau_f : \tau_f}$$

This lemma states that the backtranslation $\mathscr{F}^F_{\tau_i \rightsquigarrow \tau_f}$ must relate to a cast $\tau_i \Rightarrow \tau_f$ after arbitrary substitution with a function that satisfies $\mathcal{H}[\![F]\!]_{\preccurlyeq}$. This definition resembles the extension of our relation from closed to open expressions, i.e., eq. (4), in which eq. (3) is analogous to eq. (5). The proof of this generalized lemma is by induction on the alternative consistency relation (fig. 9). For more details see our Coq formalization.

As an aside, we briefly remark that the compatibility lemmas in fig. 12 (not including lemma 4.7), together with the adequacy of logical relations, imply the superset criterion and hence the proof of reflection sketched in fig. 3.

5 DISCUSSION

5.1 Contextual Equivalence in GTLC_μ

Equivalence by Equi-termination. In this paper, we have stated the full abstraction of the embedding from STLC_μ to GTLC_μ with respect to standard notions of contextual equivalence by equi-termination in both STLC_μ and GTLC_μ.

As mentioned in §2.2, such an equivalence in GTLC_μ does not distinguish between divergence and dynamic type errors. More concretely, for contextually equivalent e and e' ($\Gamma \vdash e \simeq_{ctx} e' : \tau$), there can still be a gradual context, say $C : (\Gamma; \tau) \Rightarrow (\cdot; \mathbb{1})$, for which $C[e]$ diverges and $C[e']$ gives a runtime error. For instance, the two terms, $\lceil\lceil \lambda f : \mathbb{B} \to \mathbb{B}.\pi_2(f\ \mathtt{true}, \Omega)\rceil\rceil$ and $\lceil\lceil \lambda f : \mathbb{B} \to \mathbb{B}.\Omega\rceil\rceil$, are still equivalent even though they could be instantiated in a context $[\cdot]\ f_{\mathsf{CastError}}$ where $f_{\mathsf{CastError}}$ is defined such that upon application it will always return a cast error.

Equating divergence with runtime errors allows us to get away with a subtle difference in expressiveness between gradual and static contexts: the former have the ability to cause dynamic errors while the latter do not. This is appropriate however, if we consider cast errors as a debugging mechanism of the language, which is not observable in production settings.

Equivalence that Distinguishes Cast Errors and Divergence. We might be interested in this slight difference in expressiveness; formally, we have the following definition.

Definition 5.1 (Contextual equivalence distinguishing cast errors and divergence). Two terms e_1, e_2 in GTLC_μ are equivalent iff $\Gamma \vdash e_1, e_2 : \tau$ and for all gradual contexts $C : (\Gamma; \tau) \Rightarrow (\cdot; \mathbb{1})$, we have both ($C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$) and ($C[e_1] \Uparrow$ iff $C[e_2] \Uparrow$).

With respect to this definition, full abstraction of the embedding fails as the two terms from the previous paragraph are not equivalent anymore.

As an interesting future work one could try to address this by designing the gradual language to prevent static terms from being exposed to `CastError`s. Alternatively, one might introduce an equivalent form of dynamic errors in the static language. This might seem like a way to "define the problem away", but this is not the case: it simply makes it clear to source language developers that they should take the possibility of dynamic errors into account when reasoning about their code. We have not proven this, but we believe our existing backtranslation can easily be adjusted

to prove full abstraction with respect to Definition 5.1 by backtranslating dynamic errors to the equivalent error in the static language.

Equivalence Accounting for Blame. Going further, one can wonder how blame would effect contextual equivalence and full abstraction of the embedding. However, defining contextual equivalence that is aware of blame seems quite challenging. When translating a pair of similarly-behaving terms to the cast calculus, we would somehow need to smartly assign blame-labels to both these terms so as to recover equivalence for non-trivial cases.

Whether or not a meaningful definition of blame-aware equivalence exists, how it would look like, and how it would affect full abstraction are all challenging questions that we leave as future work.

5.2 Partiality in the Source Language

It is also worth observing that our proof crucially relies on the existence of recursive types in the source language. The definition of the universal type $\mathscr{U}$, the backtranslation between recursive types, and the backtranslation of cast errors to diverging Ω terms rely on this. But why have we not worked with a total simply typed lambda calculus STLC, and does GTLC also satisfy the FAE criterion with respect to this?

To answer the latter question first, consider that contextual equivalence is strictly stronger in the total STLC than in STLC_μ. For example, the term $\lambda f : \mathbb{B} \to \mathbb{B}.\, \pi_2\ (f\ \mathit{true}, 43)$ is contextually equivalent to $\lambda f : \mathbb{B} \to \mathbb{B}.\, 43$, essentially because invoking f and ignoring its result is equivalent to not invoking it if we are sure that f will terminate. In GTLC, we can apply both functions to a function that diverges or fails upon application, breaking the two terms' contextual equivalence.[15] Hence, the GTLC (as a gradualization of the total STLC) does not satisfy the FAE criterion.

Is this a counterargument to the FAE criterion? No! It merely highlights an important property of the GTLC, namely that it enforces a specific semantics of static STLC types and that this semantics is not the one from STLC but the one from a version of STLC *which has some source of partiality* (like the recursive types in STLC_μ). More concretely, the GTLC does not make any effort to enforce termination, which is part of the semantics of function types in STLC. Because of this, GTLC can only be considered as a gradualization of a partial variant of STLC (like STLC_μ), not of the total STLC.

In other words, the failure of GTLC with respect to STLC should not be considered as a weakness of the FAE criterion or a clever choice of example by us. Instead, this failure of the criterion accurately demonstrates that GTLC does not enforce the semantics of types from STLC (but does so for the semantics of types in e.g. STLC_μ). This immediately suggests another interesting question that we will not go into further: can we gradualize the STLC? In other words, is it possible to construct a gradual language that enforces the interpretation of types from STLC, i.e., the totality of functions?

5.3 Recursive Types vs. Term Recursion?

So it is clear that GTLC should be considered the gradualization of a partial STLC. But still, that does not make it clear why we have used STLC_μ. "Why bother with recursive *types*," the reader may wonder, "if adding recursion to STLC with just a *fix* operator will do the trick?" In other words, we may ask ourselves whether or not GTLC_{fix}, the natural gradualization of STLC_{fix} (a version of the STLC without recursive types but with a builtin term-level fixpoint operator *fix*) satisfies the FAE criterion. If it does, it is immediately clear that it will need a separate argument. As we have

[15] Consider, for instance, the gradual term $e = \lambda x : \mathbb{B}.\, (\lambda y : \mathbb{B}.\, y)\ ((\lambda y : ?.\, y)\ ((\lambda y : \mathbb{N}.\, y)\ ((\lambda y : ?.\, y)\ x)))$. The term $\lceil\lceil (\lambda f : \mathbb{B} \to \mathbb{B}.\, 43) \rceil\rceil\ e$ evaluates to 43, while $\lceil\lceil (\lambda f : \mathbb{B} \to \mathbb{B}.\, \pi_2\ (f\ \text{true}, 43)) \rceil\rceil\ e$ fails.

pointed out above, our proof of FAE does crucially rely on recursive types for constructing the universal type $\mathscr{U}$, so it clearly does not extend to STLC_{fix}.

Even so, we do believe that the embedding from STLC_{fix} into GTLC_{fix} is fully abstract. We are aware of ongoing work to prove a fully abstract embedding from STLC_{fix} into $\text{STLC}_{\mu,\text{fix}}$ using the approximate backtranslation technique by Devriese et al. [2016, 2017b]. Because full abstraction is preserved upon composition, this result could then be composed with ours to prove that the FAE criterion holds between $\text{GTLC}_{\mu,\text{fix}}$ and STLC_{fix}. Moreover, as the operational semantics of GTLC_{fix} is trivially contained in $\text{GTLC}_{\mu,\text{fix}}$, every context in GTLC_{fix} can be seen as a context in $\text{GTLC}_{\mu,\text{fix}}$; we can conclude therefore that the embedding from STLC_{fix} into GTLC_{fix} preserves contextual equivalences.

In other words, we believe we should be able to recover the result about STLC_{fix} by composing the current result with a separately interesting one from other work, whose proof requires a different proof technique (approximate backtranslation [Devriese et al. 2016]). By making the current result only cover STLC_{μ}, we have been able to separate concerns and obtain separately interesting but composable results.

5.4 Robust Relational Hyperproperty Preservation for GTLC_{μ}

It is interesting to note that in our FAE-proof for GTLC_{μ}, our backtranslation on contexts is actually a bit overspecified; for any gradual context, we define a backtranslation that correctly emulates it, *independent of the term that is plugged into it.* Therefore, we have proven something stronger than is absolutely necessary. In principle, it would be enough to construct a backtranslation of a context such that it only works for two given terms under consideration in fig. 4. This is what happens, for instance, in backtranslations that are based on trace semantics, e.g. Patrignani et al. [2015].

In fact, the existence of an arbitrary backtranslation is actually documented as a proposed stronger formal criterion in the field of secure compilation. It is the strongest robust-property-preservation property from a broad range introduced by Abate et al. [2019]: *Robust Relational Hyperproperty Preservation.* Abate et al. [2019] give an alternative "property-free" characterization which (in this setting) exactly corresponds to theorem 4.1.

Interestingly, this suggests that GTLC does not just preserve equivalences from STLC_{μ}, but also unary (safety) and k-ary properties. Unfortunately, the framework of Abate et al. [2019] can only be applied to languages with traces modeling the interaction of a program with the outside world. In this context, such traces would either be artificial or trivial so we stick to the current binary setting. However, we plan to extend the FAE criterion to include preservation of unary and k-ary properties in follow-up work.

5.5 Other Gradual Languages

We believe our proposal raises interesting questions about existing gradual languages. We have already talked a bit about gradual parametricity and gradual security types, but what about other gradualizations? Many other gradual languages enforce static type systems with properties that go beyond type safety, for example ownership types [Sergey and Clarke 2012], effect systems [Bañados Schwerter et al. 2014], session types [Igarashi et al. 2019] etc.

It would be interesting to investigate in more detail whether these gradualizations enforce source language type-based reasoning and if so, to what extent. Do they satisfy the FAE criterion with respect to their source language? If they do not, are there possibilities to enforce more of the source types semantics, and if so, how? In some cases, the dynamic language also offers useful abstractions, and it might also be interesting to consider preservation of the dynamic language's properties, i.e. fully abstract embedding for the dynamic language.

5.6 The Pursuit of FAE

When gradualizing a static language, we should strive to preserve its reasoning principles as much as possible. To this end, adherence to the FAE criterion provides a fixed, unambiguous, and general goal. As discussed, FAE implies that two terms in the static language are equivalent in the static language if, and only if they are equivalent in the gradual language. This means that all static-based reasoning principles involving contextual equivalences would remain sound when applied to the fully static parts in a dynamic codebase, regardless of what the surrounding untyped code may do. In other words, not only would static refactorings remain valid when performed in a dynamic codebase, the static abstractions (expressed as contextual equivalences) would remain valid in the presence of dynamic code (e.g., the untyped code cannot peer through abstract data types).

Another way to look at FAE is to see what the implications are of FAE failing. In such a situation, we have none of the aforementioned advantages, e.g. refactoring the code to use an alternative implementation of an abstract data type is not necessarily safe. Moreover, if FAE fails for a particular gradualization, then some static equivalences must necessarily be broken in the presence of untyped code! Therefore, failure of FAE should be carefully investigated. Such an investigation can have one of the following two outcomes. It might actually suggest a better design that satisfies FAE. For gradualizing the total STLC for instance, we have pointed out such a possibility in §5.2. Alternatively, the investigation might also make clear however that enforcing FAE is not really feasible at a reasonable cost. In this case, pointing this out makes it (formally) clear that some reasoning is lost in the gradualization process; an important message for programmers. One can furthermore try to contextualize this failure, as we will illustrate below for GTLC as a gradualization of the total STLC.[16]

Investigating Failure of FAE for GTLC *w.r.t. the Total* STLC. We can investigate which weaker properties (regarding the preservation of reasoning principles) are still satisfied. For instance, we might restrict ourselves to a meaningful subset of equivalences and prove that at the least, they are still preserved upon embedding. In the case of GTLC as a gradualization of total STLC, we might, for instance, consider only those equivalences that also hold in STLC_μ and prove that at least they are preserved upon embedding.

Alternatively, we might shift towards a different static language, one whose equivalences are preserved more naturally upon embedding. We have shifted from GTLC as a gradualization of total STLC to GTLC_μ as a gradualization of STLC_μ. As remarked upon in §5.3, other partial variants like STLC_{fix} are also possible. We might also theorize that adding a non-recoverable error term to the total STLC (thereby making it partial) will also give us a gradualization (à la GTLC) that satisfies FAE.

Lastly, we might wonder what can or cannot happen to the static equivalences that are broken upon embedding. More concretely, we may wonder whether given any two equivalent static terms, we could still prove that their embeddings satisfy some other, much weaker (but not meaningless) relation. For instance, given two equivalent static terms in the STLC, we can hypothesize that the pair of embeddings in the GTLC still satisfies the relation $\mathcal{R}$ defined below.

$$\mathcal{R}_{(\Gamma,\tau)}(e,e') \;\; \textit{iff} \;\; \nexists C : (\Gamma;\tau) \Rightarrow (\cdot\,;\mathbb{B}).\ (C[e] \rightarrow^* \texttt{true}) \wedge (C[e'] \rightarrow^* \texttt{false})$$

To be clear, we do not think of this constraint as one whose adherence constitutes a desirable language; rather, we think of it as a bare minimum.

[16] Remember that in §5.2, we showed that the GTLC with respect to the total STLC does not satisfy FAE.

6 RELATED WORK

The FAE criterion has already been put forward by Devriese et al. [2017a], although the authors only demonstrate that it does not hold for gradual parametric calculi, as discussed in Section 3.2.

It is not feasible to provide a comprehensive overview of gradual typing here, so we will focus on research about correctness properties for gradual languages. However, it is worth discussing the two lineages of gradual typing that Greenberg [2019] identifies: the dynamic-first approach (which starts from an untyped language and adds a type system to it) versus the static-first approach (which starts from a static type system and makes it safely interoperable with untyped code). It is fair to say that our approach aligns more naturally with the latter approach, as we are motivated by preservation of reasoning principles in a pre-existing static type system. Nevertheless, this does not mean that FAE is incompatible with the dynamic-first approach. For example, dynamic idioms like occurrence typing [Tobin-Hochstadt and Felleisen 2008] are not necessarily contradictory with FAE; as long as the language under consideration offers useful abstractions, a gradual language can be expected to enforce them.

Correctness Properties for Gradual Type Systems. We have already discussed the refined criteria of Siek et al. [2015] for gradual typing in the introduction and we will not repeat this here. The proposal of Garcia and Tanter [2020], stating that gradual languages should not just preserve type safety but type soundness, has also been discussed there.

New et al. [2019] have proposed *graduality*, a semantic interpretation of the gradual guarantee [Siek et al. 2015]. While adherence to graduality could enforce new constraints on the gradual language, it is clear that in general, it does not imply FAE. Indeed, in the gradual language PolyG^v [New et al. 2019], the type $\lceil\lceil \exists Y.\, \forall X.\, (X \to Y) \times (Y \to X) \rceil\rceil$ is non-degenerately inhabited (their parametricity uses a type-world logical relation that is weaker than the traditional one originally used by Reynolds [1983]), hence the argument in §3.2 remains applicable.

Greenman et al. [2019] have proposed a Complete Monitoring property for gradual type systems. They define when a value is owned by a component and then require that the gradual language checks contracts whenever a value crosses a component boundary. The property is intended to aid in debugging dynamic type errors by requiring correct blame assignment and to highlight the benefits of more precise but more costly strategies for tracking blame. As such, the property captures a different requirement than ours and as such, should be considered largely orthogonal.

Fully Abstract Compilation and Backtranslations. The criterion of fully abstract compilation is a well-known concept in the field of secure compilation. In that setting, fully abstract compilation captures the requirement that a secure compiler should enforce source language abstractions. The concept itself and the many examples in the literature have been surveyed by Patrignani et al. [2019b]. Most closely related to our work are the results by Devriese et al. [2016] and New et al. [2016], who also use a universal type in the construction of a backtranslation.

In addition to compiler security, fully abstract compilers (translations) have also been used to study the expressiveness of program calculi [Parrow 2008]. In a certain sense, our criterion requires that the gradual language is equally expressive than the static language.

Logical Relations in Iris. The program logic Iris [Jung et al. 2018] which we have used to prove the correctness of our backtranslation, is a versatile program logic used for different applications. Several projects have used Iris for modeling logical relations which inspired the construction of ours [Frumin et al. 2018; Timany and Birkedal 2019; Timany et al. 2017a,b].

7 CONCLUSION

In the introduction, we argued that the refined criteria of Siek et al. [2015] are not a sufficient condition for a good gradual language. This has been argued before by Garcia and Tanter [2020] and it is by now not a very controversial statement in the gradual typing community. Rather than requiring the preservation of a source-language-specific and designer-selected meaning of types, this paper explores a generic and ambitious alternative, previously suggested by Devriese et al. [2017a]: the fully abstract embedding (FAE) of the static language into the gradual one. This paper offers a first exploration; we have shown that the criterion is both useful (identifies problems in unsatisfactory gradualizations) and realistic (attainable and provable for real gradualizations). We believe that future gradual language designers should think about the criterion to evaluate the consequences of their design choices.

ACKNOWLEDGMENTS

This work was funded in part by Internal Funds KU Leuven grant C14/18/064. Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0054. This work was partly supported by the Fund for Scientific Research - Flanders (FWO).

REFERENCES

Martín Abadi. 1998. Protection in Programming-Language Translations: Mobile Object Systems. In *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science)*. Springer Berlin Heidelberg, 291–291. https://doi.org/10.1007/3-540-49255-0_70

Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming*. Springer-Verlag. https://doi.org/10.1007/BFb0055109

Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *Computer Security Foundations Symposium*. https://doi.org/10.1109/CSF.2019.00025

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011a. Blame for All. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 201–214. https://doi.org/10.1145/1926385.1926409

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and without Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 39:1–39:28. https://doi.org/10.1145/3110283

Amal Ahmed, Lindsey Kuper, and Jacob Matthews. 2011b. Parametric polymorphism through run-time sealing, or, Theorems for low, low prices! http://www.ccs.neu.edu/home/amal/papers/paramseal-tr.pdf

Andrew W Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001), 657–683. https://doi.org/10.1007/978-3-642-00590-9_1

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, 283–295. https://doi.org/10.1145/2628136.2628149

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 164–177. https://doi.org/10.1145/2837614.2837618

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017a. Parametricity versus the universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 38. https://doi.org/10.1145/3158126

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2020. Two Parametricities versus Three Universal Types. (2020). http://soft.vub.ac.be/~dodevrie/poly-seal-no-j-201910.pdf Submitted to the Journal of the ACM.

Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017b. Modular, Fully-Abstract Compilation by Approximate Back-Translation. *Logical Methods in Computer Science* 13, 4 lmcs:4011 (Oct. 2017). https://doi.org/10.23638/LMCS-13(4:2)2017 arXiv:1703.09988 [cs.PL]

D. Dreyer, A. Ahmed, and L. Birkedal. 2009. Logical Step-Indexed Logical Relations. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*. 71–80. https://doi.org/10.1109/LICS.2009.34

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 442–451. https://doi.org/10.1145/3209108.3209174

Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 429–442. https://doi.org/10.1145/2914770.2837670

Ronald Garcia and Éric Tanter. 2015. Deriving a Simple Gradual Security Language. *arXiv preprint arXiv:1511.01399* (2015).

Ronald Garcia and Éric Tanter. 2020. Gradual Typing as if Types Mattered. (2020). Workshop on Gradual Typing.

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *Summit on Advances in Programming Languages (SNAPL) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:20. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete Monitors for Gradual Types. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 122:1–122:29. https://doi.org/10.1145/3360548

Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. 2019. Gradual Session Types. *Journal of Functional Programming* 29 (2019). https://doi.org/10.1017/S0956796819000169

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices! LNCS, Vol. 4960. 16–31. https://doi.org/10.1007/978-3-540-78739-6_2

Max S New, William J Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 103–116. https://doi.org/10.1145/3022670.2951941

Max S. New, Dustin Jamner, and Amal Ahmed. 2019. Graduality and Parametricity: Together Again for the First Time. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 46:1–46:32. https://doi.org/10.1145/3371114

Joachim Parrow. 2008. Expressiveness of Process Algebras. *Elec. Not. Theo. Comp. Sci.* 209, 0 (2008), 173 – 186. https://doi.org/10.1016/j.entcs.2008.04.011

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, 2 (April 2015), 6:1–6:50. https://doi.org/10.1145/2699503

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019a. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (Feb. 2019), 125:1–125:36. https://doi.org/10.1145/3280984

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019b. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36. https://doi.org/10.1145/3280984

J. C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*. North Holland, 513–523.

Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

Jeremy Siek. 2019. GitHub - jsiek/gradual-typing-in-agda: Formalizations of Gradually Typed Languages in Agda. https://github.com/jsiek/gradual-typing-in-agda. (Accessed on 10/18/2019).

Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Amin Timany and Lars Birkedal. 2019. Mechanized Relational Verification of Concurrent Programs with Continuations. *Proc. ACM Program. Lang.* 3, ICFP, Article 105 (July 2019), 28 pages. https://doi.org/10.1145/3341709

Amin Timany, Robbert Krebbers, and Lars Birkedal. 2017a. Logical relations in Iris. In *CoqPL, Date: 2017/01/21-2017/01/21, Location: Paris*.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017b. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158152

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. *ACM SIGPLAN Notices* 43, 1 (Jan. 2008), 395–406. https://doi.org/10.1145/1328897.1328486

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4 (Dec. 2018), 16:1–16:55. https://doi.org/10.1145/3229061

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 17:1–17:30. https://doi.org/10.1145/3290330

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming*. Springer, 1–16.

A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093