Fully Abstract from Static to Gradual

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

What is a good gradual language? Siek et al. have previously proposed the refined criteria, which specify certain guarantees about semantic correspondence and preservation of well-typedness and type-safety in the presence of untyped code. However, because of their exclusive focus on syntactic properties, they are not the whole story. Rich semantic properties like parametricity or non-interference, which hold in the static language, should also continue to hold upon gradualisation.

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. We illustrate in a simple setting that the criterion is useful; it can weed out an erroneous gradualisation that satisfies the refined criteria. We illustrate that it is realistic, by giving a proof sketch for the natural gradualisation of STLC_μ.

ACM Reference Format:
Koen Jacobs, Amin Timany, and Dominique Devriese. 2020. Fully Abstract from Static to Gradual. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

What do we want from a gradually typed language? To formally answer this question, Siek et al. have put forward a set of refined criteria [14]. While these do a great job at formally describing some of the intuitive requirements for a gradual language, they are not the whole story.

Generally, a gradual language should preserve the guarantees offered by the static language, even when interacting with untyped code. In an unpublished draft, Tanter et al. have argued that the refined criteria successfully capture this intuition, but only for one type of guarantees: those concerning type safety [7].[1] In many settings, static type systems offer other types of guarantees. For example, the parametricity in System F and related languages is a crucial language property, but can be broken without compromising type safety. Indeed, in a paper on gradual parametricity, Toro et al. report that a naive evidence-based semantics produces a well-behaved gradual language satisfying all the refined criteria but not parametricity. They continue to design a gradual language satisfying parametricity but not the refined criteria. The situation is remarkably similar in the context of gradual security type systems, where Toro et al. again sacrifice one of the refined criteria to salvage the static language's non-interference guarantee [15].

As an aside, we note that both for gradual parametric type systems and for gradual security type systems, it has been suggested that type-based reasoning principles (parametricity resp. non-interference) are incompatible with one of the refined criteria. However, this incompatibility is not currently agreed upon: at least in the setting of gradual parametricity, we are aware of promising ongoing work by two separate groups to reconcile them.

In this work, we propose and investigate a new formal criterion for the gradualisation of a static language, previously suggested by Devriese et al. [5]. We propose that the embedding from the static to the gradual language should be fully abstract [1, 2]. That is, contextual equivalences in the

[1] In this text, we use the word type safety to capture syntactic type safety properties, typically embodied by progress and preservation theorems.

WGT20, January 25, 2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

static language should be preserved in the gradual language. Concretely, if two static terms always behave the same in the static language (i.e. no static context could ever distinguish the two), then they should always behave the same when viewed as fully static programs in the gradual language (i.e. no *gradual* context can ever distinguish the two).

Intuitively, this is a way of specifying that the gradual part of the language should not introduce more distinguishing power over static terms 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. As a concrete example, in any gradualisation of System F, we should –at the least– expect that a fully static implementation of $\forall\alpha.\alpha \to \alpha$ is always equivalent to $\Lambda\alpha.\lambda x : \alpha.x$, $\Lambda\alpha.\, \lambda x : \alpha.\, \omega$ or $\Lambda\alpha.\, \omega$. Likewise, in a gradualisation of a security-typed language, we should –at the least– still have that our secret value $6 : \mathbb{N}_H$ inside $\lambda f : \mathbb{N}_H \to \mathbb{N}_L.f\ (6 : \mathbb{N}_H)$ can never be leaked.

In this paper, it is not our ambition to revisit gradual typing in challenging settings. We also do not claim that our criterion is the definitive characterisation of a good gradual language. Instead, we aim to demonstrate that our criterion is (1) interesting and (2) realistic. We demonstrate the former in §2, by exhibiting a simple gradual security-typed language, which is clearly unsatisfactory but satisfies all the refined criteria. However, the problem is adequately detected by our criterion.

Next, in §3, we show that full abstraction is realistic, by exhibiting a proof of the property for a representative example; the GTLC_μ. Taking inspiration from similar proofs in the context of secure compilation [4, 9], we prove that there is a fully abstract embedding of STLC_μ into GTLC_μ.

We discuss a couple of remarks and future work in §4. We conclude in §5.

2 EXAMPLE IN FAVOUR OF FAE BESIDES REFINED CRITERIA

To explain why the refined criteria alone are not enough, this section presents a simple but representative example. We start with the $\text{STLC}_{\mu,\text{sec}}$, a simple variant of λ_{sec} [6], in which every type τ comes in only two flavours; τ_H and τ_L, corresponding respectively to sensitive data and non-sensitive data. A value 6 for example, can be typed both as a value of $\mathbb{N}_H$ as well as $\mathbb{N}_L$. The former denotes that 6 must be treated as sensitive data, the latter denotes insensitive data.

The typing rules are defined in such a way that sensitive data can never leak into insensitive data. That is, any computation that uses a sensitive value is also marked sensitive, *e.g.*, *if* b *then* e *else* e' where b is sensitive. Consequently, in this system a function of type $\mathbb{B}_H \to \mathbb{B}_L$ is not able to use its input argument, as doing so would leak sensitive information. In other words, such a function is necessarily a constant function. This property is generally formalised by the notion of "non-interference" [6].

Now imagine an incorrect gradualisation of this language, which does not properly respect security labels of types, for example, by allowing a cast $(v : \mathbb{B}_H \Rightarrow \star \Rightarrow \mathbb{B}_L) \to v$ to succeed. This issue alone will not break the refined criteria for gradual type systems. This is essentially because these criteria are exclusively concerned with syntactic criteria, such as well-typedness and type safety, but turn a blind eye to semantic properties such as non-interference, preservation of contextual refinements, *etc.*

Let us look at the gradual guarantee, for instance. The first part states that if a closed term, say e, is well-typed, removing type annotations will result in a less precise term, say e', that is still typed for a less precise type. Trivially, adding this erroneous cast does not invalidate this. Secondly, it states that if $e \Downarrow v$ then $e' \Downarrow v'$ with v less precise than v', or if $e \Uparrow$ then $e' \Uparrow$. In other words, stripping type-annotations from a program that does not give any run-time errors, should not change its behaviour. Again, we can add this erroneous cast rule without breaking that statement.

Similar intuition applies to the third part of the gradual guarantee, which states that adding *correct* type annotations to a gradual term does not change its behaviour.

However, embedding the $\text{STLC}_{\mu,\text{sec}}$ into such an erroneous gradual language will not be fully abstract. To understand this, remember that any function of type $\mathbb{B}_H \to \mathbb{B}_L$ must necessarily be constant. That is, the following contextual equivalence holds:

$$(\lambda f : (\mathbb{B}_H \to \mathbb{B}_L).\ f\ \texttt{true}) \approx (\lambda f : (\mathbb{B}_H \to \mathbb{B}_L).\ f\ \texttt{false})$$

However, the same equivalence will not hold in our erroneous gradual language. A context like the following can distinguish the two programs:

$$[]\ (\lambda x : \mathbb{B}_H.\ x : \mathbb{B}_H \Rightarrow \star \Rightarrow \mathbb{B}_L)$$

From a more high-level perspective, what this example shows is that the refined criteria exclusively focus on typability and syntactic type safety, consequently failing to consider more semantic aspects of type systems. Particularly, it fails to require preservation of language properties like parametricity or non-interference. Our FAE criterion does at least partially: it requires preservation of at least contextual equivalences that follow from those properties, such as the example discussed earlier. For other properties like the validity of refactorings and compiler optimizations, it does so completely.

3 REPRESENTATIVE EXAMPLE SATISFYING FAE

The previous example shows that our FAE criterion can detect problems in gradual languages that are missed by the refined criteria. But is the criterion realistic to expect from a gradual language. In this section, we argue that it is, by proving that the criterion holds for a representative gradual language: the gradually typed lambda calculus GTLC_μ, extended with (iso-)recursive types. This proof is the more technical part of this work, and we are working to prove the result mechanically in Coq, using the Iris framework [8].

First, in §3.1, we sketch GTLC_μ as the gradual extension of STLC_μ, the STLC with sum, product, and (iso-)recursive types. Next, we formally state the FAE-criterion in §3.2. In §3.3, we present a proof sketch for FAE in this setting. The proof sketch relies on the notion of a back-translation, which we lay-out in §3.4.

3.1 Definition GTLC_μ

We sketch GTLC_μ here. Its formal definition follows the pattern of GTLC defined as a ground-type cast-calculus [13, 14]. The presentation here is not entirely self-contained, and so we advise the unfamiliar reader to first get acquainted with the presentation in [14].

We have the following types.

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

Here, B denotes a base type, so we have, for instance $B ::= \mathbb{1} \mid \mathbb{N} \mid \mathbb{B}$.

We then define the following consistency relation on types.

$$A \vdash \star \sim \tau \qquad A \vdash \tau \sim \star \qquad A \vdash B \sim B \qquad \frac{A \vdash \tau_a \sim \tau_a' \qquad A \vdash \tau_r \sim \tau_r'}{A \vdash \tau_a \to \tau_r \sim \tau_a' \to \tau_r'}$$

$$\frac{A \vdash \tau_1 \sim \tau_1' \qquad A \vdash \tau_2 \sim \tau_2'}{A \vdash \tau_1 + \tau_2 \sim \tau_1' + \tau_2'} \qquad \frac{A \vdash \tau_1 \sim \tau_1' \qquad A \vdash \tau_2 \sim \tau_2'}{A \vdash \tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'} \qquad \frac{A, X \sim Y \vdash \tau \sim \tau'}{A \vdash \mu X.\tau \sim \mu Y.\tau'} \qquad \frac{X \sim Y \in A}{A \vdash X \sim Y}$$

It is mostly standard, except for the fact that we keep track of a list of assumptions in order to deal with the consistency between recursive types. So two closed types τ and τ' are consistent, written $\tau \sim \tau'$, if they are consistent in the empty list of assumptions, i.e. $[] \vdash \tau \sim \tau'$.

Following the scheme in [13, 14], we have the following ground types.

$$G ::= B \mid \star \to \star \mid \star + \star \mid \star \times \star \mid \mu X.\star$$

We have the usual factorisation of casts through their ground-types.

Below, we have the dynamics for the cast rule between sum types [13]. Casting a value between $\tau_1 + \tau_2$ and $\tau_1' + \tau_2'$, will boil down to either a cast from τ_1 to τ_1', or from τ_2 to τ_2'.

$$\begin{aligned} v : \tau_1 + \tau_2 \Rrightarrow \tau_1' + \tau_2' \to \mathtt{case}\ v\ \mathtt{of}\ \lambda x_1 : \tau_1.\ \mathtt{inl}\ (x_1 : \tau_1 \Rrightarrow \tau_1') \\ \mid \lambda x_2 : \tau_2.\ \mathtt{inr}\ (x_2 : \tau_2 \Rrightarrow \tau_2') \end{aligned}$$

Below, we have the dynamics for the cast rule between product types [13]. Casting a value from $\tau_1 \times \tau_2$ to $\tau_1' \times \tau_2'$, will boil down to a cast from τ_1 to τ_1' for the first projection, and from τ_2 to τ_2' for the second projection.

$$v : \tau_1 \times \tau_2 \Rrightarrow \tau_1' \times \tau_2' \to (\pi_1\ v : \tau_1 \Rrightarrow \tau_1',\ \pi_2\ v : \tau_2 \Rrightarrow \tau_2')$$

For casting a value from one recursive type to another, we first unfold it and perform the cast between the two types unfolded, after which we fold the result.

$$v : \mu X.\tau \Rrightarrow \mu X.\tau' \to \mathtt{fold}\ (\mathtt{unfold}\ v : \tau[X \mapsto \mu X.\tau] \Rrightarrow \tau'[X \mapsto \mu X.\tau'])$$

3.2 Formal statement and high-level overview of proof

We state the FAE-criterion formally here.

Theorem 3.1 (Fully abstract embedding). *The embedding from* STLC_μ *to* GTLC_μ *is fully abstract. That is, for all static terms* e_1, e_2 *we have that* $e_1 \approx_S e_2$ *iff* $\lceil e_1 \rceil \approx_G \lceil e_2 \rceil$.

Here, $\lceil \_ \rceil$ denotes the natural embedding from STLC_μ into GTLC_μ. The equivalence in STLC_μ/GTLC_μ is denoted by $\approx_S/\approx_G$. In both static and gradual language, we use the standard notion of contextual equivalence as equi-termination under an arbitrary context.

We give a proof that is similar to proofs given in the context of secure compilation, where fully abstract compilation is often used to characterise security of a compiler [4, 9, 10].

3.3 Proof sketch

Theorem 3.1 can be split up into the two directions.

The if-part is often denoted as reflection of equivalence. In this context, the proof is relatively easy as every static context can be seen as a gradual one, and the operational semantics of the two languages correspond adequately. For brevity, we do not go into this part of the proof.

The only-if-part – the preservation of equivalences – is more interesting. Given e_1 and e_2 such that $\Gamma \vdash e_1 \approx_S e_2 : \tau$, we have to prove $\lceil e_1 \rceil \approx_G \lceil e_2 \rceil$. In general, a gradual context can seem to have more distinguishing power than a mere static one; it is our job to prove that it is not the case. So given an arbitrary gradual context, say C_G, we have to prove that $C_G[\lceil e_1 \rceil] \Downarrow$ iff $C_G[\lceil e_2 \rceil] \Downarrow$. The proof of the if part and the only-if part are similar, here we assume $C_G[\lceil e_1 \rceil] \Downarrow$ and prove $C_G[\lceil e_2 \rceil] \Downarrow$. To do this, we will define a so-called "back-translation". This is a function that maps a gradual context to a static one that emulates its behaviour, hence proving that it indeed does not provide any more distinguishing power. Putting it formally, we want our back-translation to satisfy the following specification.

Theorem 3.2 (Spec. back-translation). *Given arbitrary well-typed static term,* $\Gamma \vdash_S e_S : \tau$*, and arbitrary target context, say* $C_G : (\Gamma;\ \tau) \rightarrow ([];\ \mathbb{1})$*, we must have that* $\langle\!\langle C_G \rangle\!\rangle[e_S] \Downarrow$ *iff* $C_G[\lceil e_S \rceil] \Downarrow$*.*

Given such a back-translation, we have that $C_G[\lceil e_1 \rceil] \Downarrow$ implies $\langle\!\langle C_G \rangle\!\rangle[e_1] \Downarrow$. Because $\langle\!\langle C_G \rangle\!\rangle$ is a source context, we know – given $e_1 \approx_S e_2$ – that $\langle\!\langle C_G \rangle\!\rangle[e_2] \Downarrow$. Using our specification one more time, we conclude $C_G[\lceil e_2 \rceil] \Downarrow$.

3.4 Back-translation

Of course, the difficult part is finding a back-translation that satisfies the specification in theorem 3.1.

We start by defining the back-translation, denoted by $\langle\!\langle \_ \rangle\!\rangle$, on types. A term of type τ will be back-translated to a term of type $\langle\!\langle \tau \rangle\!\rangle$.

$$\langle\!\langle B \rangle\!\rangle = B \qquad \langle\!\langle \tau_1 \times \tau_2 \rangle\!\rangle = \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle \qquad \langle\!\langle \tau_1 + \tau_2 \rangle\!\rangle = \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle$$

$$\langle\!\langle \tau_1 \rightarrow \tau_2 \rangle\!\rangle = \langle\!\langle \tau_1 \rangle\!\rangle \rightarrow \langle\!\langle \tau_2 \rangle\!\rangle \qquad \langle\!\langle X \rangle\!\rangle = X \qquad \langle\!\langle \mu X.\tau \rangle\!\rangle = \mu X.\langle\!\langle \tau \rangle\!\rangle \qquad \langle\!\langle \star \rangle\!\rangle = \mathscr{U}$$

The definition is mostly trivially recursive; we maintain as much information as possible when we back-translate a gradual term.

In the case of $\star$, we back-translate to a universal type in STLC_μ as inspired by [4, 9].

$$\mathscr{U} := \mu\alpha.(B + (\alpha \rightarrow \alpha) + (\alpha + \alpha) + (\alpha \times \alpha) + \alpha)$$

The last case in the disjoint sum is the case for a folded value of a recursive type.

The back-translation is most easily defined on terms of the cast-calculus itself. Most of it is simply recursive. The only interesting case is the one for casting.

$$\langle\!\langle e : \tau \Rightarrow \tau' \rangle\!\rangle = \mathscr{F}^{\Sigma}_{\tau \Rightarrow \tau'}\ \langle\!\langle e \rangle\!\rangle$$

That is, for all types τ, τ', we define a cast from τ to τ' as a –to be defined– simply-typed function $\mathscr{F}^{\Sigma}_{\tau \Rightarrow \tau'}$ of type $\langle\!\langle \tau \rangle\!\rangle \rightarrow \langle\!\langle \tau' \rangle\!\rangle$.

We first go over functions of the form $\mathscr{F}^{\Sigma}_{G \Rightarrow \star}$:

$$\begin{aligned}
\mathscr{F}^{\Sigma}_{B \Rightarrow \star} &: B \rightarrow \mathscr{U} \\
\mathscr{F}^{\Sigma}_{\star \times \star \Rightarrow \star} &: (\mathscr{U} \times \mathscr{U}) \rightarrow \mathscr{U} \\
\mathscr{F}^{\Sigma}_{\star + \star \Rightarrow \star} &: (\mathscr{U} + \mathscr{U}) \rightarrow \mathscr{U} \\
\mathscr{F}^{\Sigma}_{\star \rightarrow \star \Rightarrow \star} &: (\mathscr{U} \rightarrow \mathscr{U}) \rightarrow \mathscr{U} \\
\mathscr{F}^{\Sigma}_{\mu X.\star \Rightarrow \star} &: \mu X.\mathscr{U} \rightarrow \mathscr{U}
\end{aligned}$$

These have the effect of injecting terms into our universal type; they are all implemented by a tedious yet straightforward use of `inl`'s, `inr`'s, and `fold`s.

We now define the functions of the form $\mathscr{F}^{\Sigma}_{\star \Rightarrow G}$:

$$\begin{aligned}
\mathscr{F}^{\Sigma}_{\star \Rightarrow B} &: \mathscr{U} \rightarrow B \\
\mathscr{F}^{\Sigma}_{\star \Rightarrow \star \times \star} &: \mathscr{U} \rightarrow (\mathscr{U} \times \mathscr{U}) \\
\mathscr{F}^{\Sigma}_{\star \Rightarrow \star + \star} &: \mathscr{U} \rightarrow (\mathscr{U} + \mathscr{U}) \\
\mathscr{F}^{\Sigma}_{\star \Rightarrow \star \rightarrow \star} &: \mathscr{U} \rightarrow (\mathscr{U} \rightarrow \mathscr{U}) \\
\mathscr{F}^{\Sigma}_{\star \Rightarrow \mu X.\star} &: \mathscr{U} \rightarrow \mathscr{U}
\end{aligned}$$

These are implemented by unfolding the input argument, after which a series of case matches are introduced to determine whether the input argument is of the right form. If it is, we just return it in the unfolded form. If it is not, we just diverge.

As a sanity check, we can indeed convince ourselves of the following contextual equivalences.

$$(\lambda x : \langle\!\langle G \rangle\!\rangle.\ x) \approx_S (\lambda x : \langle\!\langle G \rangle\!\rangle.\ \mathscr{F}^{\Sigma}_{\star \Rightarrow G}\ (\mathscr{F}^{\Sigma}_{G \Rightarrow \star}\ x))$$

$$(\lambda x : \langle\!\langle G_1 \rangle\!\rangle.\ \Omega_{\langle\!\langle G_2 \rangle\!\rangle}) \approx_S (\lambda x : \langle\!\langle G_1 \rangle\!\rangle.\ \mathscr{F}^{\Sigma}_{\star \Rightarrow G_2}\ (\mathscr{F}^{\Sigma}_{G_1 \Rightarrow \star}\ x)) \text{ where } G_1 \neq G_2$$

To implement $\mathscr{F}^{\Sigma}_{\tau \Rightarrow \star}$ and $\mathscr{F}^{\Sigma}_{\star \Rightarrow \tau}$ where $\tau \neq \star$ and where τ is not equal to a ground type, we straightforwardly emulate the factorisation through its unique ground type, say G, as follows.

$$\begin{aligned}\mathscr{F}^{\Sigma}_{\tau \Rightarrow \star} &= \lambda x : \langle\!\langle \tau \rangle\!\rangle.\ \mathscr{F}^{\Sigma}_{G \Rightarrow \star}\ (\mathscr{F}^{\Sigma}_{\tau \Rightarrow G}\ x)\\ \mathscr{F}^{\Sigma}_{\star \Rightarrow \tau} &= \lambda x : \mathscr{U}.\ \mathscr{F}^{\Sigma}_{G \Rightarrow \tau}\ (\mathscr{F}^{\Sigma}_{\star \Rightarrow G}\ x)\end{aligned}$$

It remains only to define the emulation functions for casting between sum types, product types, function and recursive types. Between sum types, we have the following.

$$\begin{aligned}\mathscr{F}^{\Sigma}_{\tau_1+\tau_2 \Rightarrow \tau_1'+\tau_2'} = \lambda s : \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle.\ \mathtt{case}\ s\ \mathtt{of}\ &\lambda x_1 : \langle\!\langle \tau_1 \rangle\!\rangle.\ \mathtt{inl}\ (\mathscr{F}^{\Sigma}_{\tau_1 \Rightarrow \tau_1'}\ x_1)\\ &\lambda x_2 : \langle\!\langle \tau_2 \rangle\!\rangle.\ \mathtt{inr}\ (\mathscr{F}^{\Sigma}_{\tau_2 \Rightarrow \tau_2'}\ x_2)\end{aligned}$$

Between products, we have the following.

$$\mathscr{F}^{\Sigma}_{\tau_1 \times \tau_2 \Rightarrow \tau_1' \times \tau_2'} = \lambda p : \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle.\ (\mathscr{F}^{\Sigma}_{\tau_1 \Rightarrow \tau_1'}\ (\pi_1\ p),\ \mathscr{F}^{\Sigma}_{\tau_2 \Rightarrow \tau_2'}\ (\pi_2\ p))$$

For function types, we have the following.

$$\mathscr{F}^{\Sigma}_{\tau_a \to \tau_r \Rightarrow \tau_a' \to \tau_r'} = \lambda f : \langle\!\langle \tau_a \to \tau_r \rangle\!\rangle.\ \lambda a : \langle\!\langle \tau_a' \rangle\!\rangle.\ \mathscr{F}^{\Sigma}_{\tau_r \Rightarrow \tau_r'}\ (f\ (\mathscr{F}^{\Sigma}_{\tau_a' \Rightarrow \tau_a}\ a))$$

The superscript is used to keep track of recursive calls, and it is used for casting between recursive types.

$$\mathscr{F}^{\Sigma}_{\mu X.\tau \Rightarrow \mu X'.\tau'} = (\mathtt{rec}\ (f : \mu X.\langle\!\langle \tau \rangle\!\rangle \to \mu X'.\langle\!\langle \tau' \rangle\!\rangle)\ r = \mathtt{fold}\ (\mathscr{F}^{\Sigma, X:f}_{\tau \Rightarrow \tau'}\ (\mathtt{unfold}\ r)))$$

$$\mathscr{F}^{\Sigma,\ X:f,\ \Sigma'}_{X \Rightarrow X'} = f$$

Showing that this back-translation satisfies theorem 3.2 demands a proof, to be carried out with step-indexed logical relations, in a style similar to [4, 9]. We hope to significantly ease this proof-burden by using the state-of-the-art program logic Iris [8].

4 DISCUSSIONS

Contextual equivalence in GTLC$_\mu$. We state the full abstraction of the embedding from STLC$_\mu$ to GTLC$_\mu$ wrt the standard notion of contextual equivalence by equi-terminations in both STLC$_\mu$ and GTLC$_\mu$. Below, we have it in full for GTLC$_\mu$.

Definition 4.1 (Contextual equivalence in Gradual Language). Two terms e_1, e_2 in GTLC$_\mu$ are equivalent, written $e_1 \approx_G e_2$, iff $\Gamma \vdash e_1, e_2 : \tau$ and for all gradual contexts $C_G : (\Gamma; \tau) \to ([]; \mathbb{1})$, we have $C_g[e_1] \Downarrow$ iff $C_g[e_2] \Downarrow$.

Note that this notion entails that if $\Gamma \vdash e \approx_G e' : \tau$ for some e and e', there can be a gradual context, say $C_G : (\Gamma; \tau) \to ([]; \mathbb{1})$, for which $C_G[e]$ diverges and $C_G[e']$ gives a runtime error. Equating divergence with runtime errors allows us to simplify the formalities (otherwise, we would need to add runtime errors in STLC$_\mu$). We also believe it can often be justified, at least in settings where runtime type errors cannot be caught, as for many purposes, the difference between divergence and uncatchable runtime errors is not practically important.

Why consider STLC_μ *and not* STLC *without recursion?* Well, the embedding of STLC into the archetypal GTLC is not fully abstract. As every function in STLC is terminating, we have that, *e.g.*, $\lambda f : \mathbb{B} \to \mathbb{B}.\ 43$ and $\lambda f : \mathbb{B} \to \mathbb{B}.\ \pi_2(f\ true, 43)$ are contextually equivalent in the static language. Considering their embeddings as part of the gradual language though, they are not any more. As GTLC embeds the untyped lambda calculus, it allows general recursion, and so we can construct a gradual context $[]\ (\lambda b : \mathbb{B}.\ \Omega)$ that will be able to distinguish the two.

Is the lack of FAE for the archetypal gradualisation of STLC an argument against our criterion? We think not. It simply shows that GTLC should not be considered as the gradualisation of STLC as it does not uphold the termination guarantees which are part of the meaning of STLC types. Whether such an enforcement is possible and how it would work is, of course, a separate question. Instead, GTLC should be considered the gradualisation of a non-terminating variant of STLC. Tanter et al. [7] also discuss this and note that type soundness is broken with respect to total correctness.

This compromise can also be formalised by noting that instead of "full" full abstraction of the embedding, the embedding still satisfies full abstraction with respect to a weaker termination-insensitive contextual equivalence in the gradual language.

Why add recursive types instead of just plain recursion using a fix operator? "Why bother with recursive *types*," the reader may wonder, "if adding recursion to STLC with just a `fix` operator will do the trick". Admittedly, we believe that the natural gradualisation of STLC_{fix} satisfies the FAE-criterion.

Nevertheless, FAE seems much easier to prove for the gradualisation of STLC_μ as it allows for a true universal type [9, 10]. Futhermore, given proofs of full abstraction for the following,

(a) the embedding from $\text{STLC}_{\mu,\text{fix}}$ into its gradualisation
(b) the embedding from STLC_{fix} into STLC_μ[2]

a proof of the FAE-criterion for the gradualisation of STLC_{fix} is easily obtained.

Indeed, full abstraction is preserved upon composition, and thus we would know that the embedding from STLC_{fix} into $\text{GTLC}_{\mu,\text{fix}}$ is fully abstract. 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 could then conclude that the embedding from STLC_{fix} into GTLC_{fix} preserves contextual equivalences as well.[3]

Decomposing the proof as traced out above, we delegate the distracting technicalities (b), allowing us to focus on the crux of the matter (a).[4]

Future work. Do current state-of-the-art gradual languages or even gradualisation mechanisms satisfy the FAE criterion? In contrast to the obviously naive example in 3, we might begin to look at a serious gradualisation of an information-flow language as presented in [6]. To what extent does the FAE criterion further weed out erroneous gradualisations? That is, to what extent do we have unsatisfactory gradualisations that still satisfy the refined criteria but not the FAE-criterion? Investigating these questions for gradual languages proposed in the literature, is an interesting exercise that we leave for future work.

The FAE is essentially a secure compilation property in which the compiler is merely the trivial embedding. It would be interesting to see if other concepts from secure compilation have interesting interpretations when considering the trivial embedding as compiler. Of special interest here could

[2]Using the idea of a family of approximate back-translations[4], a proof of this seems very feasible; the possibility is explicitly mentioned in [10] and we know that the authors of that paper are currently working on it.
[3]Of course, reflection of contextual equivalences follows from the refined criteria[14].
[4]We strip the latter a bit more by considering the gradualisation of STLC_μ.

be the robust safety properties [12] and other variants of full abstraction [3, 11]. Moreover, we intend to investigate whether some of the refined criteria might be implied by such a variant.

5 CONCLUSION

As argued in the introduction, it is not a controversial statement to say that the refined criteria can be further refined. We have argued for the criterion of full abstraction as previously suggested by [5]. We have shown that it is satisfied by the natural gradualisation of STLC_μ, and that it can indeed weed out undesirable gradual languages. We expect that our new criterion excludes languages such as those mentioned in the introduction (violating parametricity and non-interference), but we have not investigated this thoroughly yet.

In summary, we propose the FAE-criterion as a sensible *requirement* that a gradual language may satisfy, worthy of further investigation.

ACKNOWLEDGMENTS

This work was funded in part by Internal Funds KU Leuven grant C14/18/064.

REFERENCES

[1] 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

[2] Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming*. Springer-Verlag.

[3] 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

[4] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 164–177.

[5] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017. Parametricity versus the universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 38.

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

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

[8] 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).

[9] 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.

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

[11] M. Patrignani and D. Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *Computer Security Foundations Symposium*. IEEE, 392–404. https://doi.org/10.1109/CSF.2017.13

[12] Marco Patrignani and Deepak Garg. 2019. Robustly safe compilation. In *European Symposium on Programming*. Springer, 469–498.

[13] 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).

[14] 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))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[15] 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