Parametricity versus the Universal Type

DOMINIQUE DEVRIESE, KU Leuven, Belgium
MARCO PATRIGNANI*, MPI-SWS & CISPA, Germany
FRANK PIESSENS, KU Leuven, Belgium

There has long been speculation in the scientific literature on how to dynamically enforce parametricity such as that yielded by System F. Almost 20 years ago, Sumii and Pierce proposed a formal compiler from System F into the cryptographic lambda calculus: an untyped lambda calculus extended with an idealised model of encryption. They conjectured that this compiler was fully abstract, i.e. that compiled terms are contextually equivalent if and only if the original terms were, a property that can be seen as a form of secure compilation. The conjecture has received attention in several other publications since then, but remains open to this day.

More recently, several researchers have been looking at gradually-typed languages that extend System F. In this setting it is natural to wonder whether embedding System F into these gradually-typed languages preserves contextual equivalence and thus parametricity.

In this paper, we answer both questions negatively. We provide a concrete counterexample: two System F terms whose contextual equivalence is not preserved by the Sumii-Pierce compiler, nor the embedding into the polymorphic blame calculus. This counterexample relies on the absence in System F of what we call a *universal* type, i.e., a type that all other types can be injected into and extracted from. As the languages in which System F is compiled have a universal type, the compilation cannot be fully abstract; this paper explains why.

We believe this paper thus sheds light on recent results in the field of gradually typed languages and it provides a perspective for further research into secure compilation of polymorphic languages.

To better explain and clarify notions, this paper uses colours, please print or view this in colours.

CCS Concepts: • **Security and privacy** → **Formal security models**; *Logic and verification*; • **Theory of computation** → *Logic and verification*;

Additional Key Words and Phrases: Fully abstract compilation, System F, sealing, parametricity, universal type

ACM Reference Format:
Dominique Devriese, Marco Patrignani, and Frank Piessens. 2018. Parametricity versus the Universal Type. *Proc. ACM Program. Lang.* 2, POPL, Article 38 (January 2018), 23 pages. https://doi.org/10.1145/3158126

1 INTRODUCTION

System F is a widely influential type system, originally defined by Reynolds [1974] and Girard [1972], featuring parametric polymorphism and an impredicative universe. Parametricity expresses the property that polymorphic functions in System F cannot behave differently when invoked at different types. It can be formalised as an (often relational) property that all inhabitants of a type

*This work was performed when the author was at MPI-SWS.

Authors' addresses: Dominique Devriese, imec-Distrinet, KU Leuven, Leuven, Belgium, name.surname@cs.kuleuven.be; Marco Patrignani, MPI-SWS & CISPA, Saarbrücken, Germany, marcopat@mpi-sws.org; Frank Piessens, imec-Distrinet, KU Leuven, Leuven, Belgium, name.surname@cs.kuleuven.be.

2475-1421/2018/1-ART38
https://doi.org/10.1145/3158126

automatically satisfy [Reynolds 1983; Wadler 1989]. Such properties are automatically satisfied for any function without the need to verify their code. A typical example is the fact that any value of type $\forall \mathbf{X}.\, \mathbf{X} \to \mathbf{X}$ must behave as the identity function (or possibly diverge in a non-terminating variant of System F).

The topic of this paper is the interaction of System F programs with programs in a weaker type-system. In such weaker systems, a System F component may receive values of a certain type from more weakly typed code. It is then not obvious that such values will satisfy the same properties as System F values, specifically parametricity. For example, if a System F component receives a function of type $\forall \mathbf{X}.\, \mathbf{X} \to \mathbf{X}$ from a weakly-typed component, will it necessarily behave as the identity function (or diverge)?

It is already worth noting that the weaker type systems we consider are all non-terminating, and when interacting with them, the only properties that have a chance of being preserved, are those that hold in the presence of non-termination. This is the reason that the variant of System F that we consider is non-terminating, obtained by adding recursive types. We discuss in more detail why this is the right choice in Section 3.2.

We will look at the question of preserving properties of System F terms in two specific settings.

Secure compilation. The field of secure compilation studies high-level programming languages that are compiled to low-level target languages where they may interact with untrusted low-level components. The goal of secure compilation is to ensure that those low-level components can only interact with the compiled code in ways that high-level components can interact with the original code. This constitutes a powerful security property, as it effectively excludes a wide variety of low-level attacks like improper stack manipulation, breaking control flow guarantees, reading from or writing to private memory of other components, inspecting or modifying the implementation of a function etc.

Formally, security of a compiler has been mostly expressed as full abstraction: given two source-level contextually equivalent programs, their target-level compilation are also contextually equivalent (and vice versa) [Abadi 1998]. Compiler full abstraction has been proven for compilers that rely on address-space layout randomisation [Abadi and Plotkin 2012; Jagadeesan et al. 2011], or secure enclaves [Agten et al. 2012; Larmuseau et al. 2015, 2016; Patrignani et al. 2015, 2016], tagged architectures [Juglaret et al. 2016, 2015], JavaScript [Fournet et al. 2013], typed closure conversion [Ahmed and Blume 2008], cryptographic primitives [Abadi et al. 1998, 1999, 2000; Bugliesi and Giunti 2007] etc.

In a paper from the year 2000, Pierce and Sumii [2000] proposed a compiler from System F to a cryptographic lambda calculus, which enforces parametricity using a form of idealised encryption primitives (sealing) called lambda-seal (λ^{σ}). They conjectured that this compiler was fully abstract, a conjecture that has received further research attention, but remains open to this day [Siek and Wadler 2016; Sumii and Pierce 2004].

Gradual typing. A second field where System F components interact with untyped code is gradual typing. In order to allow programmers to incrementally port large, untyped code bases to a typed programming language, gradual programming languages allow for typed and untyped code to interact. Such languages generally strive to preserve the benefits of the statically-typed components of an application, even when interacting with untyped components. Such benefits include performance benefits, absence of type errors but also benefits in terms of reasoning. While the literature mentions several ways to formalise the former two properties, the latter has received less attention.

A natural way to formally express that a gradual language preserves the typed language's reasoning principles is to reuse the same notion of fully abstract compilation that we mentioned

above. Specifically, if the embedding of the typed language into the gradual language is fully abstract, then similarly to secure compilation, this expresses that untyped code can only interact with typed code in ways that are also possible using just typed code.

Also in the field of gradual typing, System F's parametric polymorphism presents a formidable challenge. The observation that sealing could be useful to combine parametric polymorphism with dynamic typing was already made by Pierce and Sumii [2000]. This idea was further developed with the definition of a gradually-typed language based on this idea [Ahmed et al. 2011c; Matthews and Ahmed 2008], the addition of blame in the polymorphic blame calculus [Ahmed et al. 2011a], further developed by Igarashi et al. [2017] and Ahmed et al. [2017]. Ahmed et al. also prove a form of parametricity in the polymorphic blame calculus, but their formulation is rather different than existing ones (e.g. Dreyer et al. [2009]; Reynolds [1983]) because of its use of Kripke worlds to capture the interpretation of generated runtime types. As such, it remains unclear how this notion of parametricity is related to existing ones and whether the polymorphic blame calculus preserves all System F reasoning, particularly its contextual equivalences (i.e. whether it embeds System F in a fully abstract way).

Two questions, one answer. In this paper, we solve both open problems, answering the questions negatively. We prove that Sumii and Pierce's compiler is not fully abstract and we show the same for the embedding of System F into the polymorphic blame calculus (i.e. equivalences are not preserved under the embedding).

Our counterexamples are essentially based on the observation that System F parametricity forbids the existence of a universal type: a type which any other type can be embedded into and extracted from. [1] More precisely, consider the following type:

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

Univ can be read as expressing the existence of a universal type: a type **Y** such that for any other type **X**, there is a mapping from **X** into **Y** and vice versa. In a non-terminating variant of System F, there exist inhabitants of **Univ**, but we prove that they are all degenerate in the sense that mapping a value into the universal type and back must necessarily diverge.

However, Sumii and Pierce's compiler fails to enforce this degeneracy of **Univ**. In fact, their target language really does contain a universal type: since it is untyped we can think about it as being "uni-typed", citing Dana Scott [Statman 1991]. As a consequence, their conjecture is false, as we will formally show by constructing two System F terms $\mathbf{t_d}$ and $\mathbf{t_u}$ whose contextual equivalence relies on the degeneracy of **Univ**. We can then falsify Sumii and Pierce's conjecture by showing that these two terms are mapped to non-equivalent terms by their proposed compiler.

In the field of gradual typing, we demonstrate that existing polymorphic blame calculi also break the degeneracy of **Univ**. Like Sumii and Pierce's target language, they also provide a universal type: the type of untyped values $\star$, common to most gradual languages (also often indicated as ?). Exploiting the existence of this type, we demonstrate that the polymorphic blame calculus does not embed System F in a fully abstract way and as a result, they do not preserve System F's parametricity.

Finally, we discuss a number of consequences and perspectives that follow from our results. First, we discuss some thoughts on how Sumii and Pierce's compiler might be fixed (so that it does enforce full abstraction), and what could be modified in the polymorphic blame calculus to make it

[1] Note that by this definition, the term *universal type* is broader than Abadi's *dynamic type* [Abadi et al. 1991] or Siek and Taha's *gradual type* [Siek and Taha 2006]: the former includes *any* type that arbitrary values can be embedded into and extracted from, while the latter are specific primitive types, specifically intended for representing untyped values in a typed language.

preserve all of System F's contextual equivalences. However, in neither case there appears to be a panacea solution: potential fixes all seem to come with certain downsides. Because of this, we also discuss whether we should not instead adjust our expectations and find a way to formalise the guarantees that we do get from both Sumii and Pierce's compiler and the polymorphic blame calculus. This exercise allows us to shed some new light on recent results from the literature.

Outline. We start our discussion in Section 2 by repeating the definition of System F and presenting the terms $\mathbf{t_u}$ and $\mathbf{t_d}$. We prove they are contextually equivalent from the degeneracy of System F, relying on an existing formalisation of System F's parametricity [Dreyer et al. 2009]. Next, we disprove Sumii and Pierce's conjecture in Section 3 by explaining their compiler and how it treats $\mathbf{t_d}$ and $\mathbf{t_u}$ and fails to preserve their equivalence. In Section 4, we turn to gradual typing, introducing polymorphic blame calculi and demonstrating how $\mathbf{t_u}$ and $\mathbf{t_d}$'s contextual equivalence is also lost in the presence of these calculi's universal type. Finally, we discuss perspectives and consequences of our results in Section 5, related work in Section 6 and we conclude in Section 7. For space constraints we omit most proofs and formalisation, which can be found in the supplementary material.

To make the distinction between source and target language visually apparent, we typeset elements of System F in a **blue, bold** font, elements of λ^σ in a red, sans-serif font and elements of the blame calculus in an *orange, italic* font.

2 SYSTEM F AND THE TYPE Univ

Let us first consider System F itself, its parametricity and our claim that parametricity excludes the existence of a universal type.

2.1 The Source Language $\lambda^{\mathbf{F}}$

Figure 1 presents the variant of System F that we will be using in this paper, which we indicate with $\lambda^{\mathbf{F}}$. In addition to standard polymorphic functions ($\forall \mathbf{X}.\ \tau$) and existential packages ($\exists \mathbf{X}.\ \tau$), the variant includes recursive types $\mu \mathbf{X}.\ \tau$ (so there are no termination guarantees), as well as **Unit** and **Bool** and product $\tau_1 \times \tau_2$ types. In the figure, we present terms $\mathbf{t}$, values $\mathbf{v}$ and types τ. We show the most important typing rules $\mathbf{\Delta}; \mathbf{\Gamma} \vdash \mathbf{t} : \tau$ in terms of term and type variable contexts $\mathbf{\Gamma}$ and $\mathbf{\Delta}$ (but we omit context and type well-formedness judgements $\mathbf{\Delta}; \mathbf{\Gamma} \vdash \diamond$ and $\mathbf{\Delta} \vdash \tau$). Finally, we define call-by-value evaluation rules in terms of evaluation contexts $\mathbb{E}$.

Program contexts $\mathfrak{C}$ are defined as terms with exactly one subterm replaced by a hole $[\cdot]$. An omitted well-typedness judgement for program contexts $\mathfrak{C} : \mathbf{\Delta}; \mathbf{\Gamma}; \tau \rightarrow \mathbf{\Delta}'; \mathbf{\Gamma}'; \tau'$ guarantees that plugging a well-typed term $\mathbf{\Delta}; \mathbf{\Gamma} \vdash \mathbf{t} : \tau$ in the hole produces a well-typed result term $\mathbf{\Delta}'; \mathbf{\Gamma}' \vdash \mathfrak{C}[\mathbf{t}] : \tau'$.

Definition 2.1 ($\lambda^{\mathbf{F}}$ Contextual equivalence). For two terms $\mathbf{t_1}, \mathbf{t_2}$ that have the same type τ in the same context $\mathbf{\Delta}; \mathbf{\Gamma}$, we define that they are contextually equivalent ($\mathbf{\Delta}; \mathbf{\Gamma} \vdash \mathbf{t_1} \simeq_{ctx} \mathbf{t_2} : \tau$) iff for all $\mathfrak{C}$ such that $\vdash \mathfrak{C} : \mathbf{\Delta}; \mathbf{\Gamma}, \tau \rightarrow \emptyset; \emptyset, \tau'$, we have that $\mathfrak{C}[\mathbf{t_1}] \Uparrow$ iff $\mathfrak{C}[\mathbf{t_2}] \Uparrow$, where $\Uparrow$ indicates divergence [Plotkin 1977].

2.2 Parametricity

The main information hiding mechanism in $\lambda^{\mathbf{F}}$ is parametric polymorphism: the representation type of an existentially quantified package is invisible outside the package, and hence clients of the package cannot depend on that representation type.

Example 2.2 ($\mathbb{Z}_n$ implementation in $\lambda^{\mathbf{F}}$). We could for instance represent the type $\mathbb{Z}_n$ of integers modulo n as a tuple $\langle \langle \mathbf{zero}, \mathbf{succ} \rangle, \mathbf{zero?} \rangle$ of type $\exists \mathbf{X}.\ (\mathbf{X} \times (\mathbf{X} \rightarrow \mathbf{X})) \times (\mathbf{X} \rightarrow \mathbf{Bool})$, and then either

Syntax:

$$\begin{aligned}
\mathbf{t} &::= \mathbf{v} \mid \mathbf{x} \mid \mathbf{t}\ \mathbf{t} \mid \mathbf{t.1} \mid \mathbf{t.2} \mid \langle \mathbf{t}, \mathbf{t} \rangle \mid \Lambda \mathbf{X}.\, \mathbf{t} \mid \mathbf{t}\ \tau \mid \mathbf{if}\ \mathbf{t}\ \mathbf{then}\ \mathbf{t}\ \mathbf{else}\ \mathbf{t} \\
&\quad \mid \mathbf{pack}\ \langle \tau, \mathbf{t} \rangle\ \mathbf{as}\ \exists \mathbf{X}.\, \tau \mid \mathbf{unpack}\ \mathbf{t}\ \mathbf{as}\ \langle \mathbf{X}, \mathbf{x} \rangle\ \mathbf{in}\ \mathbf{t} \mid \mathbf{roll}\ \mathbf{t} \mid \mathbf{unroll}\ \mathbf{t} \\
\mathbf{v} &::= \mathbf{unit} \mid \mathbf{true} \mid \mathbf{false} \mid \lambda \mathbf{x} : \tau.\, \mathbf{t} \mid \langle \mathbf{v}, \mathbf{v} \rangle \mid \Lambda \mathbf{X}.\, \mathbf{t} \mid \mathbf{pack}\ \langle \tau, \mathbf{v} \rangle\ \mathbf{as}\ \exists \mathbf{X}.\, \tau \mid \mathbf{roll}\ \mathbf{v} \\
\tau &::= \mathbf{Unit} \mid \mathbf{Bool} \mid \tau \to \tau \mid \tau \times \tau \mid \mathbf{X} \mid \forall \mathbf{X}.\, \tau \mid \exists \mathbf{X}.\, \tau \mid \mu \mathbf{X}.\, \tau \\
\Gamma &::= \emptyset \mid \Gamma, (\mathbf{x} : \tau) \qquad\qquad \Delta ::= \emptyset \mid \Delta, \mathbf{X} \\
\mathbb{E} &::= [\cdot] \mid \mathbb{E}\ \mathbf{t} \mid \mathbf{v}\ \mathbb{E} \mid \mathbb{E}.\mathbf{1} \mid \mathbb{E}.\mathbf{2} \mid \langle \mathbb{E}, \mathbf{t} \rangle \mid \langle \mathbf{v}, \mathbb{E} \rangle \mid \mathbb{E}\ \tau \mid \mathbb{E}; \mathbf{t} \mid \mathbf{inl}\ \mathbb{E} \mid \mathbf{inr}\ \mathbb{E} \\
&\quad \mid \mathbf{case}\ \mathbb{E}\ \mathbf{of}\ \mathbf{inl}\ \mathbf{x}_1 \mapsto \mathbf{t} \mid \mathbf{inr}\ \mathbf{x}_2 \mapsto \mathbf{t} \mid \mathbf{if}\ \mathbb{E}\ \mathbf{then}\ \mathbf{t}\ \mathbf{else}\ \mathbf{t} \mid \mathbf{pack}\ \langle \tau, \mathbb{E} \rangle\ \mathbf{as}\ \exists \mathbf{X}.\, \tau \\
&\quad \mid \mathbf{unpack}\ \mathbb{E}\ \mathbf{as}\ \langle \mathbf{X}, \mathbf{x} \rangle\ \mathbf{in}\ \mathbf{t} \mid \mathbf{unpack}\ \mathbf{v}\ \mathbf{as}\ \langle \mathbf{X}, \mathbf{x} \rangle\ \mathbf{in}\ \mathbb{E} \mid \mathbf{roll}\ \mathbb{E} \mid \mathbf{unroll}\ \mathbb{E}
\end{aligned}$$

Typing rules (excerpt):

$$\frac{\Delta; \Gamma \vdash \diamond \quad (\mathbf{x} : \tau) \in \Gamma}{\Delta; \Gamma \vdash \mathbf{x} : \tau} \qquad \frac{\Delta; \Gamma, (\mathbf{x} : \tau) \vdash \mathbf{t} : \tau'}{\Delta; \Gamma \vdash \lambda \mathbf{x} : \tau.\, \mathbf{t} : \tau \to \tau'} \qquad \frac{\Delta, \mathbf{X}; \Gamma \vdash \mathbf{t} : \tau}{\Delta; \Gamma \vdash \Lambda \mathbf{X}.\, \mathbf{t} : \forall \mathbf{X}.\, \tau} \qquad \frac{\Delta; \Gamma \vdash \mathbf{t} : \tau' \to \tau \quad \Delta; \Gamma \vdash \mathbf{t}' : \tau'}{\Delta; \Gamma \vdash \mathbf{t}\ \mathbf{t}' : \tau}$$

$$\frac{\Delta \vdash \tau' \quad \Delta; \Gamma \vdash \mathbf{t} : \forall \mathbf{X}.\, \tau}{\Delta; \Gamma \vdash \mathbf{t}\ \tau' : \tau[\tau'/\mathbf{X}]} \qquad \frac{\Delta; \Gamma \vdash \mathbf{t} : \tau[\mu \mathbf{X}.\, \tau/\mathbf{X}]}{\Delta; \Gamma \vdash \mathbf{roll}\ \mathbf{t} : \mu \mathbf{X}.\, \tau} \qquad \frac{\Delta; \Gamma \vdash \mathbf{t} : \mu \mathbf{X}.\, \tau}{\Delta; \Gamma \vdash \mathbf{unroll}\ \mathbf{t} : \tau[\mu \mathbf{X}.\, \tau/\mathbf{X}]}$$

$$\frac{\Delta \vdash \tau \quad \Delta; \Gamma \vdash \mathbf{t} : \tau[\tau'/\mathbf{X}]}{\Delta; \Gamma \vdash \mathbf{pack}\ \langle \tau', \mathbf{t} \rangle\ \mathbf{as}\ \exists \mathbf{X}.\, \tau : \exists \mathbf{X}.\, \tau} \qquad \frac{\Delta; \Gamma \vdash \mathbf{t} : \exists \mathbf{X}.\, \tau \quad \Delta \vdash \tau' \quad \Delta, \mathbf{X}; \Gamma, \mathbf{x} : \tau \vdash \mathbf{t}_1 : \tau'}{\Delta; \Gamma \vdash \mathbf{unpack}\ \mathbf{t}\ \mathbf{as}\ \langle \mathbf{X}, \mathbf{x} \rangle\ \mathbf{in}\ \mathbf{t}_1 : \tau'}$$

Evaluation rules (excerpt):

$$\frac{\mathbf{t} \hookrightarrow_0 \mathbf{t}'}{\mathbb{E}[\mathbf{t}] \hookrightarrow \mathbb{E}[\mathbf{t}']} \qquad \frac{}{(\lambda \mathbf{x} : \tau.\, \mathbf{t})\ \mathbf{v} \hookrightarrow_0 \mathbf{t}[\mathbf{v}/\mathbf{x}]} \qquad \frac{}{(\Lambda \mathbf{X}.\, \mathbf{t})\ \tau \hookrightarrow_0 \mathbf{t}[\tau/\mathbf{X}]}$$

$$\frac{}{\mathbf{unpack}\ (\mathbf{pack}\ \langle \tau', \mathbf{v} \rangle\ \mathbf{as}\ \exists \mathbf{X}.\, \tau)\ \mathbf{as}\ \langle \mathbf{X}', \mathbf{x} \rangle\ \mathbf{in}\ \mathbf{t} \hookrightarrow_0 \mathbf{t}[\mathbf{v}/\mathbf{x}][\tau'/\mathbf{X}']} \qquad \frac{}{\mathbf{unroll}\ (\mathbf{roll}\ \mathbf{v}) \hookrightarrow_0 \mathbf{v}}$$

Fig. 1. System F syntax, typing rules and evaluation rules (excerpt). The semantics relation $\hookrightarrow$ relies on the primitive reductions indicated as $\hookrightarrow_0$.

implement this type as a k-tuple of booleans, or by means of a type of natural numbers defined using recursive types.

Dually, the type system ensures that code cannot depend on parameters of universally quantified types, for instance the only thing a function of type $\forall \mathbf{X}.\, \mathbf{X} \times \mathbf{X} \to \mathbf{X}$ can do is return one of its two arguments or diverge.

Reynolds formalised parametricity in the form of a theorem that all λ^{F} terms of a certain type satisfy a property that can be derived from their type [Reynolds 1983]. For example, if we assume a value $\mathbf{f}$ of type $\forall \mathbf{X}.\, \mathbf{X} \to \mathbf{X}$, then parametricity states that for any closed types τ_1, τ_2, any relation R between values of types τ_1 and τ_2, and any two closed values $\mathbf{v}_1, \mathbf{v}_2$ of type τ_1 and τ_2 respectively, if $(\mathbf{v}_1, \mathbf{v}_1)$ is in R, then $\mathbf{f}\ \tau_1\ \mathbf{v}_1$ and $\mathbf{f}\ \tau_2\ \mathbf{v}_2$ will either both diverge or reduce to values $(\mathbf{v}_1', \mathbf{v}_2') \in R$.

The relational property is derived from the type using what is known as a logical relation.[2] For this paper, it is not necessary to understand how this works, so we refer the interested reader to other articles that explain this well [Dreyer et al. 2009; Wadler 1989].

2.3 The Type Univ

A central role in this paper is reserved for the type **Univ**:

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

The type can be read as stating the existence of a universal type **Y**: a type that all other types can be embedded into and extracted from.

In our non-terminating variant of $\lambda^{\mathbf{F}}$, **Univ** is clearly inhabited, for example by this value:

$$\mathbf{pack}\ \langle \mathbf{Unit}, \mathbf{\Lambda X}.\, \langle \lambda\_ : \mathbf{X}.\, \mathbf{unit}, \lambda\_ : \mathbf{Unit}.\, \omega_{\mathbf{X}} \rangle \rangle\ \mathbf{as\ Univ}$$

Note that we write $\omega_{\mathbf{X}}$ for a diverging term of type **X**, which can be constructed in the standard way using recursive types. However, this value is *degenerate* in the sense that injecting a value into the packaged **Y** and extracting it again diverges.

A crucial observation for this paper is that *all* values of type **Univ** are degenerate in this sense. Intuitively, this is because a single type **Y** needs to be chosen, independently of the types **X** that will be embedded into it. Because nothing is known upfront about these **X**s and nothing can be learnt about them after invocation (because **X** must be treated parametrically), no viable choice for **Y** can be made.[3]

To understand more formally how the degeneracy of **Univ** follows from parametricity, it is instructive to consider the same type in a simpler setting. Let us consider the following lemma, in a terminating variant of $\lambda^{\mathbf{F}}$, which has universal and existential types, the unit type but no recursive types. We indicate the terminating variant of $\lambda^{\mathbf{F}}$ as $\lambda^{\mathbf{F}}_{\Downarrow}$.

Lemma 2.3 (**Univ** is not inhabited in $\lambda^{\mathbf{F}}_{\Downarrow}$). *Assume that*

- *A value* $\mathbf{v}$ *of type* **Univ**: $\emptyset; \emptyset \vdash \mathbf{v} : \mathbf{Univ}$
- $\mathbf{v}$ *satisfies unary parametricity, i.e. there exist* τ_Y, $P_Y \in \mathcal{P}(\llbracket \tau_Y \rrbracket)$ *and* $\mathbf{v}'$ *such that we have* $\mathbf{v} \equiv \mathbf{pack}\ \langle \tau_Y, \mathbf{v}' \rangle\ \mathbf{as\ Univ}$ *and for all* $\tau_X, P_X \in \mathcal{P}(\llbracket \tau_X \rrbracket)$, $\mathbf{v}_X \in P_X$ *and* $\mathbf{v}_Y \in P_Y$, *there exist* $\mathbf{v}'_X$ *and* $\mathbf{v}'_Y$ *such that* $(\mathbf{v}'\ \tau_X).1\ \mathbf{v}_X \hookrightarrow^* \mathbf{v}'_Y$ *and* $(\mathbf{v}'\ \tau_X).2\ \mathbf{v}_Y \hookrightarrow^* \mathbf{v}'_X$ *and* $\mathbf{v}'_Y \in P_Y$ *and* $\mathbf{v}'_X \in P_X$.

This is not possible.

In Lemma 2.3, we assume some value $\mathbf{v}$ of type **Univ**. Additionally, we avoid formulating a general parametricity result for this calculus but instead just postulate that $\mathbf{v}$ satisfies its unary parametricity property (assuming a set semantics for closed types $\llbracket \cdot \rrbracket$). The lemma claims that these two assumptions lead to a contradiction, so that (if we believe unary parametricity), we can conclude that **Univ** is empty.

To prove this lemma, we use the same trick as we do in the more complex proof of degeneracy of **Univ** in $\lambda^{\mathbf{F}}$: we get a single P_Y, but we instantiate P_X twice with two different predicates:

Proof.
- Take $\tau_X = \mathbf{Unit}$, $P_X = \{\mathbf{unit}\}$ and $\mathbf{v}_X = \mathbf{unit} \in P_X$. The property above gives us a $\mathbf{v}'_Y$ such that $(\mathbf{v}'\ \mathbf{Unit}).1\ \mathbf{unit} \hookrightarrow^* \mathbf{v}'_Y$ and $\mathbf{v}'_Y \in P_Y$.
- Now take $\tau_X = \mathbf{Unit}$, $P'_X = \emptyset$ and take $\mathbf{v}_Y$ to be the $\mathbf{v}'_Y \in P_Y$ from the previous point. The property above gives us a $\mathbf{v}'_X$ such that $(\mathbf{v}'\ \mathbf{Unit}).2\ \mathbf{v}'_Y \hookrightarrow^* \mathbf{v}'_X$ and $\mathbf{v}'_X \in P'_X$.
- Contradiction: $\mathbf{v}'_X \in P'_X$ but $P'_X = \emptyset$. □

[2] Unary and n-ary variants of parametricity also exist, though the relational one is the most commonly studied.

[3] Note that the situation is entirely different if we swap the quantifications in the type: $\mathbf{Triv} \stackrel{\text{def}}{=} \forall \mathbf{X}.\, \exists \mathbf{Y}.\, (\mathbf{X} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{X})$.

Essentially, this proof exploits the intuitive argument that we outlined above: a value of type $\mathbf{Univ}$ needs to provide a single interpretation for the existentially quantified $\mathbf{Y}$ that must not depend on the choice of $\mathbf{X}$. However, rather than applying this argument to the choice of types τ_Y and τ_X, it exploits parametricity by applying the argument to the choice of predicate interpretations P_Y and P_X.

2.4 Two Contextually Equivalent Terms

This degeneracy of $\mathbf{Univ}$ implies the contextual equivalence of the following two terms of type $\mathbf{Univ} \to \mathbf{Unit}$.

$$\mathbf{t_u} \stackrel{\text{def}}{=} \lambda \mathbf{x} : \mathbf{Univ}.\ \mathbf{unpack\ x\ as}\ \langle \mathbf{Y}, \mathbf{x}' \rangle\ \mathbf{in}$$
$$\mathbf{let\ x}'' : (\mathbf{Unit} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{Unit}) = \mathbf{x}'\ \mathbf{Unit\ in\ x}''.2\ (\mathbf{x}''.1\ \mathbf{unit})$$
$$\mathbf{t}'_\mathbf{d} \stackrel{\text{def}}{=} \lambda \mathbf{x} : \mathbf{Univ}.\ \omega_{\mathbf{Unit}}$$

The reason that $\mathbf{t_u}$ and $\mathbf{t}'_\mathbf{d}$ are contextually equivalent is that both will diverge when applied to any argument of type $\mathbf{Univ}$. For $\mathbf{t_u}$, this follows from the degeneracy of the type $\mathbf{Univ}$, as discussed above, while for $\mathbf{t}'_\mathbf{d}$, the term $\omega_{\mathbf{Unit}}$ in the body ensures divergence. Note that the degeneracy of $\mathbf{Univ}$ is essential: if the context were able to produce a non-degenerate value of type $\mathbf{Univ}$, then $\mathbf{t_u}$ would not diverge when applied to it, so that the context could distinguish $\mathbf{t_u}$ from $\mathbf{t}'_\mathbf{d}$.

For technical reasons, our proofs will not actually use the term $\mathbf{t}'_\mathbf{d}$ but rather the following term $\mathbf{t_d}$, which first uses the value of type $\mathbf{Univ}$ in the same way as $\mathbf{t_u}$, but then diverges afterwards.

$$\mathbf{t_d} \stackrel{\text{def}}{=} \lambda \mathbf{x} : \mathbf{Univ}.\ \mathbf{unpack\ x\ as}\ \langle \mathbf{Y}, \mathbf{x}' \rangle\ \mathbf{in}$$
$$\mathbf{let\ x}'' : (\mathbf{Unit} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{Unit}) = \mathbf{x}'\ \mathbf{Unit\ in}\ ((\mathbf{x}''.2\ (\mathbf{x}''.1\ \mathbf{unit})); \omega_{\mathbf{Unit}})$$

The only reason we do this is because it is easier to prove that $\mathbf{t_u} \simeq_{ctx} \mathbf{t_d}$ than to do the same for $\mathbf{t}'_\mathbf{d}$ (it seems like a proof for $\mathbf{t}'_\mathbf{d}$ would require both a unary and binary logical relation).

We now have the following theorem. A full proof of this result can be found in the technical appendix [4], but we provide an overview here.

Theorem 2.4 ($\mathbf{t_u}$ and $\mathbf{t_d}$ are contextually equivalent in λ^F). $\emptyset; \emptyset \vdash \mathbf{t_u} \simeq_{ctx} \mathbf{t_d} : \mathbf{Univ} \to \mathbf{Unit}$.

Proof Sketch. This proof uses a pre-existing formulation of parametricity for λ^F by Dreyer et al. [2009]. The proof essentially applies the same idea as the above proof for Lemma 2.3: we use parametricity for the argument of type $\mathbf{Univ}$, from which we obtain a single relational interpretation for the existentially quantified type $\mathbf{Y}$. We can then use parametricity with two different relational interpretations for the universally quantified type $\mathbf{X}$, and we end up with two terms related in the expression relation for an empty relation. From this, we can conclude that both must diverge. □

3 DISPROVING THE SUMII-PIERCE CONJECTURE

The first technical result of this paper is that we disprove a conjecture by Pierce and Sumii [2000] about the dynamic enforcement of parametricity in a cryptographic calculus.

Sumii and Pierce's conjecture is about a compiler from λ^F to an untyped lambda calculus with sealing (idealised encryption) called λ^σ. In this section, we first introduce the target language λ^σ (Section 3.1) and the compiler from λ^F to λ^σ (Section 3.2). We then prove that the compiler is not fully-abstract: there exist two terms that are contextually-equivalent in λ^F but whose compilation is inequivalent in λ^σ (Section 3.3).

[4] The technical appendix is available as auxiliary material for this publication through the ACM Digital Library.

Syntax:

$$t ::= v \mid x \mid t\ t \mid t.1 \mid t.2 \mid \langle t, t\rangle \mid \text{if } t \text{ then } t \text{ else } t \mid \nu x.t \mid \{t\}_t \mid \sigma \mid \text{let } \{x\}_t = t \text{ in } t \text{ else } t \mid \text{roll } t \mid \text{unroll } t \mid \text{wrong}$$

$$v ::= \text{unit} \mid \text{true} \mid \text{false} \mid \lambda x.\, t \mid \langle v, v\rangle \mid \{v\}_\sigma \mid \sigma \mid \text{roll } v$$

Evaluation rules (excerpt):

$$\frac{\sigma \notin \text{dom}(h)}{(h, \nu x.t) \hookrightarrow_0 (h; \sigma, t[\sigma/x])} \qquad \frac{\sigma \equiv \sigma'}{\text{let } \{x\}_\sigma = \{v\}_{\sigma'} \text{ in } t \text{ else } t' \hookrightarrow_0 t[v/x]}$$

$$\frac{\sigma \not\equiv \sigma'}{\text{let } \{x\}_\sigma = \{v\}_{\sigma'} \text{ in } t \text{ else } t' \hookrightarrow_0 t'} \qquad \frac{\nexists v', \sigma'.\, v \equiv \{v'\}_{\sigma'}}{\text{let } \{x\}_\sigma = v \text{ in } t \text{ else } t' \hookrightarrow_0 \text{wrong}}$$

$$\frac{\nexists \sigma.\, v \equiv \sigma}{\text{let } \{x\}_v = v' \text{ in } t \text{ else } t' \hookrightarrow_0 \text{wrong}} \qquad \frac{\nexists \sigma.\, v' \equiv \sigma}{\{v\}_{v'} \hookrightarrow_0 \text{wrong}} \qquad \frac{t \hookrightarrow_0 t'}{(h, \mathbb{E}[t]) \hookrightarrow (h, \mathbb{E}[t'])}$$

Fig. 2. λ^σ syntax and evaluation rules (excerpt). A λ^σ program state is a pair $h; t$ where h is the list of allocated seals. The semantics relation $\hookrightarrow$ relies on the beta reductions indicated as $\hookrightarrow_0$, which do not require a list of allocated seals to reduce.

Remark. Sumii and Pierce have presented both a typed [Pierce and Sumii 2000] as well as an untyped [Sumii and Pierce 2004] version of λ^σ. We use the untyped version because it is quite a bit simpler.[5] However, the typed version suffers from the same problem, as we show in detail in the technical report. Essentially, both settings break degeneracy of **Univ** because they feature a universal type: the unitype of all values in the untyped target language and the type bits of ciphertexts produced by encryption (sealing) in the typed target language.

3.1 The Cryptographic Lambda Calculus λ^σ

λ^σ (Figure 2) is an untyped λ-calculus, extended with *sealing*, which models a *dynamic* protection mechanism such as (idealized) symmetric encryption [Sumii and Pierce 2004].

Most syntactic constructs are standard for an untyped λ-calculus. The term wrong models run-time errors and is a stuck term. Sealing introduces four new syntactic constructs in the calculus: $\nu x.t$ creates a fresh seal (symmetric encryption key) and then evaluates t with x bound to the newly created seal. There is no surface syntax for seals, but the internal syntax σ represents run-time seal values created with $\nu x.t$. The construct $\{t_1\}_{t_2}$ first evaluates t_1 and t_2 to values v_1 and v_2 and then creates the sealed value $\{v_1\}_{v_2}$ (or leads to a run-time error if v_2 is not a seal). One can think of such a sealed value as v_1 encrypted under v_2. The final construct $\text{let } \{x\}_{t_1} = t_2 \text{ in } t_3 \text{ else } t_4$ is for unsealing or decrypting. It first evaluates t_1 to a seal σ_1 and t_2 to $\{v_2\}_{\sigma_2}$ (or produces a run-time error if either result is not of that form). If σ_1 and σ_2 are equal, t_3 is evaluated with x bound to the decrypted value v_2, otherwise t_4 is evaluated.

Program contexts in λ^σ are defined as for λ^F and are denoted with $\mathfrak{C}$. Contextual equivalence in λ^σ, indicated with $\simeq_{ctx}$ is defined analogously to Definition 2.1. Note that the quantified contexts

[5] The extra complexity in the typed version comes from the difficulty of erasing a polymorphic function to a simply typed language, and from the fact that the compiler protects all type variables in a separate pass rather than using a single pass for all type variables.

are not allowed to contain literal seals (as they are internal syntax), but they are allowed to allocate and use fresh seals of their own.

Sealing is the main information hiding mechanism in λ^σ: by creating a new seal σ and making sure it does not leak to the context, a term can create values $\{v\}_\sigma$ that are opaque to the context. Pierce and Sumii [2000] explain how one can use this information hiding mechanism to implement a protection similar to that offered by parametric polymorphism. For instance, the following implementation of $\mathbb{Z}_3$ from Example 2.2 in terms of pairs of booleans, uses sealing to protect the abstraction.

Example 3.1 ($\mathbb{Z}_3$ in λ^σ). First, we introduce two helper functions:

$$\mathsf{seal}_\sigma \stackrel{\text{def}}{=} \lambda y.\, \{y\}_\sigma \qquad\qquad \mathsf{unseal}_\sigma \stackrel{\text{def}}{=} \lambda y.\, \mathsf{let}\ \{x\}_\sigma = y\ \mathsf{in}\ x\ \mathsf{else}\ \mathsf{wrong}$$

Then, we can define a λ^σ correspondent of **z3** as:

$$\mathsf{z3} = \nu s.\langle\langle \mathsf{zero}, \mathsf{succ}\rangle, \mathsf{zero?}\rangle$$
$$\text{where} \begin{cases} \mathsf{zero} = \mathsf{seal}_s\ \langle \mathsf{false}, \mathsf{false}\rangle \\ \mathsf{succ} = \lambda p.\ \mathsf{if}\ (\mathsf{unseal}_s\ p).2\ \mathsf{then}\ \mathsf{seal}_s\ \langle \mathsf{false}, \mathsf{false}\rangle\ \mathsf{else} \\ \qquad\qquad \mathsf{if}\ (\mathsf{unseal}_s\ p).1\ \mathsf{then}\ \mathsf{seal}_s\ \langle \mathsf{false}, \mathsf{true}\rangle\ \mathsf{else}\ \mathsf{seal}_s\ \langle \mathsf{true}, \mathsf{false}\rangle \\ \mathsf{zero?} = \lambda p.\ \mathsf{if}\ (\mathsf{unseal}_s\ p).1\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{if}\ (\mathsf{unseal}_s p).2\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true} \end{cases}$$

Whenever values of the abstract type leave the scope of the abstract type definition, they are encrypted, and when they are passed back in they are decrypted before use. Intuitively, one can see that this protects against a context looking into or tampering with representation values, similarly to the protection offered by type checking of parametric polymorphism.

Also the protection required for the dual case, where a term calls a universally quantified function provided by the context, can be implemented in λ^σ. Consider for instance the $\lambda^{\mathbf{F}}$ term:

$$\lambda \mathbf{f} : \forall \mathbf{X}.\, \mathbf{X} \times \mathbf{X} \to \mathbf{X}.\, \mathbf{if}\ \mathbf{f}\ \mathbf{Bool}\ \langle \mathbf{true}, \mathbf{true}\rangle\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{false}$$

The polymorphic function **f** that will be passed in by the context, can only return one of its arguments (or diverge), and hence the invocation in the term above will necessarily return **true** (or diverge).

We can implement a similar protection in λ^σ. To enforce parametric behavior of a polymorphic function received from the context, we create a new seal for every invocation, and we encrypt all parameters of the quantified type with that seal. For instance, the term above could be implemented in λ^σ as follows:

$$\lambda f.\ \mathsf{if}\ \nu s.\mathsf{unseal}_s(f\ \langle \mathsf{seal}_s\ \mathsf{true}, \mathsf{seal}_s\ \mathsf{true}\rangle)\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}$$

Before calling f, its arguments are encrypted with a new seal, and the return value gets decrypted with that seal, hence all that f can do is either return one of its arguments or diverge (doing anything else would lead to a run-time error).

Again, this gives us a *dynamic* guarantee that a function has to treat its arguments opaquely, where parametric polymorphism gives us this guarantee *statically* by type checking. This technique of dynamically protecting values with appropriately scoped seals is the essence of the idea behind Sumii and Pierce's compiler for $\lambda^{\mathbf{F}}$.

3.2 Sumii and Pierce's Compiler

The compiler $[\![\cdot]\!]^{\lambda^F}_{\lambda^\sigma}$ first performs standard type erasure (function $\mathtt{erase}(\cdot)$) and then wraps it with a dynamic check ($\mathsf{protect}_{\eta;\tau}$) that will insert dynamic applications of sealing and unsealing:

$$\text{if } \emptyset;\emptyset \vdash \mathbf{t} : \tau, \text{ then } [\![\mathbf{t}]\!]^{\lambda^F}_{\lambda^\sigma} \stackrel{\text{def}}{=} \mathsf{let}\ \mathsf{x} = \mathtt{erase}(\mathbf{t})\ \mathsf{in}\ \mathsf{protect}_{\emptyset;\tau}\ \mathsf{x}$$

Note that we restrict the compiler to closed terms here. Lifting this limitation is quite straightforward, as presented by Devriese et al. [2017].

We now present these steps and discuss their meaning. These definitions are taken from Sumii and Pierce [2004], except that we add support for recursive types (see below), and that we make some notational changes and some minor technical changes.

Type erasure. This pass is mostly straightforward, the only non-trivial aspect is how to deal with the quantified types. Type abstraction and application are erased to a dummy lambda abstraction and an application to a unit parameter, and unpacking is erased to a let-binding.[6]

$$\mathtt{erase}(\mathbf{x}) \stackrel{\text{def}}{=} \mathsf{x} \qquad \mathtt{erase}(\mathbf{t}\ \tau') \stackrel{\text{def}}{=} \mathtt{erase}(\mathbf{t})\ \mathsf{unit}$$

$$\mathtt{erase}(\mathbf{\Lambda X.\, t}) \stackrel{\text{def}}{=} \lambda\_.\, \mathtt{erase}(\mathbf{t}) \qquad \mathtt{erase}(\mathbf{pack}\ \langle \tau', \mathbf{t}\rangle\ \mathbf{as}\ \exists \mathbf{X}.\, \tau) \stackrel{\text{def}}{=} \mathtt{erase}(\mathbf{t})$$

$$\mathtt{erase}(\lambda \mathbf{x} : \tau.\, \mathbf{t}) \stackrel{\text{def}}{=} \lambda \mathsf{x}.\, \mathtt{erase}(\mathbf{t}) \qquad \mathtt{erase}(\mathbf{unpack\ t\ as}\ \langle \mathbf{X}, \mathbf{x}\rangle\ \mathbf{in\ t'}) \stackrel{\text{def}}{=} \mathsf{let\ x} = \mathtt{erase}(\mathbf{t})\ \mathsf{in}\ \mathtt{erase}(\mathbf{t'})$$

Dynamic wrappers. The second phase of the compiler wraps compiled terms with dynamic wrappers. These are formalised as the function $\mathsf{protect}_{\eta;\tau}$, which is defined in Figure 3, together with its dual $\mathsf{confine}_{\eta;\tau}$ by mutual induction on τ. We use the names protect and confine (following Devriese et al. [2016]) to refer to the wrappers that Sumii and Pierce call $\mathcal{E}^+$ and $\mathcal{E}^-$ respectively [Sumii and Pierce 2004].

Intuitively, applying $\mathsf{protect}_{\eta;\tau}$ to a value v ensures that v cannot be used in ways that are not allowed by type τ. Dually, applying $\mathsf{confine}_{\eta;\tau}$ to a value v prevents v from behaving in a way that is not allowed by type τ. For any free type variables $\mathbf{X}$ in τ, η tells us how to protect/confine values of type $\mathbf{X}$. Concretely, $\eta(\mathbf{X}) = (\mathsf{t_p}, \mathsf{t_c})$ where $\mathsf{t_p}$ and $\mathsf{t_c}$ are untyped terms that should be applied to protect/confine (respectively) values of type $\mathbf{X}$. Formally, η has the following syntax: $\eta ::= \emptyset \mid \eta, \mathbf{X} \mapsto (\mathsf{t}, \mathsf{t})$.

For defining $\mathsf{protect}_;$ and $\mathsf{confine}_;$ we rely on the following seal_σ and unseal_σ functions, which seal and unseal values with a given seal σ.

$$\mathsf{seal}_\sigma \stackrel{\text{def}}{=} \lambda \mathsf{y}.\, \{\mathsf{y}\}_\sigma \qquad \mathsf{unseal}_\sigma \stackrel{\text{def}}{=} \lambda \mathsf{y}.\, \mathsf{let}\ \{\mathsf{x}\}_\sigma = \mathsf{y}\ \mathsf{in}\ \mathsf{x}\ \mathsf{else}\ \mathsf{wrong}$$

Protecting at a function type confines the argument and protects the result with the same polarity, and similarly for confining at a function type. Confining at ground type inserts a dynamic check that the confined value is indeed of the expected type (unit or true/false). If any of these checks fail, the term reduces to wrong. Protecting at a ground type does nothing, because there is no way for the context to use such values that is not allowed by the type.

Protecting at type $\forall \mathbf{X}.\, \tau$ does nothing but forward the dummy unit application and recursively protect at type τ. This is because there is nothing to protect: intuitively, the context cannot use a term of type $\forall \mathbf{X}.\, \tau$ in a way that is not allowed by the type. Similarly, confining at an existential type $\exists \mathbf{X}.\, \tau$ just recurses over type τ without doing anything special for values of type $\mathbf{X}$, because there is intuitively no way for a value of type $\exists \mathbf{X}.\, \tau$ to behave that is not allowed by the type.

Finally, when protecting a term of type $\exists \mathbf{X}.\, \tau$, we want to make sure that the context treats the type $\mathbf{X}$ opaquely, so values of type $\mathbf{X}$ are sealed (encrypted) with a fresh seal σ. Similarly, confining

[6] We are assuming a standard desugaring of let expressions to function applications.

$$\begin{aligned}
\mathsf{protect}_{\eta;\mathbf{Unit}}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{x}\\
\mathsf{protect}_{\eta;\mathbf{Bool}}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{x}\\
\mathsf{protect}_{\eta;\tau_1\times\tau_2}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{let}\ \mathsf{x}_1 = \mathsf{x}.1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2 = \mathsf{x}.2\ \mathsf{in}\ \langle \mathsf{protect}_{\eta;\tau_1}\ \mathsf{x}_1, \mathsf{protect}_{\eta;\tau_2}\ \mathsf{x}_2\rangle\\
\mathsf{protect}_{\eta;\tau_1\to\tau_2}\ \mathsf{x} &\stackrel{\text{def}}{=} \lambda\mathsf{y}.\ \mathsf{let}\ \mathsf{z} = \mathsf{x}\ (\mathsf{confine}_{\eta;\tau_1}\ \mathsf{y})\ \mathsf{in}\ \mathsf{protect}_{\eta;\tau_2}\ \mathsf{z}\\
\mathsf{protect}_{\eta;\forall\mathbf{X}.\tau}\ \mathsf{x} &\stackrel{\text{def}}{=} \lambda_.\ \mathsf{let}\ \mathsf{y} = \mathsf{x}\ \mathsf{unit}\ \mathsf{in}\ \mathsf{protect}_{\eta,\mathbf{X}\mapsto(\lambda\mathsf{x}.\mathsf{x},\lambda\mathsf{x}.\mathsf{x});\tau}\ \mathsf{y}\\
\mathsf{protect}_{\eta;\exists\mathbf{X}.\tau}\ \mathsf{x} &\stackrel{\text{def}}{=} \nu\mathsf{s}.\ \mathsf{protect}_{\eta,\mathbf{X}\mapsto(\mathsf{seal}_\mathsf{s},\mathsf{unseal}_\mathsf{s});\tau}\ \mathsf{x}\\
\mathsf{protect}_{\eta;\mu\mathbf{X}.\tau}\ \mathsf{x} &\stackrel{\text{def}}{=} (\mathsf{fix}_2\ (\lambda\mathsf{rec}.\ \langle\lambda\mathsf{y}.\ \mathsf{protect}_{\eta,\mathbf{X}\mapsto(\mathsf{rec}.1,\mathsf{rec}.2);\tau}\ \mathsf{y}, \lambda\mathsf{y}.\ \mathsf{confine}_{\eta,\mathbf{X}\mapsto(\mathsf{rec}.1,\mathsf{rec}.2);\tau}\ \mathsf{y}\rangle).1\ \mathsf{x}\\
\mathsf{protect}_{\eta;\mathbf{X}}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{t}_\mathsf{p}\ \mathsf{x} \qquad \text{where } \eta(\mathbf{X}) = (\mathsf{t}_\mathsf{p}, _)\\
\\
\mathsf{confine}_{\eta;\mathbf{Unit}}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{x};\ \mathsf{unit}\\
\mathsf{confine}_{\eta;\mathbf{Bool}}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\\
\mathsf{confine}_{\eta;\tau_1\times\tau_2}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{let}\ \mathsf{x}_1 = \mathsf{x}.1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2 = \mathsf{x}.2\ \mathsf{in}\ \langle \mathsf{confine}_{\eta;\tau_1}\ \mathsf{x}_1, \mathsf{confine}_{\eta;\tau_2}\ \mathsf{x}_2\rangle\\
\mathsf{confine}_{\eta;\tau_1\to\tau_2}\ \mathsf{x} &\stackrel{\text{def}}{=} \lambda\mathsf{y}.\ \mathsf{let}\ \mathsf{z} = \mathsf{x}\ (\mathsf{protect}_{\eta;\tau_1}\ \mathsf{y})\ \mathsf{in}\ \mathsf{confine}_{\eta;\tau_2}\ \mathsf{z}\\
\mathsf{confine}_{\eta;\forall\mathbf{X}.\tau}\ \mathsf{x} &\stackrel{\text{def}}{=} \lambda_.\ \nu\mathsf{s}.\ \mathsf{let}\ \mathsf{x}' = \mathsf{x}\ \mathsf{unit}\ \mathsf{in}\ \mathsf{confine}_{\eta,\mathbf{X}\mapsto(\mathsf{seal}_\mathsf{s},\mathsf{unseal}_\mathsf{s});\tau}\ \mathsf{x}'\\
\mathsf{confine}_{\eta;\exists\mathbf{X}.\tau}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{confine}_{\eta,\mathbf{X}\mapsto(\lambda\mathsf{x}.\mathsf{x},\lambda\mathsf{x}.\mathsf{x});\tau}\ \mathsf{x}\\
\mathsf{confine}_{\eta;\mu\mathbf{X}.\tau}\ \mathsf{x} &\stackrel{\text{def}}{=} (\mathsf{fix}_2\ (\lambda\mathsf{rec}.\ \langle\lambda\mathsf{y}.\ \mathsf{protect}_{\eta,\mathbf{X}\mapsto(\mathsf{rec}.1,\mathsf{rec}.2);\tau}\ \mathsf{y}, \lambda\mathsf{y}.\ \mathsf{confine}_{\eta,\mathbf{X}\mapsto(\mathsf{rec}.1,\mathsf{rec}.2);\tau}\ \mathsf{y}\rangle)).2\ \mathsf{x}\\
\mathsf{confine}_{\eta;\mathbf{X}}\ \mathsf{x} &\stackrel{\text{def}}{=} \mathsf{t}_\mathsf{c}\ \mathsf{x} \qquad \text{where } \eta(\mathbf{X}) = (_, \mathsf{t}_\mathsf{c})
\end{aligned}$$

Fig. 3. The dynamic wrappers of Sumii and Pierce's compiler. fix_2 is presented in Eq. (2) p. 12.

at type $\forall\mathbf{X}.\ \tau$ generates a fresh seal for every invocation to protect values of type $\mathbf{X}$ with. This is the idea we have explained before in Section 3.1.

Applying the compiler to $\mathbf{z3}$ from Example 3.1 results in the λ^σ term z3 from Example 2.2 (modulo some additional β-reductions).

Technical remark. Sumii and Pierce's conjectured compiler for untyped λ^σ already had a technical flaw, albeit unrelated to parametricity. In fact, they had a terminating source language $\lambda^{\mathbf{F}}_{\Downarrow}$ and an untyped – thus possibly diverging – target language. With this discrepancy it is impossible to build a fully-abstract compiler for a simple reason. Consider these two terms, which are equivalent in $\lambda^{\mathbf{F}}_{\Downarrow}$ (but not in $\lambda^{\mathbf{F}}$):

$$\lambda\mathbf{x} : \mathbf{Unit} \to \mathbf{Unit}.\ \mathbf{let}\ \_ = \mathbf{x}\ \mathbf{unit}\ \mathbf{in}\ \mathbf{unit} \qquad\qquad \lambda\mathbf{x} : \mathbf{Unit} \to \mathbf{Unit}.\ \mathbf{unit}$$

Their compilation to λ^σ should intuitively be (modulo $\mathsf{protect}_;$, which we elide for simplicity):

$$\lambda\mathsf{x}.\ \mathsf{let}\ \_ = \mathsf{x}\ \mathsf{unit}\ \mathsf{in}\ \mathsf{unit} \qquad\qquad \lambda\mathsf{x}.\ \mathsf{unit}$$

However, a target context can distinguish them by passing them a term that once applied, diverges:

$$\mathfrak{C} \stackrel{\text{def}}{=} [\cdot]\ (\lambda\mathsf{y}.\ \omega)$$

Dealing with recursive types. We avoid this problem by extending the compiler to the recursive types that we have in our non-terminating version of λ^{F}. Essentially, for recursive types $\mu \mathrm{X}.\, \tau$, protect and confine can naturally be defined as a mutually recursive pair of functions, i.e. recursively in terms of themselves *and* each other. The definition in Fig. 3 achieves this using a fix_2 function, which constructs the fixpoint of a function mapping a pair of functions to a pair of functions.

We don't go into detail for this, but we can define fix_2 as follows, using a definition of Plotkin's Z combinator[7] as fix:

$$\mathsf{fix} \stackrel{\text{def}}{=} \lambda \mathsf{f}.(\lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y}))\ (\lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y})) \tag{1}$$

$$\mathsf{fix}_2 \stackrel{\text{def}}{=} \lambda \mathsf{f}.\ \mathsf{fix}\ (\lambda \mathsf{rec}.\ \mathsf{f}\ \langle \lambda \mathsf{x}.\ (\mathsf{rec}\ \mathsf{unit}).1\ \mathsf{x}, \lambda \mathsf{x}.\ (\mathsf{rec}\ \mathsf{unit}).2\ \mathsf{x} \rangle)\ \mathsf{unit} \tag{2}$$

It is worth noting at this point that recursive types are not essential for our counterexample. In fact, instead of recursive types, we could just as well have added a fixpoint primitive to make System F non-terminating and our counterexample would continue to apply without modification.

3.3 Disproving the Sumii-Pierce Conjecture

Sumii and Pierce conjectured that their compiler $[\![\cdot]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$ is fully abstract. In other words, two λ^{F} terms are contextually equivalent if and only if they are compiled to equivalent λ^{σ} terms.

Conjecture 3.2 (Sumii and Pierce). $\emptyset; \emptyset \vdash \mathbf{t}_1 \simeq_{ctx} \mathbf{t}_2 : \tau$ *if and only if* $\emptyset \vdash [\![\mathbf{t}_1]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}} \simeq_{ctx} [\![\mathbf{t}_2]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$

However, we can now prove that this conjecture is false. A counterexample is given by the terms $\mathbf{t_u}$ and $\mathbf{t_d}$ which we proved contextually equivalent in Theorem 2.4. Compiling $\mathbf{t_u}$ and $\mathbf{t_d}$ does not produce contextually equivalent λ^{σ} terms:

Theorem 3.3 ($\mathbf{t_u}$ and $\mathbf{t_d}$ are not equivalent after compilation to λ^{σ}). $\emptyset \vdash [\![\mathbf{t_u}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}} \not\simeq_{ctx} [\![\mathbf{t_d}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$

Proof Sketch. A full proof with all details can be found in the technical report.

The terms $[\![\mathbf{t_u}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$ and $[\![\mathbf{t_d}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$ can be discriminated by the following context:

$$\mathfrak{C} \stackrel{\text{def}}{=} [\cdot]\ (\lambda \_.\ \langle \lambda \mathsf{x}.\ \mathsf{x}, \lambda \mathsf{x}.\ \mathsf{x} \rangle)$$

To understand the role of this context, recall that the contextual equivalence of $\mathbf{t_u}$ and $\mathbf{t_d}$ relies on the degeneracy of type **Univ**. The context $\mathfrak{C}$ breaks this assumption by invoking the terms with a non-degenerate value of type **Univ**, which is constructed by using the unitype of all untyped values as a universal type.

By unfolding definitions and executing the operational semantics, it is easy to check that we get the following behavior.

$$\mathfrak{C}\left[[\![\mathbf{t_u}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}\right] \hookrightarrow^{*} \mathsf{unit} \qquad \mathfrak{C}\left[[\![\mathbf{t_d}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}\right] \hookrightarrow^{*} (\lambda \mathsf{r}.\ \mathsf{r})\ \omega \hookrightarrow^{*} (\lambda \mathsf{r}.\ \mathsf{r})\ \omega \hookrightarrow^{*} \cdots \Uparrow$$

We spell out the reductions in full detail in the technical appendix. From this behavior, it follows immediately that the terms are not contextually equivalent. □

We can thus prove that Sumii and Pierce's conjecture is false as follows.

Theorem 3.4 ($[\![\cdot]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$ is not fully abstract). *It is not true that*

$$\forall \mathbf{t}_1, \mathbf{t}_2.\mathbf{t}_1 \simeq_{ctx} \mathbf{t}_2 \iff [\![\mathbf{t}_1]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}} \simeq_{ctx} [\![\mathbf{t}_2]\!]^{\lambda^{\mathrm{F}}}_{\lambda^{\sigma}}$$

Proof. Follows easily from Theorem 2.4 and the counterexample in Theorem 3.3. □

[7] Plotkin's Z combinator is a variant of the more well-known Y combinator that works in a call-by-value setting, but is restricted to constructing fixpoints that are functions, see Pierce [2002, §5.2].

The problem is in the easy case. It is worth noticing that most of the work in Sumii and Pierce's compiler (see Fig. 3) is in enforcing that existentially quantified types passed to the context are treated opaquely and, dually, that polymorphic functions received from the context are forced to treat their argument type opaquely. However, it is not in these cases that the counterexample highlights a problem.

Instead, it goes wrong in the cases where we receive an existential type from the context (and dually when we pass a polymorphic function to the context). For these seemingly simple cases, the dynamic wrappers from Fig. 3 do not perform any specific kind of enforcement (except for recursing on their body type). However, it is there that our counterexample uncovers a problem: the value that it provides as a value of $\mathbf{Univ} = \exists \mathbf{Y}.\, \forall \mathbf{X}.\, (\mathbf{X} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{X})$ does not correspond to any legal choice of $\mathbf{Y}$, but the dynamic type wrappers have no way to detect this. In fact, an alternative way to understand what goes wrong is that the value $\lambda\_.\, \langle \lambda \mathrm{x}.\, \mathrm{x}, \lambda \mathrm{x}.\, \mathrm{x} \rangle$ provided by the context, behaves as if it can choose $\mathbf{Y}$ equal to $\mathbf{X}$. However, this is not possible in λ^{F} because $\mathbf{X}$ is not in scope at the moment when $\mathbf{Y}$ needs to be chosen.

In other words, what seems to be missing in Sumii and Pierce's dynamic enforcement is an enforcement of the type variable scope of existentially quantified types. To see this, consider the following type Triv, which is identical to **Univ**, except for the order of the quantifiers:

$$\mathbf{Univ} \stackrel{\text{def}}{=} \exists \mathbf{Y}.\, \forall \mathbf{X}.\, (\mathbf{X} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{X}) \qquad \mathrm{Triv} \stackrel{\text{def}}{=} \forall \mathbf{X}.\, \exists \mathbf{Y}.\, (\mathbf{X} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{X})$$

Even though the type Triv is more liberal, as it allows to instantiate $\mathbf{Y}$ with $\mathbf{X}$, the wrappers of Fig. 3 treat the types essentially the same. From this perspective, it is no surprise that the value $\lambda\_.\, \langle \lambda \mathrm{x}.\, \mathrm{x}, \lambda \mathrm{x}.\, \mathrm{x} \rangle$ is accepted as a value of type **Univ**, because it does in fact correspond to a legal λ^{F} value of type Triv:

$$\mathbf{\Lambda X}.\, \mathbf{pack}\, \langle \mathbf{X}, \langle \lambda \mathbf{x} : \mathbf{X}.\, \mathbf{x}, \lambda \mathbf{x} : \mathbf{X}.\, \mathbf{x} \rangle \rangle\, \mathbf{as}\, \exists \mathbf{Y}.\, (\mathbf{X} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{X})$$

4 SYSTEM F EQUIVALENCES VS. GRADUAL TYPES

The second technical result in this paper is related to parametric polymorphism in gradually-typed languages. As explained in the introduction, gradually-typed languages provide a path for migrating codebases of untyped code to typed code. From this high-level idea, a number of natural design goals follow and the literature contains a number of correctness properties that formally express these objectives.

First, gradual languages are intended to preserve the semantics of existing typed and untyped code. Additionally, turning untyped programs into typed ones by adding correct (!) type signatures should not modify the semantics of programs. Without going into detail (because it is not relevant for our discussion), Siek et al. [2015] formalise these objectives as a number of formal criteria for gradually-typed languages.

Another high-level design goal of gradually-typed languages is that the typed components continue to enjoy the benefits of well-typedness in the presence of untyped other components. Wadler and Findler [2009] have proposed the Blame Theorem that expresses this property when it comes to one such benefit: the absence of type errors at runtime. Gradual languages rely on dynamic casts (which can fail at runtime) to coerce untyped values into typed ones. Wadler and Findler add a notion of blame, which is essentially a way to identify the type cast that caused a runtime type error. The Blame Theorem then expresses that well-typed components are never to blame for such failures, i.e. the property that well-typed components never cause runtime type errors remains valid in the gradual language.

For our purposes, we are interested in another benefit of well-typedness that has received less attention in the literature on gradual types: the benefits that well-typedness provides in terms of

Syntax:

Terms	$t ::= v \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \mid x \mid t\ t \mid t\ \tau \mid t.1 \mid t.2 \mid \langle t, t\rangle$
	$\mid t : \tau \overset{p}{\Rightarrow} \tau \mid t : \tau \overset{\phi}{\Rightarrow} \tau \mid \mathit{blame}\ p \mid t; t$
Values	$v ::= \mathit{unit} \mid \mathit{true} \mid \mathit{false} \mid \lambda x : \tau.\, t \mid \Lambda X.\, v \mid \langle v, v\rangle$
	$\mid v : \tau \to \tau \overset{\phi}{\Rightarrow} \tau \to \tau \mid v : \forall X.\, \tau \overset{\phi}{\Rightarrow} \forall X.\, \tau \mid v : \tau \overset{\neg\alpha}{\Longrightarrow} \alpha$
	$\mid v : \tau \to \tau \overset{p}{\Rightarrow} \tau \to \tau \mid v : \tau \overset{p}{\Rightarrow} \forall X.\, \tau \mid v : \gamma \overset{p}{\Rightarrow} \star$
Types	$\tau ::= \mathit{Unit} \mid \mathit{Bool} \mid \tau \times \tau \mid \tau \to \tau \mid \forall X.\tau \mid X \mid \alpha \mid \star$
Ground types	$\gamma ::= \mathit{Unit} \mid \mathit{Bool} \mid \alpha \mid \star \times \star \mid \star \to \star$
Convertibility labels	$\phi ::= \alpha \mid \neg\alpha$
Compatibility labels	$p ::= l \mid \neg l$

Fig. 4. The polymorphic blame calculus [Ahmed et al. 2017]. Note: we have adapted notations, added the *Unit* type and removed the *int* type to align more closely with other calculi in this paper.

reasoning. Many type systems allow us to deduce properties of components from their types, for example, the function $\lambda \mathbf{x} : \mathbf{Unit}.\, \mathbf{x}$ is easily seen to be equivalent to $\lambda \mathbf{x} : \mathbf{Unit}.\, \mathbf{unit}$, based on their types. System F parametricity similarly allows us to deduce properties from programs' types.

It is natural to wonder whether gradual type systems preserve the validity of such reasoning in the original typed language. For this paper, specifically, we are interested in the question whether contextual equivalences between typed terms continue to hold when these terms are considered in the gradual language. Formally, there is an embedding of $\lambda^{\mathbf{F}}$ terms $\mathbf{t}$ into λ^{B} terms that we will denote as $\lfloor \mathbf{t} \rfloor$. The question then becomes: if $\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$, do we have that $\lfloor \mathbf{t_1} \rfloor \simeq_{ctx} \lfloor \mathbf{t_2} \rfloor$, or, in other words, is the embedding of the typed language into the gradual language fully abstract?

In this section, we answer this question negatively for the polymorphic blame calculus, which extends System F into a gradually typed language. We first present Ahmed et al.'s λ^{B}, a gradually-typed, polymorphic lambda-calculus (Section 4.1). Next, we reconsider $\mathbf{t_u}$ and $\mathbf{t_d}$ from Theorem 2.4 in λ^{B} and present a λ^{B} context that differentiates them (Section 4.2). This shows that the embedding of $\lambda^{\mathbf{F}}$ into λ^{B} is not fully abstract.

4.1 The λ^{B} Calculus [Ahmed et al. 2017]

There exist several versions of the polymorphic blame calculus in the literature [Ahmed et al. 2017, 2011a; Igarashi et al. 2017]. We take λ^{B} to be the polymorphic blame calculus as described by Ahmed et al. [2017]. This version modifies certain behavior that was "topsy turvy" in the original version [Ahmed et al. 2011a]: a peculiar operational semantics that performs evaluation under type and value abstractions and an ad hoc postponing of run-time type generation in certain situations.

The syntax of λ^{B} is presented in Fig. 4. The calculus contains all terms and types of $\lambda^{\mathbf{F}}$ except for existentials. However, we can use a standard encoding of existentials in terms of universals as follows [Pierce 2002, §24.3, pp. 377-379]:

$$\exists X.\, \tau \overset{\text{def}}{=} \forall Y.\, (\forall X.\, \tau \to Y) \to Y$$

$$\mathit{pack}\ \langle \tau', t\rangle\ \mathit{as}\ \exists X.\, \tau \overset{\text{def}}{=} \Lambda Y.\, \lambda f : (\forall X.\, \tau \to Y).\, f\ [\tau']\ t$$

$$\mathit{unpack}\ t_1\ \mathit{as}\ \langle X, x\rangle\ \mathit{in}\ t_2 \stackrel{\text{def}}{=} t_1\ [\tau_2]\ (\Lambda X.\ \lambda x : \tau_1.\ t_2) \qquad \text{where } t_1 : \exists X.\ \tau_1 \text{ and } t_2 : \tau_2$$

Even though existentials under this encoding are not entirely equivalent to regular existentials in our non-terminating calculus (because they contain a kind of bottom value), we can adapt our counterexample without trouble.

Then, λ^B contains a number of constructs to let typed and untyped code coexist. First we have the type $\star$: the type of untyped values. Untyped code can be typed with respect to the single, universal type $\star$. λ^B provides both a notion of casts ($t : \tau \stackrel{p}{\Rightarrow} \tau'$) and a notion of conversions ($t : \tau \stackrel{\phi}{\Rightarrow} \tau$).

Casts ($t : \tau \stackrel{p}{\Rightarrow} \tau'$) represent a form of dynamic casts from τ to τ' that can potentially fail at runtime (in which case *blame* p is raised). Casts can be used to inject types τ into the universal type $\star$ and also to extract those types out of $\star$ again. More generally, they can be used to convert between any types τ and τ' that satisfy a compatibility judgement $\Sigma; \Delta \vdash \tau \prec \tau'$, but we omit details for this because they are not relevant for our discussion. As part of the design that solves certain topsy-turvy aspects of the operational semantics in previous versions, λ^B carefully defines some casts as values: those where the term being cast is a value and the cast is either (1) between function types, (2) towards a polymorphic function type or (3) from a ground type γ into $\star$.

Polymorphic function application in λ^B does not use type substitution like in System F, but uses a notion of runtime type generation instead. Full details are not relevant for our discussion, but essentially, a polymorphic function application $(\Lambda X.\ \lambda x : X.\ x)\ \mathit{Unit}$ does not reduce to $\lambda x : \mathit{Unit}.\ x$ but to $\lambda x : \alpha.\ x$ for a fresh runtime type label α, where the assignment $\alpha := \mathit{Unit}$ is remembered in a type-name store Σ. Conversions ($t : \tau \stackrel{\phi}{\Rightarrow} \tau'$) represent a notion of static casts for converting between such a runtime type label α and the type that is assigned to it in the store Σ. More generally, a convertibility judgement $\Sigma; \Delta \vdash \tau \prec^{\phi} \tau'$ defines when τ can be legally converted into τ' by expanding ($+\alpha \in \phi$) or reducing ($-\alpha \in \phi$) runtime type labels' definitions.

4.2 Embedding of $\lambda^{\mathbf{F}}$ into λ^B is not Fully Abstract

To embed $\lambda^{\mathbf{F}}$ (defined in Section 2.1) into λ^B, most constructs can simply be mapped to the corresponding construct in λ^B. However, there is a discrepancy between type abstractions in λ^B and $\lambda^{\mathbf{F}}$. The difference is that λ^B uses value polymorphism: the bodies of type abstractions are required to be values. This choice has some desirable consequences, particularly that type abstractions and applications can be removed entirely during type erasure.

The difference is essentially orthogonal to the topics of this paper, but we are unable to standardise on using value polymorphism or not, because we use the logical relation by Dreyer et al. [2009] and the polymorphic blame calculus defined by Ahmed et al. [2017], which make different choices and would both be non-trivial to modify.

Therefore, we embed polymorphic functions from $\lambda^{\mathbf{F}}$ into λ^B by introducing a form of thunking for polymorphic functions: the type $\forall \mathbf{X}.\ \tau$ is mapped to $\lfloor \forall \mathbf{X}.\ \tau \rfloor \stackrel{\text{def}}{=} \forall X.\ \mathit{Unit} \to \lfloor \tau \rfloor$ and type abstractions $\mathbf{\Lambda X.\ t}$ are mapped to $\lfloor \mathbf{\Lambda X.\ t} \rfloor \stackrel{\text{def}}{=} \Lambda X.\ \lambda\_.\ \lfloor \mathbf{t} \rfloor$.

Our two contextually equivalent System F terms $\mathbf{t_u}$ and $\mathbf{t_d}$ embed into λ^B as $\lfloor \mathbf{t_u} \rfloor$ and $\lfloor \mathbf{t_d} \rfloor$. In this section, we show that they do not remain contextually equivalent, showing that the embedding is not fully abstract. Similarly to before, we can construct a λ^B context $\mathfrak{C}^B$ that differentiates them. As before, the context simply applies the terms to a non-degenerate value of type Univ, which we can construct thanks to the existence of the universal type $\star$:

$$\mathfrak{C}^B \stackrel{\text{def}}{=} [\cdot]\ (\mathit{pack}\ \langle \star, \Lambda X.\ \langle \lambda x : X.\ x : X \stackrel{p}{\Rightarrow} \star, \lambda x : \star.\ x : \star \stackrel{p'}{\Rightarrow} X \rangle\rangle\ \mathit{as}\ \mathit{Univ})$$

The constructed Univ value simply takes $\star$ as the existentially quantified universal type and uses casts to implement the functions from an arbitrary X into $\star$ and back.

Theorem 4.1 ($\lfloor \mathbf{t_u} \rfloor$ and $\lfloor \mathbf{t_d} \rfloor$ are not equivalent in λ^B). $\lfloor \mathbf{t_u} \rfloor \not\simeq_{ctx} \lfloor \mathbf{t_d} \rfloor$.

Proof. We have that $\mathfrak{C}^B\left[\lfloor \mathbf{t_u} \rfloor\right] \hookrightarrow^* \mathit{unit}$ while $\mathfrak{C}^B\left[\lfloor \mathbf{t_d} \rfloor\right] \Uparrow$. We have verified this on paper and using the interpreter provided by Jamner and Siek to support Ahmed et al. [2017]'s results.[8] [9] We provide the literal encoding of $\mathfrak{C}^B\left[\lfloor \mathbf{t_u} \rfloor\right]$ and $\mathfrak{C}^B\left[\lfloor \mathbf{t_d} \rfloor\right]$ for use in the interpreter in the technical appendix. □

Theorem 4.2 (Embedding $\lambda^{\mathbf{F}}$ into λ^B is not fully abstract). *It is not true that*

$$\forall \mathbf{t_1}, \mathbf{t_2}.\mathbf{t_1} \simeq_{ctx} \mathbf{t_2} \iff \lfloor \mathbf{t_1} \rfloor \simeq_{ctx} \lfloor \mathbf{t_2} \rfloor$$

Proof. Follows directly from Theorem 4.1 and Theorem 2.4. □

5 DISCUSSION

So Sumii and Pierce's compiler is not fully abstract and the polymorphic blame calculus breaks contextual equivalences in System F. But what to conclude from this? Should we start looking for alternative ways to dynamically enforce parametricity or were we wrong to hope for these properties to hold in the first place? In this section we present some thoughts on the different possible options.

First, it is interesting to investigate whether full abstraction could be recovered by fixing Sumii and Pierce's compiler or the polymorphic blame calculus. We discuss in the sections below some possible paths to explore, but we believe there are no straightforward solutions.

Second, another possible conclusion from our negative results is that full abstraction is perhaps too strong a property to aim for when doing secure compilation or gradual typing. We still think the Sumii-Pierce compiler is a useful secure compiler, even if it is not fully abstract. Hence we discuss how to weaken the requirements that full abstraction imposes on a translation from System F, by modifying System F in a way that weakens its contextual equivalences.

5.1 Fixing Sumii and Pierce's Compiler?

One of the attractive features of Sumii and Pierce's original conjecture is that it relies *only* on encryption (or at least, an idealised version of encryption in the form of seals). Because it required only encryption, this suggested that System F types could even be enforced as the contract for an untrusted adversary, running at the other end of a communication channel, on an untrusted computer. However, if we analyse the counterexample, this ambition of using just encryption seems hard to maintain.

Imagine that a compiled System F term (e.g. $\mathbf{t_u}$ or $\mathbf{t_d}$) is communicating with such an adversary over a communication channel and we want to enforce that the adversary respects the contract represented by System F type **Univ**. What happens is the following:

(1) The compiled term transmits the value **unit** of type **Unit**, encrypted as $\{\mathsf{unit}\}_\sigma$ of type **X** that is kept opaque from the adversary. The seal σ represents a fresh cryptographic key that we take care not to disclose to the adversary.
(2) The adversary replies with a value of the unknown type **Y** (chosen by the adversary). The actual value transmitted back in our counterexample, is simply the encrypted value $\{\mathsf{unit}\}_\sigma$ received in step 1.

[8] Available at http://www.ccs.neu.edu/home/dijamner/paramblame/artifact/
[9] Thanks to Jeremy Siek and others, for their kind support in doing this.

(3) The compiled term does not inspect the received value but simply transmits it back to the adversary as a value of type **Y**.
(4) The adversary now takes the received value $\{\mathsf{unit}\}_\sigma$ and sends it back as a value of type **X**.
(5) The compiled term receives this value, decrypts it using the private cryptographic key σ and uses the result as a value of type **Unit**.

What goes wrong in the above communication is that the value $\{\mathsf{unit}\}_\sigma$ sent back by the adversary in step 2 is essentially illegal. The adversary should have chosen a **Y** independently of **X** and values of such a type should intuitively not be able to contain values of type **X** (unless they are themselves packed in an existential package somehow). Let us try to think of what we could change in the communication protocol to enforce this.

In a pure cryptographic setting, we believe there is little hope to fix the compiler. To understand this, consider how, in the cryptography setting, the value $\{\mathsf{unit}\}_\sigma$, that we send to the adversary in step 1, simply represents a sequence of bits that is the result of some encryption algorithm applied to value **unit**. On the other hand, the value sent back by the adversary in step 2 is another sequence of bits that represents a value of an unknown type **Y**. We want to prevent this second value from somehow including the first value $\{\mathsf{unit}\}_\sigma$, but since the type **Y** is unknown and the adversary is running on an untrusted computer, there seems to be little the compiler can check. Any sequence of bits received from the adversary could in principle be a cleverly-encoded version of $\{\mathsf{unit}\}_\sigma$: they could have XORed the value with an arbitary other bitsequence and still be able to retrieve the original afterwards.

This (informal) argument suggests that the Sumii-Pierce compiler cannot be fixed in any way, as long as the target language contains only features that can be interpreted as a form of (idealised) cryptography. This would include the original sealing primitives (whatever way they are used), but also possible extensions that model a form of idealised signing (rather than encryption) (a track we were initially exploring).

To fix the compiler, it seems like we need to add some kind of feature that takes us beyond a pure-cryptography setting. We believe such a feature could take the form of a primitive that checks whether a value directly or indirectly contains values sealed with a certain seal. Such a primitive could perhaps allow to perform the required check on the value received from the adversary in step 2. Such a primitive does not correspond to a form of idealised encryption: noticing, for example, that a value like $\{\{\mathsf{v}\}_{\sigma_1}\}_{\sigma_2}$ contains a value sealed with σ_1 would break the cryptographic interpretation of λ^σ, as it requires looking inside an encrypted value without access to the key (σ_2) and requires detecting, for example, arbitrarily XORed versions of a ciphertext. However, the primitive could still be implementable in non-cryptographic settings, like the hardware-enforced seals that are present in capability machines: a form of processor with native support for capabilities and sealing that has been developed recently [Watson et al. 2015]. Note that the attacker model in this setting is a bit different: we assume that the untrusted attacker is now running on trusted hardware.

5.2 Polymorphic Blame Calculus Without a Universal Type?

When it comes to the polymorphic blame calculus, it is the existence of the universal type $\star$ that breaks the fully abstract embedding of System F in our counterexample. So perhaps we should remove or modify the type $\star$ in order to solve the problem?

We believe it may not be needed to remove the type $\star$ entirely, but that we can instead replace it with a family of types $\star_{\{\bar{X}\}}$, indexed by a set of type variables $\bar{X}$. The type $\star_{\bar{X}}$ would only be legal (i.e. well-formed) when the type variables $\bar{X}$ are in scope. For values whose type is a type variable X', injection into $\star_{\bar{X}}$ would only be legal if X' is in the set $\bar{X}$.

For our counterexample, the effect would be that the context could no longer produce a non-degenerate value of type **Univ**, as there would no longer be a viable choice for **Y**. Any instance $\star_{\{\bar{X}\}}$ of the universal type we could choose, could not have **X** as part of $\{\bar{X}\}$, as **X** is not yet in scope at the moment where **Y** needs to be produced. More generally, we conjecture that such a modified polymorphic blame calculus would embed System F in a fully abstract way.

An interesting question at this stage is whether this calculus would still adequately embed the untyped lambda calculus. Perhaps untyped code would be embeddable for any choice of $\bar{X}$? Would such untyped code still be able to interact as we would like with typed code?

5.3 Adjusting our Expectations

The above suggested solutions seem to us like they might work, but they are not without downsides. The modified Sumii-Pierce compiler could no longer be used in a purely-cryptographic setting and the family of universal types $\star_{\{\bar{X}\}}$ would be harder to use than the original $\star$. As such, we might consider an alternative way to address the lack of full abstraction. We could also decide to simply adjust our expectations: perhaps preserving all System F equivalences is overly ambitious and we should find a way to eat what is on the table instead. Both in the case of Sumii and Pierce's compiler and the gradual lambda calculus, it seems like something non-trivial is being enforced, even though it is not the preservation of arbitrary System F contextual equivalences. A way to formalise this is to recover full abstraction by modifying the source language System F: weakening its contextual equivalences in order to make them easier to preserve.

In fact, our counterexample suggests a way to accomplish this: the problem is essentially that the type **Univ** is degenerate in System F, but not in the target language. So what if we modify System F to remove that degeneracy in the source language too? Specifically, what if we add a primitive type that all other types can be embedded into and extracted from? Interestingly, it seems like what we end up here is a simple version of the gradual type $\star$, together with injection and extraction functions.

Without working this out in more detail, we find it plausible that we can recover full abstraction with such a modification, both for Sumii and Pierce's compiler and the embedding into the polymorphic blame calculus. The question is if this is a direction we want to take: is such a modified version of System F a source language that programmers would want to work in? To answer this question, we need to understand better what sort of language we end up with. For example, does such a language still satisfy a form of parametricity?

A recent result shows that the answer to this last question is positive. Ahmed et al. [2017] formulate and prove a form of parametricity for the polymorphic blame calculus. Their formulation of parametricity uses a logical relation with a Kripke world that assigns a relational interpretation to a fresh runtime type label when it is first introduced. This resembles the assignment of an invariant to a freshly allocated reference in a logical relation for a language with mutable references (see e.g. Ahmed et al. [2009]). Interestingly, this seemingly effectful formulation seems to preclude the trick that we have used in the proof of Theorem 2.4 of assigning two different relational interpretations to a single type variable: when the new runtime type label for the type application $\mathbf{x}'\ \mathbf{Unit}$ (the subterm of $\mathbf{t_u}$) is allocated, we need to choose a single relational interpretation for it and there seems to be no way to use two different ones instead. Intuitively, this suggests that the authors are allowing for the fresh runtime type label to play a global role in the application, rather than its role being restricted to the scope of the new type variable. This makes sense considering how a runtime type label can escape its scope when it is cast into the universal type $\star$. In such a setting, the degeneracy of **Univ** does not follow, but there still is a useful form of parametricity left, as Ahmed et al. demonstrate with some examples.

In other words, it appears that the degeneracy of type **Univ** does not follow from parametricity in general, but just from the particular kind of parametricity in typical formulations of System F and that Ahmed et al. have devised a weaker, alternative formulation that holds in a polymorphic blame calculus. Note that it must be weaker because it holds in a calculus where the type **Univ** is not degenerate, whereas the more standard formulation that we have used for System F implies this degeneracy. However, the weaker form of parametricity still implies useful results: Ahmed et al. prove free theorems for types $\forall X.\, \forall Y.\, X \to Y \to X$ and $\forall X.\, X \times X \to X \times X$.

6 RELATED WORK

System F and parametricity. Parametric polymorphism was first introduced 50 years ago as an informal concept by Strachey [2000]. A few years later, System F was independently discovered by Reynolds [1974] and Girard [1972]. Reynolds [1983] later formalised Strachey's informal concept of parametricity using a logical relation for System F, as explained in Example 2.2 and Wadler [1989] popularised the property using the slogan "Theorems for Free". The property was further developed by many different researchers over the years but for space reasons, we refer to Atkey et al. [2014]; Wadler [2007] for an overview of related work.

Enforcing parametricity using dynamic sealing. Dynamic sealing/unsealing was (informally) proposed more than 40 years ago by Morris [1973a,b] as a way for dynamically enforcing type abstractions. Much later, Pierce and Sumii [2000] developed this idea into a compiler that uses sealing to enforce System F's parametricity and conjectured full abstraction of the proposed compiler. The target language in this work is a simply typed cryptographic λ-calculus. They already mention that the dynamic enforcement of parametricity may also be useful to combine parametric polymorphism and untyped languages, foreshadowing the work on parametrically polymorphic gradual type systems that we discuss next.

A few years later, the same authors report further on the simply typed cryptographic λ-calculus and construct a logical relation for proving contextual equivalences for it [Sumii and Pierce 2003]. Another year later, they report on a bisimulation that can be used to prove contextual equivalence in λ^{σ}, which they originally constructed for proving the conjectured full abstraction [Sumii and Pierce 2004]. In this last paper, the compiler is presented for an untyped version of the cryptographic λ-calculus (as in this text), which renders it significantly simpler.

Sealing was also used to enforce polymorphic contracts in PLT Scheme by Guha et al. [2007]. While technically similar, the assumptions in that work are somewhat different than in the above, as contracts are about protecting the context from misbehaving terms, while Sumii and Pierce's compiler protects trusted terms from a misbehaving context. Like other work discussed in this paper, Guha et al.'s polymorphic contracts fail to enforce the degeneracy of **Univ**, so if they satisfy a form of parametricity, it would have to be similar to Ahmed et al. [2017] in the sense that it does not imply this degeneracy.

Gradual typing. Gradual typing is a specific way of combining dynamic and static typing in a single language, intended to create a gradual migration path from untyped to typed codebases. Since it was originally proposed by Siek and Taha [2006], gradual typing has received a lot of attention: gradual extensions have been constructed for many different type systems, the notion of blame was adapted from the world of contracts [Findler and Felleisen 2002] to track the origin of a dynamic cast failure [Wadler and Findler 2009], correctness criteria were studied [Siek et al. 2015], the process of constructing a gradually-typed version of a pre-existing type system was to some extent automated [Cimini and Siek 2016; Garcia et al. 2016] and last but not least, the entire approach has also been declared dead because of severe performance issues [Takikawa et al. 2016].

Gradual typing and parametric polymorphism. As an instance of a multi-language (as proposed by Matthews and Findler [2009]), Matthews and Ahmed [2008] construct a language that can embed both Scheme (i.e. an untyped lambda calculus) and ML (i.e. a parametrically polymorphic typed lambda calculus) using a notion of dynamic casts from one to the other. They enforce parametric polymorphism using a notion of run-time type generation[10] and prove a parametricity result. An error in the proof was later corrected [Ahmed et al. 2011c].

Next, Ahmed et al. [2011a, 2009] present the first version of the polymorphic blame calculus, a gradually-typed extension of System F that includes a notion of blame. The authors prove a number of correctness results, but Perconti found an incorrectible error in one of the proofs later [Ahmed et al. 2011b]. However, they do not consider parametricity or preservation of System F contextual equivalences (i.e. fully abstract embedding).

Very recently, Ahmed et al. [2017] present a new version of the polymorphic blame calculus rectifying certain "topsy-turvy" aspects of its operational semantics (see Section 4). Additionally, they prove a parametricity result for this calculus, which we have discussed from the perspective of our results in Section 5.3.

Igarashi et al. [2017] also present a new version of the polymorphic blame calculus, as an internal runtime representation for System F_G a new gradually-typed extension of System F. They make a more substantial change to the language by adding a distinction between static and gradual type variables. The former are used when embedding System F terms and they cannot be cast into or out of the gradual type $\star$. This prevents situations where a polymorphic type like $\forall X.\, X \to X$ is cast to $\forall X.\, \star \to X$ which would cause sealing or unsealing positions to be forgotten, breaking certain correctness properties. The addition allows them to prove properties like the gradual guarantee, but it does not seem to prevent our counterexample to the fully abstract embedding of System F.

In an unpublished draft, Siek and Wadler [2016] study and interrelate three ways to achieve relational parametricity: universal types, runtime type generation and cryptographic sealing. They show translations from the polymorphic blame calculus from [Ahmed et al. 2017] into the cryptographic lambda calculus and back that they show to be simulations. They also study a calculus λ_G, obtained by removing the universal type $\star$ and casts from the polymorphic blame calculus, but keeping runtime type generation. They show that embedding System F into λ_G is also fully abstract. Both of these full abstraction results tally with our observations: both System F and λ_G lack a universal type (making **Univ** degenerate) while both the polymorphic blame calculus and the cryptographic lambda calculus feature a universal type (making **Univ** non-degenerate). As such, the embedding of λ_G into the polymorphic blame calculus will not be fully abstract, supplementing their results.

Non-parametric parametricity. In a related setting, Neis et al. [2009, 2011] study a form of runtime type generation to protect parametrically polymorphic functions when interacting with code that can use a typecase primitive. They prove a parametricity result that applies to appropriately wrapped System F values. We suspect that our results may be adapted to this setting, as in such a non-parametrically polymorphic setting, the type $\forall X.\, X$ could be used as a universal type (that every other type can be embedded into or out from).

Universal types. Universal types have been studied by Longley [2003]. Our observation and proof that System F's parametricity excludes a universal type is, to the best of our knowledge, novel.

[10] They call this sealing, but since their seals have no run-time representation, we prefer the term run-time type generation.

7 CONCLUSION

This work started out as an effort to prove Sumii and Pierce's conjecture, discussed in Section 3. However, rather than solving a decade-old open problem, this paper has turned out to point out new problems instead. We disprove the conjecture rather than proving it, and we identify what we see as an important problem in current parametrically polymorphic gradual languages: programmers reasoning about System F programs cannot trust contextual equivalences to remain valid in the gradually typed extended language. Solving those problems does not seem easy: it is not even clear in which direction such solutions had best be sought. We point out some directions in Section 5, but none seems perfect: they each come with downsides like reduced programmer convenience or the requirement of additional target language primitives. However, despite the absence of positive results, our results improve the understanding of the interaction between parametrically polymorphic languages and less strongly typed ones and may be useful to steer research into the concerned topics.

ACKNOWLEDGMENTS

Dominique Devriese holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO). This research is partially funded by project grants from the Research Fund KU Leuven, and from the Research Foundation Flanders (FWO). This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762). The authors thank Phil Wadler for interesting comments and suggestions.

REFERENCES

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

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 237–268. DOI: http://dx.doi.org/10.1145/103135.103138

Martín Abadi, Cédric Fournet, and Georges Gonthier. 1998. Secure Implementation of Channel Abstractions. In *IEEE Symposium on Logic in Computer Science*. 105–116.

Martín Abadi, Cédric Fournet, and Georges Gonthier. 1999. Secure Communications Processing for Distributed Languages. In *IEEE Symposium on Security and Privacy*. 74–88.

Martín Abadi, Cédric Fournet, and Georges Gonthier. 2000. Authentication Primitives and their Compilation. In *Principles of Programming Languages*. ACM, 302–315. DOI: http://dx.doi.org/10.1145/325694.325734

Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. *ACM Transactions on Information and System Security* 15, Article 8 (July 2012), 8:1–8:29 pages. DOI: http://dx.doi.org/10.1145/2240276.2240279

Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure Compilation to Modern Processors. In *Computer Security Foundations Symposium*. IEEE, 171–185. DOI: http://dx.doi.org/10.1109/CSF.2012.12

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming*. ACM, 157–168. DOI: http://dx.doi.org/10.1145/1411204.1411227

Amal Ahmed, Justin Damner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free. In *ICFP*.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent Representation Independence. In *Principles of Programming Languages*. ACM, 340–353. DOI: http://dx.doi.org/10.1145/1480881.1480925

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011a. Blame for All. In *Principles of Programming Languages*. 201–214.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011b. *Blame for All.* Technical Report. https://plt.eecs.northwestern.edu/blame-for-all/

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

Amal Ahmed, Jacob Matthews, Robert Bruce Findler, and Philip Wadler. 2009. Blame for All. In *Workshop on Script-to-Program Evolution (STOP)*. 1–13.

Robert Atkey, Neil Ghani, and Patricia Johann. 2014. A relationally parametric model of dependent type theory. In *Principles of Programming Languages*. ACM, 503–515. DOI: http://dx.doi.org/10.1145/2535838.2535852

Michele Bugliesi and Marco Giunti. 2007. Secure Implementations of Typed Channel Abstractions. In *Principles of Programming Languages*. ACM, 251–262. DOI:http://dx.doi.org/10.1145/1190216.1190253

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Principles of Programming Languages*. ACM. DOI:http://dx.doi.org/10.1145/2837614.2837632

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

Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, Fully-Abstract Compilation by Approximate Back-Translation. to appear in LMCS. (2017). http://arxiv.org/abs/1703.09988

D. Dreyer, A. Ahmed, and L. Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Logic In Computer Science*. 71–80.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *International Conference on Functional Programming*. ACM. DOI:http://dx.doi.org/10.1145/581478.581484

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Principles of Programming Languages*. ACM, 371–384. DOI:http://dx.doi.org/10.1145/2429069.2429114

Ronald Garcia, Alison M. Clark, and ÃĽric Tanter. 2016. Abstracting Gradual Typing. In *Principles of Programming Languages*. ACM. DOI:http://dx.doi.org/10.1145/2837614.2837670

Jean-Yves Girard. 1972. *Interprétation Fonctionelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur*. Ph.D. Dissertation. Université Paris VII.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric Polymorphic Contracts. In *Symposium on Dynamic Languages*. ACM. DOI:http://dx.doi.org/10.1145/1297081.1297089

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *International Conference on Functional Programming*. ACM.

Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. 2011. Local Memory via Layout Randomization. In *Computer Security Foundations Symposium*. IEEE Computer Society, 161–174. DOI:http://dx.doi.org/10.1109/CSF.2011.18

Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *Computer Security Foundations Symposium*.

Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Towards a Fully Abstract Compiler Using Micro-Policies: Secure Compilation for Mutually Distrustful Components. *CoRR* abs/1510.00697 (2015). http://arxiv.org/abs/1510.00697

Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. 2015. A Secure Compiler for ML Modules. In *Programming Languages and Systems - 13th Asian Symposium*. 29–48. DOI:http://dx.doi.org/10.1007/978-3-319-26529-2_3

Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. 2016. Implementing a Secure Abstract Machine. In *Symposium on Applied Computing*. ACM, 2041–2048. DOI:http://dx.doi.org/10.1145/2851613.2851796

John R. Longley. 2003. Universal Types and What They are Good For. In *Domain Theory, Logic and Computation*. Springer, Dordrecht, 25–63. DOI:http://dx.doi.org/10.1007/978-94-017-1291-0_2

Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices! LNCS, Vol. 4960. 16–31.

Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-language Programs. *ACM Trans. Program. Lang. Syst.* 31, 3 (April 2009), 12:1–12:44. DOI:http://dx.doi.org/10.1145/1498926.1498930

James H. Morris, Jr. 1973a. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21. DOI:http://dx.doi.org/10.1145/361932.361937

James H. Morris, Jr. 1973b. Types Are Not Sets. In *Principles of Programming Languages*. 120–124.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric Parametricity. In *ICFP '09*. 135–148.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *Journal of Functional Programming* 21 (2011), 497–562.

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, Article 6 (April 2015), 6:1–6:50 pages. DOI:http://dx.doi.org/10.1145/2699503

Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully Abstract Compilation. In *Computer Security Foundations Symposium*.

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

Benjamin Pierce and Eijiro Sumii. 2000. Relating Cryptography and Polymorphism. (2000). http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/infohide.pdf manuscript.

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

John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium*. Lecture Notes in Computer Science, Vol. 19. Springer-Verlag, 408–423.

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

Jeremy Siek and Philip Wadler. 2016. The key to blame: Gradual typing meets cryptography. (2016). http://homepages.inf.ed.ac.uk/wadler/papers/blame-key/blame-key.pdf draft.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *SCHEME*. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *Summit on Advances in Programming Languages (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 274–293. DOI:http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.274

Richard Statman. 1991. *A local translation of untyped [lambda] calculus into simply typed [lambda] calculus.* Technical Report. http://repository.cmu.edu/cgi/viewcontent.cgi?article=1454&context=math

Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1-2 (April 2000), 11–49. DOI:http://dx.doi.org/10.1023/A:1010000313106

Eijiro Sumii and Benjamin C. Pierce. 2003. Logical Relations for Encryption. *J. Comput. Secur.* 11, 4 (July 2003), 521–554.

Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *Principles of Programming Languages*. 161–172.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Principles of Programming Languages*. ACM. DOI:http://dx.doi.org/10.1145/2837614.2837630

Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture*. ACM, 347–359.

Philip Wadler. 2007. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science* 375, 1 (2007). DOI:http://dx.doi.org/10.1016/j.tcs.2006.12.042

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 1–16. DOI:http://dx.doi.org/10.1007/978-3-642-00590-9_1

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*. DOI:http://dx.doi.org/10.1109/SP.2015.9