Two Parametricities versus Three Universal Types[*]

DOMINIQUE DEVRIESE, KU Leuven, Belgium
MARCO PATRIGNANI, University of Trento, Italy
FRANK PIESSENS, KU Leuven, Belgium

The formal calculus System F models the essence of polymorphism and abstract data types, features that exist in many programming languages. The calculus' core property is parametricity: a theorem expressing the language's abstractions and validating important principles like information hiding and modularity.

When System F is combined with features like recursive types, mutable state, continuations or exceptions, the formulation of parametricity needs to be adapted to follow suit, for example using techniques like step-indexing, Kripke world-indexing or biorthogonality. However, it is less clear how this formulation should change when System F is combined with untyped languages, gradual types, dynamic sealing and runtime type analysis (typecase) alongside type generation. Extensions of System F with these features have been proven to satisfy forms of parametricity (with Kripke worlds carrying semantic interpretations of types). However, the relative power of the modified formulations of parametricity with respect to others and the relative expressiveness of System F with and without these extensions are unknown.

In this paper, we explain that the aforementioned different settings have a common characteristic: they do not enforce or preserve the lexical scope of System F's type variables. Formally, this results in the existence of a *universal type* (note: this is not the same as a *universally-quantified* type). We explain why standard parametricity is incompatible with such a type and how type worlds resolve this. Building on these insights, we answer two open conjectures from the literature, negatively, and we point out a deficiency in current proposals for combining System F with gradual types.

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. 2022. Two Parametricities versus Three Universal Types. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (June 2022), 43 pages. https://doi.org/10.1145/3539657

To make the distinction between languages visually apparent, we typeset elements of System F in a **blue, bold** *font, elements of λ^{σ} in a* red, sans-serif *font, elements of* `G` *in a* `emerald`, `verbatim` *font and elements of the blame calculus in an orange, italic font. This kind of syntax highlighting has been proven effective for colourblind and black&white readers too [Patrignani 2020].*

[*]A *universal type* is not the same as a *universally-quantified type*. Following Longley [2003] (and category theory intuition), we use *universal type* as a term for a type which any other type can be embedded into and extracted from.

Authors' addresses: Dominique Devriese, DistriNet, KU Leuven, Celestijnenlaan 200A - bus 2402, Leuven, 3001, Belgium, name.surname@kuleuven.be; Marco Patrignani, University of Trento, Via sommarive 9, Trento, I-38123, Italy, name.surname@unitn.it; Frank Piessens, DistriNet, KU Leuven, Celestijnenlaan 200A - bus 2402 , Leuven, 3001, Belgium, name.surname@cs.kuleuven.be.

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. The core property of System F is *parametricity*. Parametricity guarantees that polymorphic functions in System F cannot behave differently when invoked at different types. Parametricity is formalised using a logical relation (LR): an (often relational) property about values and terms, derived from their type [Reynolds 1983]. A Fundamental Theorem of Logical Relations or Abstraction Theorem then states that the LR properties are automatically satisfied by any term, without the need to verify their code. For this reason, Wadler [1989] has described them as free theorems. A canonical example is the fact that any value of type $\forall X.\, X \rightarrow X$ must behave as the identity function.

It is well known that the formulation of parametricity must be adapted when language features are added to System F. For example, consider a value f of type $\forall X.\, X \rightarrow X$. In vanilla System F, it must behave as the identity function, i.e., return its argument in every invocation. If we add general recursion, then there is another possibility: f can also be the function that diverges on every invocation. Adding mutable state changes the situation again: it is now possible for f to return its argument in some invocations and to diverge in others, even though the choice can still not depend on the argument. If we add continuations, then it may be possible for f to return more than once, but still: on every return, the return value must be the one it received as an argument. These behavioural differences are reflected in the different definition of the logical relation used to formalize parametricity. Thus, formulations of parametricity using logical relations capture semantic guarantees about the language as a whole.[1]

In many cases (like the addition of state mentioned above), it is well understood that a particular change in the formulation of parametricity introduces changes both in the set of equivalences it implies as well as in the logical relation used to state it (see e.g., [Dreyer et al. 2010]). However, there are also cases where a certain formulation of parametricity is motivated by technical considerations, but it is not clear what effects this change has on equivalence reasoning and whether the change could have been somehow avoided. Specifically, in this paper, we look at the work by Sumii and Pierce [2003] on logical relations for encryption, the work by Neis et al. [2009, 2011] on parametricity in the presence of a dynamic type inspection primitive and runtime type generation, and the work by Ahmed et al. [2017], Toro et al. [2019], and New et al. [2019a] on parametricity in gradually-typed variants of System F. Although they work in different contexts, these authors prove (a form of) parametricity with a logical relation indexed by System F types (or a superset of these types), with the exception of Sumii and Pierce who index with simple types for cryptographic primitives. Additionally, the logical relation used by all these parametricity results (including Sumii and Pierce's) follow a particular pattern: they are indexed by a *type world*. For this reason, we refer to these LRs as type-world logical relations (TWLRs). TWLRs should not be regarded as a clearly delineated subclass of logical relations, but rather as a design pattern that can be applied and varied upon in the design of a logical relation.

The type worlds in TWLRs are a form of Kripke worlds: they capture a set of assumptions with respect to which the logical relation holds (or does not hold). There is also an order relation on worlds that expresses when one world (a "future world") represents a stronger set of assumptions

[1] In this paper we strive to reserve the term parametricity for the informal property that polymorphic functions must preserve relatedness of values of parametric types, following Strachey's [Strachey 2000] and Reynolds' [Reynolds 1983] use of the term. However, it is sometimes hard to formally draw the line between this restricted interpretation of parametricity and other properties (like purity or forms of type safety) that happen to be formulated together with the intuitive notion of parametricity in the Fundamental Theorem of Logical Relations for a specific logical relation. We will refer to this formal theorem as "a formulation of parametricity". Because the line between (informal) parametricity and a formal formulation of it is not very clear, we are not always very strict about this distinction.

than another and whenever the logical relation holds with respect to a world, it automatically also holds for future worlds. While traditional logical relations (e.g., Dreyer et al. [2011b]; Reynolds [1983]) keep track of semantic interpretations for type variables in a type environment (as we discuss later in Section 2.3), the logical relations used by Ahmed et al.; Neis et al.; New et al.; Sumii and Pierce; Toro et al. carry semantic interpretations for dynamically-allocated opaque type variables or seals in the Kripke world. Additionally, instantiating polymorphic functions (or allocating fresh seals in the case of Sumii and Pierce [2003]) results in terms that are only related in a world that stores the relation between the applied types as the type interpretation for the instantiated type (or seal).

So why is the same kind of TWLR used in these three different domains? In this paper, we suggest this is because they are used in settings where the lexical scope of type variables is not enforced. Thus, in types like $\forall X. X \to A$ (with X not free in A), it is possible for values of type X to escape from their scope, be stored as values of a type that does not mention X (such as A) and somehow be recovered from there as a value of type X again.

Instead of TWLRs, other work that predates TWLRs used logical relations that enforced such lexical scoping, similar to Reynolds' original formulation [Reynolds 1983]. Thus, we refer to these traditional logical relations as Reynolds-style logical relations (RLRs).

Although some of our insights might appear obvious, the importance of understanding the pattern of type-world logical relations can be seen in the literature, in two ways. First, if we look at the history of one of the aforementioned results [Ahmed et al. 2017], then we see that an RLR was actually used in a precursor of the work [Matthews and Ahmed 2008]. However, a flaw was later discovered in this proof [Ahmed et al. 2017]. Few details are available about this flaw, but our results make it clear that the RLR could not possibly have been compatible with the language's non-lexically scoped type variables, i.e., the theorem was wrong, not just the proof. The flaw was resolved by the authors in an unpublished draft by moving to a TWLR [Ahmed et al. 2011c] and the whole effort culminated in the mature TWLR of Ahmed et al. [2017].

A second consequence is that researchers had wrong expectations of how program equivalence changes when not enforcing lexical scope of type variables in extensions of System F. Concretely, our insights allow us to disprove two long-standing conjectures made by experts in published literature. These conjectures are respectively related to *secure compilation* of System F using dynamic sealing (read, idealised encryption) and to the *enforcement of parametricity* in the presence of dynamic type analysis and runtime type generation. Additionally, we also identify a concern in the *interaction between gradually typed languages and polymorphic types*. Let us take a closer look at these three topics.

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 target-level components. The goal of secure compilation is to ensure that those target-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, secure compilation has been expressed as full abstraction: given two source-level contextually equivalent programs, their target-level compilation are also contextually equivalent (and vice versa) [Abadi 1998].[2] Compiler full abstraction has been proven for compilers that rely

[2]Other formal characterisations of secure compilation have been proposed more recently [Abate et al. 2018, 2019; Patrignani and Garg 2017, 2019]. We discuss their relation to our work in Section 9.

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. 2015, 2016], dynamic type checking in 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, we refer the interested reader to the survey of Patrignani et al. [2019].

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, owing part of the complexity of such a proof to the target language being more expressive than the source. Their conjecture has received further research attention, but remains open to this day [Siek and Wadler 2016; Sumii and Pierce 2004]. In other work, the same authors proposed a TWLR for the cryptographic lambda calculus [Sumii and Pierce 2003].

Non-parametric parametricity. Some programming languages include a way to perform *intensional type analysis* through a type cast operator [Abadi et al. 1995; Rossberg 2003]. This appears to be in direct conflict with parametric polymorphism, possibly violating parametricity and representation independence guarantees [Mitchell 1986]. Researchers have proposed runtime type generation as a way to regain parametricity for languages with a type cast operator. Ideally, when an abstract type is defined, a fresh type name should also be generated at runtime. Such a name should then be used as a runtime representative of the abstract type for type analysis purposes.

Runtime type generation has been proven to indeed provide parametricity guarantees for System F terms that interact with terms that can perform type casts [Neis et al. 2009]. This guarantee has been dubbed *non-parametric parametricity.*

Neis et al. [2009] also conjectured that their way of using runtime type generation preserves *any* System F abstraction in their type cast language. Although they do not explain the reasons behind this conjecture, we have an idea of the intuition behind it. Parametricity is the most powerful abstraction programmers think that System F has, so it seems logical to believe that once that has been preserved, all other abstractions will follow. These authors also use a TWLR for stating parametricity in their system, and our results let us conclude that their version is weaker than the parametricity one states with RLR.

Gradual typing. The final field where we contribute new insight is gradual typing. In order to allow programmers to incrementally migrate 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 runtime type errors but also benefits for reasoning [de Amorim et al. 2020; New et al. 2019b; Toro et al. 2018]. While the literature mentions several ways to formalise the former two properties, the latter has received less attention.

Based on a suggestion in the conference version of this paper [Devriese et al. 2018], Jacobs et al. [2021] have recently proposed to formally express that a gradual language preserves the reasoning principles of the typed language by reusing 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.[3]

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. 2011b], further developed by Igarashi et al. [2017], Ahmed et al. [2017], Toro et al. [2019] and New et al. [2019a]. The latter two papers also formulate parametricity using a TWLR in the polymorphic blame calculus. In these cases it is also unclear what are the benefits of relying on TWLRs has as opposed to relying on RLR.

Three questions, one answer. To answer these questions, we have to define non-lexical scoping of type variables. Our best formal characterisation is based on the existence of what we call a *universal type.*[4] A universal type is a type which any other type can be embedded into and extracted from. Thus, by this definition, the term *universal type* is broader than Abadi et al. [1991]'s *dynamic type* or Siek and Taha [2006]'s *gradual type*: 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.[5] We point out that enforcing the lexical scope of type variables forbids the existence of a universal type such as the following one:

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

Type $\underline{\mathbf{Univ}}$ expresses the existence of a universal 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 $\underline{\mathbf{Univ}}$, but we prove that they are all degenerate in the sense that mapping a value into the universal type and back must necessarily diverge. Our key finding is that this degeneracy can be proven using RLRs (and this illustrates the incompatibility of RLRs with universal types) but it cannot be proven using TWLRs.

Thus, by clarifying the subtleties of TWLRs and RLRs, we solve these aforementioned open problems and answer these questions negatively. Concretely, we prove that Sumii and Pierce's compiler is not fully abstract, and that System F does not embed fully abstractly into either Neis et al.'s language with a type cast, or any of the published polymorphic blame calculi.

More in detail, Sumii and Pierce's compiler fails to enforce this degeneracy of $\underline{\mathbf{Univ}}$. 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 fully abstract compilation conjecture is false, as we will formally show by constructing two System F terms $\mathbf{t_u}$ and $\mathbf{t_\omega}$ whose contextual equivalence relies on the degeneracy of $\underline{\mathbf{Univ}}$. 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 Neis et al.'s language with non-parametric parametricity, we show that type cast operators also break the degeneracy of $\underline{\mathbf{Univ}}$, despite the presence of runtime type generation primitives.

[3]This is related to the notion that fully abstract compilation can also be used to reason about language expressiveness in general, beyond just language security (which is what is done in the case of secure compilation) [Felleisen 1991; Mitchell 1993; Parrow 2008].

[4]This seems to imply non-lexical scoping of type variables, so it suffices for our results. Ultimately, we think that the non-lexical scope of type variables is the more fundamental characteristic of the systems we are interested in.

[5]The term universal type was introduced by Longley [2003] based on a similarity to *universal objects* in category theory and related terms have been used in the literature [New et al. 2016a]. As mentioned before, the term universal type should be clearly distinguished from *universally-quantified* types, i.e., types of the form $\forall X.\, \tau$.

The type $\forall Z.\ Z$, which is normally only inhabited by diverging terms, becomes a universal type in the presence of a dynamic type cast, as any value can be embedded in it and extracted from it. By relying on this type, System F does not embed fully abstractly into this language, contradicting Neis et al.'s conjecture.

Finally, in the field of gradual typing, we demonstrate that existing polymorphic blame calculi also break the degeneracy of **Univ**.[6] 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.

To close, 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 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, Neis et al.'s sealing wrappers and the polymorphic blame calculus.

Outline. We start our discussion by repeating the definition of System F and defining two logical relations for it (an RLR and a TWLR), which we will use in subsequent proofs (Section 2). Then we present type **Univ** and two key terms: $\mathbf{t_u}$ and $\mathbf{t_\omega}$, and we discuss their contextual equivalence (Section 3). We then prove that **Univ** is degenerate and that these terms are contextually equivalent using the previously-defined logical relations (Section 4). Next, we present Sumii and Pierce's compiler and disprove their conjecture by explaining how it treats $\mathbf{t_\omega}$ and $\mathbf{t_u}$ and fails to preserve their equivalence (Section 5). We then present $\mathtt{G}$, an extension of System F with type casts and runtime type generation and demonstrate how embedding $\mathbf{t_u}$ and $\mathbf{t_\omega}$ into $\mathtt{G}$ breaks contextual equivalence (Section 6). Next, we turn to gradual typing, introducing polymorphic blame calculi and demonstrating how the contextual equivalence of $\mathbf{t_u}$ and $\mathbf{t_\omega}$ is also lost in the presence of these calculi's universal type (Section 7). Finally, we discuss perspectives and consequences of our results (Section 8), related work (Section 9) and we conclude (Section 10). We omit only few auxiliary lemmas, their proofs and tedious, long reductions used in the main proofs, all of this can be found in the supplementary material.

Relation with the Previous Version. This paper extends a paper by the same authors that was published at POPL 2018 [Devriese et al. 2018]. In this version, the main changes include (1) a more detailed analysis of the type **Univ** and how its degeneracy varies in the presence of effects and value polymorphism (2) a presentation of a Reynolds-style or lexically-scoped (Section 2.3), Kripke (Section 2.4) and type-world logical relation (Section 2.5), (3) a more elegant proof of the degeneracy of **Univ** (Section 4), (4) a detailed analysis of the relation between the different logical relations, non-lexically-scoped type variables and degeneracy of the universal type (Section 4.2), and (5) the disproval of the conjecture by Neis et al. [2011] in Section 6.

2 SYSTEM F

We now consider System F itself (Section 2.1), the standard formulation of parametricity for it (Section 2.2) and the different kinds of logical relations for it: Reynolds-style or lexically-scoped

[6]Unlike the previous two, this was not an existing conjecture we debunk.

(Section 2.3) and type-world (Section 2.5). To introduce type-world logical relations, we first explain their more general variant: Kripke logical relations (Section 2.4)

Note on divergence and recursive types. Technically, the language we should be using is System F with recursive types since that is the language considered by the conjectures we examine [Pierce and Sumii 2000]. We choose against including recursive types in our technical development since their presence is not central to our argument for disproving existing conjectures. Recursive types are required only insofar as System F is allowed some way to diverge. Fortunately, there are simpler ways to allow a language to diverge, as for example the addition of a diverging term (a solution also proposed by Pierce and Sumii [2000] and that we also adopt). Moreover, well-founded logical relations for languages with recursive types require *step indices*, and the addition of steps makes the technical development noisy without adding particular insights. Thus, in this paper we remove recursive types from System F and instead add a diverging term ω.

2.1 The Source Language λ^{F}

Figure 1 presents the variant of System F that we will be using in this paper, which we indicate with λ^{F}. In addition to standard polymorphic functions ($\forall X.\, \tau$) and existential packages ($\exists X.\, \tau$), the variant includes Unit and Bool and product $\tau_1 \times \tau_2$ types. In the figure, we present types τ, values v and terms t, here the most peculiar addition is ω_τ, which is a diverging term of type τ . We show the most important typing rules $\Delta; \Gamma \vdash t : \tau$ in terms of term and type variable contexts Γ and Δ (but we omit context and type well-formedness judgements $\Delta; \Gamma \vdash \diamond$ and $\Delta \vdash \tau$). Finally, we define call-by-value evaluation rules in terms of evaluation contexts E. There, we indicate the usual capture-avoiding substitution of value v (or type τ') for variable x (or type variable X) in term t (or type τ) as $t[v/x]$ (as $\tau[\tau'/X]$). We indicate lists of such substitutions as γ.

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

Definition 2.1 (λ^{F} Contextual equivalence). For two terms t_1, t_2 that have the same type τ in the same contexts Δ and Γ, we define that they are contextually equivalent ($\Delta; \Gamma \vdash t_1 \simeq t_2 : \tau$) iff for all C such that $\vdash C : \Delta; \Gamma, \tau \to \emptyset; \emptyset, \tau'$, we have that $C[t_1] \Uparrow$ iff $C[t_2] \Uparrow$, where $\Uparrow$ indicates divergence [Plotkin 1977].

Value polymorphism. An interesting aspect of this definition of System F is that the body of a polymorphic functions $\Lambda X.\, t$ is a general term t. This means that it is possible for polymorphic functions to diverge or perform effects when instantiated. This means, for example, that the polymorphic type $\forall X.\, \bot$ (for an empty type $\bot$) is inhabited by the function $\Lambda X.\, \omega$. In effectful variants of System F, we could similarly write a function of type $\forall X.\, \mathrm{Ref}\ (\mathrm{Maybe}\ X)$: $\Lambda X.\, \mathrm{ref\ nothing}$. For this function, it is important that different applications of this function are evaluated separately and produce different mutable reference locations, as otherwise, we could use a single location to store a value of one type and read a value of another, leading to a type error. In particular, this means that polymorphic lambdas and type applications cannot be erased away in an untyped execution scheme.

Although the choice of allowing arbitrary terms in the body of a polymorphic function is standard in formal accounts of System F, imperative polymorphic languages like ML make a different choice. They use *value polymorphism*, originally proposed by Wright [1995], which restricts ML polymorphism to values. This choice can be modeled in System F by restricting the syntax of polymorphic functions from $\Lambda X.\, t$ to $\Lambda X.\, v$, i.e., the body of the polymorphic function must be a

Syntax:

$$\begin{array}{rl} \mathsf{t} ::= & \mathsf{v} \mid \mathsf{x} \mid \mathsf{t}\;\mathsf{t} \mid \mathsf{t.1} \mid \mathsf{t.2} \mid \langle \mathsf{t}, \mathsf{t}\rangle \mid \mathsf{t}\;\tau \mid \mathbf{if}\;\mathsf{t}\;\mathbf{then}\;\mathsf{t}\;\mathbf{else}\;\mathsf{t} \mid \mathbf{pack}\;\langle \tau, \mathsf{t}\rangle\;\mathbf{as}\;\exists \mathsf{X}.\,\tau \\ & \mid \mathbf{unpack}\;\mathsf{t}\;\mathbf{as}\;\langle \mathsf{X}, \mathsf{x}\rangle\;\mathbf{in}\;\mathsf{t} \mid \omega_\tau \\ \mathsf{Val} \ni \mathsf{v} ::= & \mathbf{unit} \mid \mathbf{true} \mid \mathbf{false} \mid \lambda \mathsf{x} : \tau.\,\mathsf{t} \mid \langle \mathsf{v}, \mathsf{v}\rangle \mid \Lambda \mathsf{X}.\,\mathsf{t} \mid \mathbf{pack}\;\langle \tau, \mathsf{v}\rangle\;\mathbf{as}\;\exists \mathsf{X}.\,\tau \\ \tau ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \tau \to \tau \mid \tau \times \tau \mid \mathsf{X} \mid \forall \mathsf{X}.\,\tau \mid \exists \mathsf{X}.\,\tau \\ \Gamma ::= & \emptyset \mid \Gamma, (\mathsf{x} : \tau) \qquad\qquad \Delta ::= \emptyset \mid \Delta, \mathsf{X} \\ \mathsf{E} ::= & [\cdot] \mid \mathsf{E}\;\mathsf{t} \mid \mathsf{v}\;\mathsf{E} \mid \mathsf{E.1} \mid \mathsf{E.2} \mid \langle \mathsf{E}, \mathsf{t}\rangle \mid \langle \mathsf{v}, \mathsf{E}\rangle \mid \mathsf{E}\;\tau \mid \mathbf{if}\;\mathsf{E}\;\mathbf{then}\;\mathsf{t}\;\mathbf{else}\;\mathsf{t} \mid \mathbf{unpack}\;\mathsf{E}\;\mathbf{as}\;\langle \mathsf{X}, \mathsf{x}\rangle\;\mathbf{in}\;\mathsf{t} \end{array}$$

Typing rules (excerpts):

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \diamond \\ (\mathsf{x} : \tau) \in \Gamma\end{array}}{\Delta; \Gamma \vdash \mathsf{x} : \tau} \qquad \frac{\Delta; \Gamma, \mathsf{x} : \tau \vdash \mathsf{t} : \tau'}{\Delta; \Gamma \vdash \lambda \mathsf{x} : \tau.\,\mathsf{t} : \tau \to \tau'} \qquad \frac{\Delta, \mathsf{X}; \Gamma \vdash \mathsf{t} : \tau}{\Delta; \Gamma \vdash \Lambda \mathsf{X}.\,\mathsf{t} : \forall \mathsf{X}.\,\tau}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathsf{t} : \tau' \to \tau \\ \Delta; \Gamma \vdash \mathsf{t}' : \tau'\end{array}}{\Delta; \Gamma \vdash \mathsf{t}\;\mathsf{t}' : \tau} \qquad \frac{\Delta \vdash \tau' \qquad \Delta; \Gamma \vdash \mathsf{t} : \forall \mathsf{X}.\,\tau}{\Delta; \Gamma \vdash \mathsf{t}\;\tau' : \tau[\tau'/\mathsf{X}]}$$

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

$$\frac{\Delta; \Gamma \vdash \mathsf{t} : \exists \mathsf{X}.\,\tau \qquad \Delta \vdash \tau' \qquad \Delta, \mathsf{X}; \Gamma, \mathsf{x} : \tau \vdash \mathsf{t}_1 : \tau'}{\Delta; \Gamma \vdash \mathbf{unpack}\;\mathsf{t}\;\mathbf{as}\;\langle \mathsf{X}, \mathsf{x}\rangle\;\mathbf{in}\;\mathsf{t}_1 : \tau'} \qquad \frac{}{\Delta; \Gamma \vdash \omega_\tau : \tau}$$

Evaluation rules (excerpts):

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

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

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

syntactic value [Pitts 1998]. Value polymorphism is not central to our technical development, but we will come back to this in Section 3.2 when discussing the aforementioned universal type in more general settings.

2.2 Parametricity

The main information hiding mechanism in λ^{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 λ^{F}). We could for instance represent the type $\mathbb{Z}_n$ of integers modulo n as a tuple $\langle\langle \mathsf{zero}, \mathsf{succ}\rangle, \mathsf{zero?}\rangle$ of type $\exists \mathsf{X}.\,\mathsf{X} \times (\mathsf{X} \to \mathsf{X}) \times (\mathsf{X} \to \mathsf{Bool})$, and then implement this type as a k-tuple of booleans.

Example 2.3 (Example polymorphic function type in λ^{F}). Dually, the type system ensures that code cannot depend on parameters of universally quantified types, for instance the only thing a function of type $\forall X.\, X{\times}X{\to}X$ can do is return one of its two arguments or diverge.

Reynolds formalised (relational) 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 f of type $\forall X.\, X{\to}X$, 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 v_1, v_2 of type τ_1 and τ_2 respectively, if (v_1, v_2) is in R, then $f\ \tau_1\ v_1$ and $f\ \tau_2\ v_2$ will either both diverge or reduce to values $(v_1', v_2') \in R$. The relational property is derived from the type using what is known as a logical relation.

2.3 Reynolds-style Logical Relation

As we explained in the introduction, this logical relation can be defined in various ways. For our purposes, it is instructive to first consider Reynolds' original definition. For simplicity and because it suffices for our purposes, we present a unary variant.[7]

The Reynolds-style or lexically-scoped Logical Relation (RLR) defines a relation on values $\mathcal{V}\,[\![\cdot]\!]$ and on terms $\mathcal{E}\,[\![\cdot]\!]$. Both are indexed with a type environment ρ, which maps type variables to a closed type τ and a relation R on values of type τ. The value relation $\mathcal{V}\,[\![\cdot]\!]^{\rho}$ for type variables X defers to the appropriate entry in ρ for type variables X. Otherwise, the value relation accepts boolean and unit values when they are of canonical form and pairs when both components are in the appropriate relation themselves. For function types, the value relation accepts appropriately typed lambdas that map related values to related terms. We apply the type environment to the type τ (denoted with $\rho(\tau)$) to indicate type τ with any type variable X replaced with its binding in ρ, i.e., with $\rho(X).1$. Polymorphic functions are accepted when they can be applied to an arbitrary type τ' and an arbitrary relation R on τ' to obtain a term in the appropriate term relation, with ρ extended with R. Finally, existential packages must contain a type τ' and a value related at the appropriate type, extending ρ with some relation R on τ'. Values in the value relation are also required to be syntactically well-typed, as denoted by function $\texttt{oftype}\,(v, \tau)$, which simply checks $\emptyset; \emptyset \vdash v : \tau$. The term relation accepts terms that diverge or produce a suitable value.

$$
\begin{aligned}
\rho &\in \left\{ \overline{X \mapsto (\tau, R)} \;\middle|\; R \in \texttt{Rel}\,(\tau) \right\} \\
\texttt{Rel}\,(\tau) &= \mathcal{P}\,(\{ v \mid \emptyset \vdash v : \tau \}) \\
\mathcal{V}\,[\![X]\!]^{\rho} &= \rho(X).R \\
\mathcal{V}\,[\![\mathrm{Unit}]\!]^{\rho} &= \{\mathbf{unit}\} \\
\mathcal{V}\,[\![\mathrm{Bool}]\!]^{\rho} &= \{\mathbf{true}, \mathbf{false}\} \\
\mathcal{V}\,[\![\tau \to \tau']\!]^{\rho} &= \left\{ \lambda x : \rho(\tau).\, t \;\middle|\; \begin{array}{l} \texttt{oftype}\,(\lambda x : \rho(\tau).\, t, \tau \to \tau') \text{ and} \\ \quad \forall v.\ \text{if } v \in \mathcal{V}\,[\![\tau]\!]^{\rho} \text{ then } t[v/x] \in \mathcal{E}\,[\![\tau']\!]^{\rho} \end{array} \right\} \\
\mathcal{V}\,[\![\tau \times \tau']\!]^{\rho} &= \left\{ \langle v, v' \rangle \;\middle|\; \begin{array}{l} \texttt{oftype}\,(\langle v, v' \rangle, \tau \times \tau') \text{ and} \\ \quad v \in \mathcal{V}\,[\![\tau]\!]^{\rho} \text{ and } v' \in \mathcal{V}\,[\![\tau']\!]^{\rho} \end{array} \right\} \\
\mathcal{V}\,[\![\forall X.\, \tau]\!]^{\rho} &= \left\{ \Lambda X.\, t \;\middle|\; \begin{array}{l} \texttt{oftype}\,(\Lambda X.\, t, \forall X.\, \tau) \text{ and} \\ \quad \forall R \in \texttt{Rel}\,(\tau').\, t\big[\tau'/X\big] \in \mathcal{E}\,[\![\tau]\!]^{\rho, X \mapsto (\tau', R)} \end{array} \right\}
\end{aligned}
$$

[7]We will continue to use words like relation, related etc. despite the fact that they are unary.

$$\mathcal{V}\left[\!\left[\exists \mathrm{X}.\,\tau\right]\!\right]^{\rho} = \left\{ \mathbf{pack}\ \langle \tau', \mathrm{v}\rangle\ \mathbf{as}\ \exists \mathrm{X}.\,\rho(\tau) \;\middle|\; \begin{array}{r} \texttt{oftype}\,(\mathbf{pack}\ \langle \tau', \mathrm{v}\rangle\ \mathbf{as}\ \exists \mathrm{X}.\,\rho(\tau), \exists \mathrm{X}.\,\tau)\text{ and} \\ \exists \mathrm{R} \in \texttt{Rel}\,(\tau').\,\mathrm{v} \in \mathcal{V}\left[\!\left[\tau\right]\!\right]^{\rho, \mathrm{X}\mapsto(\tau',\mathrm{R})} \end{array} \right\}$$

$$\mathcal{E}\left[\!\left[\tau\right]\!\right]^{\rho} = \left\{ \mathrm{t} \;\middle|\; \text{if } \mathrm{t} \hookrightarrow^{*} \mathrm{v} \text{ then } \mathrm{v} \in \mathcal{V}\left[\!\left[\tau\right]\!\right]^{\rho} \right\}$$

We can then define relations on environments $\mathcal{G}\left[\!\left[\cdot\right]\!\right]$ and on type environments $\mathcal{D}\left[\!\left[\cdot\right]\!\right]$. The former accepts environments which map free variables to values in the appropriate value relation. The latter requires a type and a relation on that type for every free type variable.

$$\begin{aligned} \mathcal{G}\left[\!\left[\emptyset\right]\!\right]^{\rho} &= \{\emptyset\} \\ \mathcal{G}\left[\!\left[\Gamma, (\mathrm{x} : \tau)\right]\!\right]^{\rho} &= \left\{\gamma; [\mathrm{v}/\mathrm{x}] \;\middle|\; \gamma \in \mathcal{G}\left[\!\left[\Gamma\right]\!\right]^{\rho} \text{ and } \mathrm{v} \in \mathcal{V}\left[\!\left[\tau\right]\!\right]^{\rho}\right\} \\ \mathcal{D}\left[\!\left[\emptyset\right]\!\right] &= \{\emptyset\} \\ \mathcal{D}\left[\!\left[\Delta; \alpha\right]\!\right] &= \left\{\rho, \alpha \mapsto (\tau, \mathrm{R}) \;\middle|\; \rho \in \mathcal{D}\left[\!\left[\Delta\right]\!\right] \text{ and } \mathrm{R} = \texttt{Rel}\,(\tau)\right\} \end{aligned}$$

With these ingredients, we can define the relation $\Delta; \Gamma \Vdash \mathrm{t} : \tau$ on open terms. This relation accepts terms t which are in the term relation after appropriately closing their free variables and type variables (Definition 2.4).

Definition 2.4 (Reynolds-style Logical Relation).

$$\Delta; \Gamma \Vdash \mathrm{t} : \tau \overset{\text{def}}{=} \forall \rho \in \mathcal{D}\left[\!\left[\Delta\right]\!\right], \forall \gamma \in \mathcal{G}\left[\!\left[\Gamma\right]\!\right]^{\rho}, \mathrm{t}\gamma \in \mathcal{E}\left[\!\left[\tau\right]\!\right]^{\rho}$$

We rely on a few results for the RLR, which are listed below. The fundamental property (Theorem 2.5) states that syntactically well-typed terms are in the relation (i.e., they are *semantically* well-typed). We do not provide proofs of these lemmas as they are quite standard [Dreyer et al. 2011b].

Theorem 2.5 (Fundamental property for RLR). *if* $\Delta; \Gamma \vdash \mathrm{t} : \tau$ *then* $\Delta; \Gamma \Vdash \mathrm{t} : \tau$

Proving the fundamental property relies on a number of standard lemmas. We only mention a few which we will need in the rest of this paper.

First, we have an antireduction lemma (Lemma 2.6) which states that a term t is in the term relation if a term t' that it reduces to is.

Lemma 2.6 (Antireduction). *If* $\mathrm{t} \hookrightarrow^{*} \mathrm{t}'$ *and* $\mathrm{t}' \in \mathcal{E}\left[\!\left[\tau\right]\!\right]^{\rho}$, *then* $\mathrm{t} \in \mathcal{E}\left[\!\left[\tau\right]\!\right]^{\rho}$.

Next, we mention two compatibility lemmas: one for functions (Lemma 2.7) and one for applications (Lemma 2.8). Essentially, they state that lambdas and applications are in the logical relation if their subterms are (at appropriate types).

Lemma 2.7 (Compatibility for functions).

$$\textit{If } \Delta; \Gamma, \mathrm{x} : \tau' \Vdash \mathrm{t} : \tau \textit{ then } \Delta; \Gamma \Vdash \lambda \mathrm{x} : \tau'.\,\mathrm{t} : \tau' \to \tau$$

Lemma 2.8 (Compatibility for applications).

$$\textit{If } \Delta; \Gamma \Vdash \mathrm{t} : \tau' \to \tau \textit{ and } \Delta; \Gamma \Vdash \mathrm{t}' : \tau' \textit{ then } \Delta; \Gamma \Vdash \mathrm{t}\,\mathrm{t}' : \tau$$

We also mention the Boring lemma (Lemma 2.9)[8], which states that the semantic type relation ρ can be altered freely as long as the type variables mentioned in the type τ are left untouched.

Lemma 2.9 (Boring lemma). *If* ρ_1 *and* ρ_2 *agree on the free type variables of* τ, *then*

$$\mathcal{E}\left[\!\left[\tau\right]\!\right]^{\rho_1} = \mathcal{E}\left[\!\left[\tau\right]\!\right]^{\rho_2}$$

[8]For lack of a better name, we call this lemma as in Dreyer et al. [2011b]'s lecture notes. Neis et al. [2011] call this the *irrelevance* lemma.

Finally, it is worth mentioning that this logical relation can be easily adapted to the use of value polymorphism, discussed in Section 2.1. This requires only a change to the case for polymorphic types $\forall X.\,\tau$, which then looks as follows:

$$\mathcal{V}\,[\![\forall X.\,\tau]\!]^{\rho} = \left\{ \Lambda X.\,v \;\middle|\; \begin{array}{l} \mathtt{oftype}\,(\Lambda X.\,v, \forall X.\,\tau) \text{ and} \\ \quad \forall R \in \mathtt{Rel}\,(\tau').\,v\big[\tau'/X\big] \in \mathcal{V}\,[\![\tau]\!]^{\rho, X\mapsto(\tau',R)} \end{array} \right\}$$

The change is limited: polymorphic function bodies are now restricted to values and the instantiated bodies are now required to be in the value relation rather than the term relation, as one might expect.

2.4 Kripke Logical Relations

For languages with additional features, particularly effects like higher-order state, we can replace the basic logical relation from the previous section with a Kripke logical relation. The idea is to index the logical relation with a form of *possible worlds* or *Kripke worlds* W which represent a set of shared assumptions, often about shared state like the heap. Possible worlds are partially ordered by a relation $\sqsubseteq$, where $W' \sqsupseteq W$ expresses that W' represents a stronger set of assumptions than W. A Kripke version of the logical relation from before looks like the one below (as a notation convention, we typeset elements of Kripke LRs with a grey background, to distinguish them from elements of the RLR, which have no background). Note that we do not give a concrete definition of worlds in this section, since the goal here is just to make the reader familiar with their treatment – the next section will provide more concrete details about worlds.

$$\begin{array}{rcl}
\mathcal{V}\,[\![X]\!]^{\rho} & = & \rho(X).R \\
\mathcal{V}\,[\![\mathrm{Unit}]\!]^{\rho} & = & \{(W, \mathbf{unit})\} \\
\mathcal{V}\,[\![\mathrm{Bool}]\!]^{\rho} & = & \{(W, \mathbf{true}), (W, \mathbf{false})\} \\
\mathcal{V}\,[\![\tau \to \tau']\!]^{\rho} & = & \left\{ (W, \lambda x : \rho(\tau).\,t) \;\middle|\; \begin{array}{l} \mathtt{oftype}\,(\lambda x : \rho(\tau).\,t, \tau \to \tau') \text{ and } \forall W' \sqsupseteq W.\,\forall v. \\ \quad \text{if } (W', v) \in \mathcal{V}\,[\![\tau]\!]^{\rho} \text{ then } (W', (\lambda x : \tau.\,t)\ v) \in \mathcal{E}\,[\![\tau']\!]^{\rho} \end{array} \right\} \\
\mathcal{V}\,[\![\tau \times \tau']\!]^{\rho} & = & \left\{ (W, \langle v, v'\rangle) \;\middle|\; \begin{array}{l} \mathtt{oftype}\,(\langle v, v'\rangle, \tau \times \tau') \text{ and} \\ \quad (W, v) \in \mathcal{V}\,[\![\tau]\!]^{\rho} \text{ and } (W, v') \in \mathcal{V}\,[\![\tau']\!]^{\rho} \end{array} \right\} \\
\mathcal{V}\,[\![\forall X.\,\tau]\!]^{\rho} & = & \left\{ (W, \Lambda X.\,t) \;\middle|\; \begin{array}{l} \mathtt{oftype}\,(\Lambda X.\,t, \forall X.\,\tau) \text{ and } \forall W' \sqsupseteq W.\,\forall \tau', R \in \mathtt{Rel}\,\big[\tau'\big]. \\ \qquad (W', t[\tau'/X]) \in \mathcal{E}\,[\![\tau]\!]^{\rho, X\mapsto(\tau',R)} \end{array} \right\} \\
\mathcal{V}\,[\![\exists X.\,\tau]\!]^{\rho} & = & \left\{ (W, \mathbf{pack}\ \langle \tau', v\rangle\ \mathbf{as}\ \exists X.\,\rho(\tau)) \;\middle|\; \begin{array}{l} \mathtt{oftype}\,(\mathbf{pack}\ \langle \tau', v\rangle\ \mathbf{as}\ \exists X.\,\rho(\tau), \exists X.\,\tau) \\ \qquad \text{and } \exists R \in \mathtt{Rel}\,\big[\tau'\big]. \\ \qquad\qquad (W, v) \in \mathcal{V}\,[\![\tau]\!]^{\rho, X\mapsto(\tau',R)} \end{array} \right\} \\
\mathcal{E}\,[\![\tau]\!]^{\rho} & = & \left\{ (W, t) \;\middle|\; \forall v.\ \text{if } t \hookrightarrow^{*} v \text{ then } \exists W' \sqsupseteq W.\,(W', v) \in \mathcal{V}\,[\![\tau]\!]^{\rho} \right\} \\
\mathcal{G}\,[\![\emptyset]\!]^{\rho} & = & \{\emptyset\} \\
\mathcal{G}\,[\![\Gamma, (x : \tau)]\!]^{\rho} & = & \left\{ (W, \gamma; [v/x]) \;\middle|\; (W, \gamma) \in \mathcal{G}\,[\![\Gamma]\!]^{\rho} \text{ and } (W, v) \in \mathcal{V}\,[\![\tau]\!]^{\rho} \right\}
\end{array}$$

The differences with before are that all cases of the logical relation now mention the world $\mathbf{W}$ with respect to which values are related. Additionally, the expression relation existentially quantifies over a future world in which resulting values will be related. This generally models the fact that operational steps may introduce fresh assumptions, for example about freshly allocated mutable variables, which the resulting values' relation depends on. Finally, the value relation for function types $\tau \to \tau'$ and $\forall X.\, \tau$ is polymorphically quantified over an arbitrary world $W' \sqsupseteq W$ that extends the assumptions of the current world W. This represents the fact that these values must be valid whenever they are invoked, including when additional assumptions may have been introduced.

Again, the logical relation is easy to adapt to a value-polymorphic variant of System F, by modifying only the case for polymorphic function types:

$$\mathcal{V}\,[\![\forall X.\,\tau]\!]^{\rho} = \left\{ (W, \Lambda X.\, v) \;\middle|\; \begin{array}{r} \texttt{oftype}\,(\Lambda X.\, v, \forall X.\,\tau) \text{ and } \forall W' \sqsupseteq W.\, \forall \tau', R \in \texttt{Rel}\,[\tau']. \\ (W', v[\tau'/X]) \in \mathcal{V}\,[\![\tau]\!]^{\rho, X \mapsto (\tau', R)} \end{array} \right\}$$

As before, the instantiated function body is now required to be in the value relation rather than the term relation.

2.5 Logical Relation with Type Worlds

In this paper, we are not so interested in general effects but we focus on non-lexically-scoped type variables. For this reason, we are interested in a pattern in logical relation definitions that we dub Type World Logical Relations (TWLRs) [Ahmed et al. 2017; Neis et al. 2009; New et al. 2019a; Sumii and Pierce 2003; Toro et al. 2019]. Note that the term is not intended to denote a clearly delineated subclass of logical relations but rather a design pattern that may be used and varied upon in the definition of logical relations. A TWLR is a form of Kripke logical relation which uses Kripke worlds W to represent the interpretations of type variables instead of (or in addition to) the semantic type relations ρ. Additionally, when related polymorphic functions are instantiated with related types, the resulting terms are only related in a world that stores the relation between the applied types as the type interpretation for the instantiated type. This different treatment of type environments removes the requirement to enforce the lexical scope of type quantifiers and makes the logical relation compatible with universal types, as we will explain in Section 3.

In a TWLR, worlds are used to store the type τ and predicate R on values of type τ that are bound to a type variable, i.e., the information that was stored in ρ in the RLR (see Section 2.3). Here, we present a TWLR variant of the logical relation from Section 2.3 which uses worlds that contain types and relations for instantiated type variables. The future world relation $\sqsubseteq$ enforces that once a binding is added to a world, it can never be removed (this is true for Kripke LRs too, but in Section 2.4 we did not have to define what constitutes worlds).

$$\begin{aligned}
\text{World} \ni W &= \emptyset \mid (W; (X, \tau, R)) \\
W' \sqsupseteq W &= W' \supseteq W \\
W + (X, \tau, R) &= (W; (X, \tau, R)) \qquad \text{if } X \notin \text{dom}(W) \\
R &\in \texttt{Rel}\,[\tau] \\
\texttt{Rel}\,[\tau] &= \{ R \in \mathcal{P}(\text{World} \times \text{Val}) \mid \forall (W, v) \in R.\, \forall W' \sqsupseteq W.\, (W', v) \in R \text{ and } \emptyset; \emptyset \vdash v : \tau \}
\end{aligned}$$

The attentive reader may notice that our definition of worlds is actually cyclic: worlds map type variables to types and relations, and the relations are themselves world-indexed. As a result, the worlds presented here are not actually well-defined. Fortunately, this is a well-known problem and a standard solution exists: step-indexing [Ahmed 2004; Ahmed et al. 2009a; Dreyer et al. 2011a].

Essentially, the idea is to solve the cyclicity by indexing worlds with a number of steps which indicates up to which level they are defined. By carefully (and often tediously) ensuring that all world-indexed definitions only depend on the world up to a suitable number of steps, cyclic reasoning can be ruled out without otherwise changing the argumentation. Because step-indexing tends to complicate the technicalities while otherwise contributing little insight to the reasoning, we choose not to use it here. Instead, we simply ignore the problem in this text and stick to our illegal but comprehensible definition. To readers who wish to understand how the cyclicity can be solved without breaking the basic rules of mathematics, we recommend consulting Ahmed et al. [2017].

Having defined worlds, we can present the TWLR, which follows the same intuition of the RLR save for replacing ρ with worlds W. As before, replacing all type variables X with their bindings in W in type τ is denoted as $\mathrm{W}(\tau)$.

$$
\begin{aligned}
\mathcal{V}\,[\![\mathrm{X}]\!] &= \{(\mathrm{W},\mathrm{v}) \mid (\mathrm{W},\mathrm{v}) \in \mathrm{W}(\mathrm{X}).\mathrm{R}\}\\
\mathcal{V}\,[\![\mathbf{Unit}]\!] &= \{(\mathrm{W},\mathbf{unit})\}\\
\mathcal{V}\,[\![\mathbf{Bool}]\!] &= \{(\mathrm{W},\mathbf{true}),(\mathrm{W},\mathbf{false})\}\\
\mathcal{V}\,[\![\tau\to\tau']\!] &= \left\{(\mathrm{W},\lambda\mathrm{x}:\mathrm{W}(\tau).\,\mathrm{t}) \;\middle|\; \begin{array}{l}\mathtt{oftype}\,(\lambda\mathrm{x}:\mathrm{W}(\tau).\,\mathrm{t},\tau\to\tau') \text{ and } \forall\mathrm{W}'\sqsupseteq\mathrm{W}.\,\forall\mathrm{v}.\\ \quad\text{if } (\mathrm{W}',\mathrm{v})\in\mathcal{V}\,[\![\tau]\!] \text{ then } (\mathrm{W}',(\lambda\mathrm{x}:\tau.\,\mathrm{t})\ \mathrm{v})\in\mathcal{E}\,[\![\tau']\!]\end{array}\right\}\\
\mathcal{V}\,[\![\tau\times\tau']\!] &= \left\{(\mathrm{W},\langle\mathrm{v},\mathrm{v}'\rangle) \;\middle|\; \begin{array}{l}\mathtt{oftype}\,(\langle\mathrm{v},\mathrm{v}'\rangle,\tau\times\tau') \text{ and}\\ \quad(\mathrm{W},\mathrm{v})\in\mathcal{V}\,[\![\tau]\!] \text{ and } (\mathrm{W},\mathrm{v}')\in\mathcal{V}\,[\![\tau']\!]\end{array}\right\}\\
\mathcal{V}\,[\![\forall\mathrm{X}.\,\tau]\!] &= \left\{(\mathrm{W},\Lambda\mathrm{X}.\,\mathrm{t}) \;\middle|\; \begin{array}{r}\mathtt{oftype}\,(\Lambda\mathrm{X}.\,\mathrm{t},\forall\mathrm{X}.\,\tau) \text{ and } \forall\mathrm{W}'\sqsupseteq\mathrm{W}.\,\forall\tau',\mathrm{R}\in\mathtt{Rel}\,[\tau'].\\ (\mathrm{W}'+(\mathrm{X},\tau',\mathrm{R}),\mathrm{t}[\tau'/\mathrm{X}])\in\mathcal{E}\,[\![\tau]\!]\end{array}\right\}\\
\mathcal{V}\,[\![\exists\mathrm{X}.\,\tau]\!] &= \left\{(\mathrm{W},\mathbf{pack}\ \langle\tau',\mathrm{v}\rangle\ \mathbf{as}\ \exists\mathrm{X}.\,\mathrm{W}(\tau)) \;\middle|\; \begin{array}{r}\mathtt{oftype}\,(\mathbf{pack}\ \langle\tau',\mathrm{v}\rangle\ \mathbf{as}\ \exists\mathrm{X}.\,\mathrm{W}(\tau),\exists\mathrm{X}.\,\tau)\\ \text{and } \exists\mathrm{R}\in\mathtt{Rel}\,[\tau'].\\ (\mathrm{W}+(\mathrm{X},\tau',\mathrm{R}),\mathrm{v})\in\mathcal{V}\,[\![\tau]\!]\end{array}\right\}\\
\mathcal{E}\,[\![\tau]\!] &= \left\{(\mathrm{W},\mathrm{t}) \;\middle|\; \forall\mathrm{v}.\ \text{if } \mathrm{t}\hookrightarrow^{*}\mathrm{v} \text{ then } \exists\mathrm{W}'\sqsupseteq\mathrm{W}.\,(\mathrm{W}',\mathrm{v})\in\mathcal{V}\,[\![\tau]\!]\right\}\\
\mathcal{G}\,[\![\emptyset]\!] &= \{\emptyset\}\\
\mathcal{G}\,[\![\Gamma,(\mathrm{x}:\tau)]\!] &= \left\{(\mathrm{W},\gamma;[\mathrm{v}/\mathrm{x}]) \;\middle|\; (\mathrm{W},\gamma)\in\mathcal{G}\,[\![\Gamma]\!] \text{ and } (\mathrm{W},\mathrm{v})\in\mathcal{V}\,[\![\tau]\!]\right\}
\end{aligned}
$$

It is crucial to compare the case for $\mathcal{V}\,[\![\forall\mathrm{X}.\,\tau]\!]$ against the one from Section 2.3. Contrary to there, the instantiated term $\mathrm{t}[\tau'/\mathrm{X}]$ is only required to be related in worlds $\mathrm{W}'+(\mathrm{X},\tau',\mathrm{R})$ that store the relation R as the semantic interpretation for the instantiated type variable X.

The last piece we need to formalise is what it means for a world W to agree with a type environment Δ, which we denote with $\mathrm{W}\vdash\Delta$. This intuitively replaces the type environment relation $\mathcal{D}\,[\![\cdot]\!]$ present in RLR. A world agrees with a type environment when it maps all the type variables of the latter to valid relations, formally:

$$\emptyset \vdash \emptyset$$

$$\mathrm{W}, (\mathrm{X}, \tau, \mathrm{R}) \vdash \Delta, \mathrm{X} \text{ if } \mathrm{W} \vdash \Delta \text{ and } \mathrm{R} \in \mathtt{Rel}\,[\tau]$$

With these ingredients we can define our Type-World Logical Relation.

Definition 2.10 (Type-World Logical Relation).

$$\Delta; \Gamma \vDash \mathrm{t} : \tau \overset{\text{def}}{=} \forall\, \mathrm{W} \vdash \Delta, \forall\, \gamma \in \mathcal{G}\,[\![\Gamma]\!]\,,\, (\mathrm{W}, \mathrm{t}\gamma) \in \mathcal{E}\,[\![\tau]\!]$$

The fundamental property of TWLRs is analogous to that for RLR. As in the previous section, we do not provide proofs of these lemmas, which can be derived from similar (binary) statements in the works of e.g., Ahmed et al. [2017]; Neis et al. [2009].

Theorem 2.11 (Fundamental property for TWLR). *if* $\Delta; \Gamma \vdash \mathrm{t} : \tau$ *then* $\Delta; \Gamma \vDash \mathrm{t} : \tau$

The main result we need from this logical relation is an analogous to Lemma 2.9 (Boring lemma). If we try to naively state such a result for this TWLR too we obtain the wrong statement below (Lemma 2.12).

Lemma 2.12 (**Wrong** Boring lemma for TWLR). *If* W_1 *and* W_2 *agree on the free type variables of* τ*, then*

$$(\mathrm{W}_1, \mathrm{t}) \in \mathcal{V}\,[\![\tau]\!] \iff (\mathrm{W}_2, \mathrm{t}) \in \mathcal{V}\,[\![\tau]\!]$$

Contrary to the RLR, this lemma does not hold for our TWLR. If we were to try and prove it, we would quickly get stuck in the cases for lambdas and big lambdas where we get a future world W' of, for example, W_1 but we have no way to relate it to world W_2.

The correct statement of Lemma 2.9 (Boring lemma) applied to the TWLR is the following one, which respects world monotonicity.

Lemma 2.13 (Boring lemma for TWLR). *If* $\mathrm{W}_2 \sqsupseteq \mathrm{W}_1$*, then,* $\forall \tau, \mathrm{v}$

$$\text{if } (\mathrm{W}_1, \mathrm{v}) \in \mathcal{V}\,[\![\tau]\!] \text{ then } (\mathrm{W}_2, \mathrm{v}) \in \mathcal{V}\,[\![\tau]\!]$$

The premise has changed in this lemma, in fact we are only allowed to *extend* a world, but not remove bindings from it.

The difference between the RLR of Section 2.3 and the TWLR of this section may appear technical. However, we will see in the next section that this technical change from RLR to TWLR renders a formulation of parametricity compatible with universal types. This fact is witnessed by several TWLR-based parametricity proofs for languages with such types [Ahmed et al. 2011c, 2017; Neis et al. 2009, 2011; New et al. 2019a; Sumii and Pierce 2003; Toro et al. 2019]. However, none of those papers observed that using a TWLR reduced the set of equivalences that the LR implies and that this reduction was necessary in the presence of non-lexically-scoped type variables.

Before concluding, it is again interesting to consider what our TWLR would look like for a value-polymorphic language:

$$\mathcal{V}\,[\![\forall \mathrm{X}.\,\tau]\!] = \left\{ (\mathrm{W}, \Lambda \mathrm{X}.\,\mathrm{v}) \;\middle|\; \begin{array}{r} \mathtt{oftype}\,(\Lambda \mathrm{X}.\,\mathrm{v}, \forall \mathrm{X}.\,\tau) \text{ and } \forall \mathrm{W}' \sqsupseteq \mathrm{W}.\, \forall \tau', \mathrm{R} \in \mathtt{Rel}\,[\tau'].\\ (\mathrm{W}' + (\mathrm{X}, \tau', \mathrm{R}), \mathrm{v}[\tau'/\mathrm{X}]) \in \mathcal{V}\,[\![\tau]\!] \end{array} \right\}$$

Observe that here, the instantiated body is still required to be related by the value relation and thus not able to allocate additional assumptions. This makes sense, since the body cannot allocate, for example, fresh mutable heap locations. However, one particular assumption is still added to the world in which the body is related: the instantiated of $\mathbf{X}$ to type τ' and relation $\mathbf{R}$.

In the next Sections, we provide further insight into how the use of a TWLR underlines a compatibility with the universal type in the language. We do this by demonstrating the degeneracy of the universal type with LRLS and explaining why such a proof fails using a TWLR. Before showing these technical proofs, we must define the universal type.

3 THE TYPE $\underline{\mathbf{Univ}}$

As mentioned, in this paper we rely heavily on the type $\underline{\mathbf{Univ}}$:

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

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

In our non-terminating variant of λ^{F}, $\underline{\mathbf{Univ}}$ is clearly inhabited, for example by this value (recall that $\omega_{\mathbf{X}}$ is a diverging term of type $\mathbf{X}$):

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

However, this value is *degenerate* in the sense that injecting a value into the packaged $\mathbf{Y}$ and extracting it again diverges. Note that it is not necessarily the function of type $\mathbf{Y} \to \mathbf{X}$ that diverges. For example, we can construct the following inhabitants:

$$\mathbf{pack}\ \langle (\forall \mathbf{Z}.\, \mathbf{Z}), \Lambda\mathbf{X}.\, \langle \lambda\_ : \mathbf{X}.\, \omega_{\forall \mathbf{Z}.\mathbf{Z}}, \lambda \mathbf{y} : (\forall \mathbf{Z}.\, \mathbf{Z}).\, \mathbf{y}\ \mathbf{X} \rangle \rangle\ \mathbf{as}\ \underline{\mathbf{Univ}}$$
$$\mathbf{pack}\ \langle (\forall \mathbf{Z}.\, \mathbf{Z}), \Lambda\mathbf{X}.\, \langle \lambda\_ : \mathbf{X}.\, \Lambda\mathbf{Z}.\, \omega_{\mathbf{Z}}, \lambda \mathbf{y} : (\forall \mathbf{Z}.\, \mathbf{Z}).\, \mathbf{y}\ \mathbf{X} \rangle \rangle\ \mathbf{as}\ \underline{\mathbf{Univ}}$$

These other inhabitants instantiate $\mathbf{Y}$ to $\forall \mathbf{Z}.\, \mathbf{Z}$ and make the function of type $\mathbf{Y} \to \mathbf{X}$ simply use the received value of type $\forall \mathbf{Z}.\, \mathbf{Z}$ to obtain the required value of type $\mathbf{X}$. It is now the function of type $\mathbf{X} \to \mathbf{Y}$ that diverges, either directly or after being applied to a type $\mathbf{Z}$.

A crucial observation for this paper is that *all* System F values of type $\underline{\mathbf{Univ}}$ are degenerate in the above sense. Intuitively, this is because a single type $\mathbf{Y}$ needs to be chosen, independently of the types $\mathbf{X}$ that will be embedded into it. Because nothing is known upfront about these $\mathbf{X}$s and nothing can be learnt about them after invocation (because $\mathbf{X}$ must be treated parametrically), no viable choice for $\mathbf{Y}$ can be made.[9]

From another perspective, the degeneracy results from the lexical scope of type variable $\mathbf{X}$ in the polymorphic function of type $\forall \mathbf{X}.\, (\mathbf{X} \to \mathbf{Y}) \times (\mathbf{Y} \to \mathbf{X})$. In λ^{F}, implementations of this function are required to respect this lexical scope and importantly, the variable is not in scope at the point where a choice for existential variable $\mathbf{Y}$ needs to be made. A non-degenerate implementation of $\underline{\mathbf{Univ}}$ essentially must somehow pass a value of type $\mathbf{X}$ as the result of the function of type $\mathbf{X} \to \mathbf{Y}$ and recover it from the argument to the function of type $\mathbf{Y} \to \mathbf{X}$. This would require that value to survive exiting and reentering the lexical scope of type variable $\mathbf{X}$, and degeneracy expresses exactly the impossibility of this, i.e. λ^{F}'s respect for the lexical scope of type variables like $\mathbf{X}$.

Note that in this paper, lexical scoping of type variables in System F is regarded as an informal property. Our best attempt to characterize it uses the degeneracy of $\underline{\mathbf{Univ}}$ and the existence of a universal type as a sufficient criterion.

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

3.1 Two Contextually Equivalent Terms

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

$$\begin{aligned}
\mathrm{t_u} &\overset{\mathrm{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\,\mathrm{unpack\ x\ as\ } \langle \mathrm{Y}, \mathrm{x'} \rangle \mathrm{\ in} \\
&\quad \mathrm{let\ x''} : (\mathrm{Unit} \to \mathrm{Y}) \times (\mathrm{Y} \to \mathrm{Unit}) = \mathrm{x'\ Unit\ in\ x''.2\ (x''.1\ unit)} \\
\mathrm{t}_\omega &\overset{\mathrm{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\,\omega_{\mathrm{Unit}}
\end{aligned}$$

The reason that $\mathrm{t_u}$ and t_ω are contextually equivalent is that both will diverge when applied to any argument of type $\underline{\mathrm{Univ}}$. For $\mathrm{t_u}$, this follows from the degeneracy of the type $\underline{\mathrm{Univ}}$, as we demonstrate below, while for t_ω, the term ω_{Unit} in the body ensures divergence. Note that the degeneracy of $\underline{\mathrm{Univ}}$ is essential: if the context were able to produce a non-degenerate value of type $\underline{\mathrm{Univ}}$, then $\mathrm{t_u}$ would not diverge when applied to it, so that the context could distinguish $\mathrm{t_u}$ from t_ω.
Thus, we have the following theorem.

Theorem 3.1 ($\mathrm{t_u}$ and t_ω are contextually equivalent in λ^{F}). $\emptyset; \emptyset \vdash \mathrm{t_u} \simeq \mathrm{t}_\omega : \underline{\mathrm{Univ}} \to \mathrm{Unit}$.

Note that we have chosen $\mathrm{t_u}$ and t_ω because their equivalence is relatively easy to prove. However, we could have taken many other equivalences which follow from the degeneracy of $\underline{\mathrm{Univ}}$ and particularly, we could have taken example terms which will both terminate with different results in the presence of a non-degenerate universal type:

$$\begin{aligned}
\mathrm{t_1} &\overset{\mathrm{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\,\mathrm{unpack\ x\ as\ } \langle \mathrm{Y}, \mathrm{x'} \rangle \mathrm{\ in} \\
&\quad \mathrm{let\ x''} : (\mathrm{Unit} \to \mathrm{Y}) \times (\mathrm{Y} \to \mathrm{Unit}) = \mathrm{x'\ Unit\ in\ x''.2\ (x''.1\ unit)}; 1 \\
\mathrm{t_2} &\overset{\mathrm{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\,\mathrm{unpack\ x\ as\ } \langle \mathrm{Y}, \mathrm{x'} \rangle \mathrm{\ in} \\
&\quad \mathrm{let\ x''} : (\mathrm{Unit} \to \mathrm{Y}) \times (\mathrm{Y} \to \mathrm{Unit}) = \mathrm{x'\ Unit\ in\ x''.2\ (x''.1\ unit)}; 2
\end{aligned}$$

3.2 $\underline{\mathrm{Univ}}$ in Other Settings

Before we look at proving Theorem 3.1, it is useful to build a better intuition of the meaning and the cause of the degeneracy of $\underline{\mathrm{Univ}}$. To this end, it is useful to consider the type's properties in variants of System F.

For example, if we imagine a variant of System F with errors or exceptions, converting a function of type X to Y and back, must no longer necessarily diverge, but can now also result in an error or an exception. Intuitively, this is perhaps still a form of degeneracy, but formally, it is no longer true that $\mathrm{t_u} \simeq \mathrm{t}_\omega$ and thus, the two terms no longer form a counterexample to full abstraction. However, while the concrete counterexample is broken, the more general observation remains: System F implies properties about the type $\underline{\mathrm{Univ}}$ that are no longer true in the languages we discuss in the next sections, even when both languages are extended with errors or exceptions. Concretely, we can easily adapt the counterexample by making t_ω not always diverge, but diverge after invoking the two functions of the $\underline{\mathrm{Univ}}$ value.

$$\begin{aligned}
\mathrm{t_u} &\overset{\mathrm{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\,\mathrm{unpack\ x\ as\ } \langle \mathrm{Y}, \mathrm{x'} \rangle \mathrm{\ in} \\
&\quad \mathrm{let\ x''} : (\mathrm{Unit} \to \mathrm{Y}) \times (\mathrm{Y} \to \mathrm{Unit}) = \mathrm{x'\ Unit\ in\ x''.2\ (x''.1\ unit)} \\
\mathrm{t}'_\omega &\overset{\mathrm{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\,\mathrm{unpack\ x\ as\ } \langle \mathrm{Y}, \mathrm{x'} \rangle \mathrm{\ in} \\
&\quad \mathrm{let\ x''} : (\mathrm{Unit} \to \mathrm{Y}) \times (\mathrm{Y} \to \mathrm{Unit}) = \mathrm{x'\ Unit\ in\ x''.2\ (x''.1\ unit)}; \omega_{\mathrm{Unit}}
\end{aligned}$$

Our results should not be interpreted as strictly related to the specific example terms $\mathrm{t_u}$ and t_ω and not even to the type $\underline{\mathrm{Univ}}$. It appears clear at least that $\underline{\mathrm{Univ}}$ is not the only type for which an

RLR implies properties that do not follow from a TWLR (see Section 4). For example, it appears clear that similar properties will hold for a variant of Univ with one component duplicated:

$$\exists Y.\ \forall X.\ (X \to Y) \times (X \to Y) \times (Y \to X)$$

Our modified example shows that the phenomenon also manifests itself in the presence of effects like errors and exceptions.

It is hard to identify a root cause for the phenomenon: some of the settings we consider feature a form of type generativity, all feature a universal type, but it is hard to say whether these are symptom or disease. The use of TWLRs suggests (at least to us) that it is essentially the lexical scoping of type variables in System F which is not enforced in the different settings we consider. However, as mentioned before, we don't know how to formalize this intuitive property or argue that it is the root cause.

Extending System F with ML-like references breaks the counterexample in a seemingly more severe way. Concretely, consider extending System F in a standard way with types $\mathbf{Ref}\ \tau$ for heap references and constructs for allocating ($\mathbf{ref}_\tau\ v$), assigning ($x := v$) and dereferencing ($!x$) heap variables), see, e.g., Pierce [2002]. In this case, the type Univ gains inhabitants like the following:

$$\mathbf{pack}\ \left\langle \mathbf{Unit}, \Lambda X.\ \begin{array}{l} \mathbf{let}\ r : \mathbf{Ref}\ (\mathbf{Maybe}\ X) = \mathbf{ref}_X\ \mathbf{None}\ \mathbf{in} \\ \quad \langle \lambda x : X.\ r := \mathbf{Some}\ x; \mathbf{unit}, \lambda\_ : \mathbf{Unit}.\ !r \rangle \end{array} \right\rangle\ \mathbf{as}\ \mathrm{Univ}$$

Rather than attempting to store values of type X in an appropriately chosen Y in some way, this inhabitant instantiates Y to the non-informative type Unit. Instead, it allocates a mutable variable of type X, shared between the two functions. The first function then stores its argument in the heap variable for the second function to retrieve.

The existence of this inhabitant shows that ML references break our results as well and in this case, it is not as obvious how to recover equivalences that rely on lexical scoping of type variables. One might interpret this to mean that the degeneracy of Univ is an artifact of the purity of System F, and that it only holds in the absence of effects like mutable state. Perhaps what we call non-lexically-scoped type variables should simply be interpreted as impurity of polymorphic function applications and as such, not different from other effects?

We think it is not so simple and one way to see this is to consider what happens with the above example in a value-polymorphic language. Interestingly, value polymorphism breaks the Univ inhabitant mentioned above: it is no longer possible to allocate a reference cell that is shared by the functions from $X \to Y$ and $Y \to X$. In fact, we expect degeneracy of Univ to hold in value-polymorphic variants of System F, even in the presence of ML references or other effects.

Nevertheless, it will be clear in Sections 5 to 7 that value polymorphism does not similarly break the inhabitants of Univ we discuss there. This demonstrates that even if one interprets the non-lexical scope of type variables in those languages as a form of effect, it is at least a form of effect that behaves rather differently than other effects, as it does not disappear from polymorphic functions with the introduction of value-polymorphism. From that point of view, we think our results remain relevant in the presence of effects, although value polymorphism appears essential when those effects can be used to create a communication channel for transmitting values of type X between the two functions in Univ (as discussed for ML references).

4 PROVING DEGENERACY OF Univ

This section attempts to prove Theorem 3.1 (the contextual equivalence of t_u and t_ω which we recap below) using both the RLR and the TWLR and in doing so, it focusses on the key result for this proof: showing that Univ is degenerate. This result is doable with the RLR (from Section 2.3), highlighting why they are incompatible with non-lexically-scoped type variables but the same fact cannot be shown with the TWLR (from Section 2.5).

$$\begin{aligned}\mathrm{t_u} &\overset{\text{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\ \mathbf{unpack}\ \mathrm{x}\ \mathbf{as}\ \langle \mathrm{Y}, \mathrm{x}' \rangle\ \mathbf{in}\\ &\quad\ \mathbf{let}\ \mathrm{x}'' : (\mathrm{Unit} \to \mathrm{Y}) \times (\mathrm{Y} \to \mathrm{Unit}) = \mathrm{x}'\ \mathrm{Unit}\ \mathbf{in}\ \mathrm{x}''.2\ (\mathrm{x}''.1\ \mathrm{unit})\\ \mathrm{t}_\omega &\overset{\text{def}}{=} \lambda \mathrm{x} : \underline{\mathrm{Univ}}.\ \omega_{\mathrm{Unit}}\end{aligned}$$

The first step of the proof of Theorem 3.1 is captured by Lemma 4.1 (Diverging functions are contextually equivalent to an omega function), which states that a function that diverges for every argument is equivalent to a function whose body is the omega term.

Lemma 4.1 (Diverging functions are contextually equivalent to an omega function).

$$\textit{If } \emptyset;\emptyset \vdash \lambda \mathrm{x} : \tau'.\,\mathrm{t} : \tau' \to \mathrm{Unit} \textit{ and } (\textit{for all } \emptyset;\emptyset \vdash \mathrm{v} : \tau' \textit{ we have that } (\lambda \mathrm{x} : \tau'.\,\mathrm{t})\ \mathrm{v}\Uparrow)$$
$$\textit{Then } \emptyset;\emptyset \vdash \lambda \mathrm{x} : \tau'.\,\mathrm{t} \simeq \lambda \mathrm{x} : \tau'.\,\omega_{\mathrm{Unit}} : \tau' \to \mathrm{Unit}$$

We do not think this lemma is very hard to believe. It can be proven using standard techniques, either using an ad hoc simulation argument or by relying on existing binary logical relations for System F, like the one by Dreyer et al. [2011a]. To avoid distracting from the main point of our paper, we do not offer a proof here.

With Lemma 4.1, it suffices to prove that $\mathrm{t_u}$ always diverges when supplied with a value of type $\underline{\mathrm{Univ}}$ in order to conclude Theorem 3.1. This result we derive from the "degeneracy" of $\underline{\mathrm{Univ}}$, i.e., from the fact that $\underline{\mathrm{Univ}}$ is only inhabited by diverging terms. We show this can be proven with the aid of the RLR in Lemma 4.2 below (Section 4.1) and also show that this is not provable with the TWLR (Section 4.2). We then discuss how this proof is carried out with other kinds of LR (Section 4.3).

4.1 $\underline{\mathrm{Univ}}$ Degeneracy Using an RLR

Lemma 4.2 ($\underline{\mathrm{Univ}}$ is degenerate). *For all* $\emptyset;\emptyset \vdash \mathrm{v} : \underline{\mathrm{Univ}}$, *we have that* $\mathrm{t_u}\ \mathrm{v}\Uparrow$.

Proof. The proof proceeds largely in a standard way: we unfold the definition of value relation for the value of universal type and we rely on antireduction and compatibility lemmas in order to reason about terms after they evaluate. The goal is to show divergence of term $\mathrm{t_u}\ \mathrm{v}$. This can be achieved by showing that that term belongs to the term relation for a type variable whose set of inhabitants is empty. Specifically, we will prove that it belongs to $\mathcal{E}\,[\![\mathrm{X}]\!]^{\cdots,\mathrm{X}\mapsto(\mathrm{Unit},\emptyset)}$. By definition this means that the term must diverge or reduce to a value in $\mathcal{V}\,[\![\mathrm{X}]\!]^{\cdots,\mathrm{X}\mapsto(\mathrm{Unit},\emptyset)}$. Since that value relation is empty, the latter is not possible, so $\mathrm{t_u}\ \mathrm{v}$ must diverge.

To prove that $\mathrm{t_u}\ \mathrm{v}$ is in $\mathcal{E}\,[\![\mathrm{X}]\!]^{\cdots,\mathrm{X}\mapsto(\mathrm{Unit},\emptyset)}$, the main trick we use relies on having two relations for X to inhabit. We will then instantiate the quantification over X with two different semantic interpretations.

(1) the first one $\mathrm{RU} = \{\mathrm{unit}\}$;
(2) the second one $\mathrm{RE} = \emptyset$.

Now take $\emptyset;\emptyset \vdash \mathrm{v} : \underline{\mathrm{Univ}}$. By Theorem 2.5 (Fundamental property for RLR) with $\emptyset;\emptyset \vdash \mathrm{v} : \underline{\mathrm{Univ}}$, we have (HLR)[10] $\emptyset;\emptyset \Vdash \mathrm{v} : \underline{\mathrm{Univ}}$. By Definition 2.4 (Reynolds-style Logical Relation) with HLR (taking $\rho = \emptyset$ and $\gamma = \emptyset$), we have $\mathrm{v} \in \mathcal{E}\,[\![\underline{\mathrm{Univ}}]\!]^{\emptyset}$. Because v is a value, we have $\mathrm{v} \in \mathcal{V}\,[\![\underline{\mathrm{Univ}}]\!]^{\emptyset}$.

By unfolding the definition of $\underline{\mathrm{Univ}}$ and the value relation of the related type(s), it follows for some τ_{Y}, v' and $\mathrm{R_Y} \in \mathtt{Rel}\,(\tau_{\mathrm{Y}})$, that

[10]For ease of discussion, we use this notation to give names to hypotheses as they become available during the proof and use the name later in the proof when we use the hypothesis.

- $v = \text{pack } \langle \tau_Y, v' \rangle \text{ as } \underline{\text{Univ}}$
- (HPVP) $v' \in \mathcal{V}\,[\![\forall X.\,(X \to Y) \times (Y \to X)]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y)}$.

Let us now consider the reductions of $t_u\ v$:

$$\begin{array}{l}
t_u\ v \\
\equiv \left| \begin{array}{l} (\lambda x : \underline{\text{Univ}}.\ \text{unpack } x \text{ as } \langle Y, x' \rangle \text{ in} \\ \text{let } x'' : (\text{Unit} \to Y) \times (Y \to \text{Unit}) = x'\ \text{Unit in} \\ \qquad\qquad\qquad x''.2\ (x''.1\ \text{unit}))\ v \end{array} \right. \\
\hookrightarrow \text{unpack } v \text{ as } \langle Y, x' \rangle \text{ in let } x'' : (\text{Unit} \to Y) \times (Y \to \text{Unit}) = x'\ \text{Unit in } x''.2\ (x''.1\ \text{unit}) \\
\equiv \left| \begin{array}{l} \text{unpack } (\text{pack } \langle \tau_Y, v' \rangle \text{ as } \underline{\text{Univ}}) \text{ as } \langle Y, x' \rangle \text{ in} \\ \text{let } x'' : (\text{Unit} \to Y) \times (Y \to \text{Unit}) = x'\ \text{Unit in} \\ \qquad\qquad\qquad x''.2\ (x''.1\ \text{unit}) \end{array} \right. \\
\hookrightarrow \text{let } x'' : (\text{Unit} \to \tau_Y) \times (\tau_Y \to \text{Unit}) = v'\ \text{Unit in } x''.2\ (x''.1\ \text{unit})
\end{array}$$

By Lemma 2.6 (Antireduction) and Lemma 2.9 (Boring lemma), to conclude our thesis ($t_u\ v \in \mathcal{E}\,[\![X]\!]^{\emptyset, X \mapsto (\text{Unit}, \text{RE})}$) it is sufficient to show that:

$$\text{let } x'' : (\text{Unit} \to \tau_Y) \times (\tau_Y \to \text{Unit}) = v'\ \text{Unit in } x''.2\ (x''.1\ \text{unit}) \in \mathcal{E}\,[\![X]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$$

Now assume that the term

$$\text{let } x'' : (\text{Unit} \to \tau_Y) \times (\tau_Y \to \text{Unit}) = v'\ \text{Unit in } x''.2\ (x''.1\ \text{unit})$$

terminates to a value v_r. By definition of the term relation, it suffices to prove that

$$v_r \in \mathcal{V}\,[\![X]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$$

It is easy to see that there must be an intermediate value v_U such that $v'\ \text{Unit} \hookrightarrow^* v_U$ and

$$\text{let } x'' : (\text{Unit} \to \tau_Y) \times (\tau_Y \to \text{Unit}) = v_U \text{ in } x''.2\ (x''.1\ \text{unit}) \hookrightarrow^* v_r$$

From HPVP, we know that

$$v'\ \text{Unit} \in \mathcal{E}\,[\![X \to Y \times Y \to X]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$$

By definition of the term relation, we have that (HPVU) $v_U \in \mathcal{V}\,[\![X \to Y \times Y \to X]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$. It will be useful later that the same property holds if we replace RE with RU (HPVU2).

By definition of the term relation, it suffices to prove that

$$v_U.2\ (v_U.1\ \text{unit}) \in \mathcal{E}\,[\![X]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$$

This is a top-level application, so by Lemma 2.8 (Compatibility for applications), it suffices to prove the following (recall that $\text{RE} = \emptyset$):

(1) $v_U.2 \in \mathcal{E}\,[\![Y \to X]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$
(2) $v_U.1\ \text{unit} \in \mathcal{E}\,[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})}$

The former follows easily from HPVU by unfolding the definition of the value relation for pairs.

To prove Item 2 we apply our main trick. By Lemma 2.9 (Boring lemma), we can replace the relation with an equivalent one. We first drop the first relation for X in the term relation for Y (since variable X is not mentioned in type Y), then add the second relation and all the term relations are equivalent:

$$\mathcal{E}\,[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RE})} = \mathcal{E}\,[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y)} = \mathcal{E}\,[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\text{Unit}, \text{RU})}$$

So instead of proving:

$$\mathbf{v_U.1\ unit} \in \mathcal{E}\left[\!\left[\mathbf{Y}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RE})}$$

we can just prove (notice the change in the semantic interpretation for $\mathbf{X}$):

$$\mathbf{v_U.1\ unit} \in \mathcal{E}\left[\!\left[\mathbf{Y}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RU})}$$

This is necessary in order to reason about the function application within this term, as we explain below.

Again, we apply Lemma 2.8 (Compatibility for applications) and it suffices to prove the following:

(3) $\mathbf{v_U.1} \in \mathcal{E}\left[\!\left[\mathbf{X}\rightarrow\mathbf{Y}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RU})}$

(4) $\mathbf{unit} \in \mathcal{E}\left[\!\left[\mathbf{X}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RU})}$

Similarly to how we derived HPVU from HPVP and the definition of the term relation, Item 3 follows from the fact that $\mathbf{v_U} \in \mathcal{V}\left[\!\left[\mathbf{X}\rightarrow\mathbf{Y}\times\mathbf{Y}\rightarrow\mathbf{X}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RU})}$ (HPVU2) and then unfolding the definition of the value relation for pairs.

Proving Item 4 is not hard either. The term relation includes the value relation, so it suffices to prove:

$$\mathbf{unit} \in \mathcal{V}\left[\!\left[\mathbf{X}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RU})}$$

By definition of the value relation for type variables, this holds if $\mathbf{unit}$ inhabits the semantic interpretation for $\mathbf{X}$, i.e., $\mathbf{RU} = \{\mathbf{unit}\}$. □

If we had not performed our main trick, all other proof obligations would hold, but proving Item 4 would not be possible. In fact, there we would have had to prove that:

$$\mathbf{unit} \in \mathcal{V}\left[\!\left[\mathbf{X}\right]\!\right]^{\emptyset,\mathbf{Y}\mapsto(\tau_\mathbf{Y},\mathbf{R_Y}),\mathbf{X}\mapsto(\mathbf{Unit},\mathbf{RE})}$$

i.e., (according to the value relation for type variables) that $\mathbf{unit}$ is in the current semantic interpretation for $\mathbf{X}$, i.e., $\emptyset$, which is not possible.

4.2 Why Universal Types Require Type Worlds

The proof from Section 3.1 shows that Reynolds-style or lexically-scoped logical relations are incompatible with a universal type. But to thoroughly understand what is going on, it is useful to take a closer look at the counterexample.

Recall the definition of the type $\underline{\mathbf{Univ}}$.

$$\underline{\mathbf{Univ}} \overset{\text{def}}{=} \exists\mathbf{Y}.\,\forall\mathbf{X}.\,(\mathbf{X}\rightarrow\mathbf{Y})\times(\mathbf{Y}\rightarrow\mathbf{X})$$

According to the RLR, any value of this type is of the form $\mathbf{pack}\ \langle\tau_\mathbf{Y},\mathbf{v}\rangle\ \mathbf{as}\ \underline{\mathbf{Univ}}$ and comes with a predicate $\mathbf{R_Y} \in \mathtt{Rel}\ (\tau_\mathbf{Y})$, i.e., a predicate on values of type $\tau_\mathbf{Y}$. Importantly, this predicate needs to be chosen independently of choices that are made later, particularly the choice of the type $\mathbf{Unit}$ for $\mathbf{X}$ and the predicate $\mathbf{R_X} \in \mathtt{Rel}\ (\mathbf{Unit})$ which the universal quantification will be instantiated with.

It is this independence (of the choice of $\mathbf{R_Y}$) that is fundamentally incompatible with the existence of a universal type. Imagine there were a universal type $\mathbf{U}$ in System F with total injection and extraction functions $\mathbf{in_Z} : \mathbf{Z} \longrightarrow \mathbf{U}$ and $\mathbf{out_Z} : \mathbf{U} \longrightarrow \mathbf{Z}$ for arbitrary types $\mathbf{Z}$. Then we could define a value of type $\underline{\mathbf{Univ}}$ by taking $\mathbf{Y} = \mathbf{U}$ and constructing a pair with $\mathbf{in_X}$ and $\mathbf{out_X}$:

$$\mathbf{pack}\ \langle\mathbf{U},\Lambda\mathbf{X}.\,\langle\mathbf{in_X},\mathbf{out_X}\rangle\rangle\ \mathbf{as}\ \underline{\mathbf{Univ}}$$

Now, to make a choice for the predicate $\mathbf{R_U}$, one thing to decide is whether or not the predicate should accept the result of $\mathbf{in_X\ unit}$ when $\mathbf{X}$ is later instantiated to $\mathbf{Unit}$ (as in the counterexample). However, whether or not this value can be legally embedded in $\mathbf{U}$ fundamentally depends on the choice for $\mathbf{R_X} \in \mathtt{Rel}\ (\mathbf{Unit})$. Particularly, if $\mathbf{R_X} = \emptyset$, then it should be rejected, but if $\mathbf{R_X} = \{\mathbf{unit}\}$

then it should be accepted. Clearly, this is impossible if we need to choose R_Y before we know what R_X will be instantiated with.

So how do type-worlds fit into this picture? In a TWLR like that of Section 2.5, we still need to make a choice for R_Y before a choice is made for R_X. However, the type of R_Y is different now: $R_Y \in \mathcal{P}(\mathbf{World} \times \mathbf{Val})$ with

$$\forall (W, v) \in R. \forall W' \sqsupseteq W.\ (W', v) \in R \text{ and } \emptyset; \emptyset \vdash v : \tau_Y$$

Another way to think of this set is as $\mathbf{World} \xrightarrow{mon} \mathcal{P}(\mathbf{Val})$: the set of monotone functions from **World** to sets of values. In other words, R_Y is now a family of relations, and can decide which values to accept based on the current type world W. This world W will contain choices of predicates for other type variables, particularly the choice for R_X. In each of the TWLRs where a universal type is at play, the logical relation will accept different values depending on the world W, thus solving the conundrum outlined above.

It is also interesting to consider what this means in practice, when proving theorems using a formulation of parametricity. Many proofs go through with TWLRs as they do with RLRs, as demonstrated, for example, by Ahmed et al. [2017]. However, this is not the case for all proofs and Lemma 4.2 (Univ is degenerate) from Section 3.1 is a perfect example. To see this, let us try to adapt the proof to use the TWLR from Section 2.5 instead of the RLR from Section 2.3, and see where this fails. Interestingly, the place we get stuck is at the application of Lemma 2.9 (Boring lemma), suggesting the theorem is not as boring as the name suggests.

When we look at how Lemma 2.9 (Boring lemma) was applied in the proof of Lemma 4.2, we see that we were able to prove the following:

$$v_U.1\ \mathbf{unit} \in \mathcal{E}[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\mathrm{Unit}, RU)}$$

Then, we used Lemma 2.9 (Boring lemma) twice, the first time to forget a binding for X, and the second time to add a different binding for the same X:

$$\mathcal{E}[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\mathrm{Unit}, RU)} = \mathcal{E}[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y)} = \mathcal{E}[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\mathrm{Unit}, RE)}$$

We could then conclude that

$$v_U.1\ \mathbf{unit} \in \mathcal{E}[\![Y]\!]^{\emptyset, Y \mapsto (\tau_Y, R_Y), X \mapsto (\mathrm{Unit}, RE)}$$

With the TWLR, these applications of the lemma cannot be replicated. Consider the following worlds:

- $W_{yx1} = ((Y, \tau_Y, R_Y); (X, \mathrm{Unit}, RU))$
- $W_{yx2} = ((Y, \tau_Y, R_Y); (X, \mathrm{Unit}, RE))$

Clearly, $W_{yx2} \not\sqsupseteq W_{yx1}$ (HPNF).

We can replicate the first steps of the proof until we have the following relation:

$$(W_{yx1}, v_U).1\ \mathbf{unit}) \in \mathcal{E}[\![Y]\!]$$

but the step to W_{yx2} no longer holds:

$$(W_{yx2}, v_U).1\ \mathbf{unit}) \notin \mathcal{E}[\![Y]\!]$$

We cannot use Lemma 2.13 (Boring lemma for TWLR) because of (HPNF).

In other words, the use of a TWLR means that different terms may be valid at type Y in world $\mathrm{W}_{\mathrm{yx1}}$ than in $\mathrm{W}_{\mathrm{yx2}}$. This dependence of the relation for Y on the world is precisely what a TWLR purposefully allows. As a result, it is impossible to transfer the result about $\mathrm{v}'\ [\mathrm{Unit}]$ from the world $\mathrm{W}_{\mathrm{yx1}}$ (with the permissive predicate RU for X) to the world $\mathrm{W}_{\mathrm{yx2}}$ (with the more restrictive predicate RE for X).

4.3 Degeneracy and Other Variants of Logical Relations

If we consider the other variants of the logical relation discussed in Section 2.4, our expectations are essentially confirmed. We have seen in Section 3.2 that the degeneracy of Univ does not hold in the presence of higher-order effects and indeed, the above proof fails if we use the Kripke logical relation from Section 2.4, for a similar reason as for the TWLR.

For some W, we would be able to obtain the following two facts:

$$(\mathrm{W}, \mathrm{v}'\ \mathrm{Unit}) \in \mathcal{E}\,[\![\mathrm{X}{\rightarrow}\mathrm{Y} \times \mathrm{Y}{\rightarrow}\mathrm{X}]\!]^{\emptyset, \mathrm{Y}\mapsto(\tau_{\mathrm{Y}}, \mathrm{R}_{\mathrm{Y}}), \mathrm{X}\mapsto(\mathrm{Unit}, \mathrm{RE})}$$
$$(\mathrm{W}, \mathrm{v}'\ \mathrm{Unit}) \in \mathcal{E}\,[\![\mathrm{X}{\rightarrow}\mathrm{Y} \times \mathrm{Y}{\rightarrow}\mathrm{X}]\!]^{\emptyset, \mathrm{Y}\mapsto(\tau_{\mathrm{Y}}, \mathrm{R}_{\mathrm{Y}}), \mathrm{X}\mapsto(\mathrm{Unit}, \mathrm{RU})}$$

Combined with $\mathrm{v}'\ \mathrm{Unit} \hookrightarrow^{*} \mathrm{v}_{\mathrm{U}}$, this gives us some W', for which

$$(\mathrm{W}', \mathrm{v}_{\mathrm{U}}) \in \mathcal{V}\,[\![\mathrm{X}{\rightarrow}\mathrm{Y} \times \mathrm{Y}{\rightarrow}\mathrm{X}]\!]^{\emptyset, \mathrm{Y}\mapsto(\tau_{\mathrm{Y}}, \mathrm{R}_{\mathrm{Y}}), \mathrm{X}\mapsto(\mathrm{Unit}, \mathrm{RE})}$$

From the second expression relation above, we also get a W'' such that

$$(\mathrm{W}'', \mathrm{v}_{\mathrm{U}}) \in \mathcal{V}\,[\![\mathrm{X}{\rightarrow}\mathrm{Y} \times \mathrm{Y}{\rightarrow}\mathrm{X}]\!]^{\emptyset, \mathrm{Y}\mapsto(\tau_{\mathrm{Y}}, \mathrm{R}_{\mathrm{Y}}), \mathrm{X}\mapsto(\mathrm{Unit}, \mathrm{RU})}$$

However, these two worlds W' and W'' are potentially distinct and one is not (necessarily) a future world of the other, which will later prevent us from combining both facts later in the proof.

In Section 3.2, we have also seen that the value-polymorphic version of the same calculus with higher-order state should preserve degeneracy of Univ. And indeed, using the value-polymorphic KLR, we can proceed differently and deduce that $\mathrm{v}' = \Lambda\mathrm{X}.\,\mathrm{v}''$ and

$$(\mathrm{W}, \mathrm{v}''[\mathrm{Unit}/\mathrm{X}]) \in \mathcal{V}\,[\![\mathrm{X}{\rightarrow}\mathrm{Y} \times \mathrm{Y}{\rightarrow}\mathrm{X}]\!]^{\emptyset, \mathrm{Y}\mapsto(\tau_{\mathrm{Y}}, \mathrm{R}_{\mathrm{Y}}), \mathrm{X}\mapsto(\mathrm{Unit}, \mathrm{RE})}$$
$$(\mathrm{W}, \mathrm{v}''[\mathrm{Unit}/\mathrm{X}]) \in \mathcal{V}\,[\![\mathrm{X}{\rightarrow}\mathrm{Y} \times \mathrm{Y}{\rightarrow}\mathrm{X}]\!]^{\emptyset, \mathrm{Y}\mapsto(\tau_{\mathrm{Y}}, \mathrm{R}_{\mathrm{Y}}), \mathrm{X}\mapsto(\mathrm{Unit}, \mathrm{RU})}$$

In other words, value polymorphism gives us these two facts immediately in the same world and as a result, contrary to above, we can combine them in the remainder of the proof.

Thus, the different logical relation variants that we have discussed in Section 2 either prevent or allow the proof of degeneracy of Univ, as appropriate for the language variant they are intended for. In addition to confirming our analysis of the universal type in Section 3, this also provides more insight into how the different logical relation variants influence the particular proof techniques used in the proof outlined here.

In addition to clarifying the relation between TWLRs and non-lexically-scoped type variables, this paper also discusses some consequences that follow from these insights. Particularly, the example from Section 3.1 can be used to disprove two open conjectures and expose a previously unmentioned deficiency of polymorphic blame calculi. In the next three sections, we discuss these three topics in turn, starting with the secure compilation of System F in the cryptographic λ-calculus, a conjecture by Pierce and Sumii [2000]; Sumii and Pierce [2004].

5 ENFORCING PARAMETRICITY IN AN UNTYPED TARGET LANGUAGE

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 5.1) and the compiler from λ^{F} to λ^{σ} (Section 5.2). 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 5.3).

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.[11] 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.

5.1 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].

$\boxed{\text{Syntax:}}$

$$\begin{aligned}
\mathsf{t} ::=\ & \mathsf{v} \mid \mathsf{x} \mid \mathsf{t}\ \mathsf{t} \mid \mathsf{t.1} \mid \mathsf{t.2} \mid \langle \mathsf{t},\mathsf{t}\rangle \mid \mathsf{if\ t\ then\ t\ else\ t}\\
&\mid \nu\mathsf{x.t} \mid \{\mathsf{t}\}_{\mathsf{t}} \mid \sigma \mid \mathsf{let}\ \{\mathsf{x}\}_{\mathsf{t}} = \mathsf{t\ in\ t\ else\ t} \mid \mathsf{roll\ t} \mid \mathsf{unroll\ t} \mid \mathsf{wrong}\\
\mathsf{v} ::=\ & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid \lambda\mathsf{x.\,t} \mid \langle \mathsf{v},\mathsf{v}\rangle \mid \{\mathsf{v}\}_{\sigma} \mid \sigma \mid \mathsf{roll\ v}
\end{aligned}$$

$\boxed{\text{Evaluation rules (excerpts):}}$

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

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

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

$$\frac{\mathsf{t} \hookrightarrow_0 \mathsf{t'}}{(\mathsf{h}, \mathsf{E}[\mathsf{t}]) \hookrightarrow (\mathsf{h}, \mathsf{E}[\mathsf{t'}])}$$

Fig. 2. λ^{σ} syntax and evaluation rules (excerpts). A λ^{σ} program state is a pair $\mathsf{h};\mathsf{t}$ 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.

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:

[11]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.

$\nu x.t$ 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 C. Contextual equivalence in λ^σ, indicated with $\simeq$, is defined analogously to Definition 2.1. Note that the quantified contexts 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 5.1 ($\mathbb{Z}_3$ in λ^σ). First, we introduce two helper functions:

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

Then, we can define a λ^σ correspondent of $\mathbf{z3}$ as:

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

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 λ^F term:

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

The polymorphic function $\mathbf{f}$ 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 $\mathbf{true}$ (or diverge).

We can implement a similar protection in λ^σ. To enforce parametric behaviour 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:

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

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 λ^{F}.

5.2 Sumii and Pierce's Compiler

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

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

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 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.[12]

$$\mathsf{erase}\,(\mathbf{x}) \overset{\text{def}}{=} \mathsf{x} \qquad \mathsf{erase}\,(\mathbf{t}\ \boldsymbol{\tau}') \overset{\text{def}}{=} \mathsf{erase}\,(\mathbf{t})\ \mathsf{unit}$$

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

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

Dynamic wrappers. The second phase of the compiler wraps compiled terms with dynamic wrappers. These are formalised as the function $\mathsf{protect}_{\eta;\boldsymbol{\tau}}$, which is defined in Figure 3, together with its dual $\mathsf{confine}_{\eta;\boldsymbol{\tau}}$ by mutual induction on $\boldsymbol{\tau}$. 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 $\mathsf{protect}_{\eta;\boldsymbol{\tau}}$ to a value v ensures that v cannot be used in ways that are not allowed by type $\boldsymbol{\tau}$. Dually, applying $\mathsf{confine}_{\eta;\boldsymbol{\tau}}$ to a value v prevents v from behaving in a way that is not allowed by type $\boldsymbol{\tau}$. For any free type variables $\mathbf{X}$ in $\boldsymbol{\tau}$, 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 $\mathsf{protect}_{\cdot}$ and $\mathsf{confine}_{\cdot}$ we rely on the following seal_σ and unseal_σ functions, which seal and unseal values with a given seal σ.

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

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.

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

$$
\begin{aligned}
\mathsf{protect}_{\eta;\mathbf{Unit}}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{x} \\
\mathsf{protect}_{\eta;\mathbf{Bool}}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{x} \\
\mathsf{protect}_{\eta;\tau_1\times\tau_2}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{let\ x_1 = x.1\ in\ let\ x_2 = x.2\ in}\ \left\langle \mathsf{protect}_{\eta;\tau_1}\ \mathsf{x_1}, \mathsf{protect}_{\eta;\tau_2}\ \mathsf{x_2} \right\rangle \\
\mathsf{protect}_{\eta;\tau_1\to\tau_2}\ \mathsf{x} &\overset{\text{def}}{=} \lambda \mathsf{y}.\ \mathsf{let\ z = x\ (confine}_{\eta;\tau_1}\ \mathsf{y)\ in\ protect}_{\eta;\tau_2}\ \mathsf{z} \\
\mathsf{protect}_{\eta;\forall \mathbf{X}.\tau}\ \mathsf{x} &\overset{\text{def}}{=} \lambda\_.\ \mathsf{let\ y = x\ unit\ in\ protect}_{\eta,\mathbf{X}\mapsto(\lambda \mathsf{x.x},\lambda \mathsf{x.x});\tau}\ \mathsf{y} \\
\mathsf{protect}_{\eta;\exists \mathbf{X}.\tau}\ \mathsf{x} &\overset{\text{def}}{=} \nu \mathsf{s}.\ \mathsf{protect}_{\eta,\mathbf{X}\mapsto(\mathsf{seal_s},\mathsf{unseal_s});\tau}\ \mathsf{x} \\
\mathsf{protect}_{\eta;\mathbf{X}}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{t_p}\ \mathsf{x} \qquad\qquad \text{where } \eta(\mathbf{X}) = (\mathsf{t_p}, \_)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{confine}_{\eta;\mathbf{Unit}}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{x; unit} \\
\mathsf{confine}_{\eta;\mathbf{Bool}}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{if\ x\ then\ true\ else\ false} \\
\mathsf{confine}_{\eta;\tau_1\times\tau_2}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{let\ x_1 = x.1\ in\ let\ x_2 = x.2\ in}\ \left\langle \mathsf{confine}_{\eta;\tau_1}\ \mathsf{x_1}, \mathsf{confine}_{\eta;\tau_2}\ \mathsf{x_2} \right\rangle \\
\mathsf{confine}_{\eta;\tau_1\to\tau_2}\ \mathsf{x} &\overset{\text{def}}{=} \lambda \mathsf{y}.\ \mathsf{let\ z = x\ (protect}_{\eta;\tau_1}\ \mathsf{y)\ in\ confine}_{\eta;\tau_2}\ \mathsf{z} \\
\mathsf{confine}_{\eta;\forall \mathbf{X}.\tau}\ \mathsf{x} &\overset{\text{def}}{=} \lambda\_.\ \nu \mathsf{s}.\ \mathsf{let\ x' = x\ unit\ in\ confine}_{\eta,\mathbf{X}\mapsto(\mathsf{seal_s},\mathsf{unseal_s});\tau}\ \mathsf{x'} \\
\mathsf{confine}_{\eta;\exists \mathbf{X}.\tau}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{confine}_{\eta,\mathbf{X}\mapsto(\lambda \mathsf{x.x},\lambda \mathsf{x.x});\tau}\ \mathsf{x} \\
\mathsf{confine}_{\eta;\mathbf{X}}\ \mathsf{x} &\overset{\text{def}}{=} \mathsf{t_c}\ \mathsf{x} \qquad\qquad \text{where } \eta(\mathbf{X}) = (\_, \mathsf{t_c})
\end{aligned}
$$

Fig. 3. The dynamic wrappers of Sumii and Pierce's compiler.

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 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 5.1.

Applying the compiler to z3 from Example 5.1 results in the λ^σ term z3 from Example 2.2 (modulo some additional β-reductions).

5.3 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 5.2 (Sumii and Pierce). $\emptyset; \emptyset \vdash \mathbf{t}_1 \simeq \mathbf{t}_2 : \tau$ *if and only if* $\emptyset \vdash [\![\mathbf{t}_1]\!]^{\lambda^{\mathrm{F}}}_{\lambda^\sigma} \simeq [\![\mathbf{t}_2]\!]^{\lambda^{\mathrm{F}}}_{\lambda^\sigma}$

However, we can now prove that this conjecture is false. A counterexample is given by the terms $\mathbf{t_u}$ and $\mathbf{t}_\omega$ which we proved contextually equivalent in Theorem 3.1. Compiling $\mathbf{t_u}$ and $\mathbf{t}_\omega$ does not produce contextually equivalent λ^σ terms:

Theorem 5.3 ($\mathbf{t_u}$ and $\mathbf{t}_\omega$ are not equivalent after compilation to λ^σ). $\emptyset \vdash [\![\mathbf{t_u}]\!]^{\lambda^{\mathrm{F}}}_{\lambda^\sigma} \not\simeq [\![\mathbf{t}_\omega]\!]^{\lambda^{\mathrm{F}}}_{\lambda^\sigma}$

Proof Sketch. A full proof with all details can be found in the supplementary material.

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

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

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 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 behaviour.

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

We spell out the reductions in full detail in the supplementary material. From this behaviour, 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 5.4 ($[\![\cdot]\!]^{\lambda^F}_{\lambda^\sigma}$ is not fully abstract). *It is not true that*

$$\forall \mathbf{t_1}, \mathbf{t_2}. \mathbf{t_1} \simeq \mathbf{t_2} \iff [\![\mathbf{t_1}]\!]^{\lambda^F}_{\lambda^\sigma} \simeq [\![\mathbf{t_2}]\!]^{\lambda^F}_{\lambda^\sigma}$$

Proof. Follows easily from Theorem 3.1 and the counterexample in Theorem 5.3. □

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 $\lambda\_.\ \langle \lambda \mathsf{x}.\,\mathsf{x}, \lambda \mathsf{x}.\,\mathsf{x}\rangle$ 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:

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

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 $\lambda\_.\ \langle \lambda \mathsf{x}.\,\mathsf{x}, \lambda \mathsf{x}.\,\mathsf{x}\rangle$ is accepted as a value of type Univ, because it does in fact correspond to a legal λ^F value of type Triv:

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

6 ENFORCING PARAMETRICITY IN THE PRESENCE OF TYPE CASTS

The second conjecture we disprove is by Neis et al. [2009, 2011]. These authors study a form of runtime type generation to protect parametrically polymorphic functions when interacting with code that can use a type cast primitive. They prove a parametricity result that applies to appropriately wrapped System F values once embedded in $\mathtt{G}$, a language with a type cast primitive. They conjecture [Neis et al. 2009, section 10] that this wrapping is fully abstract, and while they prove equivalence reflection, they only conjecture equivalence preservation due to a lack of sophistication of the proof techniques available at the time.[13]

It turns out that our results may be adapted to this setting, as in such a non-parametrically polymorphic setting, the type $\forall \mathtt{X}.\, \mathtt{X}$ can be used as a universal type (that every other type can be embedded into or out from). In other words, we disprove this conjecture too: embedding System F into $\mathtt{G}$ à la Neis-Dreyer-Rossberg (NDR, in the sequel) is not fully abstract.

We first present $\mathtt{G}$ (Section 6.1) and the wrapper for protecting polymorphic functions from interacting with $\mathtt{G}$ terms (Section 6.2). Then we demonstrate how $\forall \mathtt{Z}.\, \mathtt{Z}$ is a universal type in $\mathtt{G}$ (Section 6.3) and prove that due to that type, the wrapper does not preserve contextual equivalence (Section 6.4).

6.1 The $\mathtt{G}$ Language

$\mathtt{G}$ extends System F with two primitives (Figure 4). The first one casts values of type τ_1 to values of type τ_2. The second one generates a fresh type name $\mathtt{X}$ that is registered to be an equivalent classifier to a type τ although data of the two types is not equivalent (so casting between $\mathtt{X}$ and τ will not succeed).

6.2 Enforcing Parametricity in $\mathtt{G}$

To avoid much code repetition, we use the same notation for the wrapper as that used by Neis et al. [2009]. Instead of writing two recursive functions such as $\mathsf{protect}_{;}$ and $\mathsf{confine}_{;}$, we annotate the wrapper with a *polarity*: positive polarity (+) is analogous to $\mathsf{protect}_{;}$, negative polarity (−) is analogous to $\mathsf{confine}_{;}$. When the *other* function is invoked, the polarity is switched (i.e., from ± to ∓).

$$
\begin{aligned}
\mathrm{W}^{\pm}_{\boldsymbol{\tau}}(\mathtt{t}) &= \mathtt{let}\ \mathtt{x} = \mathtt{t}\ \mathtt{in}\ \mathrm{Wrap}^{\pm}_{\boldsymbol{\tau}}(\mathtt{x})\\
\mathrm{Wrap}^{\pm}_{\mathbf{X}}(\mathtt{v}) &= \mathtt{v}\\
\mathrm{Wrap}^{\pm}_{\mathbf{Bool}}(\mathtt{v}) &= \mathtt{v}\\
\mathrm{Wrap}^{\pm}_{\boldsymbol{\tau}\to\boldsymbol{\tau}'}(\mathtt{v}) &= \lambda \mathtt{x} : \tau.\, \mathrm{W}^{\pm}_{\boldsymbol{\tau}'}\left((\mathtt{v}\ \mathrm{Wrap}^{\mp}_{\boldsymbol{\tau}}(\mathtt{x}))\right)\\
\mathrm{Wrap}^{\pm}_{\boldsymbol{\tau}\times\boldsymbol{\tau}'}(\mathtt{v}) &= \left\langle \mathrm{W}^{\pm}_{\boldsymbol{\tau}}(\mathtt{v}.1), \mathrm{W}^{\pm}_{\boldsymbol{\tau}'}(\mathtt{v}.2)\right\rangle\\
\mathrm{Wrap}^{\pm}_{\forall \mathbf{X}.\boldsymbol{\tau}}(\mathtt{v}) &= {}^{\sim}\mathtt{X}.\, \mathrm{NewTy}^{\mp}\mathtt{X}\ \mathrm{in}\ \mathrm{W}^{\pm}_{\boldsymbol{\tau}}\left((\mathtt{v}\ \mathtt{X})\right)\\
\mathrm{Wrap}^{\pm}_{\exists \mathbf{X}.\boldsymbol{\tau}}(\mathtt{v}) &= \mathtt{unpack}\ \mathtt{v}\ \mathtt{as}\ \langle \mathtt{X}, \mathtt{x}\rangle\ \mathtt{in}\ \mathrm{NewTy}^{\pm}\mathtt{X}\ \mathrm{in}\ \mathtt{pack}\ \left\langle \mathtt{X}, \mathrm{Wrap}^{\pm}_{\boldsymbol{\tau}}(\mathtt{x})\right\rangle\ \mathtt{as}\ \exists \mathtt{X}.\, \tau\\
\mathrm{NewTy}^{+}\mathtt{X}\ \mathrm{in}\ \mathtt{t} &= \mathtt{new}\ \mathtt{Y} \approx \mathtt{X}\ \mathtt{in}\ \mathtt{t}[\mathtt{Y}/\mathtt{X}]\\
\mathrm{NewTy}^{-}\mathtt{X}\ \mathrm{in}\ \mathtt{t} &= \mathtt{t}
\end{aligned}
$$

The wrapper consists of three parts. The first one, $\mathrm{W}^{\pm}_{\boldsymbol{\tau}}(\mathtt{t})$, is the term wrapper, it reduces a term $\mathtt{t}$ of source type $\boldsymbol{\tau}$ to a value and applies the value wrapper. The second one, $\mathrm{Wrap}^{\pm}_{\boldsymbol{\tau}}(\mathtt{v})$, is the value wrapper, it is responsible for generating the fresh type variables for outgoing universal values and

[13]A claim that was indeed true, since much of the research in proof techniques for fully abstract compilation postdates that paper [Abate et al. 2019; Ahmed and Blume 2011; Devriese et al. 2016; Fournet et al. 2013; New et al. 2016b; Patrignani and Garg 2019; Schmidt-Schauß et al. 2015]. An exception is the work by Ahmed and Blume [2008] on typed closure conversion.

Syntax:

$$\begin{aligned} \mathtt{t} &::= \cdots \mid \mathtt{cast}\ \tau_1\ \tau_2 \mid \mathtt{new}\ \mathtt{X} \approx \tau\ \mathtt{in}\ \mathtt{t} \\ \text{´} &::= \emptyset \mid \text{´}, \mathtt{X} \mid \text{´}, \mathtt{X} \approx \tau \\ \text{˚} &::= \emptyset \mid \text{˚}, \mathtt{X} \approx \tau \end{aligned}$$

Typing rules (excerpts):

$$\frac{\text{´} \vdash \tau_1 \qquad \text{´} \vdash \tau_2}{\text{´}; \text{`} \vdash \mathtt{cast}\ \tau_1\ \tau_2 : \tau_1 \to \tau_2 \to \tau_2} \qquad \frac{\text{´}, \mathtt{X} \approx \tau; \text{`} \vdash \mathtt{t} : \tau'}{\text{´}; \text{`} \vdash \mathtt{new}\ \mathtt{X} \approx \tau\ \mathtt{in}\ \mathtt{t} : \tau'}$$

$$\frac{\text{´}; \text{`} \vdash \mathtt{t} : \tau' \qquad \text{´} \vdash \tau \approx \tau'}{\text{´}; \text{`} \vdash \mathtt{t} : \tau} \qquad \frac{\tau \approx \tau' \in \text{´}}{\text{´} \vdash \tau \approx \tau'}$$

Evaluation rules (excerpts):

$$\frac{\tau_1 = \tau_2}{\text{˚} \triangleright \mathtt{cast}\ \tau_1\ \tau_2 \hookrightarrow_0 \text{˚} \triangleright \lambda \mathtt{x}_1 : \tau_1 . \lambda \mathtt{x}_2 : \tau_2 . \mathtt{x}_1}$$

$$\frac{\tau_1 \neq \tau_2}{\text{˚} \triangleright \mathtt{cast}\ \tau_1\ \tau_2 \hookrightarrow_0 \text{˚} \triangleright \lambda \mathtt{x}_1 : \tau_1 . \lambda \mathtt{x}_2 : \tau_2 . \mathtt{x}_2} \qquad \frac{}{\text{˚} \triangleright \mathtt{new}\ \mathtt{X} \approx \tau\ \mathtt{in}\ \mathtt{t} \hookrightarrow_0 \text{˚}, (\mathtt{X} \approx \tau) \triangleright \mathtt{t}}$$

Fig. 4. The G language: syntax, typing rules and evaluation rules (excerpts). The semantics relation $\hookrightarrow$ relies on the primitive reductions indicated as $\hookrightarrow_0$ and it relates configurations of the form ˚ $\triangleright$ t

incoming existential packages. The third one, $\mathrm{NewTy}^{\pm}\mathtt{X}$ in t is the code that effectively generates the fresh type variables, depending on the polarity of the invocation. As the authors themselves note, the wrapper functions in a way analogous to the dynamic checks inserted by the Sumii-Pierce compiler.

The compiler from $\lambda^{\mathbf{F}}$ to G, denoted with $\wr\cdot\wr$, leaves the term untouched (which we indicate with $\equiv$) and wraps it with a wrapper of the appropriate type.

$$\wr \mathtt{t} \wr \overset{\text{def}}{=} \mathrm{W}^{+}_{\boldsymbol{\tau}}(\mathtt{t}) \qquad\qquad \text{if } \emptyset; \emptyset \vdash \mathbf{t} : \boldsymbol{\tau} \text{ and } \mathtt{t} \equiv \mathbf{t}$$

The NDR conjecture (Conjecture 6.1) states that this wrapper is fully abstract. Contextual equivalence in G, indicated with $\simeq$, is defined analogously to Definition 2.1.

Conjecture 6.1 (NDR Conjecture). $\forall \mathbf{t}_1, \mathbf{t}_2 . \mathbf{t}_1 \simeq \mathbf{t}_2 \iff \wr \mathbf{t}_1 \wr \simeq \wr \mathbf{t}_2 \wr$

We will prove that this statement is not true (Theorem 6.3).

6.3 G has an Instance of an Universal Type: ∀Z. Z

To explain how we disprove the NDR conjecture, we need to explain that G indeed has a universal type (unlike System F). This type is $\forall \mathtt{Z}. \mathtt{Z}$ and it functions as a universal type because we can take any other type X and (i) embed a value v of type X into $\forall \mathtt{Z}. \mathtt{Z}$, (ii) extract the same value v from $\forall \mathtt{Z}. \mathtt{Z}$ at type X. It is possible to do this while remaining parametric in the type X that these functions work with, i.e., keeping type variable X free in these terms and binding it in a larger term using both these functions.

These two functionalities will be key for building a distinguishing context for G which is analogous to the distinguishing contexts of for λ^{σ} and λ^{B} (the latter will be presented later). A value v of an

$$
\begin{array}{l}
\left(\left(\Lambda X.\, \lambda x : X. \left(\begin{array}{l}(\lambda x_2 : \forall X_0.\, X_0.\, x2\ X)\\ \left(\left(\lambda x : X.\, {\sim}Y. \left(\begin{array}{l}\mathtt{cast}\ (\mathtt{Unit} \rightarrow X)\ (\mathtt{Unit} \rightarrow Y)\\ (\lambda\_ : \mathtt{Unit}.\, x)\ (\lambda\_ : \mathtt{Unit}.\, \omega_X)\end{array}\right)\ \mathtt{unit}\right)\ x\right)\end{array}\right)\right)\ \mathtt{Bool}\right)\ \mathtt{true}\\
\hookrightarrow \left(\lambda x : \mathtt{Bool}. \left(\begin{array}{l}(\lambda x_2 : \forall X_0.\, X_0.\, x2\ \mathtt{Bool})\\ \left(\left(\lambda x : \mathtt{Bool}.\, {\sim}Y. \left(\begin{array}{l}\mathtt{cast}\ (\mathtt{Unit} \rightarrow \mathtt{Bool})\ (\mathtt{Unit} \rightarrow Y)\\ (\lambda\_ : \mathtt{Unit}.\, x)\ (\lambda\_ : \mathtt{Unit}.\, \omega_X)\end{array}\right)\ \mathtt{unit}\right)\ x\right)\end{array}\right)\right)\ \mathtt{true}\\
\hookrightarrow \left(\begin{array}{l}(\lambda x_2 : \forall X_0.\, X_0.\, x2\ \mathtt{Bool})\\ \left(\left(\lambda x : \mathtt{Bool}.\, {\sim}Y. \left(\begin{array}{l}\mathtt{cast}\ (\mathtt{Unit} \rightarrow \mathtt{Bool})\ (\mathtt{Unit} \rightarrow Y)\\ (\lambda\_ : \mathtt{Unit}.\, x)\ (\lambda\_ : \mathtt{Unit}.\, \omega_X)\end{array}\right)\ \mathtt{unit}\right)\ \mathtt{true}\right)\end{array}\right)\\
\hookrightarrow \left(\begin{array}{l}(\lambda x_2 : \forall X_0.\, X_0.\, x2\ \mathtt{Bool})\\ ({\sim}Y.\, (\mathtt{cast}\ (\mathtt{Unit} \rightarrow \mathtt{Bool})\ (\mathtt{Unit} \rightarrow Y)\ (\lambda\_ : \mathtt{Unit}.\, \mathtt{true})\ (\lambda\_ : \mathtt{Unit}.\, \omega_X)\ \ )\ \mathtt{unit})\end{array}\right)\\
\hookrightarrow (\ ({\sim}Y.\, (\mathtt{cast}\ (\mathtt{Unit} \rightarrow \mathtt{Bool})\ (\mathtt{Unit} \rightarrow Y)\ (\lambda\_ : \mathtt{Unit}.\, \mathtt{true})\ (\lambda\_ : \mathtt{Unit}.\, \omega_X)\ \ )\ \mathtt{unit})\ \mathtt{Bool}\quad)\\
\hookrightarrow (\mathtt{cast}\ (\mathtt{Unit} \rightarrow \mathtt{Bool})\ (\mathtt{Unit} \rightarrow \mathtt{Bool})\ (\lambda\_ : \mathtt{Unit}.\, \mathtt{true})\ (\lambda\_ : \mathtt{Unit}.\, \omega_X))\ \mathtt{unit}\\
\hookrightarrow ((\lambda y_1 : (\mathtt{Unit} \rightarrow \mathtt{Bool}).\, \lambda y_2 : (\mathtt{Unit} \rightarrow \mathtt{Bool}).\, y_1)\ (\lambda\_ : \mathtt{Unit}.\, \mathtt{true})\ (\lambda\_ : \mathtt{Unit}.\, \omega_X))\ \mathtt{unit}\\
\hookrightarrow^{*} (\lambda\_ : \mathtt{Unit}.\, \mathtt{true})\ \mathtt{unit}\\
\hookrightarrow \mathtt{true}
\end{array}
$$

Fig. 5. Reductions of the combination of the two functionalities.

arbitrary type τ is represented as a value of type $\forall Z.\, Z$. The cast operator in G allows us to make this function behave differently when it is applied to type τ than when it is applied to other types. In the former case, we will make the function return v itself, while in the latter case, we simply make the function diverge.

Concretely, to extract a value from $\forall Z.\, Z$ into X, we simply apply the polymorphic function to type X:

$$\lambda z : (\forall Z.\, Z).\, z\ X$$

Injecting a value of type from X into $\forall Z.\, Z$ is a bit more complex. What we would like to write is the following:

$$\lambda x : X.\, {\sim}Z.\, \mathtt{cast}\ X\ Z\ x\ \omega_Z$$

This term uses the $\mathtt{cast}\ \cdot\ \cdot$ primitive to cast a value of type X into $\forall Z.\, Z$. If everything goes well, the requested type Z is the same as the type X of the value contained, and the constructed function returns x. Otherwise, it diverges by calling ω_Z.

Unfortunately, the above does not work as intended because we are in a call-by-value setting, so ω_Z is evaluated before checking the type equality and the whole term always diverges. Fortunately, this can be resolved by the standard trick of "thunking" the term and the expected value x into lambdas that expect a $\mathtt{Unit}$ value, and pass $\mathtt{unit}$ to them after the cast:

$$\lambda x : X.\, {\sim}Z.\, (\mathtt{cast}\ (\mathtt{Unit} \rightarrow X)\ (\mathtt{Unit} \rightarrow Z)\ (\lambda\_ : \mathtt{Unit}.\, x)\ (\lambda\_ : \mathtt{Unit}.\, \omega_Z))\ \mathtt{unit}$$

To see how these functions work in practice in our counterexample, we build a term that uses them in order to inject $\mathtt{true}$ of type $\mathtt{Bool}$ into $\forall Z.\, Z$ and extract $\mathtt{true}$ back. The term and its reductions are in Figure 5.

6.4 Disproving the NDR Conjecture

We now have all the technical machinery to disprove the NDR conjecture.

Compiling $\mathbf{t_u}$ to G results in a complex term that is equivalent to the following one, where, for simplicity, we have elided some redundant sub-terms that do not alter the semantics of the overall-term. As a convention we name variables and type variables from the wrapper with y. The first line is the $\mathbf{t_u}$ term ported to G, the second and third lines (as well as the wrapping lambda) are wrapper-generated code. This code will take the parameter that needs to be passed to $\mathbf{t_u}$ and perform a series of unpacking and type renaming which are intended to preserve parametricity.

$$\lambda x_1 : \underline{\mathsf{Univ}}. \left(\begin{array}{l} (\lambda x : \underline{\mathsf{Univ}}.\, \mathsf{unpack}\ x\ \mathsf{as}\ \langle Y, x' \rangle\ \mathsf{in\ let}\ x'' = x'\ \mathsf{Unit\ in}\ x''.2\ (x''.1\ \mathsf{unit})) \\ \left(\begin{array}{l} \mathsf{unpack}\ x_1\ \mathsf{as}\ \langle X_1, x_4 \rangle\ \mathsf{in} \\ \mathsf{pack}\ \left\langle X_1, {}^{\sim}X_2.\, \mathsf{new}\ X_3 \approx X_2\ \mathsf{in} \left\langle \begin{array}{l} \lambda x_5 : X_3.(x_4\ X_3).1\ x_5, \\ \lambda x_6 : X_1.(x_4\ X_3).2\ x_6 \end{array} \right\rangle \right\rangle\ \mathsf{as}\ \underline{\mathsf{Univ}} \end{array} \right) \end{array} \right)$$

The distinguishing context for G passes a parameter of type $\underline{\mathsf{Univ}}$ to the term in the hole (which will be either $\wr\mathbf{t_u}\wr$ or $\wr\mathbf{t_\omega}\wr$). The goal of this parameter is to make $\wr\mathbf{t_u}\wr$ terminate and $\wr\mathbf{t_\omega}\wr$ diverge. Let us discuss the passed parameter. Since $\underline{\mathsf{Univ}}$ is an existential type at the top level, the parameter is a $\mathsf{pack}\ \langle\cdot\rangle\ \mathsf{as}\ \cdot$. The packed type is the universal type that exists in G: $\forall Z.\, Z$. After the existential, $\underline{\mathsf{Univ}}$ has a universal quantification, and thus the body of the existential package contains a ${}^{\sim}Z.\,\cdot$. Then comes the pair of functions for projecting into and extracting from the universal type $\forall Z.\, Z$, as explained in Section 6.3. As a convention, we name variables and type variables from the context with z:

$$C^{G} \stackrel{\text{def}}{=} [\cdot] \left(\mathsf{pack} \left\langle \forall Z.\, Z, {}^{\sim}Z1. \left\langle \begin{array}{l} \lambda z1 : Z1.\, \Lambda Z2. \left(\begin{array}{l} \mathsf{cast}\ (\mathsf{Unit} \longrightarrow Z1)\ (\mathsf{Unit} \longrightarrow Z2) \\ (\lambda\_ : \mathsf{Unit}.\, z1)\ (\lambda\_ : \mathsf{Unit}.\, \omega_{Z2}) \end{array} \right) \mathsf{unit}, \\ \lambda z2 : (\forall Z.\, Z).\, z2\ Z1 \end{array} \right\rangle \right\rangle\ \mathsf{as}\ \underline{\mathsf{Univ}} \right)$$

Theorem 6.2 ($\wr\mathbf{t_u}\wr$ and $\wr\mathbf{t_\omega}\wr$ are not equivalent in G). $\wr\mathbf{t_u}\wr \not\simeq \wr\mathbf{t_\omega}\wr$

Proof. We have that $C^{G}\ [\wr\mathbf{t_u}\wr] \hookrightarrow^{*} \mathsf{unit}$ while $C^{G}\ [\wr\mathbf{t_\omega}\wr]\Uparrow$. □

Theorem 6.3 (Embedding λ^{F} into G is not fully abstract). *It is not true that*

$$\forall \mathbf{t_1}, \mathbf{t_2}.\mathbf{t_1} \simeq \mathbf{t_2} \iff \wr\mathbf{t_1}\wr \simeq \wr\mathbf{t_2}\wr$$

Proof. Follows directly from Theorem 6.2 and Theorem 3.1. □

Additional instances of universal types in G. $\forall Z.\, Z$ is not the only universal type in G. The type $\exists X.X$ works just as well and might be more intuitive to some readers. The context below is analogous to the one above and can also differentiate between the compilation of $\mathbf{t_u}$ and of $\mathbf{t_\omega}$.

$$\begin{aligned} C^{G} &\stackrel{\text{def}}{=} [\cdot]\ \ (\mathsf{pack}\ \langle \exists X.X, {}^{\sim}Z.\, \langle \mathsf{inj}, \mathsf{ext} \rangle \rangle\ \mathsf{as}\ \underline{\mathsf{Univ}}) \\ \mathsf{inj} &\stackrel{\text{def}}{=} \lambda z : Z.\, \mathsf{pack}\ \langle Z, z \rangle\ \mathsf{as}\ \exists X.X \\ \mathsf{ext} &\stackrel{\text{def}}{=} \lambda z : (\exists X.X).\, \mathsf{unpack}\ z\ \mathsf{as}\ \langle U, u \rangle\ \mathsf{in} \left(\begin{array}{l} \mathsf{cast}\ (\mathsf{Unit} \longrightarrow U)\ (\mathsf{Unit} \longrightarrow Z) \\ (\lambda\_ : \mathsf{Unit}.\, u)\ (\lambda\_ : \mathsf{Unit}.\, \omega_{Z}) \end{array} \right) \mathsf{unit} \end{aligned}$$

7 SYSTEM F EQUIVALENCES VS. GRADUAL TYPES

After discussing these conjectures, we turn our attention to polymorphic gradual calculi: gradually typed languages featuring parametric polymorphism. 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, which include the *gradual guarantee*.

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.

Jacobs et al. [2021] have recently argued the importance of on another benefit of well-typedness that has received less attention so far: the benefits that well-typedness provides in terms of reasoning. Many type systems allow us to deduce properties of components from their types, for example, the function $\boldsymbol{\lambda}\mathbf{x} : \mathbf{Unit}.\, \mathbf{x}$ is easily seen to be equivalent to $\boldsymbol{\lambda}\mathbf{x} : \mathbf{Unit}.\, \mathbf{unit}$, based on the terms' types. Formulations of System F parametricity similarly allows us to deduce properties from programs' types.

Jacobs et al. propose a criterion that requires gradual type systems to preserve the validity of such reasoning in the original typed language. Specifically, the criterion requires that 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 criterion then becomes: if $\mathbf{t_1} \simeq \mathbf{t_2}$, do we have that $\lfloor \mathbf{t_1} \rfloor \simeq \lfloor \mathbf{t_2} \rfloor$, or, in other words, is the embedding of the typed language into the gradual language fully abstract? Based on this view, Jacobs et al. refer to their criterion as the Fully Abstract Embedding (FAE) criterion.

In this section, we show that the criterion fails 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 7.1). Next, we reconsider $\mathbf{t_u}$ and $\mathbf{t_\omega}$ from Theorem 3.1 in λ^{B} and present a λ^{B} context that differentiates them (Section 7.2). This shows that the embedding of $\lambda^{\mathbf{F}}$ into λ^{B} is not fully abstract.[14]

7.1 The λ^{B} Calculus

There exist several versions of polymorphic gradual languages in the literature [Ahmed et al. 2011b, 2017; Igarashi et al. 2017; New et al. 2019a; Toro et al. 2019; Xie et al. 2018]. We take λ^{B} to be the polymorphic blame calculus as described by Ahmed et al. [2017]. This version modifies certain behaviour that was "topsy turvy" in the original version [Ahmed et al. 2011b]: 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. 6. 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

[14]As mentioned, to the best of our knowledge this is not an existing conjecture that we disprove.

$\boxed{\text{Syntax:}}$

Terms	$t ::= v \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \mid x \mid t\ t \mid t\ \tau \mid t.1 \mid t.2 \mid \langle t, t\rangle$
	$\mid t : \tau \overset{p}{\Longrightarrow} \tau \mid t : \tau \overset{\phi}{\Longrightarrow} \tau \mid \mathit{blame}\ p$
Values	$v ::= \mathit{unit} \mid \mathit{true} \mid \mathit{false} \mid \lambda x : \tau.\, t \mid \Lambda X.\, v \mid \langle v, v\rangle$
	$\mid v : \tau \to \tau \overset{\phi}{\Longrightarrow} \tau \to \tau \mid v : \forall X.\, \tau \overset{\phi}{\Longrightarrow} \forall X.\, \tau \mid v : \tau \overset{\neg\alpha}{\Longrightarrow} \alpha$
	$\mid v : \tau \to \tau \overset{p}{\Longrightarrow} \tau \to \tau \mid v : \tau \overset{p}{\Longrightarrow} \forall X.\, \tau \mid v : \gamma \overset{p}{\Longrightarrow} \star$
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. 6. 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.

follows [Pierce 2002, §24.3, pp. 377-379]:

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

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

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

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 ($t : \tau \overset{p}{\Longrightarrow} \tau'$) and a notion of conversions ($t : \tau \overset{\phi}{\Longrightarrow} \tau$).

Casts ($t : \tau \overset{p}{\Longrightarrow} \tau'$) represent a form of dynamic casts from τ to τ' that can potentially fail at runtime (in which case $\mathit{blame}\ p$ 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 ($t : \tau \overset{\phi}{\Longrightarrow} \tau'$) 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.

7.2 Embedding of λ^{F} into λ^{B} is not Fully Abstract

To embed λ^{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 λ^{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 the non-parametrically polymorphic language $\mathtt{G}$ from the previous section and the polymorphic blame calculus λ^{B} make different choices and would both be non-trivial to modify.

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

Our two contextually equivalent System F terms $\mathbf{t_u}$ and $\mathbf{t_\omega}$ embed into λ^{B} as $\lfloor \mathbf{t_u} \rfloor$ and $\lfloor \mathbf{t_\omega} \rfloor$. 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 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 $\star$:

$$C^{B} \overset{\text{def}}{=} [\cdot] \left(\mathit{pack} \left\langle \star, \Lambda X. \left\langle \lambda x : X.\, x : X \overset{p}{\Longrightarrow} \star, \lambda x : \star.\, x : \star \overset{p'}{\Longrightarrow} X \right\rangle \right\rangle \mathit{as}\ \underline{\mathit{Univ}} \right)$$

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.

In the following, contextual equivalence in λ^{B}, indicated with $\simeq$, is defined analogously to Definition 2.1.

Theorem 7.1 ($\lfloor \mathbf{t_u} \rfloor$ and $\lfloor \mathbf{t_\omega} \rfloor$ are not equivalent in λ^{B}). $\lfloor \mathbf{t_u} \rfloor \not\simeq \lfloor \mathbf{t_\omega} \rfloor$.

Proof. We have that $C^{B}\left[\lfloor \mathbf{t_u} \rfloor\right] \hookrightarrow^{*} \mathit{unit}$ while $C^{B}\left[\lfloor \mathbf{t_\omega} \rfloor\right] \Uparrow$. We have verified this on paper and using the interpreter provided by Jamner and Siek to support Ahmed et al. [2017]'s results.[15] [16] We provide the literal encoding of $C^{B}\left[\lfloor \mathbf{t_u} \rfloor\right]$ and $C^{B}\left[\lfloor \mathbf{t_\omega} \rfloor\right]$ for use in the interpreter in the supplementary material. □

Theorem 7.2 (Embedding λ^{F} into λ^{B} is not fully abstract). *It is not true that*

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

Proof. Follows directly from Theorem 7.1 and Theorem 3.1. □

Although the above discussion looks just at λ^{B} by Ahmed et al. [2017], our results also apply to the more recent polymorphic gradual languages proposed by Toro et al. [2019], New et al. [2019a] and Xie et al. [2018].

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

8 DISCUSSION

So Sumii and Pierce's compiler is not fully abstract, the polymorphic blame calculus breaks contextual equivalences in System F and enforcing parametricity in a non-parametric calculus also breaks contextual equivalence. 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.

8.1 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 $\mathbf{t_\omega}$) 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 $\underline{\mathbf{Univ}}$. What happens is the following:

(1) The compiled term transmits the value **unit** of type **Unit**, encrypted as $\{\mathsf{unit}\}_\sigma$ of type $\mathbf{X}$ 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 $\mathbf{Y}$ (chosen by the adversary). The actual value transmitted back in our counterexample, is simply the encrypted value $\{\mathsf{unit}\}_\sigma$ received in step 1.
(3) The compiled term does not inspect the received value but simply transmits it back to the adversary as a value of type $\mathbf{Y}$.
(4) The adversary now takes the received value $\{\mathsf{unit}\}_\sigma$ and sends it back as a value of type $\mathbf{X}$.
(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 $\mathbf{Y}$ independently of $\mathbf{X}$ and values of such a type should intuitively not be able to contain values of type $\mathbf{X}$ (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 $\mathbf{Y}$. We want to prevent this second value from somehow including the first value $\{\mathsf{unit}\}_\sigma$, but since the type $\mathbf{Y}$ 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.

8.2 Polymorphic Blame Calculus Without a Universal Type?

Whether or not it is feasible to construct a gradual polymorphic language that fully abstractly embeds System F is unclear. In fact, even simply reconciling type safety results like the Blame Theorem (see Section 7) or the Dynamic Gradual Guarantee (which expresses that removing type annotations from a term should never cause the term to fail at runtime) with parametricity is the topic of ongoing research. In fact,in the design of their GSF calculus, Toro et al. [2019] explicitly abandon the dynamic gradual guarantee in order to salvage parametricity, formulated using a TWLR. New et al. [2019a] have recently reconciled (a TWLR-based form of) parametricity with the dynamic gradual guarantee, by requiring explicit sealing annotations in the source.

All published gradual polymorphic calculi make use of dynamic sealing and as such, do not satisfy the FAE criterion, except perhaps for one. Labrada et al. [2022] have investigated an alternative design, based on a form of lexically-scoped sealing (in addition to some other new ideas like plausible sealing). It would lead us too far to expain the details here, but the design replaces the type $\star$ with a family of types $\star_{\{\bar{X}\}}$, indexed by a set of type variables $\bar{X}$. The type $\star_{\bar{X}}$ is only legal (i.e., well-formed) when the type variables $\bar{X}$ are in scope. Injecting values of type X' into $\star_{\bar{X}}$ is only legal if X' is in the set $\bar{X}$. For our FAE counterexample, the effect is that the context can no longer produce a non-degenerate value of type <u>**Univ**</u>, as there would no longer be a viable choice for $\mathbf{Y}$. Any instance $\star_{\{\bar{X}\}}$ of the universal type we could choose, could not have $\mathbf{X}$ as part of $\{\bar{X}\}$, as $\mathbf{X}$ is not yet in scope at the moment where $\mathbf{Y}$ needs to be produced.

Although the new design still has some restrictions, it satisfies a form of RLR-based parametricity. As such, the jury is still out, but we believe the FAE criterion may not be out of reach in the context of gradual parametricity.

8.3 Adjusting our Expectations

For us, the solutions suggested above look 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 $\star_{\{\bar{X}\}}$ might be harder to use than the original $\star$. As such, we might consider alternative ways to address the lack of full abstraction.

One alternative is to abandon the choice of fully abstract compilation and rely on other notions. More concretely, perhaps a secure compiler or gradually typed language should not preserve arbitrary System F equivalences but only some of them, namely those that follow from a TWLR-based formulation of parametricity?

Another alternative is to keep relying on fully abstract compilation and 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. Ahmed et al. [2017], Toro et al. [2019] and New et al. [2019a] have shown that such a variant of System F still satisfies useful (TWLR-based) parametricity results and that useful free theorems follow from it, suggesting it is a suitable language for programmers to work in.

9 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 TWLR-based.

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] and Tobin-Hochstadt and Felleisen [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].

Prior to the work on gradual typing, calculi which combined statically-typed languages with a universal type for interacting with untyped code, have been proposed by Henglein [1994] and Abadi et al. [1991].

As mentioned in Section 7, formal criteria for gradual type systems have been proposed by Siek et al. [2015], including the gradual guarantee. Garcia and Tanter [2020] have later argued that gradually typed languages should additionally preserve reasoning principles of the static type system. Jacobs et al. [2021] have proposed that this can be formulated in a standard way in the form of the Fully Abstract Embedding (FAE) criterion and have established the criterion for the GTLC, a basic gradual type system.

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[17] and prove a parametricity result. An error in the proof was later corrected [Ahmed et al. 2011c].

Next, Ahmed et al. [2009b, 2011b] 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. 2011a]. However, they do not consider parametricity or preservation of System F contextual equivalences (i.e., fully abstract embedding).

Ahmed et al. [2017] present λ_B: a new version of the polymorphic blame calculus rectifying certain "topsy-turvy" aspects of its operational semantics (see Section 7). Additionally, they prove a parametricity result for this calculus, which we have discussed from the perspective of our results in Section 8.3.

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

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 certain special restrictions to the consistency and precision relation of the system (later criticised by others [Toro et al. 2019]), which allow them to prove a version of the gradual guarantee. However, 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.

Xie et al. [2018] present a gradually typed language with parametric polymorphism, focusing on the interplay of gradual typing with the subtyping relation in the presence of implicit polymorphism. The result of their work is a source language that elaborates to λ_B, which we discussed before. As such, the language provides the same form of parametricity than λ_B and we think our results apply to it as well.

More recently, Toro et al. [2019] proposed a new gradually typed calculus with explicit polymorphism, based on the AGT methodology by Garcia et al. [2016]. The methodology allows them to construct a system that satisfies the refined criteria by Siek et al. [2015], except for the Dynamic Gradual Guarantee. They show that this property is in conflict with parametricity in their system, but they demonstrate a weaker property which they do satisfy. The gradual type is also a universal type in their system and they prove a TWLR-based parametricity.

Finally, New et al. [2019a] have developed PolyG$^\nu$, the first gradual language to support both parametricity and the dynamic gradual guarantee (which they refer to as graduality) at the same time. The syntax of PolyG$^\nu$ departs from System F, requiring programmers to write explicit sealing and unsealing annotations. Like previous work, New et al. use dynamic sealing and they prove a form of parametricity based on a TWLR logical relation, although it is closer to standard System F logical relations in some other respects.

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.

Alternatives for full abstraction. The property of full abstraction was proposed by Abadi [1998]. Abate et al. [2019] provide a lattice of secure compilation criteria (dubbed robust compilation) that preserve classes of hyperproperties [Clarkson and Schneider 2010], i.e., arbitrary behaviours. The general nature of the framework makes it hard to make precise statements, but we believe our counterexamples would also disprove most of these alternative properties.

10 CONCLUSION

This work started out years ago as an effort to prove Sumii and Pierce's conjecture, discussed in Section 5. Our failure to do so has perhaps proven more interesting than a success might have been. Specifically, we disprove the conjecture rather than proving it, we disprove another conjecture for languages with non-parametric parametricity 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. In addition to highlighting these problems, we discuss in Section 8 some ideas about how the issues might be solved. None of them seem easy to solve and the solutions we propose have downsides of their own, but we do believe they might be worth exploring further.

During the work on the Sumii-Pierce conjecture, we also gained some more high-level insights about variations of parametricity (type-world LRs versus lexically-scoped LRs) and their relation to the lexical scope of type variables and the existence of universal types. These insights were not mentioned explicitly in the previous version of this paper [Devriese et al. 2018] and in discussions with experts in the field, we found that these insights were unclear. For this reason, this paper focuses very clearly on these more high-level insights and we hope they may improve understanding of the issues involved.

ACKNOWLEDGMENTS

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 Air Force Office of Scientific Research under award number FA9550-21-1-0054. 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). This work was partially supported by the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019). The authors thank Phil Wadler for interesting comments and suggestions.

REFERENCES

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

Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15:8:1–8:29, July 2012. ISSN 1094-9224. doi: 10.1145/2240276.2240279.

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, April 1991. ISSN 0164-0925. doi: 10.1145/103135.103138. URL http://doi.acm.org/10.1145/103135.103138.

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 5:92–103, 1995.

Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *IEEE Symposium on Logic in Computer Science*, pages 105–116, 1998.

Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, pages 74–88, 1999.

Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Principles of Programming Languages*, pages 302–315. ACM, 2000. doi: 10.1145/325694.325734.

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. CCS '18, 2018.

Carmine Abate, Roberto Blanco, Deepak Garg, Marco Hriţcu, Cătălin andPatrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32th Computer Security Foundations Symposium*, CSF 2019, June 2019.

Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium*, pages 171–185. IEEE, 2012. doi: 10.1109/CSF.2012.12.

Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, USA, 2004. AAI3136691.

Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming*, pages 157–168. ACM, 2008. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411227.

Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 431–444. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034830. URL http://doi.acm.org/10.1145/2034773.2034830.

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

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

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. Technical report, 2011a. URL https://plt.eecs.northwestern.edu/blame-for-all/.

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

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

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

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

Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In *Principles of Programming Languages*, pages 251–262. ACM, 2007. doi: 10.1145/1190216.1190253.

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

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. Reconciling noninterference and gradual typing. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 116–129. ACM, 2020. doi: 10.1145/3373718.3394778. URL https://doi.org/10.1145/3373718.3394778.

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

Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science*, 13(4 lmcs:4011), October 2017. doi: 10.23638/LMCS-13(4:2)2017.

Dominique Devriese, Marco Patrignani, and Frank Piessens. Parametricity versus the universal type. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2018, Los Angeles, CA, USA, 2016*, 2018.

Derek Dreyer, Georg Neis, and Lars Birkedal. The Impact of Higher-order State and Control Effects on Local Relational Reasoning. In *International Conference on Functional Programming*, pages 143–156. ACM, 2010. doi: 10.1145/1863543.1863566.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011a.

Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. Semantics of type systems lecture notes, 2011b. Available at: https://plv.mpi-sws.org/semantics/2017/lecturenotes.pdf.

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

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

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

Ronald Garcia and Éric Tanter. Gradual typing as if types mattered. Workshop on Gradual Typing, 2020. URL https://wgt20.irif.fr/wgt20-final28-acmpaginated.pdf.

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

Jean-Yves Girard. *Interprétation Fonctionelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur.* PhD thesis, Université Paris VII, 1972.

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

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. ISSN 0167-6423. doi: 10.1016/0167-6423(94)00004-2.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. In *International Conference on Functional Programming*. ACM, 2017.

Koen Jacobs, Amin Timany, and Dominique Devriese. Fully abstract from static to gradual. *Proceedings of the ACM on Programming Languages*, 5(POPL):7:1–7:30, January 2021. doi: 10.1145/3434288.

Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *Computer Security Foundations Symposium*, pages 161–174. IEEE Computer Society, 2011. doi: 10.1109/CSF.2011.18.

Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Towards a fully abstract compiler using micro-policies: Secure compilation for mutually distrustful components. *CoRR*, abs/1510.00697, 2015. URL http://arxiv.org/abs/1510.00697.

Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *Computer Security Foundations Symposium*, 2016.

Elizabeth Labrada, Matías Toro, Éric Tanter, and Dominique Devriese. Plausible sealing for gradual parametricity. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA), 2022. To appear.

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

Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. Implementing a secure abstract machine. In *Symposium on Applied Computing*, pages 2041–2048. ACM, 2016. ISBN 978-1-4503-3739-7. doi: 10.1145/2851613.2851796. URL http://doi.acm.org/10.1145/2851613.2851796.

John R. Longley. Universal types and what they are good for. In *Domain Theory, Logic and Computation*, Semantic Structures in Computation, pages 25–63. Springer, Dordrecht, 2003. doi: 10.1007/978-94-017-1291-0_2.

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! volume 4960 of *LNCS*, pages 16–31. 2008.

Jacob Matthews and Robert Bruce Findler. Operational Semantics for Multi-language Programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, April 2009. ISSN 0164-0925. doi: 10.1145/1498926.1498930.

John C. Mitchell. Representation independence and data abstraction. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 263–276, New York, NY, USA, 1986. ACM. doi: 10.1145/512644.512669. URL http://doi.acm.org/10.1145/512644.512669.

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

James H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, January 1973a. ISSN 0001-0782. doi: 10.1145/361932.361937.

James H. Morris, Jr. Types are not sets. In *Principles of Programming Languages*, pages 120–124, 1973b.

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

Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *Journal of Functional Programming*, 21: 497–562, 2011. ISSN 1469-7653.

Max New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *ICFP'16*, 2016a.

Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming*, pages 103–116. ACM, 2016b. doi: 10.1145/2951913.2951941.

Max S. New, Dustin Jamner, and Amal Ahmed. Graduality and parametricity: Together again for the first time. *Proceedings of the ACM on Programming Languages*, 4(POPL):46:1–46:32, December 2019a. doi: 10.1145/3371114.

Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *Proc. ACM Program. Lang.*, 3(POPL):15:1–15:31, 2019b. doi: 10.1145/3290328. URL https://doi.org/10.1145/3290328.

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

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

Marco Patrignani and Deepak Garg. Secure Compilation and Hyperproperties Preservation. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium CSF 2017, Santa Barbara, USA*, CSF 2017, 2017.

Marco Patrignani and Deepak Garg. Robustly safe compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019*, ESOP'19, 2019.

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37:6:1–6:50, April 2015. ISSN 0164-0925. doi: 10.1145/2699503.

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

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

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

Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism. manuscript, 2000. URL http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/infohide.pdf.

Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 309–326.

Springer Berlin, Heidelberg, 1998. doi: 10.1007/BFb0055063.

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

John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, pages 513–523. North Holland, 1983.

Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pages 241–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-705-2. doi: 10.1145/888251.888274. URL http://doi.acm.org/10.1145/888251.888274.

Manfred Schmidt-Schauß, David Sabel, Joachim Niehren, and Jan Schwinghammer. Observational program calculi and the correctness of translations. *Theoretical Computer Science*, 577:98 – 124, 2015. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2015.02.027.

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

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *SCHEME*, pages 81–92, 2006.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *Summit on Advances in Programming Languages*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.SNAPL.2015.274.

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

Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1-2): 11–49, April 2000. ISSN 1388-3690, 1573-0557. doi: 10.1023/A:1010000313106.

Eijiro Sumii and Benjamin C. Pierce. Logical Relations for Encryption. *J. Comput. Secur.*, 11(4):521–554, July 2003. ISSN 0926-227X.

Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Principles of Programming Languages*, pages 161–172, 2004.

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

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 964–974, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593491X. doi: 10.1145/1176617.1176755. URL https://doi.org/10.1145/1176617.1176755.

Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Trans. Program. Lang. Syst.*, 40(4):16:1–16:55, 2018. doi: 10.1145/3229061. URL https://doi.org/10.1145/3229061.

Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.*, 3(POPL): 17:1–17:30, January 2019. ISSN 2475-1421. doi: 10.1145/3290330.

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

Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1), 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.042.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Programming Languages and Systems*, pages 1–16. Springer, Berlin, Heidelberg, March 2009. doi: 10.1007/978-3-642-00590-9_1.

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015. doi: 10.1109/SP.2015.9.

Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, December 1995. ISSN 1573-0557. doi: 10.1007/BF01018828.

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent Subtyping for All. In Amal Ahmed, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 3–30. Springer International Publishing, 2018.