On Polymorphic Gradual Typing

YUU IGARASHI, Kyoto University
TARO SEKIYAMA, IBM Research - Tokyo
ATSUSHI IGARASHI, Kyoto University

We study an extension of gradual typing—a method to integrate dynamic typing and static typing smoothly in a single language—to parametric polymorphism and its theoretical properties, including conservativity of typing and semantics over both statically and dynamically typed languages, type safety, blame-subtyping theorem, and the gradual guarantee—the so-called refined criteria, advocated by Siek et al. We develop System F_G, which is a gradually typed extension of System F with the dynamic type and a new type consistency relation, and translation to a new polymorphic blame calculus System F_C, which is based on previous polymorphic blame calculi by Ahmed et al. The design of System F_G and System F_C, geared to the criteria, is influenced by the distinction between static and gradual type variables, first observed by Garcia and Cimini. This distinction is also useful to execute statically typed code without incurring additional overhead to manage type names as in the prior calculi. We prove that System F_G satisfies most of the criteria: all but the hardest property of the gradual guarantee on semantics. We show that a key conjecture to prove the gradual guarantee leads to the Jack-of-All-Trades property, conjectured as an important property of the polymorphic blame calculus by Ahmed et al.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Polymorphism**; **Semantics**;

Additional Key Words and Phrases: gradual typing, parametric polymorphism, gradual guarantee

ACM Reference format:
Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.* 1, 1, Article 40 (September 2017), 29 pages.
https://doi.org/10.1145/3110284

1 INTRODUCTION

1.1 Gradual Typing

Types are a broadly used, lightweight tool to verify programs and many programming languages adopt either static typing or dynamic typing to prevent programs from exposing undefined behavior. Static typing, which guarantees that programs behave as types, is useful for early and exhaustive detection of certain classes of errors and makes it possible to generate more efficient machine code, but programmers cannot run their programs until they pass static typechecking, which sometimes prevents writing programs as they hope. Dynamic typing enables rapid software development and flexible programming, while it often consumes more computational resources and it is not easy to support exhaustive error detection. In summary, they are complementary.

Integration of static and dynamic typing has been studied widely, from theory [Abadi et al. 1991, 1995; Ahmed et al. 2011; Matthews and Ahmed 2008; Matthews and Findler 2009; Siek and Taha 2006; Thatte 1990; Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009] to practice [Bierman et al. 2014, 2010; Bonnaire-Sergeant et al. 2016; Facebook 2017; Gronski et al. 2006; Meijer and Drayton 2004; Tobin-Hochstadt and Felleisen 2008; Vitousek et al. 2014], to cite a few, and *gradual typing* [Siek and Taha 2006] is a methodology for achieving a full spectrum between fully statically

2475-1421/2017/9-ART40
https://doi.org/10.1145/3110284

typed and fully dynamically typed programs. For example, gradual typing enables programmers to write a fully dynamically typed program, as a prototype, at the early stage of development, change its pieces to statically typed code gradually, and finally obtain a fully statically typed program, which has no run-time typing errors, in a single programming language. Since Siek and Taha [2006] proposed gradual typing for the simply typed lambda calculus, their idea has been applied to many programming concepts such as objects [Siek and Taha 2007], generics [Ina and Igarashi 2011], effects [Bañados Schwerter et al. 2014], and type inference [Garcia and Cimini 2015; Siek and Vachharajani 2008]. More recently, even general approaches to gradual typing, which give a gradually typed version of a statically typed language, have been studied [Cimini and Siek 2016, 2017; Garcia et al. 2016].

Gradual typing "gradualizes" statically typed languages with two key components: the dynamic type and type consistency. The dynamic type, denoted by $\star$, is the type of dynamically typed code. All statically typed expressions can be injected to $\star$ and, conversely, values of $\star$ can be projected to any type with run-time checking. In fact, gradual typing allows interaction between not only a fully static type and the dynamic type but also types on the spectrum, such as $\star \rightarrow \mathsf{Int}$ and $\mathsf{Int} \rightarrow \star$, by using *type consistency*. Type consistency is the same as type equality except that it allows the dynamic type to match with any type. By using type consistency instead of type equality, gradually typed languages achieve fine-grained transition between program pieces on the spectrum, while all typing errors in a program without the dynamic type are detected statically.

1.2 Our Work

This work studies a gradually typed language with parametric polymorphism. We propose System $\mathrm{F_G}$, a gradual extension of System F [Girard 1972; Reynolds 1974]. The design of System $\mathrm{F_G}$, in particular its type consistency, is geared to the so-called "refined criteria" proposed by Siek et al. [2015], which are properties that gradually typed languages should satisfy. In System $\mathrm{F_G}$, type variables are classified into two kinds—*static type variables* and *gradual type variables*—as first observed by Garcia and Cimini [2015]. Gradual type variables can be consistent with the dynamic type whereas static type variables cannot. The distinction of static and gradual type variables makes the dynamic semantics of System $\mathrm{F_G}$ "pay-as-you-go" in the sense of Siek and Taha [2006], that is, statically typed code can be executed without much additional overhead—in particular, the run-time system of System $\mathrm{F_G}$ would be able to run well-typed System F terms in almost the same way as System F.

The dynamic semantics of System $\mathrm{F_G}$ is defined by translation to a new polymorphic blame calculus System $\mathrm{F_C}$, which is an extension of (a subset of) the simply typed blame calculus [Wadler and Findler 2009] and aims at dynamic enforcement of parametricity. Our blame calculus is largely based on previous polymorphic blame calculi [Ahmed et al. 2011, 2017] which use type bindings for enforcing parametricity at run time, but System $\mathrm{F_C}$ inherits the distinction between static and gradual type variables from System $\mathrm{F_G}$ and has slightly different cast semantics.

We show that System $\mathrm{F_G}$ and System $\mathrm{F_C}$ meet almost all of the refined criteria [Siek et al. 2015], except the hardest property called the gradual guarantee on semantics, which is left as a conjecture. Our contributions are summarized as follows.

- We carefully define type consistency of System $\mathrm{F_G}$ so that it is a conservative extension over System F in terms of typing. Our key observation is that a polymorphic type can be consistent with a nonpolymorphic type only if the nonpolymorphic type contains the dynamic type. We also show that untyped terms can be embedded in System $\mathrm{F_G}$.
- We introduce a new polymorphic blame calculus System $\mathrm{F_C}$ and give formal translation from System $\mathrm{F_G}$ to System $\mathrm{F_C}$. The semantics of System $\mathrm{F_C}$ is conservative over both System F and

Syntax $\boxed{T \triangleright T}$ **Matching**

Types	$A, B ::= \iota \mid \star \mid A \to B$	Terms	$e ::= c \mid x \mid \lambda x{:}A.\ e \mid e\ e$	$A \to A' \triangleright A \to A'$
Values	$v ::= c \mid \lambda x{:}A.\ e$	Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : A$	$\star \triangleright \star \to \star$

$\boxed{A \sim B}$ **Type consistency**

$$A \sim A \ (\text{C\_Refl}) \qquad \star \sim A \ (\text{C\_StarL}) \qquad A \sim \star \ (\text{C\_StarR}) \qquad \frac{A \sim A' \qquad B \sim B'}{A \to B \sim A' \to B'} \ (\text{C\_Arrow})$$

$\boxed{\Gamma \vdash_G e : A}$ **Typing**

$$\frac{ty(c) = A}{\Gamma \vdash_G c : A} \ (\text{T\_Const\_G}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_G x : A} \ (\text{T\_Var\_G}) \qquad \frac{\Gamma, x : A \vdash_G e : B}{\Gamma \vdash_G \lambda x{:}A.\ e : A \to B} \ (\text{T\_Abs\_G})$$

$$\frac{\Gamma \vdash_G e_1 : A_1 \qquad \Gamma \vdash_G e_2 : A_2 \qquad A_1 \triangleright A_{11} \to A_{12} \qquad A_{11} \sim A_2}{\Gamma \vdash_G e_1\ e_2 : A_{12}} \ (\text{T\_App\_G})$$

Fig. 1. Static semantics of GTLC

the untyped lambda calculus. We show type safety of System F_G via progress and subject reduction of System F_C as well as soundness of the translation.

- We discuss type-erasing semantics of System F_C and implement a prototype interpreter[1] based on it in OCaml, demonstrating that a "pay-as-you-go" implementation is possible.
- We show properties relevant to the Blame Theorem [Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009], which justifies the intuition that all run-time typing errors are triggered by only untyped code, for System F_C. We define our subtyping relations so that they are subsets of type consistency because we are interested in only well-typed terms. We have bidirectional factoring of ordinary subtyping, a.k.a., Tangram [Wadler 2015].
- We extend precision of types and terms to formalize the gradual guarantee [Siek et al. 2015], which is concerned with invariance of typing and semantics through code evolution. Although the gradual guarantee on typing is proved, the gradual guarantee on semantics is still left as a conjecture. We discuss a few key lemmas that have been proved and one key conjecture, which turns out to be related to another tough conjecture, *Jack-of-All-Trades* [Ahmed et al. 2013], an important property of the prior polymorphic blame calculi.

System F_G is fully explicitly typed—every variable has to declare its type and type abstraction/application is explicit—so it may be far from practical polymorphically typed languages, where some type information can be omitted thanks to type inference. Ideally, we should design a surface language where many type annotations are inferred and give elaboration from the surface language to a blame calculus, for which operational semantics is defined. System F_G can be considered an intermediate language after type inference and before translation to a blame calculus. The design of a surface language and study of type inference (translation to System F_G) are left as future work.

The rest of the paper is organized as follows. We review a simple gradually typed λ-calculus, a simply typed blame calculus, and the refined criteria in Section 2. We present an overview of System F_G and System F_C in Section 3. The definitions of System F_G and System F_C with their properties are presented in Sections 4 and 5, respectively. After we discuss type erasure in Section 6, we prove the Blame Theorem in Section 7 and discuss the gradual guarantee in Section 8. After discussing related work in Section 9, we conclude in Section 10.

In this paper, we omit the details of proofs; they can be found in the supplementary material.

[1]`https://github.com/SoftwareFoundationGroupAtKyotoU/SystemFg`

2 GRADUALLY TYPED λ-CALCULUS

In this section, we review a simple gradually typed language, a blame calculus, which works as an internal language for the first language, and Siek et al.'s criteria that gradually typed languages should meet.

2.1 The Source Language

We show the source language with its syntax and type system in Fig. 1. The definition is mostly the same as the Gradually Typed Lambda Calculus (GTLC) by Siek et al. [2015]. In this paper, GTLC refers to the definition in Fig. 1.

The syntax is a standard simply-typed lambda calculus extended with the dynamic type $\star$. The metavariable ι ranges over base types, which include at least Int and Bool. A term is a constant c, a variable x, a lambda abstraction $\lambda x{:}A.\ e$, or an application $e\ e$. A constant includes at least integers and Booleans. A value is a constant or a lambda abstraction.

A notable feature of gradually typed languages is type consistency $A \sim B$, which is defined by relaxing type equality, used in typechecking, say, function applications. The dynamic type is considered as a wild card in type consistency for allowing dynamically typed code to always pass typecheck. Formally, type consistency is defined as the least reflexive and compatible relation including $\star \sim A$ for any type A (rules (C_StarL) and (C_StarR)). For example, we can derive $\star \rightarrow \mathrm{Int} \sim \mathrm{Bool} \rightarrow \star$. It is easy to see that type consistency is symmetric but *not* transitive.

Typing judgment $\Gamma \vdash_G e : A$ means that term e has type A in type environment Γ. All the rules but (T_App_G) are standard. The function ty in (T_Const_G) assigns a constant a first-order type of the form $\iota_1 \rightarrow \cdots \rightarrow \iota_n$. In an application, the type of e_1 is normalized to a function type $A_{11} \rightarrow A_{12}$ by *matching* $A_1 \triangleright A_{11} \rightarrow A_{12}$. By the normalization, a term of $\star$ can be applied to any term as if it is a function; it is checked at run-time whether it is really a function. In (T_App_G), type consistency replaces usual type equality between the function argument type A_{11} and the actual argument type A_2, allowing, for example, a dynamically typed expression to be passed to a statically typed function.

2.2 The Internal Language: Blame Calculus

We define the dynamic semantics of GTLC by cast insertion translation to (a subset of) the *blame calculus* by Wadler and Findler [2009]. The definition of the blame calculus, adapted from Siek et al. [2015], is found in Fig. 2. Throughout this paper, the *blame calculus* refers to the definition in Fig. 2.

Types are the same as GTLC. A ground type, used as a tag of injection ($w : G \Rightarrow^p \star$), is either a base type or $\star \rightarrow \star$. A blame label, which decorates a cast, abstracts a program location such as the numbers of the row and the column. We have an involutive operator $\overline{p}$ (called negation) on blame labels, that is, $\overline{\overline{p}} = p$. This is used to blame the context (not the subject of a cast). Terms, ranged over by f, are GTLC terms augmented with casts, $f : A \Rightarrow^p B$, and blame, blame p, which is an uncatchable exception to notify cast failure. We abbreviate a sequence of two casts $((f : A \Rightarrow^{p_1} B) : B \Rightarrow^{p_2} C)$ to $f : A \Rightarrow^{p_1} B \Rightarrow^{p_2} C$. A value, ranged over by w, is a constant, a lambda abstraction, a value with a cast from a ground type to $\star$, or a value with a cast between function types. Evaluation contexts indicate that the evaluation is left-to-right and call-by-value.

Typing rules are straightforward. Rules (T_Const_C), (T_Var_C), and (T_Abs_C) are the same as those of GTLC. Rule (T_App_C) requires the actual and formal argument types to be equal and (T_Cast_C) allows only casts for pairs of consistent types. Blame is given any type (T_Blame_C).

The dynamic semantics of GTLC is defined by cast insertion ($\rightsquigarrow$) and reduction ($\longrightarrow$) in the blame calculus. We write $\longrightarrow^\star$ for the reflexive transitive closure of $\longrightarrow$.

Syntax

Types	$A, B, C ::= \iota \mid \star \mid A \to B$
Ground types	$G, H ::= \iota \mid \star \to \star$
Blame labels	$p, q ::= \ell$
Terms	$f ::= c \mid x \mid \lambda x{:}A.\, f \mid f\, f \mid f : A \Rightarrow^p B \mid \text{blame } p$
Values	$w ::= c \mid \lambda x{:}A.\, f \mid w : G \Rightarrow^p \star \mid w : A \to B \Rightarrow^p A' \to B'$
Evaluation contexts	$\mathrm{E} ::= [\,] \mid \mathrm{E}\, f \mid w\, \mathrm{E} \mid \mathrm{E} : A \Rightarrow^p B$
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : A$

$\boxed{\Gamma \vdash_C f : A}$ **Typing**

$$\frac{ty(c) = A}{\Gamma \vdash_C c : A}\ (\text{T\_Const\_C}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_C x : A}\ (\text{T\_Var\_C}) \qquad \frac{\Gamma, x : A \vdash_C f : B}{\Gamma \vdash_C \lambda x{:}A.\, f : A \to B}\ (\text{T\_Abs\_C})$$

$$\frac{\Gamma \vdash_C f_1 : A \to B \qquad \Gamma \vdash_C f_2 : A}{\Gamma \vdash_C f_1\, f_2 : B}\ (\text{T\_App\_C}) \qquad \frac{\Gamma \vdash_C f : A \qquad A \sim B}{\Gamma \vdash_C f : A \Rightarrow^p B : B}\ (\text{Cast}) \qquad \Gamma \vdash_C \text{blame } p : A\ (\text{T\_Abort\_C})$$

$\boxed{\Gamma \vdash e \rightsquigarrow f : A}$ **Cast insertion**

$$\frac{\Gamma \vdash e_1 \rightsquigarrow f_1 : A_1 \qquad \Gamma \vdash e_2 \rightsquigarrow f_2 : A_2 \qquad A_1 \triangleright A_{11} \to A_{12} \qquad A_{11} \sim A_2}{\Gamma \vdash e_1\, e_2 \rightsquigarrow (f_1 : A_1 \Rightarrow^{\ell}_{?} A_{11} \to A_{12})\ (f_2 : A_2 \Rightarrow^{\ell}_{?} A_{11}) : A_{12}}\ (\text{CI\_App})$$

$\boxed{f \longrightarrow f'}$ **Reduction**

$$\begin{array}{lll}
c_1\, c_2 \longrightarrow \delta(c_1, c_2) & & (\text{R\_Delta}) \\
(\lambda x{:}A.\, f)\, w \longrightarrow f[x := w] & & (\text{R\_Beta}) \\
w : G \Rightarrow^p G \longrightarrow w & \text{if } G \neq \star \to \star & (\text{R\_Id}) \\
w : \star \Rightarrow^p \star \longrightarrow w & & (\text{R\_IdStar}) \\
(w : A \to B \Rightarrow^p A' \to B')\, w' \longrightarrow (w\, (w' : A' \Rightarrow^{\bar{p}} A)) : B \Rightarrow^p B' & & (\text{R\_Wrap}) \\
w : A \to A' \Rightarrow^p \star \longrightarrow w : A \to A' \Rightarrow^p \star \to \star \Rightarrow^p \star & \text{if } A \to A' \neq \star \to \star & (\text{R\_Ground}) \\
w : \star \Rightarrow^p A \to A' \longrightarrow w : \star \Rightarrow^p \star \to \star \Rightarrow^p A \to A' & \text{if } A \to A' \neq \star \to \star & (\text{R\_Extend}) \\
w : G \Rightarrow^q \star \Rightarrow^p G \longrightarrow w & & (\text{R\_Collapse}) \\
w : G \Rightarrow^q \star \Rightarrow^p H \longrightarrow \text{blame } p & \text{if } G \neq H & (\text{R\_Conflict})
\end{array}$$

$$\frac{\mathrm{E} \neq [\,]}{\mathrm{E}[\text{blame } p] \longrightarrow \text{blame } p}\ (\text{R\_Blame}) \qquad \frac{f \longrightarrow f'}{\mathrm{E}[f] \longrightarrow \mathrm{E}[f']}\ (\text{R\_Env})$$

Fig. 2. Blame Calculus

Definition 2.1 (Dynamic semantics of GTLC). For e with $\emptyset \vdash_G e : A$, $e \Downarrow_G w \overset{\text{def}}{\iff} \emptyset \vdash e \rightsquigarrow f : A$ and $f \longrightarrow^{\star} w$.

Cast insertion $\Gamma \vdash e \rightsquigarrow f : A$ shown in the middle of Fig. 2 is a syntax-directed translation from GTLC terms to blame calculus terms. The only interesting rule is for application: casts are inserted so as to reflect the results of matching and consistency. Here, $f : A \Rightarrow^p_? B$ denotes f when $A = B$, otherwise $f : A \Rightarrow^p B$. Translation for the other forms of terms are trivial (and homomorphic) because no casts are inserted, and so rules are omitted.

Finally, reduction rules are at the bottom of Fig. 2. The first two rules (R_Delta) and (R_Beta) are standard. Application of a primitive function $c_1\ c_2$ is processed by the oracle δ, which produces a constant as a result of application (R_Delta). We suppose that the behavior of δ is compatible with its type. The other rules are concerned about casts and blame. Identity casts for ground types (except for $\star \to \star$) and $\star$ disappear immediately ((R_Id) and (R_IdStar), respectively). A cast between function types breaks down into a cast of the domain types and a cast of the codomain types, when it is applied to an argument (R_Wrap). The negation of the label means that the target of the blame changes from w (a term contained in the cast) to w' (a context containing the cast). Casts between a function type (except for $\star \to \star$) and $\star$ is factored into two casts with $\star \to \star$ in the middle ((R_Ground) and (R_Extend)). Although the previous blame calculus by Wadler and Findler [2009] deals with a more general case $w : A \Rightarrow^p \star$ (where A is neither ground nor $\star$), we consider only the case $w : A \to A' \Rightarrow^p \star$ because this is the only case. If we had more type constructors, a general rule would be useful. Rules (R_Collapse) and (R_Conflict) describe run-time checks triggered by composition of injection and projection, which may result in blame. Blame in an evaluation context results in the blame (R_Abort). Finally, (R_Env) describes reduction under an evaluation context.

2.3 Refined Criteria in GTLC

We review the properties desired for GTLC, advocated by Siek et al. [2015], who called them refined criteria. We provide brief explanations and discuss their desirability here. For their formal statements, some of which are omitted here, readers are referred to Siek et al. [2015].

Conservative extension of STLC and the untyped lambda calculus. GTLC is a superset of the simply typed lambda calculus (STLC) and the untyped lambda calculus (DTLC). Conservativity over STLC states that for terms in STLC, that is, terms without $\star$, typing and semantics of GTLC is equivalent to those of STLC. In other words, as long as programmers do not use $\star$ explicitly, GTLC is just as STLC. Conservativity over DTLC states that there exists an embedding of an untyped lambda term to a GTLC term of type $\star$ and the results of reduction in both languages agree. In practice, these properties ensure that programmers can use existing statically/dynamically typed code or libraries in a gradualized language without modification or semantic changes and, conversely, that statically typed code developed in a gradually typed language can be brought back to the statically typed world as it is.

Type safety. Every well-typed term in GTLC results in either a value, blame, or divergence. This argues that a well-typed term does not cause untrapped errors.

Blame-Subtyping Theorem. This is a theorem of soundness about blame. It states that a cast between two types in the subtyping relation, whose definition is omitted, never raises blame. The subtyping relation, which characterizes casts that never fail, can be used for static analysis and maybe optimization to remove redundant casts.

The Gradual Guarantee. In a gradual type system, adding more static types to a program serves only to expose more errors either statically or dynamically and should not change the behavior of a program. In other words, for two programs, which are the same except that one is given more static type annotations than the other, the one with more static types is (1) less likely to be statically typechecked (due to more static errors) and (2) more likely to get blame but does not change the result (the value or divergence) otherwise. The gradual guarantee formulates this expectation.

First, to formalize the relation between programs using more or less $\star$, *precision* $\sqsubseteq$, precongruence on types and terms in GTLC, is introduced. Intuitively, $\sqsubseteq$ means that the right-hand side has more dynamic types. Here are all the rules of type precision and the key rule of term precision.

$$A \sqsubseteq A \qquad A \sqsubseteq \star \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'} \qquad\qquad \frac{e \sqsubseteq e' \qquad A \sqsubseteq A'}{\lambda x{:}A.\ e \sqsubseteq \lambda x{:}A'.\ e'}$$

Type precision is related to consistency in the following sense [Siek and Wadler 2010]: $A \sim B$ if and only if there is the greatest lower bound of A and B with respect to precision.

Now, we are ready to show formal statements of the gradual guarantee. Here, $e \Uparrow_G$ means that e (after cast insertion translation) diverges. (1) and (2) state that making types of well-typed terms less precise does not affect typeability and semantics; and (3) states that making types more precise may result in more blame; but otherwise the semantics is the same (modulo precision).

Theorem 2.2 (The Gradual Guarantee [Siek et al. 2015]). *Suppose* $\emptyset \vdash_G e : A$ *and* $e \sqsubseteq e'$.

(1) $\emptyset \vdash_G e' : A'$ *and* $A \sqsubseteq A'$ *for some* A'.
(2) *If* $e \Downarrow_G w$ *then* $e' \Downarrow_G w'$ *and* $w \sqsubseteq w'$ *for some* w'. *If* $e \Uparrow_G$ *then* $e' \Uparrow_G$.
(3) *If* $e' \Downarrow_G w'$ *then* $e \Downarrow_G w$ *and* $w \sqsubseteq w'$ *for some* w' *or* $e \Downarrow_G \textsf{blame}\, p$ *for some* p. *If* $e' \Uparrow_G$ *then* $e \Uparrow_G$ *or* $e \Downarrow_G \textsf{blame}\, p$ *for some* p.

We can show that, if e' invokes blame, so does e as a corollary of (2), while we cannot say much about e' when e triggers blame because that blame may or may not remain in e'.

3 KEY IDEAS OF SYSTEM F_G AND SYSTEM F_C

We will design a polymorphic extension of GTLC so that it meets (an adapted version of) the criteria discussed in the last section. In this section, we give the key ideas behind our source language System F_G and internal language System F_C informally.

3.1 Type consistency

An advantage of gradually typed languages is that dynamically typed code can incorporate with statically typed code, so we design type consistency in System F_G and System F_C so that dynamically typed and polymorphically typed terms can interact smoothly, as follows (we use $\forall X.A$ and $\Lambda X.\ v$ for polymorphic types and type abstractions, respectively):

$$(\lambda y{:}\forall X.X \to X.\ y\ \text{Int}\ 1)\ (\lambda x{:}\star\ .\ x)$$

where nonpolymorphic value $\lambda x{:}\star\ .\ x$ is passed to a function that expects polymorphic arguments. To accept this expression, type consistency has to allow $\forall X.X \to X \sim \star \to \star$ to be derived. Our type consistency supports more general cases: it relates a polymorphic type and a non-$\forall$ type.[2] Then, since the dynamic type is consistent with any type—even if it is a type variable—we can derive $\forall X.X \to X \sim \star \to \star$.

We have to, however, take care of what non-$\forall$ types can be consistent with polymorphic types. In particular, allowing a fully static non-$\forall$ type to be consistent with a fully static polymorphic type would result in violation of conservativity over typing of System F, one of Siek et al.'s refined criteria. For example, a polymorphic gradually typed language with type consistency that relates $\text{Int} \to \text{Int}$ to $\forall X.\text{Int} \to \text{Int}$ would accept $(\lambda y{:}\forall X.\text{Int} \to \text{Int}.\ y\ \text{Int}\ 1)\ (\lambda x{:}\text{Int}.\ x)$ even though it is ill-typed in System F. To prevent such a term to be accepted, our type consistency requires non-$\forall$ types which are consistent with polymorphic types to have one or more occurrences of $\star$. We coin *quasi-polymorphic* types for such non-$\forall$ types in the sense that it may be used where a polymorphic type is expected. For example, $\star \to \star$ is a quasi-polymorphic but $\text{Int} \to \text{Int}$ is not. Thus, that requirement allows us to derive $\forall X.X \to X \sim \star \to \star$ while disallowing deriving $\forall X.\text{Int} \to \text{Int} \sim \text{Int} \to \text{Int}$.

By adding a structural rule for polymorphic types, there are two additional consistency rules (and a symmetric variant for the first):

[2]A non-$\forall$ type refers to a type whose top-most type constructor is not $\forall$.

$$\frac{A \sim B \qquad B \neq \forall X.B' \qquad \star \in \mathrm{Types}(B) \qquad X \notin \mathrm{Ftv}(B)}{\forall X.A \sim B} \qquad \frac{A \sim B}{\forall X.A \sim \forall X.B}$$

In the left rule, the predicate $\star \in \mathrm{Types}(B)$ means that $\star$ occurs somewhere in B, and so combining it with the side condition $B \neq \forall X.B'$ says that B is a quasi-polymorphic type. $\mathrm{Ftv}(A)$ denotes the set of free type variables in A and the condition $X \notin \mathrm{Ftv}(B)$ avoids that X captures free type variables in B. These rules allow us to derive, for example, the following relations:

$$\forall X.X \to \mathrm{Bool} \sim \star \to \mathrm{Bool} \qquad \forall X.X \to \mathrm{Int} \sim \forall X.X \to \star$$

in addition to $\forall X.X \to X \sim \star \to \star$. On the other hand, as intended, the following consistency relations *do not* hold:

$$\forall X.X \to \mathrm{Bool} \sim \mathrm{Bool} \to \mathrm{Bool} \qquad \mathrm{Int} \to \mathrm{Int} \sim \forall X.\mathrm{Int} \to \mathrm{Int}$$

because non-$\forall$ types in the examples above do not involve $\star$. Though we will refine type consistency with distinction of type variables in Section 3.3, the rules above are the heart of our type consistency.

Ahmed et al. [2011] introduce type compatibility $A \prec B$ (where a cast from A to B is allowed. It also relates a polymorphic type to a non-$\forall$ type, for a polymorphic blame calculus instead of type consistency, and Ahmed et al. [2017] also follows it. We considered whether we could use it as type consistency in the source language but we soon found that type compatibility would break conservativity over typing of System F because their type compatibility allows $\forall X.\mathrm{Int} \prec \mathrm{Int}$, for example.[3] Their type compatibility is asymmetric: For example, $\forall X.A \prec [B/X]A$ (for any B) but not vice versa. For source language following the GTLC, we stick to a symmetric relation.

3.2 Type bindings

As well known, naively adding the dynamic type to a polymorphically typed language results in the lack of parametricity. For example, let us consider $f_1 = \Lambda X.\ \lambda x{:}\mathrm{Int}.\ (x : \mathrm{Int} \Rightarrow^p \star \Rightarrow^q X)$, which performs run-time check that x of Int is at X via the injection to $\star$. Usual, substitution-based type application makes this term nonparametric:

$$\begin{array}{llll} f_1\ \mathrm{Int}\ 1 & \longrightarrow^\star & 1 : \mathrm{Int} \Rightarrow^p \star \Rightarrow^q \mathrm{Int} & \longrightarrow \quad 1 \\ f_1\ \mathrm{Bool}\ 1 & \longrightarrow^\star & 1 : \mathrm{Int} \Rightarrow^p \star \Rightarrow^q \mathrm{Bool} & \longrightarrow \quad \mathrm{blame}\ q \end{array}$$

Inhabitants of type variable X should be only expressions typed at X in the source code, so $x : \mathrm{Int} \Rightarrow^p \star \Rightarrow^q X$ should raise blame. However, type substitution erases the information that the target type in the cast was X.

The prior polymorphic blame calculus [Ahmed et al. 2011] introduces *type bindings* to preserve the information about type variables. In their work, type applications generate type bindings $X := A$, which record that X is instantiated with A, instead of substituting A for X. They retain type bindings locally in the form of a term $\nu X := A.t$. Ahmed et al. [2017] refine Ahmed et al.'s "local" type bindings and propose "global" ones, which we follow in this work. In the "global" approach, generated type bindings are immediately moved to the global store. For that, we extend reduction to quaternary relation $\Sigma \triangleright f \longrightarrow \Sigma' \triangleright f'$, where Σ is a global store having type bindings $X := A$, and type applications reduce as follows:

$$\Sigma \triangleright (\Lambda X.\ w)\ A \longrightarrow \Sigma, X := A \triangleright w.$$

Thanks to the avoidance of type substitution, the examples above raise blame as we expected:

$$\begin{array}{lllll} \Sigma \triangleright f_1\ \mathrm{Int}\ 1 & \longrightarrow^\star & \Sigma, X := \mathrm{Int} \triangleright 1 : \mathrm{Int} \Rightarrow^p \star \Rightarrow^q X & \longrightarrow & \Sigma, X := \mathrm{Int} \triangleright \mathrm{blame}\ q \\ \Sigma \triangleright f_1\ \mathrm{Bool}\ 1 & \longrightarrow^\star & \Sigma, X := \mathrm{Bool} \triangleright 1 : \mathrm{Int} \Rightarrow^p \star \Rightarrow^q X & \longrightarrow & \Sigma, X := \mathrm{Bool} \triangleright \mathrm{blame}\ q \end{array}$$

[3] Although one could argue that conservativity is not that important, we stick to it in favor of the refined criteria.

Type bindings also enforce parametric behavior for untyped terms used as parametric values. For example, let us consider using $f_2 = \lambda x{:}\star.\ ((x : \star \Rightarrow^{p_1} \mathrm{Int}) + 1) : \mathrm{Int} \Rightarrow^{p_2} \star$ as a polymorphic identity function. Since f_2 is not an identity function actually, it should be blamed, and it *is* thanks to type bindings.

$$\begin{array}{ll} & \Sigma \triangleright (f : \star \to \star \Rightarrow^{p_3} \forall X.X \to X)\ \mathrm{Int}\ 1 \\ \longrightarrow & \Sigma, X := \mathrm{Int} \triangleright (f : \star \to \star \Rightarrow^{p_3} X \to X)\ 1 \\ \longrightarrow^{\star} & \Sigma, X := \mathrm{Int} \triangleright ((1 : X \Rightarrow^{\overline{p_3}} \star \Rightarrow^{p_1} \mathrm{Int}) + 1) : \mathrm{Int} \Rightarrow^{p_2} \star \Rightarrow^{p_3} X \\ \longrightarrow & \Sigma, X := \mathrm{Int} \triangleright \mathsf{blame}\ p_1 \end{array}$$

On the other hand, an untyped identity function can behaves as a parametric one:

$$\begin{array}{ll} & \Sigma \triangleright ((\lambda x{:}\star.\ x) : \star \to \star \Rightarrow^{p} \forall X.X \to X)\ \mathrm{Int}\ 1 \\ \longrightarrow & \Sigma, X := \mathrm{Int} \triangleright ((\lambda x{:}\star.\ x) : \star \to \star \Rightarrow^{p} X \to X)\ 1 \\ \longrightarrow^{\star} & \Sigma, X := \mathrm{Int} \triangleright (1 : X \Rightarrow^{\overline{p}} \star \Rightarrow^{p} X) \\ \longrightarrow & \Sigma, X := \mathrm{Int} \triangleright 1 \end{array}$$

3.3 Static and gradual type variables

Although type bindings enable run-time enforcement of parametricity, they are generated even when unnecessary—for example, in reduction of pure System F terms—which leads to incurring additional run-time overhead, while type substitution in System F can be erased by type-erasing semantics [Pierce 2002, Section 23.7]. Our key observation to avoid the overhead of type binding generation is that a type binding for X is needed only when X occurs on casts together with $\star$. In other words, the type binding is unnecessary if X is not required to be consistent with $\star$. For example, we need to generate type bindings in the following examples for enforcing parametricity:

$$\Lambda X.\ \lambda x{:}\mathrm{Bool}.\ x : \mathrm{Bool} \Rightarrow^{p} \star \Rightarrow^{q} X \qquad \Lambda X.\ \lambda x{:}X.\ \lambda y{:}\star \to \star.\ y\ (x : X \Rightarrow^{p} \star)$$

whereas we do not in the examples below:

$$\Lambda X.\ \lambda x{:}X.\ x \qquad \Lambda X.\ \lambda x{:}X \to \star.\ (x : X \to \star \Rightarrow^{p} X \to \mathrm{Int})$$

The second expression does not need a type binding despite that it involves a cast accompanying X, because the cast is decomposed into a reflexive cast for X, which always succeeds even in type-substitution semantics.

To distinguish usage of type variables, we introduce the concepts of *static type variables* and *gradual type variables*, first discussed in [Garcia and Cimini 2015].[4] Static type variables are ones that are consistent only with themselves, whereas gradual type variables are ones that are consistent also with $\star$, just like other base types. In System F$_\mathrm{G}$ and System F$_\mathrm{C}$, bound type variables are annotated by the two labels, $\mathcal{S}$ (for static) and $\mathcal{G}$ (for gradual). We write $X{::}\mathcal{S}$ for a static type variable and $X{::}\mathcal{G}$ for a gradual type variable. Corresponding to the annotation of bound type variables, $(\Lambda X{::}\mathcal{S}.\ w)$ and $(\Lambda X{::}\mathcal{G}.\ w)$ are called a static type abstraction and a gradual type abstraction, respectively. Below are examples of type abstractions augmented with labels.

$$\begin{array}{ll} \Lambda X{::}\mathcal{G}.\ \lambda x{:}\mathrm{Bool}.\ x : \mathrm{Bool} \Rightarrow^{p} \star \Rightarrow^{q} X & \Lambda X{::}\mathcal{G}.\ \lambda x{:}X.\ \lambda y{:}\star \to \star.\ y\ (x : X \Rightarrow^{p} \star) \\ \Lambda X{::}\mathcal{S}.\ \lambda x{:}X.\ x & \Lambda X{::}\mathcal{S}.\ \lambda x{:}X \to \star.\ (x : X \to \star \Rightarrow^{p} X \to \mathrm{Int}) \end{array}$$

Note that $\mathcal{S}$ can be replaced with $\mathcal{G}$, but the converse may not hold.

The distinction of type abstractions meets the needs. By the nature of static type variables, they always appear in the same positions in the source and target types in a cast (if they ever appear), resulting in reflexive casts in the end. Thus, we do not have to generate a type binding for a type

[4]They are called static and gradual type *parameters*; the distinction of type parameters plays a relatively minor role to describe a finer-grained notion of principal type schemes in their work, but it plays a central role in our work.

application of a static type abstraction and so it reduces by type substitution, while applying a gradual type abstraction generates a type binding still.

$$\Sigma \triangleright (\Lambda X{::}\mathcal{S}.\ w)\ A \longrightarrow \Sigma \triangleright w[X := A] \qquad \Sigma \triangleright (\Lambda X{::}\mathcal{G}.\ w)\ A \longrightarrow \Sigma, X := A \triangleright w$$

In Section 6, we demonstrate that even the cost of type substitution can be erased under type-erasing semantics; so, the overhead is only generation of fresh type names for gradual type abstractions. This leads us to the "pay-as-you-go" implementation of a polymorphic gradually typed language, where there is no run-time type information at all, if a program is fully statically typed, that is, without any $\star$. Moreover, the distinction of type abstractions makes clear the correspondence between our calculus and System F (by regarding System-F type abstractions as static).

It is ensured by type consistency that usage of bound type variables follows their labels. To this end, we extend the type consistency relation with a type environment Γ, where type variables are annotated. Our type consistency relation is actually a ternary relation $\Gamma \vdash A \sim B$. The three rules are extended with Γ as follows:

$$\frac{\forall X \in \mathrm{Ftv}(A).\ X{::}\mathcal{G} \in \Gamma}{\Gamma \vdash \star \sim A.} \qquad \frac{\Gamma, X{::}\mathcal{G} \vdash A \sim B \quad B \neq \forall X.B' \quad \star \in \mathrm{Types}(B) \quad X \notin \mathrm{Ftv}(B)}{\Gamma \vdash \forall X.A \sim B} \qquad \frac{\Gamma, X{::}\mathcal{S} \vdash A \sim B}{\Gamma \vdash \forall X.A \sim \forall X.B.}$$

In the first rule, which is from (C_StarL), the side condition verifies that A does not contain static type variables. This restriction disallows static type variables to be consistent with $\star$ as expected. The second rule, which relates a polymorphic type and non-$\forall$ type, chooses $\mathcal{G}$ as a label for X, because X does not occur in B and so X should be consistent with only $\star$ in $A \sim B$. The choice in the third rule is more subtle. In fact, from the perspective of type safety, both $\mathcal{S}$ and $\mathcal{G}$ would be reasonable, but choosing $\mathcal{S}$ leads us to rejecting some failing casts and avoiding counterexamples of the gradual guarantee; we discuss in Section 5.2 in detail.

Finally, we end this section with a few remarks on labels. First, the distinction of type variables is not kinding in the sense that it does not restrict the range of type variables; we can apply both sorts of type abstractions to *any* type including $\star$. So, we do not call them kinds but simply static and gradual *labels*. For the same reason, we do not annotate $\forall X.A$ with a label. The distinction arises only when we compare two forall types as above two consistency rules. Second, we always require an explicit label for type abstractions in our calculi, but we are sure that it is easily inferred by observing derivation of typing. In fact, our interpreter performs label inference along with cast insertion translation. Programmers will enjoy the pay-as-you-go property with no care about labels at all, though its formalization and correctness are future work. Third, the distinction of type variables is also used in the definition of term precision when we discuss the gradual guarantee in Section 8. It is interesting to see that the distinction has found another use, which is not directly related to our original motivation of a "pay-as-you-go" implementation.

4 SYSTEM $\mathrm{F_G}$: A GRADUALLY TYPED EXTENSION OF SYSTEM F

Having described key ideas in the last section, we formally define System $\mathrm{F_G}$, which is a gradually typed extension of System F and also a polymorphic extension of GTLC, with its syntax and type system and show its conservativity over typing of System F and DTLC.

The definition of System $\mathrm{F_G}$ is found in Fig. 3. Types are augmented (from GTLC) with polymorphic types $\forall X.A$ and type variables X. Terms are extended with type abstractions $\Lambda X{::}L.\ v$ and type applications $e\ A$, where L denotes a *label*, $\mathcal{S}$ or $\mathcal{G}$, meaning a static type variable or a gradual type variable, respectively. We restrict the body of a type abstraction to a value, following the latest polymorphic blame calculus [Ahmed et al. 2017]. A type abstraction is a value. Type environments are extended with a type variable with a label.

Syntax

Types	$A, B ::= \iota \mid \star \mid A \to B \mid \forall X.A \mid X$
Labels	$L ::= \mathcal{S} \mid \mathcal{G}$
Terms	$e ::= c \mid x \mid \lambda x{:}A.\ e \mid e\ e \mid \Lambda X{::}L.\ v \mid e\ A$
Values	$v ::= c \mid \lambda x{:}A.\ e \mid \Lambda X{::}L.\ v$
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, X{::}L$

$\boxed{T \rhd T}$ **Matching**

$$A \to A' \rhd A \to A' \qquad \star \rhd \star \to \star$$

$$\forall X.A \rhd \forall X.A \qquad A \rhd \forall X.A \quad \text{if QPoly } (A) \text{ and } X \notin \mathrm{Ftv}(A)$$

$\boxed{\Gamma \vdash A \sim B}$ **Type consistency**

$$\Gamma \vdash A \sim A \ (\text{C\_Refl}) \qquad \frac{\Gamma \vdash A \sim A' \qquad \Gamma \vdash B \sim B'}{\Gamma \vdash A \to B \sim A' \to B'} \ (\text{C\_Arrow}) \qquad \frac{\text{GType } (\Gamma, A) \qquad A \neq \forall Y.A'}{\Gamma \vdash \star \sim A} \ (\text{C\_StarL})$$

$$\frac{\Gamma, X{::}\mathcal{S} \vdash A \sim B}{\Gamma \vdash \forall X.A \sim \forall X.B} \ (\text{C\_AllS}) \qquad \frac{\Gamma, X{::}\mathcal{G} \vdash A \sim B \qquad \text{QPoly } (B) \qquad X \notin \mathrm{Ftv}(B)}{\Gamma \vdash \forall X.A \sim B} \ (\text{C\_AllGL})$$

$\boxed{\Gamma \vdash A}$ **Type well-formedness**

$$\Gamma \vdash \iota \ (\text{Tw\_Base}) \qquad \Gamma \vdash \star \ (\text{Tw\_Star}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B} \ (\text{Tw\_Arrow}) \qquad \frac{X{::}L \in \Gamma}{\Gamma \vdash X} \ (\text{Tw\_Var}) \qquad \frac{\Gamma, X{::}L \vdash A}{\Gamma \vdash \forall X.A} \ (\text{Tw\_All})$$

$\boxed{\Gamma \vdash_G e : A}$ **Typing**

$$\frac{ty(c) = A}{\Gamma \vdash_G c : A} \ (\text{T\_Const\_G}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_G x : A} \ (\text{T\_Var\_G}) \qquad \frac{\Gamma, x : A \vdash_G e : B \qquad \Gamma \vdash A}{\Gamma \vdash_G \lambda x{:}A.\ e : A \to B} \ (\text{T\_Abs\_G})$$

$$\frac{\Gamma \vdash_G e_1 : A_1 \qquad \Gamma \vdash_G e_2 : A_2 \qquad A_1 \rhd A_{11} \to A_{12} \qquad \Gamma \vdash A_{11} \sim A_2}{\Gamma \vdash_G e_1\ e_2 : A_{12}} \ (\text{T\_App\_G})$$

$$\frac{\Gamma, X{::}L \vdash_G v : B}{\Gamma \vdash_G \Lambda X{::}L.\ v : \forall X.B} \ (\text{T\_Tyabs\_G}) \qquad \frac{\Gamma \vdash_G e : B \qquad B \rhd \forall X.B' \qquad \Gamma \vdash A}{\Gamma \vdash_G e\ A : B'[X := A]} \ (\text{T\_Tyapp\_G})$$

Fig. 3. System $\mathsf{F_G}$

As we have discussed, type consistency is a ternary relation $\Gamma \vdash A \sim B$. Γ is used to know whether each of type variables is static or gradual. (C_Refl) and (C_Arrow) are the same as before except for the addition of Γ. In (C_StarL), we verify that A does not contain static type variables by predicate GType, defined by:

$$\text{GType } (\Gamma, A) \overset{\text{def}}{\iff} \forall X \in \mathrm{Ftv}(A).\ X{::}\mathcal{G} \in \Gamma\ .$$

We call a type without static type variables a *gradual type*. Readers should note that we may also call even a type without $\star$ as a gradual type. When we compare two polymorphic types, we record X as a static type variable in the type environment (C_AllS). In comparison of a polymorphic type and a non-$\forall$ type, a type variable X is recorded as a gradual type variable ((C_AllGL)). QPoly (B) is the set of side conditions to check that type B is quasi-polymorphic:

$$\text{QPoly } (A) \overset{\text{def}}{\iff} A \neq \forall Y.A' \text{ and } \star \in \text{Types}(A).$$

The condition $X \notin \mathrm{Ftv}(B)$ is to avoid the capture of free occurrences of X in the premise. We omit symmetric variants of (C_StarL) and (C_AllGL), in which the types on both sides of $\sim$ are swapped, for brevity.

Type well-formedness verifies that all the free type variables in A are declared in Γ. In typing rules, we always check type well-formedness when new types appear in a term.

Typing rules are obtained by a mostly straightforward combination of GTLC and System F. Γ is passed to type consistency in (T_App_G). Type application (T_Tyapp_G) is gradualized similarly to function application by (extended) matching $B \triangleright \forall X.B'$. If B is a polymorphic type, matching returns B as it is. When B is a quasi-polymorphic type, matching returns $\forall X.B$, where X is a fresh type variable. The conservativity over typing of System F is retained thanks to the side condition QPoly (B). This rule may be interesting also because the Gradualizer [Cimini and Siek 2016] does not generate it.

The new (T_Tyapp_G) rule allows flexible prototyping. For example, type application like $(\lambda x{:}\star . x)\ A$ is allowed, where $\star \to \star$, the type of $(\lambda x{:}\star . x)$, is quasi-polymorphic. Type application makes no essential effects in this case yet. In this calculus, programmers can evolve code from $(\lambda x{:}\star . x)$ to $(\Lambda X.\ \lambda x{:}X.\ x)$ and $(\Lambda X.\ \lambda x{:}X.\ x)\ A$ is naturally well-typed. The context $[\ ]\ A$ deals with both cases. This context would enable programmers to write flexible code which accepts both polymorphic and untyped prototype code, the latter of which will evolve to polymorphic code.

We describe correspondence of our calculus with System F and DTLC. By System F, we formally mean System $\mathrm{F_G}$ without $\star$ and $(\Lambda X{::}\mathcal{G}.\ v)$ from the syntax. An ordinary type abstraction $(\Lambda X.\ v)$ is regarded as an abbreviation of $(\Lambda X{::}\mathcal{S}.\ v)$. Typing $\Gamma \vdash_S e : A$ in System F is as usual.

Then, our type system is a conservative extension of that of System F.

Theorem 4.1 (Conservativity over typing of System F). *Suppose e is a closed term of System F. $\emptyset \vdash_G e : A$ iff $\emptyset \vdash_S e : A$.*

Proof. Because types do not include $\star$, consistency and matching are mere equality in System $\mathrm{F_G}$. Especially, (C_AllGL) and (C_AllGR) cannot be used by the restriction $\star \in \mathrm{Types}(B)$. □

System $\mathrm{F_G}$ also includes fully dynamic typing—that is, there exists an embedding $\lceil e \rceil$ for an untyped term e. We assume a pure, call-by-value untyped lambda calculus with constants. The embedding is defined as follows:

$$\lceil c \rceil = (\lambda x{:}\star . x)\ c \qquad \lceil x \rceil = x \qquad \lceil \lambda x.\ e \rceil = (\lambda x{:}\star . x)\ (\lambda x{:}\star . \lceil e \rceil) \qquad \lceil e_1\ e_2 \rceil = \lceil e_1 \rceil\ \lceil e_2 \rceil$$

Theorem 4.2 (Embedding DTLC [Siek et al. 2015]). *If e is a closed DTLC term, then $\vdash_G \lceil e \rceil : \star$.*

5 SYSTEM $\mathrm{F_C}$: YET ANOTHER POLYMORPHIC BLAME CALCULUS

We formally define System $\mathrm{F_C}$, an internal cast calculus of System $\mathrm{F_G}$, based on the previous polymorphic blame calculi by Ahmed et al. [2011] and Ahmed et al. [2017]. The dynamic semantics of System $\mathrm{F_G}$ is defined by a cast insertion translation to System $\mathrm{F_C}$. We show that System $\mathrm{F_G}$ satisfies conservativity over the semantics of System F and DTLC; we also show type safety.

5.1 Definition

The definition of System $\mathrm{F_C}$ is found in Fig. 4. The syntax of System $\mathrm{F_C}$ is an extension of the blame calculus in Fig. 2 with the features for polymorphism in System $\mathrm{F_G}$. A type variable is a ground type. A value with a cast to a polymorphic type $(w : A \Rightarrow^p \forall X.B)$ is a value. This new value form comes from the latest polymorphic blame calculus by Ahmed et al. [2017]. This makes it possible to restrict the body of a type abstraction to a value throughout reduction. A type environment has a type binding $X := A$, which means that X is definitionally equal to A. Σ is a *store*, a type environment which consists of only type bindings. The predicate GType is extended along with this extension.

$$\mathrm{GType}\ (\Gamma, A) \overset{\text{def}}{\iff} \forall X \in \mathrm{Ftv}(A).\ (X{::}\mathcal{G} \in \Gamma \text{ or } X := B \in \Gamma)$$

The bound type variable in a type binding is always considered gradual because a type binding is generated only when a gradual type abstraction is reduced. The premise of (Tw_Var) rule is also extended to "$X{::}L \in \Gamma$ or $X := A \in \Gamma$" (this rule is not displayed in Fig. 4, though).

Syntax

Types	$A, B ::= \iota \mid \star \mid A \to B \mid \forall X.A \mid X$
Ground types	$G, H ::= \iota \mid \star \to \star \mid X$
Blame labels	$p, q ::= l$
Labels	$L ::= \mathcal{S} \mid \mathcal{G}$
Terms	$f ::= c \mid x \mid \lambda x{:}A.\ f \mid f\ f \mid \Lambda X{::}L.\ w \mid f\ A \mid f : A \Rightarrow^p B \mid \mathsf{blame}\ p$
Values	$w ::= c \mid \lambda x{:}A.\ f \mid \Lambda X{::}L.\ w \mid w : G \Rightarrow^p \star \mid w : A \to B \Rightarrow^p A' \to B' \mid w : A \Rightarrow^p \forall X.B$
Context	$\mathrm{E} ::= [\,] \mid \mathrm{E}\,f \mid w\,\mathrm{E} \mid \mathrm{E}\,A \mid \mathrm{E} : A \Rightarrow^p B$
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, X{::}L \mid$
Store	$\Sigma ::= \emptyset \mid \Sigma, X := A$

$\boxed{\Gamma \vdash e \rightsquigarrow f : A}$ **Cast insertion**

$$\frac{\Gamma \vdash e_1 \rightsquigarrow f_1 : A_1 \qquad \Gamma \vdash e_2 \rightsquigarrow f_2 : A_2 \qquad A_1 \rhd A_{11} \to A_{12} \qquad \Gamma \vdash A_{11} \sim A_2}{\Gamma \vdash e_1\ e_2 \rightsquigarrow (f_1 : A_1 \Rightarrow^{\ell}_{?} A_{11} \to A_{12})\ (f_2 : A_2 \Rightarrow^{\ell}_{?} A_{11}) : A_{12}}\ (\text{CI\_App})$$

$$\frac{\Gamma \vdash e \rightsquigarrow f : B \qquad B \rhd \forall X.B' \qquad \Gamma \vdash A}{\Gamma \vdash e\ A \rightsquigarrow (f : B \Rightarrow^{\ell}_{?} \forall X.B')\ A : B'[X := A]}\ (\text{CI\_Tyapp})$$

$\boxed{\Sigma \rhd f \longrightarrow \Sigma' \rhd f'}$ **Reduction**

$$(\Lambda X{::}\mathcal{S}.\ w)\ A \longrightarrow w[X := A] \qquad (\text{R\_TybetaS})$$

$$\Sigma \rhd (\Lambda X{::}\mathcal{G}.\ w)\ A \longrightarrow \Sigma, X := A \rhd w \qquad (\text{R\_TybetaG})$$

$$(w : \forall X.A \Rightarrow^p \forall X.B)\ C \longrightarrow (w\ C) : A[X := C] \Rightarrow^p B[X := C] \qquad (\text{R\_Content})$$

$$\Sigma \rhd (w : A \Rightarrow^p \forall X.B)\ C \longrightarrow \Sigma, X := C \rhd w : A \Rightarrow^p B \quad \text{if } A \neq \forall X'.A' \text{ and } X \text{ is fresh.} \qquad (\text{R\_Generalize})$$

$$w : \forall X.A \Rightarrow^p B \longrightarrow (w\ \star) : A[X := \star] \Rightarrow^p B \quad \text{if } B \neq \forall X'.B' \qquad (\text{R\_Instantiate})$$

$\boxed{\Gamma \vdash_C f : A}$ **Typing**

$$\frac{ty(c) = A}{\Gamma \vdash_C c : A}\ (\text{T\_Const\_C}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_C x : A}\ (\text{T\_Var\_C}) \qquad \frac{\Gamma, x : \Gamma(A) \vdash_C f : B \qquad \Gamma \vdash A}{\Gamma \vdash_C \lambda x{:}A.\ f : \Gamma(A) \to B}\ (\text{T\_Abs\_C})$$

$$\frac{\Gamma \vdash_C f_1 : A \to B \qquad \Gamma \vdash_C f_2 : A}{\Gamma \vdash_C f_1\ f_2 : B}\ (\text{T\_App\_C}) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash_C \mathsf{blame}\ p : A}\ (\text{T\_Abort\_C}) \qquad \frac{\Gamma, X{::}L \vdash_C w : B}{\Gamma \vdash_C \Lambda X{::}L.\ w : \forall X.B}\ (\text{T\_TyAbs\_C})$$

$$\frac{\Gamma \vdash_C f : \forall X.B \qquad \Gamma \vdash A}{\Gamma \vdash_C f\ A : B[X := \Gamma(A)]}\ (\text{T\_Tyapp\_C}) \qquad \frac{\Gamma \vdash_C f : \Gamma(A) \qquad \Gamma \vdash B \qquad \Gamma \vdash A \sim B}{\Gamma \vdash_C f : A \Rightarrow^p B : \Gamma(B)}\ (\text{T\_Cast\_C})$$

Fig. 4. System $\mathrm{F_C}$

We show only main rules of cast insertion; the full definition is found in the supplementary material. Type consistency is extended to a ternary relation in (CI_App). A cast may be inserted by following the matching for a polymorphic type (CI_Tyapp).

The reduction relation takes the form $\Sigma \rhd f \longrightarrow \Sigma' \rhd f'$. As we mentioned already, type bindings generated by reduction are immediately stored in Σ, inspired by the *configuration reduction* by Ahmed et al. [2017]. In the definition, $f \longrightarrow f'$ is an abbreviation of $\Sigma \rhd f \longrightarrow \Sigma \rhd f'$. All the reduction rules of the blame calculus in Section 2.2 are implicitly adopted to System $\mathrm{F_C}$ similarly. It is worth noting that casts do not look through the store: that is, $X := \mathrm{Int} \rhd (1 : \mathrm{Int} \Rightarrow^{p_1} \star \Rightarrow^{p_2} X)$ raises blame. We explain the rules specific to System $\mathrm{F_C}$ here. Type application to a static type abstraction results in type substitution (R_TybetaS). Type application to a gradual type abstraction immediately stores a type binding in the store (R_TybetaG). Rule (R_Content) comes from Belo et al. [2011] and Sekiyama et al. [2017], where a cast between polymorphic types reduces without type arguments supplied. Our rule can be understood in terms of their rules as follows:

$$(w : \forall X.A \Rightarrow^p \forall X.B)\ C \longrightarrow (\Lambda X{::}\mathcal{S}.\ (w\ X) : A \Rightarrow^p B)\ C \longrightarrow (w\ C) : A[X := C] \Rightarrow^p B[X := C]$$

A static type abstraction would be created in the intermediate state because the last applied rule in the derivation of $\Gamma \vdash \forall X.A \sim \forall X.B$ is (C_AllS). Then, we would proceed by (R_TybetaS). Rule (R_Generalize) can be understood similarly. Programmers should take care that gradual type abstractions, which cause overhead of name generation, may be implicitly introduced by (R_Generalize), even if gradual type abstractions do not appear explicitly in a program. If a polymorphic type appears only on the left-hand side of a cast, we immediately instantiate the source term by $\star$ (R_Instantiate). Rules (R_Generalize) and (R_Instantiate) come from the latest polymorphic blame calculus [Ahmed et al. 2017]. We discuss further about the latter rule in Section 8 with the *Jack-of-All-Trades* conjecture [Ahmed et al. 2013]. As in GTLC, the dynamic semantics of System F_G is defined by cast insertion, followed by reduction in System F_C. $\Sigma \triangleright f \longrightarrow^\star \Sigma' \triangleright f'$ is the reflexive transitive closure of $\Sigma \triangleright f \longrightarrow \Sigma' \triangleright f'$.

Definition 5.1 (Dynamic semantics of System F_G). For e with $\emptyset \vdash_G e : A$, $e \Downarrow_G \Sigma \triangleright w \overset{\text{def}}{\iff} \emptyset \vdash e \rightsquigarrow f : A$ and $\emptyset \triangleright f \longrightarrow^\star \Sigma \triangleright w$.

For typing, we take an approach slightly different from the previous polymorphic blame calculi. One can find the difficulties of typing with type bindings in reduction by (R_TybetaG) such as

$$\emptyset \triangleright (\Lambda X{::}\mathcal{G}.\ (\lambda x{:}X.\ x))\ \mathsf{Int} \longrightarrow X := \mathsf{Int} \triangleright (\lambda x{:}X.\ x).$$

The type of the left term is clearly $\mathsf{Int} \to \mathsf{Int}$, but how about the right term? In terms of type preservation, the type of the right term should also be $\mathsf{Int} \to \mathsf{Int}$, but we could not assign the type only by the term $(\lambda x{:}X.\ x)$. We have to take $X := \mathsf{Int}$ into account and cause type application in the derivation of typing somehow. We develop the following typing rule for a lambda abstraction.

$$\frac{\Gamma, x : \Gamma(A) \vdash_C f : B \qquad \Gamma \vdash A}{\Gamma \vdash_C \lambda x{:}A.\ f : \Gamma(A) \to B}\ (\text{T\_Abs\_C})$$

The type environment application $\Gamma(A)$ applies all type bindings in Γ to a type A as follows.

$$(\emptyset)(A) = A \qquad (\Gamma, x : B)(A) = \Gamma(A) \qquad (\Gamma, X{::}L)(A) = \Gamma(A) \qquad (\Gamma, X := B)(A) = \Gamma(A[X := B])$$

Type bindings are applied sequentially from the innermost one so that accidental capture or escape of type variables cannot happen. Now, we can derive $X := \mathsf{Int} \vdash_C \lambda x{:}X.\ x : \mathsf{Int} \to \mathsf{Int}$ as expected. Type environment application appears in rules (T_Tyapp_C) and (T_Cast_C) for the same reason.

In the previous polymorphic blame calculus, *static casts* [Ahmed et al. 2011], which syntactically record where type substitution and abstraction occur, is used for syntax-directed typing, instead of type environment application, to make metatheory more tractable. We believe our approach makes the type system and reduction even simpler and easier to work with: we have achieved syntax-directed typing by type environment application without introducing additional devices.

5.2 Why do we choose $\mathcal{S}$ in (C_AllS)?

The consistency rule (C_AllS), which relates two polymorphic types, assigns $\mathcal{S}$ to the bound type variable. We expect that using $\mathcal{G}$ instead of $\mathcal{S}$ would work well for type safety, but it would be the source of two subtle issues.

First, choosing $\mathcal{G}$ would result in acceptance of casts which always raise blame. For example, let us consider $f = (\mathsf{id} : \forall X.X \to X \Rightarrow^p \forall X.\star \to X)$ where id is a polymorphic identity function. Giving $\mathcal{G}$ to bound type variables in the consistency rule for polymorphic types would allow for the cast in f and change the reduction rule for polymorphic types as follows (recall that the information of type variables with label $\mathcal{G}$ is stored in Σ):

$$\Sigma \triangleright (w : \forall X.A \Rightarrow^p \forall X.B)\ C \longrightarrow \Sigma, X := C \triangleright (w\ X) : A \Rightarrow^p B.$$

Then, for any A, G, and w, term $f\ A\ (w : G \Rightarrow^q \star)$ triggers blame:

$$\begin{array}{rll} \Sigma \triangleright f\ A\ (w : G \Rightarrow^q \star) & \longrightarrow & \Sigma, X := A \triangleright ((\mathsf{id}\ X) : X \to X \Rightarrow^p \star \to X)\ (w : G \Rightarrow^q \star) \quad (X \neq G) \\ & \longrightarrow^\star & \Sigma, X := A \triangleright ((\lambda x{:}X.\ x)\ (w : G \Rightarrow^q \star \Rightarrow^{\overline{p}} X)) : X \Rightarrow^p X \\ & \longrightarrow & \mathsf{blame}\ \overline{p} \end{array}$$

Choosing $\mathcal{S}$ can reject such failing casts statically.

Second, perhaps even worse, choosing $\mathcal{G}$ would give rise to a counterexample to the gradual guarantee of semantics ((2) of Theorem 2.2)—that is, there would be a case that a less precisely typed term fail. As mentioned above, if we chose $\mathcal{G}$, $\forall X.\star \to X \sim \forall X.X \to X$ would hold, and so $(\lambda x{:}\forall X.\star \to X.\ x)$ id would be a well-typed term. We would also have $\forall X.X \to X \sqsubseteq \forall X.\star \to X$[5] and $(\lambda x{:}\forall X.X \to X.\ x)$ id Int 42 $\sqsubseteq$ $(\lambda x{:}\forall X.\star \to X.\ x)$ id Int 42. As discussed above, however, using the result of a cast from $\forall X.X \to X$ to $\forall X.\star \to X$, which occurs on the right-hand side, would trigger blame, whereas the term on the left-hand side should evaluate to 42—the gradual guarantee of semantics is broken! The rule (C_AllS) avoids such troublesome cases by rejecting $\forall X.X \to X \sqsubseteq \forall X.\star \to X$ and the term precision above.

5.3 Properties

In this section, we show properties of System $\mathrm{F_C}$: conservativity over semantics of System F (Theorem 5.2) and DTLC (Theorem 5.3) and type safety (Theorem 5.8).

We assume standard reduction rules of System F [Pierce 2002, Chapter 23]. Let $\longrightarrow_S$ be the reduction relation of System F and $\longrightarrow_S^\star$ be the reflexive transitive closure of $\longrightarrow_S$. The dynamic semantics of our calculus is a conservative extension of that of System F.

Theorem 5.2 (Conservativity over reduction of System F). *Suppose e is a closed term of System F. $e \Downarrow_G \Sigma \triangleright w$ iff $e \longrightarrow_S^\star w$.*

Proof. First, no casts are inserted to e in System $\mathrm{F_G}$ because e has no dynamic types. Possible reduction rules for e in System $\mathrm{F_C}$ are (R_Delta), (R_Beta), or (R_TybetaS). These rules are the same as those in System F. □

The reduction rules of DTLC are also as usual. However, we assume that a term is reduced to the special term **wrong** if reduction is stuck. The dynamic semantics of our calculus is a conservative extension of that of DTLC. The metavariable r denotes either a value or **wrong** in DTLC.

Theorem 5.3 (Conservativity over reduction of DTLC [Siek et al. 2015]). *There exists an embedding $\lceil\cdot\rceil_C$ from DTLC into System $\mathrm{F_C}$ such that, if e is a closed term of DTLC, then $\lceil e \rceil \Downarrow_G \emptyset \triangleright \lceil r \rceil_C$ iff $e \Downarrow_D r$.*

Proof. The same result holds in GTLC and our calculus is a superset of GTLC. □

If a term is well-typed in System $\mathrm{F_G}$, cast insertion always succeeds and the resulting term has the same type in System $\mathrm{F_C}$.

Lemma 5.4. *If $\Gamma \vdash_G e : A$, then $\Gamma \vdash e \rightsquigarrow f : A$ and $\Gamma \vdash_C f : A$ for some f.*

We state the canonical forms lemma below. In some cases, the type of a value depends on Σ.

Lemma 5.5 (Canonical Forms). *If $\Sigma \vdash_C w : A$, then*

[5] Recall the relationship between type precison and consistency, discussed in Section 2.3.

- $w = c$ *and* $A = ty(c)$,
- $w = \lambda x{:}A'.\, f$ *and* $A = \Sigma(A') \to B$,
- $w = w' : B \to B' \Rightarrow^p C \to C'$ *and* $A = \Sigma(C) \to \Sigma(C')$,
- $w = \Lambda X{::}L.\ w'$ *and* $A = \forall X.A'$,
- $w = (w' : B \Rightarrow^p \forall X.A')$ *and* $A = \forall X.\Sigma(A')$, *or*
- $w = (w' : G \Rightarrow^p \star)$ *and* $A = \star$.

A usual progress property holds: a well-typed term in System $\mathrm{F_C}$ is a value, blame, or reduced.

Lemma 5.6 (Progress). *If* $\Sigma \vdash_C f : A$, *then (1)* $f = w$, *(2)* $f = \mathsf{blame}\ p$ *for some* p, *or (3)* $\Sigma \triangleright f \longrightarrow \Sigma' \triangleright f'$ *for some* Σ' *and* f'.

System $\mathrm{F_C}$ satisfies also type preservation under the store Σ. As a note, type bindings in reduction are essential only for this property. In the next section, we argue that only type names are required to reduce a term.

Lemma 5.7 (Type preservation). *If* $\Sigma \vdash_C f : A$ *and* $\Sigma \triangleright f \longrightarrow \Sigma' \triangleright f'$, *then* $\Sigma' \vdash_C f' : A$.

Then, type safety, one of the desired criteria, holds in our calculus.

Theorem 5.8 (Type Safety). *If* $\emptyset \vdash_G e : A$, *then (1)* $e \Downarrow_G \Sigma \triangleright w$ *and* $\Sigma \vdash_C w : A$ *for some* Σ *and* w, *(2)* $e \Downarrow_G \Sigma \triangleright \mathsf{blame}\ p$ *for some* Σ *and* p, *or (3)* $e \Uparrow_G$.

6 TYPE ERASURE

In this section, we discuss how we can implement System $\mathrm{F_C}$ without most of the overhead of maintaining run-time type information. In fact, well-typed System F terms are subject to the same erasure translation as discussed by Pierce [2002, Section 23.7]: a static type abstraction is encoded into a function abstracted by a dummy variable of type unit and a type application is into an application to the unit value. For gradual type abstractions, we adopt an idea which is very similar to that used for encoding a polymorphic blame calculus into a language with dynamic sealing [Siek and Wadler 2016] with some additional twists. The cost of dynamic types is "pay-as-you-go" in the following sense. The baseline cost, at which a fully statically typed term aims, is the same as type-erasing semantics of System F; a static type abstraction basically meets this baseline because it does not cause additional casts (or blame) and name generation by its nature. Additional overhead is caused only when a gradual type abstraction is used. We have already implemented a prototype of a type-erasing interpreter in OCaml.

As we saw in rule (R_TybetaG) in Section 5, type application of a gradual type abstraction generates a type binding. During reduction, a type binding records the actual type which the generated type name points at. However, in fact, the information of the actual type is required to prove type safety but not very essential for results of reduction. For example, we can blame following casts without knowledge about the actual types to which X and Y are bound.

$$1 : \mathsf{Int} \Rightarrow^{p_1} \star \Rightarrow^{p_2} X \qquad \mathsf{true} : X \Rightarrow^{p_1} \star \Rightarrow^{p_2} Y$$

Based on this observation, we no longer keep an actual type argument at type application. It suffices to generate a fresh ID to identify type variables introduced at each type application. The fresh ID is reminiscent of one of the cryptographic keys, used to express type abstraction of polymorphic blame calculus λB in λK, cryptographic lambda calculus [Siek and Wadler 2016]. We drop the other key to represent type binding because we erase run-time types.

Our type system does not decide whether a term which is applied to a type results in a static type abstraction or a gradual. A possible implementation is to check whether a type abstraction is static or gradual at every type application, but we do not so. We create a thunk with a dummy variable of type unit for every type abstraction and we put fresh ID generation into the thunk of a gradual type abstraction. As a result, we implement type application uniformly as application to a unit value without any checks.[6] This strategy, i.e., fresh ID generation in a *callee* side, resembles the implementation of relationally-parametric polymorphic contracts by Guha et al. [2007], where a polymorphic contract is implemented by a function that generates an opaque container corresponding to a fresh ID every time an abstracted variable is instantiated by a concrete contract. This is in contrast to Siek and Wadler [2016], where fresh keys are generated at a *caller* (that is, type application) and passed to type abstractions. (The caller-side generation would make it easy to prove properties of encoding, though.) Cast rules such as (R_Content) and (R_Generalize) can be implemented as in static and gradual type abstractions, respectively.

Now we show the core of the evaluator implemented in OCaml in Listing 1. The function `eval` is the main function which uses the infix operator ($\Longrightarrow$), which implements casts. It is not a tagless interpreter where values of statically typed expression come without run-time tags. Our main purpose is to demonstrate that static type variables can be completely erased and gradual type variables can be implemented in a lightweight manner. As the term constructor `Var` takes an integer, we use de Bruijn indices to represent variable bindings, although the names of identifiers are still recorded in an AST for a debugging purpose (the type `id` in `FunExp` and so on). For simplicity, error handling code is omitted. We do not detail how blame is implemented—in Listing 1, blame is replaced with raising a simple exception. Implementing blame requires more costs due to manipulation of blame labels. Interested readers are referred to our prototype interpreter.

Type abstractions of both kinds (`TSFunExp` and `TGFunExp`) evaluates to functions of type `unit -> value` (line 16 and 17 in `eval`). In the case of type abstraction by static type variables, the type argument is just discarded and the body is evaluated. In the case of type abstraction by gradual type variables, when it is applied, a fresh ID is generated by `ref ()` and recorded in the environment, which maps (term) variables to values and gradual type variables to a reference to the unit value.

A cast from a gradual type variable to $\star$ results in a value tagged with the ID of the type variable, retrieved from the environment (line 27). A cast from $\star$ to a gradual type variable will compare two IDs; if they disagree, blame is raised (lines 30–31). Lines 34–35 represent (R_Content); (static) type variables in terms `t1` and `t2` become free but their names will not be examined during evaluation (except at the reflexive case, which is omitted). Lines 36–39 implement (R_Generalize) and (R_Instantiate). In the former case, a fresh ID is generated just like type application of $\Lambda X{::}\mathcal{G}.\ v$.

7 BLAME THEOREM

The Blame Theorem, advocated by Tobin-Hochstadt and Felleisen [2006] and dubbed by Wadler and Findler [2009], is a standard property of a blame calculus and justifies the reasonable intuition that run-time typing errors are never triggered by statically typed code, which indicates that the cast semantics of the calculus is plausible. In fact, the Blame Theorem goes a step further: *more precisely* typed code never invokes run-time errors.

We show the Blame Theorem by following Wadler and Findler [2009]: we introduce a few subtyping relations—positive subtyping, negative subtyping, and precision (also called naive subtyping)—and show the relationship between them; and, finally, we prove that the "more precisely typed" side, which is formalized by precision, never causes run-time errors via the relationship. In addition, we introduce ordinary subtyping and show Blame-Subtyping Theorem, one of the refined criteria by Siek et al. [2015], which says that a cast from a type to its supertype never fails. We design all

[6] Ideally, it is best to avoid even creation of a thunk, considering we adopted value restriction; we leave it for future work.

Listing 1. Interpreter

```
type ty    = Int | Bool | Arr of ty * ty | TyVar of int | Forall of id * ty | Dyn
type term  = Var of int | ··· | FunExp of id * ty * term | AppExp of term * term
           | TSFunExp of id * term | TGFunExp of id * term | TAppExp of term * ty
           | CastExp of term * ty * ty
type tag   = I | B | Ar | TV of unit ref           (* ground types *)
type value = IntV of int | BoolV of bool | Fun of (value → value)
           | TFun of (unit → value) | Tagged of tag * value
and  env   = Empty | VB of value * env | TB of unit ref * env

(* eval : env → term → value *)
let rec eval env = function
  | Var idx              → lookup idx env
  ...
  | FunExp (id, _, e)    → Fun (fun v → eval (VB (v, env)) e)
  | AppExp (e1, e2)      → (match eval env e1 with Fun f → f eval env e2)
  | TSFunExp (id, e)     → TFun (fun () → eval env e)
  | TGFunExp (id, e)     → TFun (fun () → eval (TB (ref (), env)) e)
  | TAppExp (e, _)       → (match eval env e with TFun f → f ())
  | CastExp (e, ty1, ty2) → let v = eval env e in (ty1 ⟹ ty2) env v
(* (⟹) : ty → ty → env → value → value *)
and (⟹) t1 t2 env v = match t1, t2 with
    Int, Int → v                                            (* R_ID *)
  | Int, Dyn → Tagged (I, v)
  | Dyn, Int → (match v with
      Tagged(I, v0) → v0                                    (* R_COLLAPSE *)
    | Tagged(_, _) → failwith "Blame!")                     (* R_CONFLICT *)
  | TyVar idx, Dyn → Tagged (TV (lookupty idx env), v)
  | Dyn, TyVar idx → (match v with
    | Tagged(TV r, v0) → if lookupty idx env == r then v0   (* R_COLLAPSE *)
                         else failwith "Blame!"             (* R_CONFLICT *)
    | Tagged(_, _) → failwith "Blame!")                     (* R_CONFLICT *)
  | Arr(s1,t1), Arr(s2,t2) → (match v with Fun f →          (* R_WRAP *)
     Fun (fun w → (t1 ⟹ t2) env (f ((s2 ⟹ s1) env w))))
  | Forall(id1, t1), Forall(id2, t2) → (match v with TFun f →  (* R_CONTENT *)
     TFun (fun () → (t1 ⟹ t2) env (f ())))
  | ty1, Forall(id2, ty2) →                                  (* R_GENERALIZE *)
     TFun (fun () → (typeShift 1 0 ty1 ⟹ ty2) (TB(ref (), env)) v)
  | Forall(id1, ty1), ty2 → (match v with TFun f →           (* R_INSTANTIATE *)
     (typeInst ty1 Dyn ⟹ ty2) env (f ()))
  ...
```

subtyping relations so that they are subsets of type consistency because we are interested in only well-typed programs.

7.1 The Blame Theorem

The Blame Theorem says that the "more precisely typed" side of a cast never triggers run-time errors. Namely, if A is a more precise type than B, then neither cast $f : A \Rightarrow^{p} B$ nor $f : B \Rightarrow^{\overline{p}} A$ invokes blame p. Intuitively, A is more precise than B if A is a result type of replacing occurrences of the dynamic type in B with some types. Precision $\Gamma \vdash A \sqsubseteq B$ formalizes this notion, that is, it means that A is more precise than B under Γ. The inference rules of precision are shown at the top of Fig. 5 and the same as the rules of type consistency except for the lack of a few rules for symmetry: we choose rules from type consistency so that the left-hand side is more precisely typed.

$\boxed{\Gamma \vdash A \sqsubseteq B}$ **Precision**

$$\Gamma \vdash A \sqsubseteq A \ (\text{P\_Refl}) \qquad \frac{\text{GType}\ (\Gamma, A) \qquad A \neq \forall Y.A'}{\Gamma \vdash A \sqsubseteq \star}\ (\text{P\_Star}) \qquad \frac{\Gamma \vdash A \sqsubseteq A' \qquad \Gamma \vdash B \sqsubseteq B'}{\Gamma \vdash A \to B \sqsubseteq A' \to B'}\ (\text{P\_Arrow})$$

$$\frac{\Gamma, X{::}\mathcal{S} \vdash A \sqsubseteq B}{\Gamma \vdash \forall X.A \sqsubseteq \forall X.B}\ (\text{P\_AllS}) \qquad \frac{\Gamma, X{::}\mathcal{G} \vdash A \sqsubseteq B \qquad \text{QPoly}\ (B) \qquad X \notin \text{Ftv}(B)}{\Gamma \vdash \forall X.A \sqsubseteq B}\ (\text{P\_AllG})$$

$\boxed{\Gamma \vdash A <:^+ B}$ **Positive Subtyping**

$$\Gamma \vdash A <:^+ A\ (\text{Ps\_Refl}) \qquad \frac{\text{GType}\ (\Gamma, A) \qquad A \neq \forall Y.A'}{\Gamma \vdash A <:^+ \star}\ (\text{Ps\_Ground}) \qquad \frac{X{::}\mathcal{G}^+ \in \Gamma}{\Gamma \vdash \star <:^+ X}\ (\text{Ps\_GVar})$$

$$\frac{\Gamma \vdash A' <:^- A \qquad \Gamma \vdash B <:^+ B'}{\Gamma \vdash A \to B <:^+ A' \to B'}\ (\text{Ps\_Arrow}) \qquad \frac{\Gamma, X{::}\mathcal{S} \vdash A <:^+ B}{\Gamma \vdash \forall X.A <:^+ \forall X.B}\ (\text{Ps\_AllS})$$

$$\frac{\Gamma, X{::}\mathcal{G} \vdash A <:^+ B \qquad \text{QPoly}\ (B) \qquad X \notin \text{Ftv}(B)}{\Gamma \vdash \forall X.A <:^+ B}\ (\text{Ps\_AllGL}) \qquad \frac{\Gamma, X{::}\mathcal{G} \vdash A <:^+ B \qquad \text{QPoly}\ (A) \qquad X \notin \text{Ftv}(A)}{\Gamma \vdash A <:^+ \forall X.B}\ (\text{Ps\_AllGR})$$

$\boxed{\Gamma \vdash A <:^- B}$ **Negative Subtyping**

$$\Gamma \vdash A <:^- A\ (\text{Ns\_Refl}) \qquad \frac{\Gamma \vdash A <:^- G \qquad \text{GType}\ (\Gamma, A) \qquad A \neq \forall X.A'}{\Gamma \vdash A <:^- \star}\ (\text{Ns\_Ground})$$

$$\frac{A \neq \forall Y.A' \qquad \text{GType}\ (\Gamma, A)}{\Gamma \vdash \star <:^- A}\ (\text{Ns\_Star}) \qquad \frac{\Gamma \vdash A' <:^+ A \qquad \Gamma \vdash B <:^- B'}{\Gamma \vdash A \to B <:^- A' \to B'}\ (\text{Ns\_Arrow})$$

$$\frac{\Gamma, X{::}\mathcal{S} \vdash A <:^- B}{\Gamma \vdash \forall X.A <:^- \forall X.B}\ (\text{Ns\_AllS}) \qquad \frac{\Gamma, X{::}\mathcal{G}^+ \vdash A <:^- B \qquad \text{QPoly}\ (B) \qquad X \notin \text{Ftv}(B)}{\Gamma \vdash \forall X.A <:^- B}\ (\text{Ns\_AllGL})$$

$$\frac{\Gamma, X{::}\mathcal{G} \vdash A <:^- B \qquad \text{QPoly}\ (A) \qquad X \notin \text{Ftv}(A)}{\Gamma \vdash A <:^- \forall X.B}\ (\text{Ns\_AllGR})$$

Fig. 5. Precision, positive subtyping, and negative subtyping.

Following the prior work [Ahmed et al. 2011; Wadler and Findler 2009], our proof of the Blame Theorem is based on two subtyping relations: positive and negative subtyping. The intention is that, if A is a positive or negative subtype of B, cast $f : A \Rightarrow^p B$ does not invoke blame p or blame $\overline{p}$, respectively.

Before showing the rules of positive and negative subtyping, we discuss a subtle observation in polymorphic blame calculi and describe an extension of type environments to address it. In general, we should not be able to derive $\star <:^+ X$ because a cast from $\star$ to X could invoke blame. However, there are cases that X is instantiated with the dynamic type at run time—by (R_Instantiate)—in which case a cast from $\star$ to X should never cause blame. For example, cast $w : \forall X.X \to \star \Rightarrow^p \star \to \star$ never causes blame, in particular blame $\overline{p}$, despite that a cast from $\star$ to X, obtained by decomposing the domain types contravariantly, does not always succeed. To capture such cases, we extend type environments with type variables given new gradual label $\mathcal{G}^+$:

$$\Gamma \quad ::= \quad \ldots \mid X{::}\mathcal{G}^+$$

Giving $\mathcal{G}^+$ to X means that X will be instantiated with the dynamic type at run time. The new label $\mathcal{G}^+$ is gradual and we extend GType (Γ, A) as follows:

$$\text{GType}\ (\Gamma, A) \overset{\text{def}}{\iff} \forall X \in \text{Ftv}(A).\ (X{::}\mathcal{G} \in \Gamma \text{ or } X := B \in \Gamma \text{ or } X{::}\mathcal{G}^+ \in \Gamma)$$

The positive subtyping relation takes the form $\Gamma \vdash A <:^{+} B$ and its rules are shown in the middle of Fig. 5. (Ps_Ground) means that values of any type can be injected into the dynamic type safely, with (Ps_AllGL) for cases where the left-hand side is a polymorphic type. (Ps_GVar) says that $f : \star \Rightarrow^{p} X$ always succeeds if X is given $\mathcal{G}^{+}$ because X of $\mathcal{G}^{+}$ should be replaced with the dynamic type. As we will see soon, $\mathcal{G}^{+}$ in the type environment is introduced by one of the negative subtyping rules. Note that, in general, we cannot expect that $f : \star \Rightarrow^{p} A$ succeeds if A is neither $\star$ nor X with $\mathcal{G}^{+}$. As ordinary subtyping, rule (Ps_Arrow), applied to function types, is contravariant on their domain types and covariant on their codomain types. It also reverses the polarity of subtyping in the domain types, corresponding to the negation of blame labels on casts between domain types in (R_Wrap). Other three rules (Ps_AllS), (Ps_AllGL), and (Ps_AllGR) follow from type consistency.

Negative subtyping $\Gamma \vdash A <:^{-} B$ is used to identify casts $f : A \Rightarrow^{p} B$ which do not invoke blame $\overline{p}$. The rules of negative subtyping are shown at the bottom in Fig. 5. The first four rules are standard [Wadler and Findler 2009], except conditions to make applicable rules syntax-directed and restrict negative subtyping to be a subset of type consistency. (Ns_Ground) requires a type on the left-hand side to be a negative subtype of some ground type G. This condition is needed for the case that A is a function type. In that case, $f : A \Rightarrow^{p} \star$ will reduce to $f : A \Rightarrow^{p} \star \to \star \Rightarrow^{p} \star$ by (R_Ground), so we have to show that $f : A \Rightarrow^{p} \star \to \star$ does not invoke blame $\overline{p}$ for proving that $f : A \Rightarrow^{p} \star$ does not. Negative supertype G of function types is always $\star \to \star$, so the condition $\Gamma \vdash A <:^{-} G$ guarantees that $f : A \Rightarrow^{p} \star \to \star$ never triggers blame $\overline{p}$, as expected. (Ns_Arrow) is dual to (Ps_Arrow). (Ns_AllGL) gives $\mathcal{G}^{+}$ to the bound type variable X, because, if B is not a polymorphic type, cast $f : \forall X.A \Rightarrow^{p} B$ will reduce to $(f\ \star) : A[X := \star] \Rightarrow^{p} B$ by reduction rule (R_Instantiate), and so X is bound to be instantiated with $\star$.

Before showing the Blame Theorem, we discuss properties about positive and negative subtyping. First of all, we show that positive and negative subtyping are a subset of type consistency.

Lemma 7.1. *Let $[\![-]\!]_{\mathcal{G}^{+}\mapsto\mathcal{G}}$ be a function over type environments such that it behaves an identity function except that it maps $X{::}\mathcal{G}^{+}$ to $X{::}\mathcal{G}$.*

(1) *If $\Gamma \vdash A <:^{+} B$, then $[\![\Gamma]\!]_{\mathcal{G}^{+}\mapsto\mathcal{G}} \vdash A \sim B$.*
(2) *If $\Gamma \vdash A <:^{-} B$, then $[\![\Gamma]\!]_{\mathcal{G}^{+}\mapsto\mathcal{G}} \vdash A \sim B$.*

Proof. By induction on the derivations of $\Gamma \vdash A <:^{+} B$ and $\Gamma \vdash A <:^{-} B$. □

Introducing $\mathcal{G}^{+}$ to type environments in (Ns_AllGL) enables positive and negative subtyping to relate more types than using just $\mathcal{G}$. For example, we can derive $\emptyset \vdash \forall X.X \to \star <:^{-} \star \to \star$ using (Ns_AllGL) as well as (Ps_Ground), as follows: we have $X{::}\mathcal{G}^{+} \vdash \star <:^{+} X$ by (Ps_GVar) and $X{::}\mathcal{G}^{+} \vdash \star <:^{-} \star$ by (Ns_Refl); thus, $X{::}\mathcal{G}^{+} \vdash X \to \star <:^{-} \star \to \star$ by (Ns_Fun); finally, we can derive $\emptyset \vdash \forall X.X \to \star <:^{-} \star \to \star$ by (Ns_AllGL). If we did not introduce $\mathcal{G}^{+}$ and used $\mathcal{G}$ instead, we could not derive that judgment because $X{::}\mathcal{G} \vdash \star <:^{+} X$ does not hold. Thus, $\mathcal{G}$ cannot be a replacement for $\mathcal{G}^{+}$, while $\mathcal{G}^{+}$ can be one for $\mathcal{G}$, as the following lemmas show.

Lemma 7.2.

(1) *If $\Gamma, X{::}\mathcal{G}, \Gamma' \vdash A <:^{+} B$, then $\Gamma, X{::}\mathcal{G}^{+}, \Gamma' \vdash A <:^{+} B$.*
(2) *If $\Gamma, X{::}\mathcal{G}, \Gamma' \vdash A <:^{-} B$, then $\Gamma, X{::}\mathcal{G}^{+}, \Gamma' \vdash A <:^{-} B$.*

Proof. By induction on the derivations of $\Gamma, X{::}\mathcal{G}, \Gamma' \vdash A <:^{+} B$ and $\Gamma, X{::}\mathcal{G}, \Gamma' \vdash A <:^{-} B$. □

As usual, we first show the relationship of precision to positive and negative subtyping.

Lemma 7.3. *If $\Gamma \vdash A \sqsubseteq B$, then $\Gamma \vdash A <:^{+} B$ and $\Gamma \vdash B <:^{-} A$.*

Proof. By induction on the derivation of $\Gamma \vdash A \sqsubseteq B$. □

$\boxed{\Gamma \vdash f \text{ sf } p}$ **Safe term**

$$\frac{\Gamma \vdash A <:^{+} B \qquad \Gamma \vdash f \text{ sf } p}{\Gamma \vdash (f : A \Rightarrow^{p} B) \text{ sf } p}\ (\textsc{Sf\_PSub}) \qquad \frac{\Gamma \vdash A <:^{-} B \qquad \Gamma \vdash f \text{ sf } p}{\Gamma \vdash (f : A \Rightarrow^{\overline{p}} B) \text{ sf } p}\ (\textsc{Sf\_NSub}) \qquad \frac{\Gamma \vdash f \text{ sf } p \qquad q \neq p, \overline{p}}{\Gamma \vdash (f : A \Rightarrow^{q} B) \text{ sf } p}\ (\textsc{Sf\_Cast})$$

$$\frac{\Gamma, X{::}L \vdash w \text{ sf } p}{\Gamma \vdash (\Lambda X{::}L.\ w) \text{ sf } p}\ (\textsc{Sf\_TyAbs}) \qquad \frac{q \neq p}{\Gamma \vdash \text{blame } q \text{ sf } p}\ (\textsc{Sf\_Blame}) \qquad \Gamma \vdash x \text{ sf } p\ (\textsc{Sf\_Var})$$

$$\frac{\Gamma \vdash f \text{ sf } p}{\Gamma \vdash \lambda x{:}A.\ f \text{ sf } p}\ (\textsc{Sf\_Abs}) \qquad \frac{\Gamma \vdash f_1 \text{ sf } p \qquad \Gamma \vdash f_2 \text{ sf } p}{\Gamma \vdash f_1\ f_2 \text{ sf } p}\ (\textsc{Sf\_App}) \qquad \frac{\Gamma \vdash f \text{ sf } p}{\Gamma \vdash f\ A \text{ sf } p}\ (\textsc{Sf\_TyApp})$$

Fig. 6. Safe terms.

While Lemma 7.3 is enough to show the Blame Theorem, it is weaker than the Factoring (also known as, Tangram [Wadler 2015]) property, which states also the reverse direction [Wadler and Findler 2009]—i.e., $A \sqsubseteq B$ iff $A <:^{+} B$ and $B <:^{-} A$. In fact, the Factoring property does not hold in our precision and subtyping relations.

Example 7.4. $\emptyset \vdash \star \to \star <:^{+} \forall X.\star \to \star$ and $\emptyset \vdash \forall X.\star \to \star <:^{-} \star \to \star$ are derivable, but $\emptyset \vdash \star \to \star \sqsubseteq \forall X.\star \to \star$ is not.

According to our precision rules, there are no non-$\forall$ types more precise than polymorphic types. Ahmed et al. [2011] claimed that their naive subtyping satisfies the Tangram property, but we have found a counterexample to it: for example, using their positive subtyping $<:^{+}$, negative subtyping $<:^{-}$, and naive subtyping $<:_{n}$, we can derive $\star \to \star <:^{+} \forall X.X \to \star$ and $\forall X.X \to \star <:^{-} \star \to \star$, but we cannot $\star \to \star <:_{n} \forall X.X \to \star$. Thus, defining naive subtyping satisfying the Tangram property in polymorphic blame calculi is left open.

Next, we show that a cast from one type to its positive (resp. negative) supertype never invokes positive (resp. negative) blame. Following Ahmed et al. [2011], we define safe terms $\Gamma \vdash f$ sf p, which indicates that (1) blame p does not appear in f and (2) all casts with p in f are from one type to its positive supertype and all ones with $\overline{p}$ are from one type to its negative supertype. The rules for safe term are shown in Fig. 6—note that type environment Γ does not need to bind term variables. We show the Blame Theorem by proving that reduction preserves term safety. We write $\Sigma \triangleright f \not\longrightarrow^{\star} \Sigma' \triangleright f'$ to denote that $\Sigma \triangleright f \longrightarrow^{\star} \Sigma' \triangleright f'$ cannot be derived.

Lemma 7.5 (Blame Preservation). *If $\Sigma \vdash f$ sf p and $\Sigma \triangleright f \longrightarrow \Sigma' \triangleright f'$, then $\Sigma' \vdash f'$ sf p.*

Theorem 7.6 (Blame Theorem). *Let f be a closed term with a subterm $f' : A \Rightarrow^{p} B$ such that neither p nor $\overline{p}$ occurs at other casts in f. Suppose that Γ is a type environment such that: (1) it captures all free type variables in A and B; and (2) type variables bound by Γ are bound in f with the same labels.*

(1) *If $\Gamma \vdash A <:^{+} B$, then $\emptyset \triangleright f \not\longrightarrow^{\star} \Sigma \triangleright$ blame p for any Σ.*
(2) *If $\Gamma \vdash A <:^{-} B$, then $\emptyset \triangleright f \not\longrightarrow^{\star} \Sigma \triangleright$ blame $\overline{p}$ for any Σ.*
(3) *If $\Gamma \vdash A \sqsubseteq B$, then $\emptyset \triangleright f \not\longrightarrow^{\star} \Sigma \triangleright$ blame p for any Σ.*
(4) *If $\Gamma \vdash B \sqsubseteq A$, then $\emptyset \triangleright f \not\longrightarrow^{\star} \Sigma \triangleright$ blame $\overline{p}$ for any Σ.*

Proof. The first and second items are shown by Blame Preservation (Lemma 7.5) and the fact that f = blame p contradicts the safety of f.[7] The third and fourth follow from Lemma 7.3 with the first and second, respectively. □

[7]Our proof does not need Blame Progress, which is a lemma used to prove the Blame Theorem in Ahmed et al. [2011], though it can be derived from Blame Preservation and that fact about safe terms.

$\boxed{\Gamma \vdash A <: B}$ **Subtyping**

$$\Gamma \vdash A <: A \;(\text{S\_Refl}) \qquad \frac{\Gamma \vdash A <: G \qquad \text{GType}\,(\Gamma, A) \qquad A \neq \forall Y.A'}{\Gamma \vdash A <: \star}\;(\text{S\_Ground}) \qquad \frac{X{::}\mathcal{G}^{+} \in \Gamma}{\Gamma \vdash \star <: X}\;(\text{S\_GVar})$$

$$\frac{\Gamma \vdash A' <: A \qquad \Gamma \vdash B <: B'}{\Gamma \vdash A \to B <: A' \to B'}\;(\text{S\_Arrow}) \qquad \frac{\Gamma, X{::}\mathcal{S} \vdash A <: B}{\Gamma \vdash \forall X.A <: \forall X.B}\;(\text{S\_AllS})$$

$$\frac{\Gamma, X{::}\mathcal{G}^{+} \vdash A <: B \qquad \text{QPoly}\,(B) \qquad X \notin \text{Ftv}(B)}{\Gamma \vdash \forall X.A <: B}\;(\text{S\_AllGL}) \qquad \frac{\Gamma, X{::}\mathcal{G} \vdash A <: B \qquad \text{QPoly}\,(A) \qquad X \notin \text{Ftv}(A)}{\Gamma \vdash A <: \forall X.B}\;(\text{S\_AllGR})$$

Fig. 7. Ordinary subtyping.

We make a remark on a slight difference from positive and negative subtyping in Ahmed et al. [2011]. While we introduce $\mathcal{G}^{+}$ to deal with casts from polymorphic types, Ahmed et al. adopts rules that, instead of using $\mathcal{G}^{+}$, substitute the dynamic type for bound type variables, following their type compatibility. Although we believe that the Blame Theorem holds under their rules (with reasonable side conditions), they would violate our design policy that a subtyping relation should be a subset of type consistency. For example, it is derivable $\forall X.X \to X <:^{-} \text{Int} \to \text{Bool}$ in Ahmed et al. [2011], though $\forall X.X \to X \sim \text{Int} \to \text{Bool}$ is not derivable in our type consistency. For the same reason, our subtyping rule, shown in Section 7.2, that takes polymorphic types on only the left-hand side uses $\mathcal{G}^{+}$, unlike the corresponding rule in Ahmed et al. [2011].

7.2 Blame-Subtyping Theorem

In this section, we investigate how to identify casts which never fail. For that, following the early work, we introduce ordinary subtyping $\Gamma \vdash A <: B$ and show that casts from a type to its ordinary supertype never fail by factoring the ordinary subtyping into positive and negative subtyping.

The rules of ordinary subtyping are shown in Fig. 7. A cast from a subtype of a ground type to the dynamic type does not fail because a cast from the ground type to the dynamic just adds a tag (S_Ground). A cast from the dynamic type to a type variable of $\mathcal{G}^{+}$ succeeds because such a type variable is instantiated with the dynamic type, as discussed in Section 7.1 (S_GVar). Rule (S_AllGL) gives $\mathcal{G}^{+}$ to X because X is instantiated with the dynamic type by (R_Instantiate).

We show that casts from a type to its ordinary supertype never trigger blame.

Theorem 7.7 (Factoring Subtyping). *$\Gamma \vdash A <: B$ iff $\Gamma \vdash A <:^{+} B$ and $\Gamma \vdash A <:^{-} B$.*

Proof. The left-to-right direction is shown by induction on the derivation of $\Gamma \vdash A <: B$. The other direction is by induction on the sum of sizes of A and B. □

Corollary 7.8 (Blame-Subtyping Theorem). *Suppose that f, A, B, p, and Γ are the same as given in the Blame Theorem. If $\Gamma \vdash A <: B$, then $\emptyset \triangleright f \not\longrightarrow^{\star} \Sigma \triangleright \text{blame}\; p$ and $\emptyset \triangleright f \not\longrightarrow^{\star} \Sigma \triangleright \text{blame}\; \overline{p}$ for any Σ.*

Proof. By Factoring Subtyping (Theorem 7.7) and the Blame Theorem (Theorem 7.6). □

8 THE GRADUAL GUARANTEE

In this section, we discuss the gradual guarantee property for System F_G. We first extend term precision of GTLC to System F_G (type precision has been presented at the top of Fig. 5) and then prove that the term precision preserves typing (corresponding to Theorem 2.2 (1)). We state a conjecture that term precision also preserves the semantics (corresponding to Theorem 2.2 (2) and (3)), show auxiliary lemmas, which have been proved, and discuss how the conjecture may be proved. Finally, we discuss that the simulation conjecture, a key lemma for the gradual guarantee, implies (the System F_C version of) the Jack-of-All-Trades property [Ahmed et al. 2011, 2013], which has been proposed as a justification of the choice of $\star$ in the (R_Instantiate) rule of the polymorphic blame calculus but is still left open.[8]

Term precision for System F_G is actually nontrivial for two reasons: (1) naively extending the GTLC term precision by adding congruence rules for type abstraction and type application would break the expected gradual guarantee; and (2) in System F_G, there is reasonable code evolution other than change in type annotations in lambda abstractions as in GTLC.

We first discuss (1). We may expect that the term precision contains the following terms.

$$(\Lambda X{::}\mathcal{G}.\ (\lambda x{:}X.\ (x : X)))\ \mathrm{Int}\ (42 : \mathrm{Int}) \sqsubseteq (\Lambda X{::}\mathcal{G}.\ (\lambda x{:}\star\ .\ (x : X)))\ \mathrm{Int}\ (42 : \star)$$

Here, the term of the form $(e : A)$, which is an abbreviation of $(\lambda z{:}A.\ z)\ e$, ascribes type A to e. Their differences are the type annotations on x and type ascription for 42. Both terms are well-typed, but, while the term on the left is reduced to a value, the term on the right results in blame. This would be a counterexample of the gradual guarantee because making types less precise introduces blame. The problem here is code evolution within a type abstraction, more concretely, the type annotations X and $\star$ on x: X should not be considered more precise than $\star$, if they appear under ΛX (and even if X is declared to be gradual). To address this problem, we index our precision by $\mathcal{X}$, which is another type environment (consisting only of $X{::}\mathcal{S}$ or $X{::}\mathcal{G}$), and give precision rules for lambda abstractions and type abstractions as follows:

$$\frac{e \sqsubseteq_{\mathcal{X}} e' \qquad \mathcal{X} \vdash A \sqsubseteq A'}{\lambda x{:}A.\ e \sqsubseteq_{\mathcal{X}} \lambda x{:}A'.\ e'} \qquad \frac{v \sqsubseteq_{\mathcal{X}, X::\mathcal{S}} v'}{\Lambda X{::}L.\ v \sqsubseteq_{\mathcal{X}} \Lambda X{::}L'.\ v'}$$

Type annotations at λ are compared under $\mathcal{X}$. The rule for type abstractions assigns $\mathcal{S}$ to X, irrespective of the label declared at Λ. The latter rule can be understood by analogy with the type consistency rule (C_All_S), where $\mathcal{S}$ is assigned to X in comparison between two polymorphic types.

We will see cases where X is given $\mathcal{G}$ soon.

Next, we discuss about (2). In GTLC, term precision captures change only in type annotation in lambda abstractions but, in System F_G in which $\forall X.X \to X \sqsubseteq \star \to \star$ holds, it is also reasonable to allow evolution from lambda abstraction, say $\lambda x{:}\star\ .\ x$, to a type abstraction $\Lambda X{::}\mathcal{S}.\ \lambda x{:}X.\ x$. Correspondingly, a more precise expression may have more type abstractions: $(\Lambda X{::}\mathcal{S}.\ \lambda x{:}X.\ x)$ Int 42 should be considered more precise than $(\lambda x{:}\star\ .\ x)$ 42. This observation leads to rules like:

$$\frac{v \sqsubseteq_{\mathcal{X}, X::\mathcal{G}} v' \quad \mathrm{QPoly}\ (\mathrm{TypeOf}\ (v'))}{\Lambda X{::}L.\ v \sqsubseteq_{\mathcal{X}} v'} \qquad \frac{e \sqsubseteq_{\mathcal{X}} e' \quad \mathrm{TypeOf}\ (e) = \forall X.B \quad \mathrm{QPoly}\ (\mathrm{TypeOf}\ (e'))}{e\ A \sqsubseteq_{\mathcal{X}} e'}$$

(where TypeOf (e) is the type of e under a certain implicitly assumed type environment). The condition QPoly(...) means that the type of the right-hand side has to be quasi-polymorphic to align with the typing rule. Note also that X in $\mathcal{X}$ is given $\mathcal{G}$, so that X is considered more precise than $\star$ in this context.

The complete definition of term precision of System F_G is found in Fig. 8. Both sides of term precision are accompanied by typing (that is, a type environment and a type) to avoid TypeOf (e), which has been informally used above and in the literature [Siek et al. 2015]. If Γ and types are ignored, the rules are as we explained.

The property corresponding to Theorem 2.2 (1) holds. As a note, environment precision $\Gamma \sqsubseteq_{\mathcal{X}} \Gamma'$ in Fig. 9 is needed. Label precision $L \sqsubseteq L'$ is a reflexive relation with $\mathcal{S} \sqsubseteq \mathcal{G}$. The second environment precision rule is essential in this proof.

[8]The proof given in Ahmed et al. [2011] has turned out to be flawed [Ahmed et al. 2013].

$$\frac{ty(c) = A}{\Gamma \vdash_G c : A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G c : A}\ (\textsc{Tp\_Const\_G}) \qquad \frac{x : A \in \Gamma \qquad x : A' \in \Gamma'}{\Gamma \vdash_G x : A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G x : A'}\ (\textsc{Tp\_Var\_G})$$

$$\frac{\Gamma, x : A \vdash_G e : B \sqsubseteq_{\mathcal{X}} \Gamma', x : A' \vdash_G e' : B' \qquad \mathcal{X} \vdash A \sqsubseteq A' \qquad \Gamma \vdash A \qquad \Gamma' \vdash A'}{\Gamma \vdash_G \lambda x{:}A.\ e : A \to B \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G \lambda x{:}A'.\ e' : A' \to B'}\ (\textsc{Tp\_Abs\_G})$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e_1 : A_1 \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e_1' : A_1' \qquad \Gamma \vdash_G e_2 : A_2 \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e_2' : A_2' \\ A_1 \rhd A_{11} \to A_{12} \qquad A_1' \rhd A_{11}' \to A_{12}' \qquad \Gamma \vdash A_{11} \sim A_2 \qquad \Gamma' \vdash A_{11}' \sim A_2'\end{array}}{\Gamma \vdash_G e_1\ e_2 : A_{12} \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e_1'\ e_2' : A_{12}'}\ (\textsc{Tp\_App\_G})$$

$$\frac{\Gamma, X{::}L \vdash_G v : A \sqsubseteq_{\mathcal{X}, X::\mathcal{G}} \Gamma' \vdash_G v' : A' \qquad \mathrm{QPoly}\ (A') \qquad X \notin \mathrm{Ftv}(A')}{\Gamma \vdash_G \Lambda X{::}L.\ v : \forall X.A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G v' : A'}\ (\textsc{Tp\_Tyabs1\_G})$$

$$\frac{\Gamma, X{::}L \vdash_G v : A \sqsubseteq_{\mathcal{X}, X::\mathcal{S}} \Gamma', X{::}L' \vdash_G v' : A' \qquad L \sqsubseteq L'}{\Gamma \vdash_G \Lambda X{::}L.\ v : \forall X.A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G \Lambda X{::}L'.\ v' : \forall X.A'}\ (\textsc{Tp\_Tyabs2\_G})$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e : B \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : B' \qquad \Gamma \vdash A \\ B \rhd \forall X.B_1 \qquad \mathrm{GType}\ (\mathcal{X}, A) \qquad \mathrm{QPoly}\ (B') \qquad X \notin \mathrm{Ftv}(B')\end{array}}{\Gamma \vdash_G e\ A : B_1[X := A] \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : B'}\ (\textsc{Tp\_Tyapp1\_G})$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e : B_1 \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : B_1' \qquad B_1 \rhd \forall X.B_1 \qquad B_1' \rhd \forall X.B_1' \\ \Gamma \vdash A \qquad \Gamma' \vdash A' \qquad \mathcal{X} \vdash A \sqsubseteq A'\end{array}}{\Gamma \vdash_G e\ A : B_1[X := A] \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e'\ A' : B_1'[X := A']}\ (\textsc{Tp\_Tyapp2\_G})$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e : \forall X.B \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : \forall X.B' \qquad \forall X.B \rhd \forall X.B \qquad \forall X.B' \rhd \forall X.B' \\ \Gamma \vdash A \qquad \Gamma' \vdash A' \qquad \mathcal{X} \vdash A \sqsubseteq A'\end{array}}{\Gamma \vdash_G e\ A : B[X := A] \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e'\ A' : B'[X := A']}\ (\textsc{Tp\_Tyapp3\_G})$$

Fig. 8. Term precision.

$$\emptyset \sqsubseteq_{\mathcal{X}} \emptyset \qquad \frac{\Gamma \sqsubseteq_{\mathcal{X}} \Gamma' \qquad \mathcal{X} \vdash A \sqsubseteq A'}{\Gamma, x : A \sqsubseteq_{\mathcal{X}} \Gamma', x : A'} \qquad \frac{\Gamma \sqsubseteq_{\mathcal{X}} \Gamma' \qquad L \sqsubseteq L'}{\Gamma, X{::}L \sqsubseteq_{\mathcal{X}} \Gamma', X{::}L'} \qquad \frac{\Gamma \sqsubseteq_{\mathcal{X}} \Gamma'}{\Gamma, X{::}L \sqsubseteq_{\mathcal{X}} \Gamma'}$$

Fig. 9. Environment precision.

Theorem 8.1 (The Gradual Guarantee on Typing). *If* $\Gamma \vdash_G e : A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : A'$ *and* $\Gamma \vdash_G e : A$ *and* $\Gamma \sqsubseteq_{\mathcal{X}} \Gamma'$, *then* $\Gamma' \vdash_G e' : A'$ *and* $\mathcal{X} \vdash A \sqsubseteq A'$.

Proof. By induction on the derivation of $\Gamma \vdash_G e : A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : A'$. □

The proof of the other gradual guarantee properties, which are concerned about the semantics, is not completed and so only conjectured here:

Conjecture 8.2 (The Gradual Guarantee on Semantics). *Suppose* $\emptyset \vdash_G e : A$ *and* $\emptyset \vdash_G e : A \sqsubseteq_{\emptyset} \emptyset \vdash_G e' : A$.

(1) *If* $e \Downarrow_G \Sigma \rhd w$, *then* $e' \Downarrow_G \Sigma' \rhd w'$ *and* $\Sigma \vdash_C w : A \sqsubseteq_{\mathcal{X}} \Sigma' \vdash_C w' : A'$ *for some* $\mathcal{X}$.
(2) *If* $e \Uparrow_G$ *then* $e' \Uparrow_G$.
(3) *If* $e' \Downarrow_G \Sigma' \rhd w'$, *then* $e \Downarrow_G \Sigma \rhd w$ *and* $\Sigma \vdash_C w : A \sqsubseteq_{\mathcal{X}} \Sigma' \vdash_C w' : A'$ *for some* $\mathcal{X}$ *or* $e \Downarrow_G \Sigma \rhd \mathrm{blame}\ p$.
(4) *If* $e' \Uparrow_G$, *then* $e \Uparrow_G$ *or* $e \Downarrow_G \Sigma \rhd \mathrm{blame}\ p$.

Our plan for proving this conjecture is to define term precision $\Sigma \vdash_C f : A \sqsubseteq_{\mathcal{X}} \Sigma' \vdash_C f' : A'$ of System $\mathrm{F_C}$ similarly (the definition is omitted) and show that cast insertion preserves term precision and, finally, that term precision is a weak simulation (modulo blame on the left). Unfortunately, we have not managed to finished proofs as of writing. In order to reduce the complexity of the proof, we might explore more sophisticated term precision so as to reflect the codomain of cast insertion translation.

Conjecture 8.3. *If* $\Gamma \vdash_G e : A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_G e' : A'$ *and* $\Gamma \sqsubseteq_{\mathcal{X}} \Gamma'$ *and* $\Gamma \vdash e \rightsquigarrow f : A$ *and* $\Gamma' \vdash e' \rightsquigarrow f' : A'$, *then* $\Gamma \vdash_C f : A \sqsubseteq_{\mathcal{X}} \Gamma' \vdash_C f' : A'$.

Conjecture 8.4. *Suppose* $\Sigma_1 \vdash_C f_1 : A \sqsubseteq_{\mathcal{X}} \Sigma_1' \vdash_C f_1' : A'$ *and* $\Sigma_1 \sqsubseteq_{\mathcal{X}} \Sigma_1'$. *If* $\Sigma_1 \triangleright f_1 \longrightarrow \Sigma_2 \triangleright f_2$, *then* $\Sigma_1' \triangleright f_1' \longrightarrow^\star \Sigma_2' \triangleright f_2'$ *and* $\Sigma_2 \vdash_C f_2 : A \sqsubseteq_{\mathcal{X}} \Sigma_2' \vdash_C f_2' : A'$ *for some* Σ_2' *and* f_2'.

Jack-of-All-Trades. Ahmed et al. [2011] formulated the *Jack-of-All-Trades* property as a justification of the rule (R_Instantiate):

$$w : \forall X.A \Rightarrow^p B \longrightarrow (w\,\star) : A[X := \star] \Rightarrow^p B$$

Instantiation by $\star$ is a very delicate design choice. Suppose we have a term $w : \forall X.X \to X \Rightarrow^{p_1} \star \to \star \Rightarrow^{p_2}$ Int $\to$ Int, it seems that instantiation by Int is more reasonable than $\star$. We do not know whether this optimistic instantiation by $\star$ preserves the behavior of instantiation by more precise types (by Int in this case). In order to trust in the (R_Instantiate) rule, we have to verify that there are no situations where $\star$ results in a blame while instantiation by a more precise type gets a value. The conjecture *the Jack-of-All-Trades* aims at such a justification and has been known as one of the big challenges of polymorphic blame calculi (See Ahmed et al. [2013] for the history). In fact, we have found that Conjecture 8.4 implies Jack-of-All-Trades in System F$_C$, for which it can be stated as follows:

Theorem 8.5 (Jack-of-All-Trades in System F$_C$). *Suppose Conjecture 8.4 holds and* $\Sigma \vdash_C E[(w\,C) : A[X := C] \Rightarrow^p A'] : B$ *and* $\Sigma \vdash_C E[(w\,\star) : A[X := \star] \Rightarrow^p A'] : B$ *and* GType $(\mathcal{X}, C)$ *where* $\mathcal{X} = \{X{::}\mathcal{S} \mid X := A \in \Sigma$ *or* X *is bound in* $E\}$.

- *If* $\Sigma \triangleright E[(w\,C) : A[X := C] \Rightarrow^p A'] \longrightarrow^\star \Sigma_1 \triangleright w_1$, *then* $\Sigma \triangleright E[(w\,\star) : A[X := \star] \Rightarrow^p A'] \longrightarrow^\star \Sigma_1' \triangleright w_1'$ *for some* Σ_1' *and* w_1'.
- *If* $E[(w\,C) : A[X := C] \Rightarrow^p A']$ *diverges, then* $E[(w\,\star) : A[X := \star] \Rightarrow^p A']$ *diverges.*

Proof. It is easy to show $\Sigma \vdash_C E[(w\,C) : A[X := C] \Rightarrow^p A'] : B \sqsubseteq_{\mathcal{X}} \Sigma \vdash_C E[(w\,\star) : A[X := \star] \Rightarrow^p A'] : B$. Then the statement follows straightforwardly from Conjecture 8.4. □

9 RELATED WORK

Adding the dynamic type to statically typed languages with polymorphism. Abadi et al. [1995] studied adding the dynamic type to polymorphic programming languages. In their work, expressions of the dynamic type are investigated by typecase operation explicitly, and neither parametricity nor smooth transition between statically and dynamically typed code were studied. Gronski et al. [2006] developed the Sage language, which extends hybrid type checking by Flanagan [2006] to advanced typing features including the dynamic type, first-class types, the Type:Type discipline [Cardelli 1986]. Polymorphic functions can be expressed in Sage, but run-time checks do not enforce parametricity. Matthews and Ahmed [2008] studied parametricity in the multi-language combining an ML-like, polymorphically typed language and a Scheme-like, untyped language. They also use dynamic sealing [Morris 1973] for enforcing parametricity of statically typed values in untyped code and show parametricity.[9] Sealing in their work sticks to casts—more precisely, type annotations given to untyped code—while we separate sealing and casts, following Ahmed et al. [2011]. Since the ML-like and Scheme-like languages are fully typed and fully untyped respectively, types on sides

[9]Its proof turned out to be flawed [Neis et al. 2011], though.

of a cast are either fully static (that is, without the dynamic type) or fully dynamic (that is, the dynamic type), while types in our work may contain the dynamic type as their subcomponents, which motivates study of precision on types and the gradual guarantee. Ina and Igarashi [2011] introduced generics, which is a typing feature to obtain polymorphism in object-oriented languages, to gradual typing, but they did not study enforcement of parametricity.

Polymorphic Blame Calculus. Ahmed et al. [2009, 2011, 2013] introduced a series of polymorphic extensions of (a subset of) the blame calculus [Wadler and Findler 2009] and developed cast semantics which dynamically enforces parametricity. The later version [Ahmed et al. 2011, 2013] used type bindings to treat instantiated type variables in a special manner but the calculus had a problem that terms that are usually considered values reduce. Ahmed et al. [2017] addressed this problem by introducing a refined polymorphic blame calculus λB, from which we also incorporate some of the improvements. They also studied a formal encoding of λB into a calculus of dynamic sealing [Morris 1973]. They use dynamic sealing for expressing type bindings and casts between type variables and the dynamic type. Our interpreter uses dynamic sealing only for the latter purpose because most type information, including type bindings, is erased. Ahmed et al. [2017] proved relational parametricity of λB by a step-indexed Kripke logical relation. We expect that their proof method can be adapted to prove parametricity of System $\mathrm{F_C}$ because it is largely based on λB.

Although the polymorphic blame calculi are studied as foundations for polymorphic gradual typing, a source language design has never been studied,[10] not to mention the refined criteria [Siek et al. 2015]. As we have discussed, their compatibility relation could not be used in the type consistency of the source language as the fact that it relates $\forall X.\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Int} \to \mathsf{Int}$ would break conservativity over typing of System F.

Static and Gradual Type Variables/Parameters. Garcia and Cimini [2015], who studied type inference for gradual typing, proposed the distinction between static and gradual type parameters, which correspond to our two kinds of type variables. In their work, type variables[11] constrained only to be consistent with $\star$ are instantiated to gradual type parameters, and type variables without any constraints are instantiated to static type parameters. The distinction of type parameters leads to the fine-grained notion of principal type schemes. They also mentioned possibility of using dynamic sealing only for gradual type parameters but did not elaborate the idea further.

General approaches to gradual typing. Cimini and Siek [2016] introduced the *Gradualizer*, a general algorithmic methodology to generate gradual type systems from static type systems, and Cimini and Siek [2017] extended it so that the Gradualizer can generate dynamic semantics of gradually typed languages automatically. Garcia et al. [2016] introduced a framework called *AGT* (standing for $\underline{\text{A}}$bstracting $\underline{\text{G}}$radual $\underline{\text{T}}$yping); they reformalized type consistency by an interpretation of gradual types on the basis of the abstract interpretation. Once an interpretation of gradual types is given, all other definitions and properties of gradual typing are obtained systematically.

In spite of a great deal of generality in these approaches, they are, unfortunately, not very satisfactory to our demand. The first approach, the Gradualizer, does not instruct us how to define type consistency, which is one of the main difficulties in designing our calculus. The second, AGT, seems to need some nontrivial extensions as we have extended type consistency to a ternary relation in order to manage the classification of type variables. It will be an interesting future direction to extend these approaches so as to handle parametric polymorphism.

[10] The blame calculus itself could be used as a source language, though, as mentioned in Ahmed et al. [2011].

[11] In Garcia and Cimini [2015], type variables refer to unknown types to be inferred, while type parameters are for expressing polymorphism.

Polymorphic Manifest Contracts. Belo et al. [2011] and, more recently, Sekiyama et al. [2017] studied polymorphic extensions of a manifest contract calculus [Flanagan 2006; Greenberg et al. 2010], which integrates refinement types and dynamic contract checking. They enforce parametricity statically; in this sense, all type variables are static there. In fact, our cast rule (R_Content) is derived from their cast semantics.

10 CONCLUSION

We have introduced System $\mathrm{F_G}$ as a foundation of polymorphic gradual typing. Type abstractions and the key relations of gradual typing, such as type consistency, subtyping, precision, and so on, are extended, along with the introduction of static and gradual type variables. We have also proposed System $\mathrm{F_C}$ as an intermediate cast calculus on the basis of the existing polymorphic blame calculi. We have informally demonstrated the "pay-as-you-go" result by showing prototype implementation of a type-erasing interpreter. It is interesting future work to study erasure translation more formally along the line of Siek and Wadler [2016]. The Blame Theorem is also shown with sophisticated subtyping relations. Almost all the desired criteria of gradual typing have been proved in our calculus except for the gradual guarantee about semantics. As future work, we are going to prove the gradual guarantee on semantics; the Jack-of-All-Trades problem will also be solved in the process. Finally, parametricity of System $\mathrm{F_C}$ should be proved.

ACKNOWLEDGMENTS

We would like to thank Jeremy Siek, Phil Wadler, and anonymous reviewers for valuable comments. This work was supported in part by the JSPS KAKENHI Grant Number JP17H01723.

REFERENCES

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. 13, 2 (1991), 237–268.

Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. 1995. Dynamic Typing in Polymorphic Languages. *J. Funct. Program.* 5, 1 (1995), 111–130.

Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. 2009. Blame for all. In *Proc. of Workshop on Stript to Program Evolution.*

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 201–214. https://doi.org/10.1145/1926385.1926409

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *Proceedings of the 22st ACM SIGPLAN International Conference on Functional Programming, ICFP 2017.*

Amal Ahmed, James T. Perconti, Jeremy G. Siek, and Philip Wadler. 2013. Blame for All (revised). (2013).

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014.* 283–295.

João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. 2011. Polymorphic Contracts. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings.* 18–37.

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings.* 257–281.

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C$^{\#}$. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings.* 76–100.

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* 68–94.

Luca Cardelli. 1986. *A Polymorphic λ-calculus with Type:Type.* Technical Report 10. DEC Systems Research Center.

Matteo Cimini and Jeremy G. Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 443–455. https://doi.org/10.1145/2837614.2837632

Matteo Cimini and Jeremy G. Siek. 2017. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 789–803.

Facebook. 2017. The Hack Programming Language. (2017). http://hacklang.org/.

Cormac Flanagan. 2006. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006.* 245–256.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 303–315. https://doi.org/10.1145/2676726.2676992

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 429–442. https://doi.org/10.1145/2837614.2837670

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'état. Université Paris VII. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts made manifest. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* 353–364.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop.*

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, October 22, 2007, Montreal, Quebec, Canada.* 29–40.

Lintaro Ina and Atsushi Igarashi. 2011. Gradual typing for generics. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011.* 609–624.

Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 16–31.

Jacob Matthews and Robert Bruce Findler. 2009. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.* 31, 3 (2009), 12:1–12:44.

Erik Meijer and Peter Drayton. 2004. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages.*

James H. Morris, Jr. 1973. Protection in Programming Languages. 16, 1 (Jan. 1973), 15–21.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *J. Funct. Program.* 21, 4-5 (2011), 497–562.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press, Chapter 23.

John Reynolds. 1974. Towards a Theory of Type Structure. In *Proc. Colloque sur la Programmation (Lecture Notes in Computer Science)*, Vol. 19. Springer-Verlag, 408–425.

Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. Syst.* 39, 1, Article 3 (Feb. 2017), 36 pages.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme And Functional Programming Workshop.* 81–92.

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings.* 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus.* 7. https://doi.org/10.1145/1408681.1408688

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA.* 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* 365–376.

Jeremy G. Siek and Philip Wadler. 2016. The key to blame: Gradual typing meets cryptography. (2016).

Satish Thatte. 1990. Quasi-static Typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990*. 367–381.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 964–974.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 395–406.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Dynamic Language Symposium*. 45–56.

Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. 309–320.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 1–16. https://doi.org/10.1007/978-3-642-00590-9_1