Principal Type Schemes for Gradual Programs

With updates and corrections since publication (Latest: May 16, 2017)

Ronald Garcia *
Software Practices Lab
Department of Computer Science
University of British Columbia
rxg@cs.ubc.ca

Matteo Cimini †
Indiana University
mcimini@indiana.edu

Abstract

Gradual typing is a discipline for integrating dynamic checking into a static type system. Since its introduction in functional languages, it has been adapted to a variety of type systems, including object-oriented, security, and substructural. This work studies its application to implicitly typed languages based on type inference. Siek and Vachharajani designed a gradual type inference system and algorithm that infers gradual types but still rejects ill-typed static programs. However, the type system requires local reasoning about type substitutions, an imperative inference algorithm, and a subtle correctness statement.

This paper introduces a new approach to gradual type inference, driven by the principle that gradual inference should only produce static types. We present a static implicitly typed language, its gradual counterpart, and a type inference procedure. The gradual system types the same programs as Siek and Vachharajani, but has a modular structure amenable to extension. The language admits let-polymorphism, and its dynamics are defined by translation to the Polymorphic Blame Calculus (Ahmed et al. 2009, 2011).

The principal types produced by our initial type system mask the distinction between static parametric polymorphism and polymorphism that can be attributed to gradual typing. To expose this difference, we distinguish static type parameters from gradual type parameters and reinterpret gradual type consistency accordingly. The resulting extension enables programs to be interpreted using either the polymorphic or monomorphic Blame Calculi.

1. Introduction

Interest in integrating static and dynamic checking is increasing among language researchers and industrial language designers. (e.g. (Bierman et al. 2010; Gronski et al. 2006; Swamy et al. 2014; Tobin-Hochstadt and Felleisen 2008; Wrigstad et al. 2010)) Among the proposed foundations for such languages, *gradual typing*, due to Siek and Taha (2006) has been used to integrate dynamic checks into a variety of type structures, including object-oriented (Siek and Taha 2007), substructural (Wolff et al. 2011), and ownership types (Sergey and Clarke 2012). The key technical pillars of gradual typing are the unknown type, ?, the *consistency* relation among gradual types, and support for the entire spectrum between purely dynamic and purely static checking. A gradual type checker rejects type inconsistencies in programs, accepts statically safe code, and instruments code that could plausibly be safe with runtime checks.

This paper applies the gradual typing approach to implicitly typed languages, like Standard ML, OCaml, and Haskell, where a type inference (a.k.a. type reconstruction) procedure is integral to type checking. This problem was investigated first by Siek and Vachharajani (2008), defining $\lambda^{?\alpha}_{\rightarrow}$, a gradual type inference system and algorithm that infers gradual types, but rejects ill-typed static programs.[1] This groundbreaking paper outlines principles that a gradual implicitly typed language should satisfy, demonstrates how several plausible approaches fail, and ultimately produces a compelling type system and type inference algorithm. Though the results satisfy the criteria for gradual typing, the type system requires special care in its handling of type variables, subtle statements and proofs of correctness, and a special-purpose imperative inference algorithm. These complexities make it unclear how to adopt, adapt, and extend this approach with modern features of implicitly typed languages, like let-polymorphism, row-polymorphism, and first-class polymorphism. Furthermore, extending an existing implicitly typed language implementation with support for gradual typing would require a rewrite or substantial overhaul of the type inferencer.

To support the extension of implicitly typed languages with support for gradual typing, this paper introduces a new foundation for gradual implicit typing. In particular, we make the following contributions:

1. We introduce a specification of static implicit typing that emphasizes the input-output modes of the type system. In particular the types of subterms are viewed as opaque, and we rely on partial functions to structure the system. This structure leads to a natural conception of gradual implicit typing that is easy to reason about and extend. The key insight underlying the design is to only allow fully static types to be implicit: gradual types, in our approach, must originate in the program text.
2. Siek and Vachharajani's type system is based explicitly on inferring gradual types and emphasizing type precision, a sub-

* Partially funded by an NSERC discovery grant.

† Partially funded by NSF grant 1360694.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676992

[1] Rastogi et al. (2012) also combine gradual typing and type inference, but focus on improving performance rather than detecting inconsistencies (Sec. 13).

stantially different conceptual foundation. Nonetheless, we prove that the two foundations coincide: both type systems accept exactly the same programs.

3. We define a corresponding constraint typing judgment and constraint solver. Central to its design is that we choose to limit the type inference problem to deducing only static types. In addition to the standard equality constraints between static types, we require consistency constraints between gradual types; our solver naturally extends unification to support them.
4. To confirm the extensibility of the type system, we extend it with support for let-polymorphism, using the standard approach off the shelf. We give the language dynamics by translating it to the Polymorphic Blame Calculus (Ahmed et al. 2009, 2011). The translation mirrors the analogous translation of Hindley/Milner typing to System F, suggesting that recent approaches to first-class polymorphism may also apply.
5. We observe that the principal types of the initial system mask the distinction between static parametric polymorphism and polymorphism due to gradual typing. To expose this difference, we distinguish static type parameters from gradual type parameters and reinterpret gradual type consistency accordingly. The resulting language can be interpreted using either a monomorphic or polymorphic intermediate language.

2. Implicit Typing and Type Inference

This section briefly introduces the Implicitly Typed Static Language (ITSL). Its main purpose is to introduce the core type system that we extend with gradual typing, as well as to briefly review the principles of implicit typing. We highlight some particular details of our presentation that are important to the overall approach.

$$
\begin{array}{rcll}
\multicolumn{4}{c}{A \in \textsc{TParam},\quad T \in \textsc{Type},\quad x \in \textsc{Var},\quad b \in \textsc{Bool},} \\
\multicolumn{4}{c}{n \in \mathbb{Z},\quad t \in \textsc{Term},\quad v \in \textsc{Value}\quad \Gamma \in \textsc{Var} \rightharpoonup \textsc{Type}} \\
T & ::= & A \mid \mathsf{Int} \mid \mathsf{Bool} \mid T \to T & \text{(types)} \\
t & ::= & n \mid t + t \mid b \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \text{(terms)} \\
 & \mid & x \mid \lambda x.t \mid \lambda x : T.t \mid t\ t \mid t :: T & \\
v & ::= & n \mid b \mid x \mid \lambda x.t \mid \lambda x : T.t & \text{(syntactic values)}
\end{array}
$$

Figure 1. Static Language (ITSL): Syntax

Fig. 1 presents the syntax for ITSL. By *implicit typing* we mean that the language is statically typed, but programmers may omit type annotations. In particular, this language provides function expressions $\lambda x.t$ that require no type annotation on their parameter, in contrast to the annotated form $\lambda x : T.t$, which it also supports. As a convenience, the language also provides type ascriptions $t :: T$, which could already be simulated as $((\lambda x : T.x)\ t)$.

The implications of these language features become clearer in the type system specification, presented in Fig. 2. The (Tλ:) rule uses the type annotation on a function parameter to type its body. The (Tλ) rule, on the other hand, types a function's body with any parameter type that works. The broader impact of this is that a function like $\lambda x : \mathsf{Int}.x$ has only one legal type, $\mathsf{Int} \to \mathsf{Int}$, but a function like $\lambda x.x$ can be given many types, specifically any type of the form $T \to T$. This flexibility reduces the annotation overhead for programmers while retaining static safety, and increases the expressiveness of the language if a polymorphic let construct is added (see Sec. 10).

Presentation Style There are some notable differences between the presentation in Fig. 2 and the typical textbook presentation (c.f. (Pierce 2002)). They do not affect the meaning of this type system definition, but they play an important structuring role that we exploit when we extend it to support gradual typing.

$$
\text{(Tx)}\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(Tn)}\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \text{(Tb)}\frac{}{\Gamma \vdash b : \mathsf{Bool}}
$$

$$
\text{(Tapp)}\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad dom(T_1) = T_2}{\Gamma \vdash t_1\ t_2 : cod(T_1)}
$$

$$
\text{(Tif)}\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_1 = \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : equate(T_2, T_3)}
$$

$$
\text{(T+)}\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 = \mathsf{Int} \quad T_2 = \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}
$$

$$
\text{(T}\lambda\text{)}\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x.t) : T_1 \to T_2}
$$

$$
\text{(T}\lambda\text{:)}\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1.t) : T_1 \to T_2} \qquad \text{(T::)}\frac{\Gamma \vdash t : T \quad T = T_1}{\Gamma \vdash (t :: T_1) : T_1}
$$

$dom : \textsc{Type} \rightharpoonup \textsc{Type}$
$dom(T_1 \to T_2) = T_1$
$dom(T)$ undefined otherwise

$cod : \textsc{Type} \rightharpoonup \textsc{Type}$
$cod(T_1 \to T_2) = T_2$
$cod(T)$ undefined otherwise

$equate : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type}$
$equate(T, T) = T$
$equate(T_1, T_2)$ undefined otherwise

Figure 2. Static Language (ITSL): Type System

The type system definition pays particular attention to the *mode* of the typing judgment, in the sense of logic programming (Debray and Warren 1988). In particular, the type context and term are interpreted as *inputs* to the typing judgment, while the term's type is viewed as an *output*. The impact of this interpretation on the structure of the definition is as follows, and can be seen in the (Tapp) rule. Though the structure of the terms t in the conclusions of rules are analyzed using pattern-matching-style syntax, the type position of each typing judgment in the premise is stated abstractly as *some* type T. For instance, the type rule for application usually has a premise $\Gamma \vdash t_1 : T_{11} \to T_{12}$, but in contrast the (Tapp) rule only assumes that the result is a type T_1. In lieu of pattern matching on a function type, the rules appeal to *partial* functions that are defined only for types with the proper shape. These partial functions name the result type of a term (e.g., $cod(T_1)$) and assert properties that must hold between interacting types (e.g., $dom(T_1) = T_2$).[2]

Our partial functions are partly responsible for imposing requirements on types. For instance, reference to $dom(T_1)$ in a rule implies that $T_1 = T_{11} \to T_{12}$ for some T_{11}, T_{12}, so this requirement does not need to be explicitly stated in the rule. If $dom(T_1)$ is undefined, then the (Tapp) rule does not apply. Using partial functions to abstract away type requirements is critical to our subsequent development of gradual typing.

A particularly interesting example of this phenomenon arises in the (Tif) rule. The requirement that T_1 must be Bool is standard, but the result type of the term is a partial function $equate(T_2, T_3)$, that is defined only when the two types are equal, in which case it is the type itself. Typically, this rule is expressed by using pattern matching on the results of typing t_2 and t_3 to force them into the same type, and then providing that type as the final result. Instead, we keep the result types abstract and appeal to a partial function to "combine" them.

In summary, we use predicates only to express relationships between types that do not imply some result type. For all cases

[2] In propositions that refer to partial functions, we interpret $=$ as Kleene equality: both sides of the equation must either be defined and equal, or undefined.

where we need to determine a type in terms of subterm types, we defer to a partial function.

2.1 Type Polymorphism

As we mentioned earlier, a function like $\lambda x.x$ can be given the types $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Bool}$, and any other type of the form $T \to T$. We would like to express this polymorphism somehow, but our language does not support first-class polymorphism: each type is concrete. For this purpose we endow the language with a set of *type parameters* A, which are uninterpreted types in the sense that the type system gives them no special treatment. As a consequence any situation where a type parameter A is used, any other particular type could appear. Let us formalize this observation.

Definition 1. *Let* $S^A \in \text{ASUBST} = \text{TPARAM} \to \text{TYPE}$ *denote type parameter substitutions, or* A-substitutions*.*

Notation. *If* S *is some sort of substitution function, then we use* $\widehat{S}(T)$*,* $\widehat{S}(\Gamma)$*, and* $\widehat{S}(t)$ *to denote the compatible closure of substitution over types, type contexts, and terms, respectively.*

The A-substitutions give meaning to type parameters. With this we can treat them as representatives for type polymorphism.

Proposition 1. *If* $\Gamma \vdash t : T$ *then* $\widehat{S^A}(\Gamma) \vdash \widehat{S^A}(t) : \widehat{S^A}(T)$ *for any* $S^A \in \text{ASUBST}$*.*

Proof. Straightforward induction on derivations of $\Gamma \vdash t : T$. □

Prop. 1 shows us that a type with parameters can be made more specific arbitrarily. Regarding the dual consideration, we can ask about the presence of a *most general* type for a program.

Definition 2 (Principal Type)**.** *If* $\Gamma \vdash t : T$*, then we say that* T *is a* principal type *of* Γ *and* t *if whenever* $\Gamma \vdash t : T_0$ *holds, then* $\widehat{S^A}(\Gamma) = \Gamma$*,* $\widehat{S^A}(t) = t$*, and* $T_0 = \widehat{S^A}(T)$ *for some* A*-substitution* S^A*.*

The first two constraints on the substitution S^A force any parameters that explicitly appear in the term or context to be treated as constants, truly uninterpreted types. The third constraint formalizes the sense that T unambiguously represents all possible types that can be assigned to the term-context pair.

Principal types succinctly summarize all possible types of a program. This particular language has principal types for every typeable program. We establish this once and for all in the gradually typed setting.

2.2 Type Inference

While this type system specification has great appeal, the question arises as to whether it can be effectively type checked. Indeed it can, and a broad array of literature has discussed approaches to doing so. We desire even more flexibility than the language provides as-is. To do so, we generalize the problem beyond purely elided type annotations.

To check implicit types, we must deduce types that are not present in the program text. To represent this problem in a broader context, we introduce notions of *type variables* and *type expressions*:

$$X \in \text{TVAR}, \quad T^{\mathtt{X}} \in \text{TYPEEXP}$$
$$T^{\mathtt{X}} ::= X \mid A \mid \mathsf{Int} \mid \mathsf{Bool} \mid T^{\mathtt{X}} \to T^{\mathtt{X}}$$

A type variable is a placeholder for one or more possible types, directly analogous to variables in secondary school algebra. Then a type expression $T^{\mathtt{X}}$ is just a type where some of its substructure has been filled with type variables. This idea extends to term expressions $t^{\mathtt{X}} \in \text{TERMEXP}$ and *type expression contexts* $\Gamma^{\mathtt{X}} \in \text{VAR} \rightharpoonup \text{TYPEEXP}$. We write $\text{Vars}(T^{\mathtt{X}})$ for the set of type variables in $T^{\mathtt{X}}$. The function Vars extends to terms and type contexts in the expected way, and in the remainder of the paper we simply write $\text{Vars}(o_0, o_1, \ldots, o_n)$ for $\text{Vars}(o_0) \cup \text{Vars}(o_1) \ldots \cup \text{Vars}(o_n)$, for some $n \geq 0$ and where o_i can be a type, a term or a type context, for $0 \leq i \leq n$.

Note that type variables and type parameters are distinct concepts. Type parameters actually *are* types, while type variables are merely placeholders, just as variables in algebra are not numbers. However, all types count as type expressions too (i.e. $\text{TYPE} \subset \text{TYPEEXP}$); the analogous status also holds for term expressions and type expression contexts.[3] We give all of these expression forms meaning by substituting types for type variables.

Definition 3. *Let* $S^{\mathtt{X}} \in \text{XSUBST} = \text{TVAR} \to \text{TYPE}$ *denote type variable substitutions, or* X*-substitutions.*

Observe that these substitutions produce real types, terms, and type contexts in accordance with the source language.

Definition 4. *Let* $S_1^{\mathtt{X}}$ *and* $S_2^{\mathtt{X}}$ *be two type variable substitutions and let* $\mathcal{X} \subseteq \text{TVAR}$*. We say that* $S_2^{\mathtt{X}}$ agrees with $S_1^{\mathtt{X}}$ over $\mathcal{X}$ *whenever for all variables* $X \in \mathcal{X}$*, it holds that* $S_1^{\mathtt{X}}(X) = S_2^{\mathtt{X}}(X)$*.*

In algebra, we use variables to write down equation expressions, which are problems that may have solutions. By analogy, we use our expressions to formalize the *type inference problem* and to characterize its solutions.

Definition 5 (Type Inference Problem)**.** *Given a type expression context* $\Gamma^{\mathtt{X}}$ *and a term expression* $t^{\mathtt{X}}$*, the* type inference problem *for ITSL asks if* $\widehat{S^{\mathtt{X}}}(\Gamma^{\mathtt{X}}) \vdash \widehat{S^{\mathtt{X}}}(t^{\mathtt{X}}) : T$ *for some* X*-substitution* $S^{\mathtt{X}}$ *and type* T*.*

Only the context and term are expressions here.

A type inferencer, then, is a decision procedure for the type inference problem. We do not present a type inference algorithm for this static language. However, such an algorithm can be extracted directly from the gradual type inferencer we present below.

3. Introduction to Gradual Typing

Gradual typing is a type discipline, introduced by Siek and Taha, to extend an existing type system with seamless support for static and dynamic checks. Such a language supports both extremes of checking, fully static and fully dynamic, as well as any point in between. The gradual type system uses available type information to detect inconsistencies between types. Typing conditions that cannot be determined statically are checked at runtime using casts.

To achieve these goals, gradual typing extends an existing type system with an *unknown type* ?. For example, the corresponding gradual types for the language from the last section are as follows:

$$U \in \text{GTYPE}$$
$$U ::= ? \mid A \mid \mathsf{Int} \mid \mathsf{Bool} \mid U \to U \quad \text{(gradual types)}$$

Based on this definition, every static type is also a gradual type.

Fig. 3 presents the type system for $\lambda^?_\to$, the simply typed gradual calculus (Siek and Taha 2006). The rules for variables and functions are standard, but the rules for function application are not. The (App?) rule applies when the operator has the unknown type. Since the result could plausibly be a function, the program type checks, and the result type of the application is unknown in turn.

The (AppU) rule applies when the operator has some function type. Unlike static typing, the rule does not require that the operand have the same type as the domain of the operator, but simply that two types be *consistent*, written $U \sim U$. We define consistency as

[3] Some presentations of type inference use a single entity for both concepts. We distinguish them for clarity and to support our development in Sec. 12.

$$\text{(Var)}\frac{x : U \in \Gamma}{\Gamma \vdash x : U} \qquad \text{(Abs)}\frac{\Gamma, x : U_1 \vdash t : U_2}{\Gamma \vdash (\lambda x : U_1.t) : U_1 \to U_2}$$

$$\text{(AppU)}\frac{\Gamma \vdash t_1 : U_{11} \to U_{12} \quad \Gamma \vdash t_2 : U_1 \quad U_{11} \sim U_1}{\Gamma \vdash t_1\ t_2 : U_{12}}$$

$$\text{(App?)}\frac{\Gamma \vdash t_1 : ? \quad \Gamma \vdash t_2 : U_2}{\Gamma \vdash t_1\ t_2 : ?}$$

Figure 3. $\lambda^{?}_{\to}$: Simply Typed Gradual Calculus

follows.

$$\frac{}{\mathsf{Int} \sim \mathsf{Int}} \qquad \frac{}{\mathsf{Bool} \sim \mathsf{Bool}} \qquad \frac{}{A \sim A} \qquad \frac{}{? \sim U}$$

$$\frac{}{U \sim ?} \qquad \frac{U_{11} \sim U_{21} \quad U_{12} \sim U_{22}}{U_{11} \to U_{12} \sim U_{21} \to U_{22}}$$

The intuition behind this definition is that two consistent gradual types need only agree on their statically known parts. Consistency is one of the key innovations of the gradual typing approach.

The type system must be a conservative extension of the underlying type system. This means that programs without any dynamism must type according to the underlying static type system. Since consistency reduces to equality in the absence of ?, it follows that the type system conservatively extends its pre-existing simply typed counterpart. A programmer, development environment, or syntactically-sugared source language then introduces dynamism by ascribing ? to program subterms (see Sec. 6).

In addition to consistency, gradual types impose a notion of *type precision* as a partial order $\sqsubseteq$, which indicates that one type is less unknown than another.

$$\frac{}{U \sqsubseteq ?} \qquad \frac{}{U \sqsubseteq U} \qquad \frac{U_{11} \sqsubseteq U_{21} \quad U_{12} \sqsubseteq U_{22}}{U_{11} \to U_{12} \sqsubseteq U_{21} \to U_{22}}$$

This relation coincides with the "naïve subtyping" relation of (Wadler and Findler 2009).[4]

To support gradual typing, programs are instrumented with type-directed runtime checks that ensure that consistency checks that cannot be fully resolved at type-checking time are dynamically verified.

In short, a gradual type system takes an optimistic approach to checking that two types are compatible, only statically rejecting combinations that exhibit inconsistencies, and dynamically verifying any remaining checks.

4. A Gradual Implicitly Typed Language

This section presents our conception of gradual typing for an implicitly typed language. The ideas from the last two sections combine to yield a design that differs from $\lambda^{?\alpha}_{\to}$ in its foundations, but is ultimately equivalent.

Fig. 4 presents the syntax of the Implicitly Typed Gradual Language (ITGL). As with standard practice, the singular difference is lifting of static types T to gradual types U. Every static type annotation can now be replaced with a gradual type annotation.

The type system, presented in Fig. 5 builds on the ITSL type system from Sec. 2. Most of the rules have the same structure, except their static types have been lifted to gradual types, and the predicates and partial functions on static types have been lifted to consistent predicates and partial functions on gradual types. A straightforward example of this is the (U+) rule, which replaces type equality with type consistency, and static types with gradual types.

[4] We avoid calling it "subtyping" because it does not characterize substitutability.

$$\begin{array}{rcll}
& A \in \textsc{TParam}, & T \in \textsc{Type}, & U \in \textsc{GType}, \quad x \in \textsc{Var}, \\
& b \in \textsc{Bool}, & n \in \mathbb{Z}, & t \in \textsc{Term}, \quad v \in \textsc{Value} \\
T & ::= & A \mid \mathsf{Int} \mid \mathsf{Bool} \mid T \to T & \text{(static types)} \\
U & ::= & ? \mid A \mid \mathsf{Int} \mid \mathsf{Bool} \mid U \to U & \text{(gradual types)} \\
t & ::= & n \mid t + t \mid b \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \text{(terms)} \\
& \mid & x \mid \lambda x.t \mid \lambda x : U.t \mid t\ t \mid t :: U & \\
v & ::= & n \mid b \mid x \mid \lambda x.t \mid \lambda x : U.t & \text{(syntactic values)}
\end{array}$$

Figure 4. Gradual Language (ITGL): Syntax

$$\text{(Ux)}\frac{x : U \in \Gamma}{\Gamma \vdash x : U} \qquad \text{(Un)}\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \text{(Ub)}\frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$\text{(Uapp)}\frac{\Gamma \vdash t_1 : U_1 \quad \Gamma \vdash t_2 : U_2 \quad \widetilde{dom}(U_1) \sim U_2}{\Gamma \vdash t_1\ t_2 : \widetilde{cod}(U_1)}$$

$$\text{(Uif)}\frac{\Gamma \vdash t_1 : U_1 \quad \Gamma \vdash t_2 : U_2 \quad \Gamma \vdash t_3 : U_3 \quad U_1 \sim \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : U_2 \sqcap U_3}$$

$$\text{(U+)}\frac{\Gamma \vdash t_1 : U_1 \quad \Gamma \vdash t_2 : U_2 \quad U_1 \sim \mathsf{Int} \quad U_2 \sim \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$\text{(U}\lambda\text{)}\frac{\Gamma, x : T_1 \vdash t : U_2}{\Gamma \vdash (\lambda x.t) : T_1 \to U_2}$$

$$\text{(U}\lambda\text{:)}\frac{\Gamma, x : U_1 \vdash t : U_2}{\Gamma \vdash (\lambda x : U_1.t) : U_1 \to U_2} \qquad \text{(U::)}\frac{\Gamma \vdash t : U \quad U \sim U_1}{\Gamma \vdash (t :: U_1) : U_1}$$

$$\begin{array}{ll}
\widetilde{dom} : \textsc{GType} \rightharpoonup \textsc{GType} & \widetilde{cod} : \textsc{GType} \rightharpoonup \textsc{GType} \\
\widetilde{dom}(U_1 \to U_2) = U_1 & \widetilde{cod}(U_1 \to U_2) = U_2 \\
\widetilde{dom}(?) = ? & \widetilde{cod}(?) = ? \\
\widetilde{dom}(U) \text{ undefined otherwise} & \widetilde{cod}(U) \text{ undefined otherwise}
\end{array}$$

$$\begin{array}{c}
\sqcap : \textsc{GType} \times \textsc{GType} \rightharpoonup \textsc{GType} \\
T \sqcap T = T \\
? \sqcap U = U \sqcap ? = U \\
(U_{11} \to U_{12}) \sqcap (U_{21} \to U_{22}) = (U_{11} \sqcap U_{21}) \to (U_{12} \sqcap U_{22}) \\
U_1 \sqcap U_2 \text{ undefined} \quad \text{otherwise}
\end{array}$$

Figure 5. Gradual Language (ITGL): Type System

If we compare this type system to $\lambda^{?}_{\to}$, we notice that the former has one rule for function application, (Uapp), while the latter has two, (AppU) and (App?). In this case, the $\widetilde{dom}$ partial function provides the necessary abstraction to combine the two rules into one. The $\widetilde{dom}$ function is defined with two cases: folding them into the (Uapp) rule, along with $\widetilde{cod}$ would recover the two rules of $\lambda^{?}_{\to}$.

Consider the (Uif) rule. Most of the structure is as expected, but its result type is $U_2 \sqcap U_3$, the greatest lower bound (or *meet*) of U_2 and U_3 according to the precision relation. In fact, the meet is the gradual partial function that corresponds to $equate$. To see how, consider the two gradual types $? \to \mathsf{Int}$ and $\mathsf{Bool} \to ?$. The only hope for some U to be a well-defined result of combining them is for U to be $\mathsf{Bool} \to \mathsf{Int}$. Naturally, one can always coerce two inconsistent branches to type-check using type ascriptions.

Implicit types must be static types The (Uλ) rule embodies implicit typing in ITGL. The corresponding ITSL rule types the body of the function using some arbitrary type, and the gradual type system does the same. To understand why the arbitrary type is static rather than gradual, consider the program $\lambda x.x\ x$. This program has no unknown type annotations, so it is in the language of the

ITSL type system, which cannot type it. As such, the gradual type system must reject it as well. If the gradual type system could arbitrarily type x as ?, however, then this function would type check, even though the programmer introduced no ? annotations. The key insight is that dynamicity is introduced only via program annotations. The type system itself must never introduce ? of its own accord. As we see next, this insight has broader implications.

4.1 Gradual Inference of Static Types

As with ITSL, we generalize type inference, taking into account the insight that our type system need only infer static types. To state the gradual type inference problem, we introduce corresponding type expressions, term expressions, and type context expressions. For brevity, we present only the gradual type expressions.

$$U^{\mathtt{X}} \in \text{GTYPEEXP}$$
$$U^{\mathtt{X}} ::= ? \mid X \mid A \mid \mathsf{Int} \mid \mathsf{Bool} \mid U^{\mathtt{X}} \to U^{\mathtt{X}}$$

Gradual type expressions $U^{\mathtt{X}}$ now include ?, since we are in a gradually-typed language. However, we do not update the definition of X-substitutions: type variables map to static types. This property generalizes the reasoning underlying the (Uλ) rule: we only allow gradual types to be explicitly introduced in a program as a means to ensure that the type system cannot introduce dynamism without prompt and thereby compromise inconsistency checking.

Definition 6 (Gradual Type Inference Problem)**.** *Given a gradual type expression context $\Gamma^{\mathcal{X}}$ and a term expression $t^{\mathcal{X}}$, the* gradual type inference problem *asks if there is an X-substitution $S^{\mathcal{X}}$ and gradual type U such that $\widehat{S^{\mathcal{X}}}(\Gamma^{\mathcal{X}}) \vdash \widehat{S^{\mathcal{X}}}(t^{\mathcal{X}}) : U$.*

In contrast, $\lambda^{?\alpha}_{\to}$ (Siek and Vachharajani 2008) infers gradual types, albeit with restrictions. In the next section, we reconcile this difference.

5. $\lambda^{?\alpha}_{\to}$ types the same programs

Siek and Vachharajani (2008) analyze the concept of gradual type inference and design a language $\lambda^{?\alpha}_{\to}$ that formalizes and substantiates their approach. The $\lambda^{?\alpha}_{\to}$ language and types correspond to our gradual type expressions, but the type theoretic foundations differ substantially. In particular, $\lambda^{?\alpha}_{\to}$ is based on inferring *gradual* types, not just static types, but in a controlled manner. To formalize this, we introduce a new notion of substitutions.

$$S^G \in \text{GSUBST} = \text{TVAR} \to \text{GTYPE}$$
$$\mathcal{X} \in \mathcal{P}(\text{TVAR})$$

The $\lambda^{?\alpha}_{\to}$ type system is presented in Fig. 6.[5] It is based on the judgment $S^G; \Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid \mathcal{X}$, which intuitively means that S^G and $\widehat{S^G}(U^{\mathtt{X}})$ solve a constrained variant of the type inference problem $\Gamma^{\mathtt{X}}$ and $t^{\mathtt{X}}$ for $\lambda^{?}_{\to}$ (Sec. 3), where extra variables $\mathcal{X}$ and their mappings are involved in the typing process. More precisely it means that in $\lambda^{?}_{\to}$, $\widehat{S^G}(\Gamma^{\mathtt{X}}) \vdash \widehat{S^G}(t^{\mathtt{X}}) : \widehat{S^G}(U^{\mathtt{X}})$ *and moreover* every type variable X used to type $t^{\mathtt{X}}$ (i.e. present in $t^{\mathtt{X}}$ or $\mathcal{X}$) must map to some $S^G(X)$ such that $S^G(X) \sqsubseteq U$ for every gradual type U that it is consistent with according to the typing derivation. The first criteria is essentially the standard type inference problem, but with gradual types in the substitution; the second criteria is their approach to preventing the type system from introducing unplanned dynamism. Every type variable's mapping is constrained by the static types it interacts with, but types that originate in the program text are unconstrained.

[5] The presentation has been adjusted to match this paper.

$\boxed{S^G; \Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid \mathcal{X}} \quad \bot(\mathcal{X}, \text{Vars}(t^{\mathtt{X}}))$

$$\text{(Gx)}\frac{x : U^{\mathtt{X}} \in \Gamma^{\mathtt{X}}}{S^G; \Gamma^{\mathtt{X}} \vdash x : U^{\mathtt{X}} \mid \emptyset} \qquad \text{(Gn)}\frac{}{S^G; \Gamma^{\mathtt{X}} \vdash n : \mathsf{Int} \mid \emptyset}$$

$$\text{(Gb)}\frac{}{S^G; \Gamma^{\mathtt{X}} \vdash b : \mathsf{Bool} \mid \emptyset}$$

$$\text{(Gapp)}\frac{\begin{array}{c} S^G; \Gamma^{\mathtt{X}} \vdash t_1 : U_1^{\mathtt{X}} \mid \mathcal{X}_1 \quad S^G; \Gamma^{\mathtt{X}} \vdash t_2 : U_2^{\mathtt{X}} \mid \mathcal{X}_2 \\ S^G \models U_1^{\mathtt{X}} \simeq U_2^{\mathtt{X}} \to X \qquad \bot(\{ X \}, \mathcal{X}_1, \mathcal{X}_2) \end{array}}{S^G; \Gamma^{\mathtt{X}} \vdash t_1\ t_2 : X \mid \{ X \} \cup \mathcal{X}_1 \cup \mathcal{X}_2}$$

$$\text{(G}\lambda\text{:)}\frac{S^G; \Gamma^{\mathtt{X}}, x : U_1^{\mathtt{X}} \vdash t : U_2^{\mathtt{X}} \mid \mathcal{X}}{S^G; \Gamma^{\mathtt{X}} \vdash (\lambda x : U_1^{\mathtt{X}}.t) : U_1^{\mathtt{X}} \to U_2^{\mathtt{X}} \mid \mathcal{X}}$$

$\boxed{S^G \models U^{\mathtt{X}} \simeq U^{\mathtt{X}}}$

$$\frac{}{S^G \models \mathsf{Bool} \simeq \mathsf{Bool}} \qquad \frac{}{S^G \models \mathsf{Int} \simeq \mathsf{Int}}$$

$$\frac{}{S^G \models A \simeq A} \qquad \frac{}{S^G \models U^{\mathtt{X}} \simeq ?} \qquad \frac{}{S^G \models ? \simeq U^{\mathtt{X}}}$$

$$\frac{S^G \models U_{11}^{\mathtt{X}} \simeq U_{21}^{\mathtt{X}} \qquad S^G \models U_{12}^{\mathtt{X}} \simeq U_{22}^{\mathtt{X}}}{S^G \models U_{11}^{\mathtt{X}} \to U_{12}^{\mathtt{X}} \simeq U_{21}^{\mathtt{X}} \to U_{22}^{\mathtt{X}}}$$

$$\frac{S^G \models \widehat{S^G}(X) \sqsubseteq U_2^{\mathtt{X}}}{S^G \models X \simeq U_2^{\mathtt{X}}} \qquad \frac{S^G \models \widehat{S^G}(X) \sqsubseteq U_1^{\mathtt{X}}}{S^G \models U_1^{\mathtt{X}} \simeq X}$$

$\boxed{S^G \models U \sqsubseteq U^{\mathtt{X}}}$

$$\frac{\widehat{S^G}(X) = U}{S^G \models U \sqsubseteq X} \qquad \frac{}{S^G \models \mathsf{Bool} \sqsubseteq \mathsf{Bool}} \qquad \frac{}{S^G \models \mathsf{Int} \sqsubseteq \mathsf{Int}}$$

$$\frac{}{S^G \models A \sqsubseteq A} \qquad \frac{}{S^G \models U \sqsubseteq ?}$$

$$\frac{S^G \models U_{11}^{\mathtt{X}} \sqsubseteq U_{21}^{\mathtt{X}} \qquad S^G \models U_{12}^{\mathtt{X}} \sqsubseteq U_{22}^{\mathtt{X}}}{S^G \models U_{11}^{\mathtt{X}} \to U_{12}^{\mathtt{X}} \sqsubseteq U_{21}^{\mathtt{X}} \to U_{22}^{\mathtt{X}}}$$

Figure 6. $\lambda^{?\alpha}_{\to}$: Type System

Notation. *We write $\bot\left(\prod_{i<n} \mathcal{X}_i\right)$ to mean that the given variable sets are mutually disjoint, i.e., $\forall i < n. \forall j < n. i \neq j \Rightarrow \mathcal{X}_i \cap \mathcal{X}_j = \emptyset$.*

Most of the typing rules are standard in structure. The (Gapp) rule for function application is the most interesting. Where $\lambda^{?}_{\to}$ imposes a consistency relation between the type of the operand and the operator, this type system appeals to a judgment $S^G \models U_1^{\mathtt{X}} \simeq U_2^{\mathtt{X}}$, which means that $\widehat{S^G}(U_1^{\mathtt{X}}) \sim \widehat{S^G}(U_2^{\mathtt{X}})$ and moreover that S^G respects the aforementioned lower-bound criteria for all variables in $U_1^{\mathtt{X}}$ and $U_2^{\mathtt{X}}$. The judgment decomposes type expressions to check for consistency. Upon reaching a type variable, it appeals to the judgment $S^G \models U \sqsubseteq U_2^{\mathtt{X}}$, which means that $U \sqsubseteq \widehat{S^G}(U_2^{\mathtt{X}})$. The key rule of this judgment is the variable case. If $U \sqsubseteq S^G(X)$, then the converse—which is our global criterion—holds only if the two types are equal.

This type system satisfies the goals of gradual type inference, but its structure is unorthodox. A substitution is incorporated explicitly into the judgment, instead of simply appearing in the statement of the type inference problem. This non-modular structure leaks into the corresponding constraint typing relation and constraint-solver. The result is a type inference procedure that differs significantly from a common type inferencer: existing type

inference systems and procedures would be hard to extend with this approach.

Fortunately, and surprisingly, the type inference problems for $\lambda_{\to}^{?\alpha}$ and ITGL coincide, even though $\lambda_{\to}^{?\alpha}$ is allowed to infer gradual types. The key to understanding this is that the $\lambda_{\to}^{?\alpha}$ type system must reject any program that has a static inconsistency even after taking its type annotations into account. The type system cannot infer a ? that resolves such a conflict. As such, any ? that appears in an inferred type must be superfluous and replaceable with a static type. The propositions below formalize this insight.[6]

Definition 7. *We use the name* unknown substitution *to refer to those A-substitutions* $S^? : \text{TPARAM} \to \text{GTYPE}$ *such that either* $S^?(A) = A$ *or* $S^?(A) = ?$.

We use unknown substitutions to show that ? is never needed in practice. The property we exploit is that all substitutions S^G can be factored into some X-substitution composed with some unknown substitution $S^G = \widehat{S^?} \circ S^{\mathtt{X}}$, and any such factoring will do.

Proposition 2.

1. *Suppose* $S^G = \widehat{S^?} \circ S^X$. *Then* $S^G \models \widehat{S^G}(U_1^X) \sqsubseteq U_2^X$ *implies* $S^X \models \widehat{S^X}(U_1^X) \sqsubseteq U_2^X$.
2. *Suppose* $S^G = \widehat{S^?} \circ S^X$. *Then* $S^G \models U_1^X \simeq U_2^X$ *implies* $S^X \models U_1^X \simeq U_2^X$.
3. *Suppose* $S^G = \widehat{S^?} \circ S^X$. *Then* $S^G; \Gamma^X \vdash t^X : U^X$ *implies* $S^X; \Gamma^X \vdash t^X : U^X$.

Proof. By induction on derivations. □

Prop. 2 tells us that with regards to solving the type inference problem for $\lambda_{\to}^{?\alpha}$, we need only look to X-substitutions $S^{\mathtt{X}}$, just as for ITGL. Thus, we restrict consideration to X-substitutions for the $\lambda_{\to}^{?\alpha}$ type system, which allows us to connect the two type systems more directly.

Proposition 3 (Completeness)**.** *If* $S^X; \Gamma^X \vdash t^X : U^X \mid \mathcal{X}$ *then* $\widehat{S^X}(\Gamma^X) \vdash \widehat{S^X}(t^X) : U$ *for some* U *such that* $\widehat{S^X}(U^X) \sqsubseteq U$.

Proof. By induction on $S^{\mathtt{X}}; \Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid \mathcal{X}$. □

The intuition behind this proposition is that since $\lambda_{\to}^{?\alpha}$ imposes only a consistency on the type of (Gapp), the result type can be an arbitrary static type if the codomain of the operator is ?; in contrast, ITGL propagates the gradual domain of the operator verbatim.

Proposition 4 (Soundness)**.** *If* $\widehat{S^X}(\Gamma^X) \vdash \widehat{S^X}(t^X) : U$ *then given any finite* $\mathcal{X}_0$ *that contains* $\text{Vars}(t^X, \Gamma^X)$, $S_0^X; \Gamma^X \vdash t^X : U^X \mid \mathcal{X}$ *where* $\bot(\mathcal{X}_0, \mathcal{X})$ *for some* S_0^X *that agrees with* S^X *except at* $\mathcal{X}$, *and* $\widehat{S_0^X}(U^X) = U$.

Proof. By induction on $\widehat{S^{\mathtt{X}}}(\Gamma^{\mathtt{X}}) \vdash \widehat{S^{\mathtt{X}}}(t^{\mathtt{X}}) : U$. □

We have designed an interesting gradual type system with support for implicit typing, and have established that it types the same programs as $\lambda_{\to}^{?\alpha}$. In Section 7 we deduce a type inference process that corresponds naturally to our type system and conservatively extends the corresponding static procedure.

[6] These theorems consider only the corresponding subset of ITGL.

6. Application: Interlanguage Migration

Now that we have seen the syntax and static semantics of ITGL, the question remains: what can we do with it? We have already seen that ITGL conservatively extends ITSL, accepting and rejecting programs in the ITSL syntax exactly as that type system does. Furthermore, the last section demonstrates that ITGL subsumes $\lambda_{\to}^{?\alpha}$, and Siek and Vachharajani (2008) demonstrate how statically-inclined programmers can leverage the flexibility of gradual typing to write programs that would not be accepted by a purely static type system. As such, ITGL shares $\lambda_{\to}^{?\alpha}$'s example programs, and we defer to that paper for static-leaning applications.

In this section, we consider the converse question: how does ITGL serve the dynamically-inclined programmer? Gradual typing's goal is to serve both communities, filling the space between static and dynamic while subsuming both. We demonstrate how ITGL addresses the dynamic side of this goal. We present a dynamic counterpart to ITSL and demonstrate by a result of Siek and Taha (2006), that it amounts to a syntactic discipline over ITGL. We then present a hybrid surface language that is suitable for migration between dynamic and static programs. These language designs are straightforward embeddings into ITGL.

6.1 A Dynamic Language

If we reconsider the ITSL static language, the syntax of its natural dynamic counterpart, which we call DL, simply amounts to eliding type information:

$$
\begin{array}{lll}
e \in \text{DYNAMIC} & & \\
e & ::= & n \mid e + e \mid b \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \quad \text{(terms)} \\
& \mid & x \mid \lambda x.e \mid e\ e
\end{array}
$$

It's immediately clear that $\text{DYNAMIC} \subset \text{TERM}$: every DL program looks exactly like an ITSL program, but looks can be deceiving. In particular, one consistent property of dynamic languages is that any program in the syntax of the language has semantics. In contrast, ITSL only gives meaning to programs that are well-typed. For instance, $(\lambda x.x\ x)$ is a fine DL program, but it is not a legal ITSL program because it is not well-typed. The field of linguistics has a term for this: two similar-looking expressions, sometimes in different languages, that might be mistaken for having the same meaning, but do not, are called *false friends* (Crystal 2008). In natural languages they lead to misunderstandings between native speakers of different languages: in programming languages, the phenomenon recurs.

How do we reconcile these syntactic similarities but semantic differences between ITSL and DL? The key lies in ITGL: we define both languages by translation to ITGL. The translation for ITSL is trivial because it embeds immediately. The translation for DL (Siek and Taha 2006), on the other hand, has substance:

$$
\begin{array}{c}
\mathcal{G}[\![\cdot]\!] : \text{DYNAMIC} \to \text{TERM} \\
\mathcal{G}[\![n]\!] = n :: ? \qquad \mathcal{G}[\![b]\!] = b :: ? \qquad \mathcal{G}[\![x]\!] = x :: ? \\
\mathcal{G}[\![e_1 + e_2]\!] = (\mathcal{G}[\![e_1]\!] + \mathcal{G}[\![e_2]\!]) :: ? \\
\mathcal{G}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] = (\mathsf{if}\ \mathcal{G}[\![e_1]\!]\ \mathsf{then}\ \mathcal{G}[\![e_2]\!]\ \mathsf{else}\ \mathcal{G}[\![e_3]\!]) :: ? \\
\mathcal{G}[\![\lambda x.e]\!] = (\lambda x : ?.\mathcal{G}[\![e]\!]) :: ? \\
\mathcal{G}[\![e_1\ e_2]\!] = (\mathcal{G}[\![e_1]\!]\ \mathcal{G}[\![e_2]\!]) :: ?
\end{array}
$$

Under this translation, $\mathcal{G}[\![\lambda x.x\ x]\!] = (\lambda x : ?.(x :: ?)\ (x :: ?) :: ?) :: ?$, which is a well-typed ITGL program, as are all embeddings of DL programs.

Proposition 5 (Siek and Taha (2006))**.** *Let* X *be the free variables of* e *and* $\Gamma = \{\, x : ? \mid x \in X \,\}$. *Then* $\Gamma \vdash \mathcal{G}[\![e]\!] : ?$.

As we desire, all terms in DYNAMIC are well-defined DL programs.

6.2 A Hybrid Language for Interlanguage Migration

Armed with a translational semantics for DL and a (trivial) translational semantics for ITSL, we can now combine them. The result is a hybrid multi-language semantics, which we call HL, that seamlessly supports migration between the DL and ITSL languages (Matthews and Findler 2009; Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009):

$$
\begin{array}{lcll}
e^d \in \textsc{Dynamic} & & e^s \in \textsc{Static} & \\
e^s & ::= & n \mid e^s + e^s \mid b \mid \mathsf{if}\ e^s\ \mathsf{then}\ e^s\ \mathsf{else}\ e^s & \text{(static)} \\
& \mid & x \mid \lambda x.e^s \mid \lambda x : T.e^s \mid e^s\ e^s \mid e^s :: T & \\
& \mid & \llcorner e^d \lrcorner & \\
e^d & ::= & n \mid e^d + e^d \mid b \mid \mathsf{if}\ e^d\ \mathsf{then}\ e^d\ \mathsf{else}\ e^d & \text{(dynamic)} \\
& \mid & x \mid \lambda x.e^d \mid e^d\ e^d \mid \ulcorner e^s \urcorner &
\end{array}
$$

The HL syntax combines ITSL and DL, allowing each to include the other. Its semantics is defined by combining the two translations to ITGL, yielding a coarse-grained gradual language where each type is either precisely static or completely unknown.

Taking the dynamic language as primary, HL clearly differentiates between the dynamic program $(\lambda x.x\ x)$, the hybrid program $\ulcorner \lambda x.\llcorner x\ x \lrcorner \urcorner$ and the static *non*-program $\ulcorner \lambda x.x\ x \urcorner$.

DL and HL have close ties to research on using types to improve the performance of dynamic languages (Cartwright and Fagan 1991; Rastogi et al. 2012; Swamy et al. 2014). By analogy, the goal in that approach is to transform a DL program into a corresponding HL program by automatically inserting transitions to the static language for as much code as possible while retaining well-typing and behaviour. This problem has been approached using type-based analyses that do infer unknown types.

7. Constraints and Constraint Generation

We have specified an implicit type system, demonstrated its expressiveness, and formalized the type inference problem for it: now we must solve the problem. To do so, we follow the approach of (Wand 1987a): generate constraints for typeability and solve them. Guided by the definition of the ITGL type system and properties of our gradual partial functions and predicates, we define a *constraint typing judgment*, which indicates what constraints must hold for a particular type expression-and-context pair to be typeable.

7.1 Constraint Typing Judgments

We first introduce a set of constraints C that need to be solved during type inference (Fig. 7). There are two kinds of constraints: the standard equality constraints on static types, and new *consistency constraints* on gradual types. Ultimately we want to determine whether a set of constraints can be instantiated affirmatively using some substitution.

Definition 8 (Constraint Satisfaction)**.**

1. $S^{\mathcal{X}} \models U_1^{\mathcal{X}} \sim U_2^{\mathcal{X}}$ *if and only if* $\widehat{S^{\mathcal{X}}}(U_1^{\mathcal{X}}) \sim \widehat{S^{\mathcal{X}}}(U_1^{\mathcal{X}})$*;*
2. $S^{\mathcal{X}} \models T_1^{\mathcal{X}} \doteq T_2^{\mathcal{X}}$ *if and only if* $\widehat{S^{\mathcal{X}}}(T_1^{\mathcal{X}}) = \widehat{S^{\mathcal{X}}}(T_2^{\mathcal{X}})$*;*
3. $S^{\mathcal{X}} \models C^\star$ *if and only if* $S^{\mathcal{X}} \models C$ *for each* $C \in C^\star$*.*

Constraint Typing Fig. 7 presents the constraint typing judgment $\Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X}$. It means that the type expression context $\Gamma^{\mathtt{X}}$ and term expression $t^{\mathtt{X}}$, can be given type expression $U^{\mathtt{X}}$ so long as the constraints $C^\star$ can be satisfied, where $\mathcal{X}$ are any extra variables needed to express the constraints. Since it involves type variables, the constraint typing judgment denotes a problem.

Definition 9 (Constraint Typing Problem)**.** *Given a gradual type expression context* $\Gamma^{\mathcal{X}}$ *and a term expression* $t^{\mathcal{X}}$*, the* constraint typing problem *asks if there is a* $S^{\mathcal{X}}$ *such that* $\Gamma^{\mathcal{X}} \vdash t^{\mathcal{X}} : U^{\mathcal{X}} \mid C^\star \mid \mathcal{X}$ *and* $S^{\mathcal{X}} \models C^\star$*.*

$$\boxed{\Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X}} \quad \bot(\mathcal{X}, \mathrm{Vars}(t^{\mathtt{X}}, \Gamma^{\mathtt{X}}))$$

$$\text{(Cx)}\ \frac{x : U^{\mathtt{X}} \in \Gamma^{\mathtt{X}}}{\Gamma^{\mathtt{X}} \vdash x : U^{\mathtt{X}} \mid \emptyset \mid \emptyset} \qquad \text{(Cn)}\ \frac{}{\Gamma^{\mathtt{X}} \vdash n : \mathsf{Int} \mid \emptyset \mid \emptyset}$$

$$\text{(Cb)}\ \frac{}{\Gamma^{\mathtt{X}} \vdash b : \mathsf{Bool} \mid \emptyset \mid \emptyset}$$

$$\text{(C+)}\ \frac{\begin{array}{c}\Gamma^{\mathtt{X}} \vdash t_1^{\mathtt{X}} : U_1^{\mathtt{X}} \mid C_1^\star \mid \mathcal{X}_1 \qquad \Gamma^{\mathtt{X}} \vdash t_2^{\mathtt{X}} : U_2^{\mathtt{X}} \mid C_2^\star \mid \mathcal{X}_2 \\ \bot(\mathcal{X}_1, \mathcal{X}_2) \qquad C^\star = C_1^\star \cup C_2^\star \qquad \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2\end{array}}{\Gamma^{\mathtt{X}} \vdash t_1^{\mathtt{X}} + t_2^{\mathtt{X}} : \mathsf{Int} \mid C^\star \cup \{\, U_1^{\mathtt{X}} \sim \mathsf{Int}, U_2^{\mathtt{X}} \sim \mathsf{Int} \,\} \mid \mathcal{X}}$$

$$\text{(Cif)}\ \frac{\begin{array}{c}\Gamma^{\mathtt{X}} \vdash t_1^{\mathtt{X}} : U_1^{\mathtt{X}} \mid C_1^\star \mid \mathcal{X}_1 \qquad \Gamma^{\mathtt{X}} \vdash t_2^{\mathtt{X}} : U_2^{\mathtt{X}} \mid C_2^\star \mid \mathcal{X}_2 \\ \Gamma^{\mathtt{X}} \vdash t_3^{\mathtt{X}} : U_3^{\mathtt{X}} \mid C_3^\star \mid \mathcal{X}_3 \qquad U_2^{\mathtt{X}} \sqcap U_3^{\mathtt{X}} \doteq U_4^{\mathtt{X}} \mid C_4^\star \mid \mathcal{X}_4 \\ \bot(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3, \mathcal{X}_4) \qquad \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3 \cup \mathcal{X}_4 \\ C^\star = C_1^\star \cup C_2^\star \cup C_3^\star \cup C_4^\star\end{array}}{\Gamma^{\mathtt{X}} \vdash \mathsf{if}\ t_1^{\mathtt{X}}\ \mathsf{then}\ t_2^{\mathtt{X}}\ \mathsf{else}\ t_3^{\mathtt{X}} : U_4^{\mathtt{X}} \mid C^\star \cup \{\, U_1^{\mathtt{X}} \sim \mathsf{Bool} \,\} \mid \mathcal{X}}$$

$$\text{(Capp)}\ \frac{\begin{array}{c}\Gamma^{\mathtt{X}} \vdash t_1^{\mathtt{X}} : U_1^{\mathtt{X}} \mid C_1^\star \mid \mathcal{X}_1 \qquad \Gamma^{\mathtt{X}} \vdash t_2^{\mathtt{X}} : U_2^{\mathtt{X}} \mid C_2^\star \mid \mathcal{X}_2 \\ \widetilde{cod}(U_1^{\mathtt{X}}) \doteq U_3^{\mathtt{X}} \mid C_3^\star \mid \mathcal{X}_3 \qquad \widetilde{dom}(U_1^{\mathtt{X}}) \sim U_2^{\mathtt{X}} \mid C_4^\star \mid \mathcal{X}_4 \\ \bot(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3, \mathcal{X}_4) \qquad \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3 \cup \mathcal{X}_4 \\ C^\star = C_1^\star \cup C_2^\star \cup C_3^\star \cup C_4^\star\end{array}}{\Gamma^{\mathtt{X}} \vdash t_1^{\mathtt{X}}\ t_2^{\mathtt{X}} : U_3^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}$$

$$\text{(C}\lambda\text{)}\ \frac{\Gamma^{\mathtt{X}}, x : X \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X} \qquad \bot(\mathcal{X}, \{\, X \,\})}{\Gamma^{\mathtt{X}} \vdash (\lambda x.t^{\mathtt{X}}) : X \to U^{\mathtt{X}} \mid C^\star \mid \mathcal{X} \cup \{\, X \,\}}$$

$$\text{(C}\lambda\text{:)}\ \frac{\Gamma^{\mathtt{X}}, x : U_1^{\mathtt{X}} \vdash t^{\mathtt{X}} : U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}{\Gamma^{\mathtt{X}} \vdash (\lambda x : U_1^{\mathtt{X}}.t^{\mathtt{X}}) : U_1^{\mathtt{X}} \to U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}$$

$$\text{(C::)}\ \frac{\Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}{\Gamma^{\mathtt{X}} \vdash (t^{\mathtt{X}} :: U_1^{\mathtt{X}}) : U_1^{\mathtt{X}} \mid C^\star \cup \{\, U^{\mathtt{X}} \sim U_1^{\mathtt{X}} \,\} \mid \mathcal{X}}$$

$$C \in \textsc{Constraint}$$

$$C^\star \in \mathcal{P}(\textsc{Constraint})$$

$$C ::= U^{\mathtt{X}} \sim U^{\mathtt{X}} \mid T^{\mathtt{X}} \doteq T^{\mathtt{X}}$$

Figure 7. Constraint Typing Judgment

$$\boxed{\widetilde{cod}(U_1^{\mathtt{X}}) \doteq U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}} \quad \bot(\mathcal{X}, \mathrm{Vars}(U_1^{\mathtt{X}}))$$

$$\frac{}{\widetilde{cod}(X) \doteq X_2 \mid \{\, X \doteq X_1 \to X_2 \,\} \mid \{\, X_1, X_2 \,\}}$$

$$\frac{}{\widetilde{cod}(U_1^{\mathtt{X}} \to U_2^{\mathtt{X}}) \doteq U_2^{\mathtt{X}} \mid \emptyset \mid \emptyset} \qquad \frac{}{\widetilde{cod}(?) \doteq ? \mid \emptyset \mid \emptyset}$$

Figure 8. Constraint Codomain Judgment

$$\boxed{\widetilde{dom}(U_1^{\mathtt{X}}) \sim U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}} \quad \bot(\mathcal{X}, \mathrm{Vars}(U_1^{\mathtt{X}}, U_2^{\mathtt{X}}))$$

$$\frac{}{\widetilde{dom}(X) \sim U_2^{\mathtt{X}} \mid \{\, X \doteq X_1 \to X_2, X_1 \sim U_2^{\mathtt{X}} \,\} \mid \{\, X_1, X_2 \,\}}$$

$$\frac{}{\widetilde{dom}(U_{11}^{\mathtt{X}} \to U_{12}^{\mathtt{X}}) \sim U_2^{\mathtt{X}} \mid \{\, U_{11}^{\mathtt{X}} \sim U_2^{\mathtt{X}} \,\} \mid \emptyset}$$

$$\frac{}{\widetilde{dom}(?) \sim U_2^{\mathtt{X}} \mid \{\, ? \sim U_2^{\mathtt{X}} \,\} \mid \emptyset}$$

Figure 9. Constraint Domain Consistency Judgment

$$\boxed{U_1^{\mathtt{X}} \sqcap U_2^{\mathtt{X}} \doteq U_3^{\mathtt{X}} \mid C^\star \mid \mathcal{X}} \quad \bot(\mathcal{X}, \mathrm{Vars}(U_1^{\mathtt{X}}, U_2^{\mathtt{X}}))$$

$$\frac{}{\mathsf{Int} \sqcap \mathsf{Int} \doteq \mathsf{Int} \mid \emptyset \mid \emptyset} \qquad \frac{}{\mathsf{Bool} \sqcap \mathsf{Bool} \doteq \mathsf{Bool} \mid \emptyset \mid \emptyset}$$

$$\frac{}{A \sqcap A \doteq A \mid \emptyset \mid \emptyset} \qquad \frac{}{U_1^{\mathtt{X}} \sqcap ? \doteq U_1^{\mathtt{X}} \mid \{\, U_1^{\mathtt{X}} \nsim ? \,\} \mid \emptyset}$$

$$\frac{}{? \sqcap U_2^{\mathtt{X}} \doteq U_2^{\mathtt{X}} \mid \{\, ? \nsim U_2^{\mathtt{X}} \,\} \mid \emptyset}$$

$$\frac{}{X_1 \sqcap U_2^{\mathtt{X}} \doteq X_1 \mid \{\, X_1 \nsim U_2^{\mathtt{X}} \,\} \mid \emptyset}$$

$$\frac{U_1^{\mathtt{X}} \notin \mathrm{TVAR}}{U_1^{\mathtt{X}} \sqcap X_2 \doteq X_2 \mid \{\, U_1^{\mathtt{X}} \nsim X_2 \,\} \mid \emptyset}$$

$$\frac{\begin{array}{ccc} U_{11}^{\mathtt{X}} \sqcap U_{21}^{\mathtt{X}} \doteq U_1^{\mathtt{X}} \mid C_1^\star \mid \mathcal{X}_1 & U_{12}^{\mathtt{X}} \sqcap U_{22}^{\mathtt{X}} \doteq U_2^{\mathtt{X}} \mid C_2^\star \mid \mathcal{X}_2 \\ C^\star = C_1^\star \cup C_2^\star & \bot(\mathcal{X}_1, \mathcal{X}_2) & \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \end{array}}{(U_{11}^{\mathtt{X}} \to U_{12}^{\mathtt{X}}) \sqcap (U_{21}^{\mathtt{X}} \to U_{22}^{\mathtt{X}}) \doteq U_1^{\mathtt{X}} \to U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}$$

Figure 10. Constraint Meet Judgment

Auxiliary Constraint Typing Constraint typing appeals to three auxiliary constraint judgments, which reduce complex conditions to consistency constraints on gradual types and equality constraints on static types. They are presented in Figs. 8,9, and 10.

The $\widetilde{cod}(U_1^{\mathtt{X}}) \doteq U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}$ judgment means that the codomain of gradual type expression $U_1^{\mathtt{X}}$ is $U_2^{\mathtt{X}}$ so long as the constraints $C^\star$ can be satisfied. The $\widetilde{dom}(U_1^{\mathtt{X}}) \nsim U_2^{\mathtt{X}} \mid C^\star \mid \mathcal{X}$ judgment means that the domain of gradual type expression $U_1^{\mathtt{X}}$ is consistent with $U_2^{\mathtt{X}}$ so long as the constraints $C^\star$ can be satisfied. Finally the $U_1^{\mathtt{X}} \sqcap U_2^{\mathtt{X}} \doteq U_3^{\mathtt{X}} \mid C^\star \mid \mathcal{X}$ judgment means that the meet of gradual type expressions $U_1^{\mathtt{X}}$ and $U_2^{\mathtt{X}}$ is $U_3^{\mathtt{X}}$ so long as the constraints $C^\star$ can be satisfied. In each case, $\mathcal{X}$ names any extra type variables needed to express the constraints. The rules for these judgments are based on the properties of the corresponding gradual operations.

These constraint judgments are not defined for all inputs. For instance, the constraint meet judgment is not defined for Int and Bool. Any result would be contrived, so rather than relating these type expressions to an arbitrary variable and unsatisfiable constraint, the judgment simply fails to hold.

7.2 Correctness and Decidability of Constraint Typing

We must prove for each constraint judgment that if a substitution can satisfy the given constraints, then that substitution, and the program type that it produces, solves the type inference problem.

Proposition 6 (Constraint Soundness)**.**
If $\Gamma^{\mathcal{X}} \vdash t^{\mathcal{X}} : U^{\mathcal{X}} \mid C^\star \mid \mathcal{X}$ *and* $S^{\mathcal{X}} \models C^\star$ *then* $\widehat{S^{\mathcal{X}}}(\Gamma^{\mathcal{X}}) \vdash \widehat{S^{\mathcal{X}}}(t^{\mathcal{X}}) : \widehat{S^{\mathcal{X}}}(U^{\mathcal{X}})$.

Proof. By induction on $\Gamma^{\mathtt{X}} \vdash t^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X}$ with lemmas to address auxiliary constraint judgments. □

Proposition 7 (Constraint Completeness)**.**
If $\widehat{S^{\mathcal{X}}}(\Gamma^{\mathcal{X}}) \vdash \widehat{S^{\mathcal{X}}}(t^{\mathcal{X}}) : U$ *then for finite* $\mathcal{X}_0$ *s.t.* $\mathrm{Vars}(\Gamma^{\mathcal{X}}, t^{\mathcal{X}}) \subseteq \mathcal{X}_0$, $\Gamma^{\mathcal{X}} \vdash t^{\mathcal{X}} : U^{\mathcal{X}} \mid C^\star \mid \mathcal{X}$ *where* $\bot(\mathcal{X}_0, \mathcal{X})$*, and furthermore there is an* $S_0^{\mathcal{X}}$ *that agrees with* $S^{\mathcal{X}}$ *except at* $\mathcal{X}$*, where* $S_0^{\mathcal{X}} \models C^\star$ *and* $\widehat{S_0^{\mathcal{X}}}(U^{\mathcal{X}}) = U$.

Proof. By induction on $\widehat{S^{\mathtt{X}}}(\Gamma^{\mathtt{X}}) \vdash \widehat{S^{\mathtt{X}}}(t^{\mathtt{X}}) : U$. with lemmas to address auxiliary constraint judgments. □

The constraint completeness proposition must account for the extra variables $\mathcal{X}$ used by constraint typing, which do not necessarily hold for an arbitrary substitution that solves the type inference problem. The key is that given an arbitrary substitution $S^{\mathtt{X}}$ that solves the type inference problem, it's always possible to massage it into a slightly different $S_0^{\mathtt{X}}$ that produces the same solution but also satisfies the constraints.

8. Gradual Unification

In this section we present a procedure that generalizes unification (Robinson 1965) to account for consistency constraints and relates constraints to solutions, where possible. Since each rule of the judgment is invertible and simplifies the constraints, together they induce a straightforward decision procedure (bottom-up proof search) for gradual constraints.

Proposition 8 (Consistency Inversion)**.**

1. *If* $U_1 \sim U_2$ *then* $U_2 \sim U_1$*;*
2. *If* $T_1 \sim T_2$ *then* $T_1 = T_2$*;*
3. *if* $U_{11} \to U_{12} \sim U_{21} \to U_{22}$*; then* $U_{11} \sim U_{21}$ *and* $U_{12} \sim U_{22}$*;*

Proof. Straightforward. □

Fig. 11 presents the gradual unification judgment $C^\star \ \boldsymbol{\mathcal{U}} \ S^{\mathtt{X}}$, which means that the X-substitution $S^{\mathtt{X}}$ satisfies the constraints in $C^\star$. Recall that satisfaction was defined in Sec. 7.1. Throughout, we assume that $C^\star \cup \{\, C \,\}$ implies $C \notin C^\star$.

Most of the cases are straightforward. The most interesting case is for $X \nsim U_1^{\mathtt{X}} \to U_2^{\mathtt{X}}$. Since X represents a static type, we expand it to $X_1 \to X_2$ and make each component consistent to its corresponding gradual type. Equality would not work in general since the type on the right hand side is gradual. Furthermore, the added consistency checks ensure that any ?s contained within the underlying types are ignored.

We let $S_{\mathcal{A}} : \mathrm{TVAR} \to \mathrm{TPARAM}$ denote an injective map from type variables to type parameters. This function is responsible for transforming any unconstrained type variables into type parameters.

Notation.

1. *The notation* $\mathcal{X}$ $\mathbf{fresh}$ *means that if the consequent of the rule is* $C^\star \ \boldsymbol{\mathcal{U}} \ S^{\mathcal{X}}$*, then* $\bot(\mathcal{X}, \mathrm{Vars}(C^\star))$.
2. *The notation* $[X \mapsto T]$ *stands for* $S_{id}[X \mapsto T]$*, the type substitution that is identity everywhere except for* X*, which it maps to* T.

Proposition 9 (Unification Soundness)**.** *If* $C^\star \ \boldsymbol{\mathcal{U}} \ S^{\mathcal{X}}$ *then* $S^{\mathcal{X}} \models C^\star$.

Proof. By induction on the structure of $C^\star \ \boldsymbol{\mathcal{U}} \ S^{\mathtt{X}}$. □

Proposition 10 (Unification Completeness)**.** *If* $S_1^{\mathcal{X}} \models C^\star$ *then* $C^\star \ \boldsymbol{\mathcal{U}} \ S_2^{\mathcal{X}}$ *for some* $S_2^{\mathcal{X}}$ *and furthermore* $S_1^{\mathcal{X}} = \widehat{S^A} \circ S_2^{\mathcal{X}}$ *for some* S^A.

Proof. By induction on the breakdown of constraint sets $C^\star$ by the unification rules. This is dependent on the decomposition structure being well-founded. □

Proposition 11 (Unification is Decidable)**.** *Backward proof search on* $C^\star \ \boldsymbol{\mathcal{U}} \ S^{\mathcal{X}}$ *driven by* $C^\star$ *is decidable.*

Proof. Consider the relation on constraint sets determined by how each rule (read bottom-up) transforms them. Given soundness and completeness, it suffices to prove that this relation is well-founded.

Unlike other unification relations, the search structure cannot be lexicographically ordered on the number of type variables. Instead,

$\mathcal{U} \in \mathcal{P}(\textsc{Constraint}) \times \textsc{XSubst}$

$$\frac{}{\emptyset\ \mathcal{U}\ S_{\mathcal{A}}}$$

$$\frac{C^\star\ \mathcal{U}\ S^{\mathtt{X}} \qquad U^{\mathtt{X}} \in \{\,\mathsf{Int}, \mathsf{Bool}\,\} \cup \textsc{TVar} \cup \textsc{TParam}}{C^\star \cup \{\,U^{\mathtt{X}} \dot{\sim} U^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{C^\star\ \mathcal{U}\ S^{\mathtt{X}}}{C^\star \cup \{\,? \dot{\sim} U^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}} \qquad \frac{C^\star\ \mathcal{U}\ S^{\mathtt{X}}}{C^\star \cup \{\,U^{\mathtt{X}} \dot{\sim} ?\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{C^\star \cup \{\,U_{11}^{\mathtt{X}} \dot{\sim} U_{21}^{\mathtt{X}}, U_{12}^{\mathtt{X}} \dot{\sim} U_{22}^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}{C^\star \cup \{\,U_{11}^{\mathtt{X}} \to U_{12}^{\mathtt{X}} \dot{\sim} U_{21}^{\mathtt{X}} \to U_{22}^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{C^\star \cup \{\,X \dot{\sim} U^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}} \quad U^{\mathtt{X}} \notin \textsc{TVar}}{C^\star \cup \{\,U^{\mathtt{X}} \dot{\sim} X\,\}\ \mathcal{U}\ S^{\mathtt{X}}} \qquad \frac{C^\star \cup \{\,X \doteq U^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}} \quad U^{\mathtt{X}} \in \{\,\mathsf{Int}, \mathsf{Bool}\,\} \cup \textsc{TVar} \cup \textsc{TParam}}{C^\star \cup \{\,X \dot{\sim} U^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{\{\,X_1, X_2\,\}\ \mathbf{fresh} \qquad X \notin \mathrm{Vars}(U_1^{\mathtt{X}} \to U_2^{\mathtt{X}}) \quad C^\star \cup \{\,X \doteq X_1 \to X_2, X_1 \dot{\sim} U_1^{\mathtt{X}}, X_2 \dot{\sim} U_2^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}{C^\star \cup \{\,X \dot{\sim} U_1^{\mathtt{X}} \to U_2^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{C^\star\ \mathcal{U}\ S^{\mathtt{X}} \qquad T^{\mathtt{X}} \in \{\,\mathsf{Int}, \mathsf{Bool}\,\} \cup \textsc{TVar} \cup \textsc{TParam}}{C^\star \cup \{\,T^{\mathtt{X}} \doteq T^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{C^\star \cup \{\,T_{11}^{\mathtt{X}} \doteq T_{21}^{\mathtt{X}}, T_{12}^{\mathtt{X}} \doteq T_{22}^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}{C^\star \cup \{\,T_{11}^{\mathtt{X}} \to T_{12}^{\mathtt{X}} \doteq T_{21}^{\mathtt{X}} \to T_{22}^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}}}$$

$$\frac{C^\star \cup \{\,X \doteq T^{\mathtt{X}}\,\}\ \mathcal{U}\ S^{\mathtt{X}} \quad T^{\mathtt{X}} \notin \textsc{TVar}}{C^\star \cup \{\,T^{\mathtt{X}} \doteq X\,\}\ \mathcal{U}\ S^{\mathtt{X}}} \qquad \frac{\widehat{[X \mapsto T^{\mathtt{X}}]}(C^\star)\ \mathcal{U}\ S^{\mathtt{X}} \qquad X \notin \mathrm{Vars}(T^{\mathtt{X}})}{C^\star \cup \{\,X \doteq T^{\mathtt{X}}\,\}\ \mathcal{U}\ \widehat{S^{\mathtt{X}}} \circ [X \mapsto T^{\mathtt{X}}]}$$

Figure 11. Unification

we can represent the set of constraints as a directed graph of constraint *paths* and show that a metric on the graph decreases at every step. □

9. Principal Types

We establish our notion of principal types using the standard notion, adapted to account for the separation between type variables and type parameters.

Definition 10 (Solution Pre-order)**.** *Let* (S_1^X, U_1) *and* (S_2^X, U_2) *be solutions for the same gradual type inference problem. Then* $(S_1^X, U_1) \leq (S_2^X, U_2)$ *if and only if* $S_2^X = \widehat{S^A} \circ S_1^X$ *and* $U_2 = \widehat{S^A}(U_1)$ *for some* A*-substitution* S^A*.*

Definition 11 (Principal Solution)**.** (S^X, U) *is a* principal solution *for a gradual type inference problem if and only if it is least among all other solutions to the problem. It directly follows that* U *is a principal type.*

Proposition 12 (Principal Types)**.** *Suppose* $\Gamma^X \vdash t^X : U^X \mid C^\star \mid \mathcal{X}$*. Then* $S_1^X \models C^\star$ *for some* S_1^X*, implies that* $C^\star\ \mathcal{U}\ S_2^X$*, and* $(S_2^X, \widehat{S_2^X}(U^X))$ *is a principal solution for* Γ^X *and* t^X*.*

Proof. Follows from Props. 7 and 10. □

This proposition establishes that our type inference procedure produces the best type possible, which implies that the type system indeed has a well-defined notion of "best" types.

10. Let Polymorphism

Implicitly typed functional programming languages usually support let-polymorphism, introduced by Milner (1978). Siek and Vachharajani (2008) left this as an open problem for gradual type inference. Here we demonstrate that supporting let polymorphism follows the same approach as for a static implicitly typed language (Ohori 1989; Wright 1995; Wright and Felleisen 1994).

To extend ITGL with let-polymorphism, we introduce two new rules:

$$(\text{Uletp})\ \frac{\Gamma \vdash v : U_1 \quad \Gamma \vdash [v/x]t : U_2}{\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ t : U_2}$$

$$(\text{Ulet})\ \frac{t_1 \notin \textsc{Value} \quad \Gamma \vdash (\lambda x.t_2)\ t_1 : U}{\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : U}$$

If the bound expression is a syntactic value, then it is treated polymorphically; otherwise, the value is given a monomorphic type according to the standard let translation.[7] These typing rules yield the corresponding constraint generation rules.

$$(\text{Cletp})\ \frac{\Gamma^{\mathtt{X}} \vdash v^{\mathtt{X}} : U_1^{\mathtt{X}} \mid C_1^\star \mid \mathcal{X}_1 \quad \Gamma^{\mathtt{X}} \vdash [v^{\mathtt{X}}/x]t^{\mathtt{X}} : U_2^{\mathtt{X}} \mid C_2^\star \mid \mathcal{X}_2 \qquad \bot(\mathcal{X}_1, \mathcal{X}_2)}{\Gamma^{\mathtt{X}} \vdash \mathsf{let}\ x = v^{\mathtt{X}}\ \mathsf{in}\ t^{\mathtt{X}} : U_2^{\mathtt{X}} \mid C_1^\star \cup C_2^\star \mid \mathcal{X}_1 \cup \mathcal{X}_2}$$

$$(\text{Clet})\ \frac{t_1^{\mathtt{X}} \notin \textsc{Value} \qquad \Gamma^{\mathtt{X}} \vdash (\lambda x.t_2^{\mathtt{X}})\ t_1^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}{\Gamma^{\mathtt{X}} \vdash \mathsf{let}\ x = t_1^{\mathtt{X}}\ \mathsf{in}\ t_2^{\mathtt{X}} : U^{\mathtt{X}} \mid C^\star \mid \mathcal{X}}$$

The substitution-based presentation of let polymorphism is simple and clear, but impractical for implementations, so we also present an equivalent presentation based on *type schemes*, representatives of type polymorphism in the language. In the next section, we use this type system to endow ITGL with dynamic semantics.

The Schematic ITGL type system is modeled after (Milner 1978). It has the same terms and types as the original, but the typing judgment changes. The key difference is that the type context now maps variables to type schemes $\forall\overline{X}.U^{\mathtt{X}}$ instead of types.

$$\begin{aligned}&\sigma \in \textsc{TypeScheme}\\ &\Gamma^\forall \in \textsc{Var} \rightharpoonup \textsc{TypeScheme}\\ &\sigma ::= \forall\overline{X}.U^{\mathtt{X}}\end{aligned}$$

This allows us to type polymorphic let expressions, without resorting to substitution, by introducing polymorphic assumptions. We can represent a type U as a vacuous type scheme $\forall\emptyset.U$. By convention, all type schemes must be closed. The type system then requires a few changes (Fig. 12).

The ($\sigma\lambda$:) and ($\sigma\lambda$) rules now introduce type schemes into the context when typing function bodies. The (σx) rule magically instantiates a variable's corresponding type scheme to some gradual type. In the substitution-based system, this variable would have been replaced with a full syntactic value, which conveys its polymorphism implicitly. As we'll see, this corresponds directly to the translation from let-polymorphism to parametric polymorphism. Type schemes with no type variables naturally instantiate to their underlying type. The (σletp) rule exploits the polymorphism of type parameters, abstracting some that are unconstrained (i.e. do not appear in the type context) to form the type scheme that is used to type the body. This formulation corresponds directly to the substitution-based specification.

Definition 12. *Let* $\forall : \textsc{GType} \to \textsc{TypeScheme}$ *be* $\forall(U) = \forall\emptyset.U$*; For notational convenience, we write* $\forall(\Gamma) \equiv \forall \circ \Gamma$*.*

[7] Here we assume a call-by-value semantics, but in general that is not necessary.

$$(\sigma\lambda{:})\frac{\Gamma^\forall, x : \forall\emptyset.U_1 \vdash^\forall t : U_2}{\Gamma^\forall \vdash^\forall \lambda x : U_1.t : U_1 \to U_2}$$

$$(\sigma\lambda)\frac{\Gamma^\forall, x : \forall\emptyset.T_1 \vdash^\forall t : U_2}{\Gamma^\forall \vdash^\forall \lambda x.t : T_1 \to U_2} \qquad (\sigma\mathrm{x})\frac{x : \forall\overline{X_i}.U^\mathtt{X} \in \Gamma^\forall}{\Gamma^\forall \vdash^\forall x : [\overline{T_i}/\overline{X_i}]U^\mathtt{X}}$$

$$(\sigma\mathsf{let})\frac{\Gamma^\forall \vdash^\forall (\lambda x.t_2)\ t_1 : U \qquad t_1 \notin \textsc{Value}}{\Gamma^\forall \vdash^\forall \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : U}$$

$$(\sigma\mathsf{letp})\frac{\Gamma^\forall \vdash^\forall v_1 : U_1 \qquad \Gamma^\forall, x : \sigma \vdash^\forall t_2 : U_2 \qquad \sigma = \forall\overline{X_i}.[\overline{X_i}/\overline{A_i}]U_1 \qquad \overline{A_i} \notin \Gamma^\forall}{\Gamma^\forall \vdash^\forall \mathsf{let}\ x = v_1\ \mathsf{in}\ t_2 : U_2}$$

Figure 12. Schematic Typing: Key Rules

$$\begin{array}{rcll} & A \in \textsc{TParam},\quad U \in \textsc{GType},\quad X \in \textsc{TVar},\quad x \in \textsc{Var}, & & \\ & b \in \textsc{Bool},\quad n \in \mathbb{Z},\quad t \in \textsc{Term},\quad v \in \textsc{Value} & & \\ U & ::= & ? \mid X \mid A \mid \mathsf{Int} \mid \mathsf{Bool} \mid U \to U \mid \forall X.U & \text{(types)} \\ t & ::= & n \mid t + t \mid b \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \text{(terms)} \\ & \mid & x \mid \lambda x : U.t \mid t\ t \mid \Lambda X.t \mid t[U] & \\ & \mid & \mathsf{let}\ x = t\ \mathsf{in}\ t \mid \langle U \Leftarrow U \rangle\ t & \\ v & ::= & n \mid b \mid x \mid \lambda x : U.t \mid \Lambda X.v & \text{(syntactic values)} \end{array}$$

Figure 13. Polymorphic Blame Calculus (PBC): Syntax

Proposition 13 (Schematic Substitution)**.** *If* $\Gamma \vdash v : [\overline{A_i}/\overline{X_i}]U^X$ *for* $\overline{A_i} \notin \Gamma$ *and* $\forall(\Gamma), x : \forall\overline{X_i}.U^X \vdash^\forall t : U$ *then* $\Gamma \vdash [v/x]t : U$

Proof. By induction on the structure of t. □

Proposition 14 (Schematic Abstraction)**.** *If* $\forall(\Gamma) \vdash^\forall [v/x]t : U_1$ *and* $\forall(\Gamma) \vdash^\forall v : U_2$ *for some* U_2 *then* $\forall(\Gamma) \vdash^\forall v : [\overline{A_i}/\overline{X_i}]U^X$ *and* $\forall(\Gamma), x : \forall\overline{X_i}.U^X \vdash^\forall t : U_1$ *for some* U^X, $\overline{A_i} \notin \forall(\Gamma)$, *and* $\overline{X_i}$.

Proof. By induction on the structure of t. □

Proposition 15. $\Gamma \vdash t : U$ *if and only if* $\forall(\Gamma) \vdash^\forall t : U$.

Proof. Both directions are by induction on the structure of the corresponding typing judgment. □

11. Dynamic Semantics

Gradually typed languages are defined by type-directed translation to an intermediate language with runtime casts. We do not need to develop a new intermediate cast calculus for ITGL: we can translate it to the Polymorphic Blame Calculus (PBC) (Ahmed et al. 2009, 2011)

Fig. 13 presents the syntax of PBC, adapted to our notation. The language extends the polymorphic lambda calculus (Girard 1972; Reynolds 1974) with support for runtime casts. At runtime, the calculus also uses runtime sealing to encapsulate polymorphically typed values in an effort to enforce representation-independence, even in the face of casts (Matthews and Ahmed 2008). The language supports first-class polymorphism using type-abstracted terms $\Lambda X.t$ and types, $\forall X.U^\mathtt{X}$, which resemble type schemes, but only abstract a single type variable, and can be open.

Fig. 14 presents the type system of the polymorphic blame calculus. For the most part the type system is standard, but it also supports casts $\langle U_2 \Leftarrow U_1 \rangle\ t$. To type casted terms, the polymorphic blame calculus presents a more complex consistency relation, but in the absence of first-class polymorphism, it reduces to $\sim$.

$$(\forall\mathrm{x})\frac{x : U \in \Gamma}{\Omega \mid \Gamma \vdash x : U} \qquad (\forall\mathrm{n})\frac{}{\Omega \mid \Gamma \vdash n : \mathsf{Int}}$$

$$(\forall\mathrm{b})\frac{}{\Omega \mid \Gamma \vdash b : \mathsf{Bool}}$$

$$(\forall\mathrm{app})\frac{\Omega \mid \Gamma \vdash t_1 : U_1 \to U_2 \qquad \Omega \mid \Gamma \vdash t_2 : U_1}{\Omega \mid \Gamma \vdash t_1\ t_2 : U_2}$$

$$(\forall\mathrm{if})\frac{\Omega \mid \Gamma \vdash t_1 : \mathsf{Bool} \qquad \Omega \mid \Gamma \vdash t_2 : U \qquad \Omega \mid \Gamma \vdash t_3 : U}{\Omega \mid \Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : U}$$

$$(\forall{+})\frac{\Omega \mid \Gamma \vdash t_1 : \mathsf{Int} \qquad \Omega \mid \Gamma \vdash t_2 : \mathsf{Int}}{\Omega \mid \Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$(\forall\lambda{:})\frac{\Omega \mid \Gamma, x : U_1 \vdash t : U_2}{\Omega \mid \Gamma \vdash (\lambda x : U_1.t) : U_1 \to U_2}$$

$$(\forall\Lambda)\frac{\Omega, X \mid \Gamma \vdash t : U}{\Omega \mid \Gamma \vdash \Lambda X.t : \forall X.U} \qquad (\forall[])\frac{\Omega \mid \Gamma \vdash t : \forall X.U}{\Omega \mid \Gamma \vdash t[U_0] : [U_0/X]U}$$

$$(\forall\mathsf{let})\frac{\Omega \mid \Gamma \vdash t_1 : U_1 \qquad \Omega \mid \Gamma, x : U_1 \vdash t_2 : U_2}{\Omega \mid \Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : U_2}$$

$$(\forall\Leftarrow)\frac{\Omega \mid \Gamma \vdash t : U_1 \qquad U_1 \sim U_2}{\Omega \mid \Gamma \vdash \langle U_2 \Leftarrow U_1 \rangle\ t : U_2}$$

Figure 14. Polymorphic Blame Calculus (PBC): Type System

Fig. 15 presents a type-directed translation from the Schematic typing judgment for ITGL to PBC. In practice, the type derivation is produced by type inference. Most of the rules are straightforward compositional translations. Type abstractions and applications are introduced where necessary. Several of the rules insert casts, as needed, in positions where consistency is needed. The $\langle\!\langle U \Leftarrow U \rangle\!\rangle t$ function inserts a cast when the type translation is nontrivial.

Definition 13 (Context Translation)**.**

$$\begin{array}{rcl} [\![\epsilon]\!]^X & ::= & \epsilon \\ [\![x : \forall\overline{X}.U^X, \Gamma^\forall]\!]^X & ::= & [\![\Gamma^\forall]\!]^X \\ [\![X, \Gamma^\forall]\!]^X & ::= & X, [\![\Gamma^\forall]\!]^X \\ [\![\epsilon]\!]^x & ::= & \epsilon \\ [\![x : \forall\overline{X}.U^X, \Gamma^\forall]\!]^x & ::= & x : \overline{\forall X}.U^X, [\![\Gamma^\forall]\!]^x \\ [\![X, \Gamma^\forall]\!]^x & ::= & [\![\Gamma^\forall]\!]^x \end{array}$$

Proposition 16 (Well-typed Translation)**.** *If* $\Gamma^\forall \vdash^\forall t \rightsquigarrow t' : U$ *then* $[\![\Gamma^\forall]\!]^X \mid [\![\Gamma^\forall]\!]^x \vdash t' : U$.

Proof. By induction on the derivation of $\Gamma^\forall \vdash^\forall t \rightsquigarrow t' : U$ □

In principle ITGL's semantics can also be defined using the substitutive approach, which essentially monomorphises programs at the expense of code explosion. Taking this approach, the image of the translation is the Blame Calculus (Siek et al. 2009; Wadler and Findler 2009), which is the monomorphic fragment of the PBC.

12. Static and Gradual Polymorphism

This section revisits Siek and Vachharajani's analysis of gradual type inference. Their intuitions about parametric polymorphism led them to design an inference algorithm that does not always yield principal types. Our analysis of that variance uncovers a distinction among type parameters. We extend our type system and inference procedure to expose this distinction, and thereby give firmer foundation to this idea.

$$(\sigma\text{x})\ \frac{x : \forall\overline{X_i}.U^{\mathtt{x}} \in \Gamma^{\forall}}{\Gamma^{\forall} \vdash^{\forall} x \rightsquigarrow x\overline{[T_i]} : \overline{[T_i/X_i]}U^{\mathtt{x}}}$$

$$(\sigma\text{n})\ \frac{}{\Gamma^{\forall} \vdash^{\forall} n \rightsquigarrow n : \mathsf{Int}} \qquad (\sigma\text{b})\ \frac{}{\Gamma^{\forall} \vdash^{\forall} b \rightsquigarrow b : \mathsf{Bool}}$$

$$(\sigma\text{app})\ \frac{\Gamma^{\forall} \vdash^{\forall} t_1 \rightsquigarrow t_1' : U_1 \quad \Gamma^{\forall} \vdash^{\forall} t_2 \rightsquigarrow t_2' : U_2 \quad \widetilde{dom}(U_1) \sim U_2}{\begin{array}{l}\Gamma^{\forall} \vdash^{\forall} t_1\ t_2 \rightsquigarrow \\ \quad (\langle\!\langle \widetilde{dom}(U_1) \rightarrow \widetilde{cod}(U_1) \Leftarrow U_1 \rangle\!\rangle t_1') \\ \quad (\langle\!\langle \widetilde{dom}(U_1) \Leftarrow U_2 \rangle\!\rangle t_2')\end{array} : \widetilde{cod}(U_1)}$$

$$(\sigma\text{if})\ \frac{\begin{array}{c}\Gamma^{\forall} \vdash^{\forall} t_1 \rightsquigarrow t_1' : U_1 \qquad \Gamma^{\forall} \vdash^{\forall} t_1 \rightsquigarrow t_2' : U_2 \\ \Gamma^{\forall} \vdash^{\forall} t_1 \rightsquigarrow t_3' : U_3 \quad U_1 \sim \mathsf{Bool} \quad U = U_2 \sqcap U_3\end{array}}{\begin{array}{l}\Gamma^{\forall} \vdash^{\forall} \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightsquigarrow \\ \quad \mathsf{if}\ \langle\!\langle \mathsf{Bool} \Leftarrow U_1 \rangle\!\rangle t_1' \\ \quad \mathsf{then}\ \langle\!\langle U \Leftarrow U_2 \rangle\!\rangle t_2'\ \mathsf{else}\ \langle\!\langle U \Leftarrow U_3 \rangle\!\rangle t_3' : U\end{array}}$$

$$(\sigma{+})\ \frac{\begin{array}{c}\Gamma^{\forall} \vdash^{\forall} t_1 \rightsquigarrow t_1' : U_1 \qquad \Gamma^{\forall} \vdash^{\forall} t_1 \rightsquigarrow t_2' : U_2 \\ U_1 \sim \mathsf{Int} \qquad\qquad U_2 \sim \mathsf{Int}\end{array}}{\Gamma^{\forall} \vdash^{\forall} t_1 + t_2 \rightsquigarrow (\langle\!\langle \mathsf{Int} \Leftarrow U_1 \rangle\!\rangle t_1') + (\langle\!\langle \mathsf{Int} \Leftarrow U_2 \rangle\!\rangle t_2') : \mathsf{Int}}$$

$$(\sigma\lambda{:})\ \frac{\Gamma^{\forall}, x : \forall\emptyset.U_1 \vdash^{\forall} t \rightsquigarrow t' : U_2}{\Gamma^{\forall} \vdash^{\forall} \lambda x : U_1.t \rightsquigarrow \lambda x : U_1.t' : U_1 \rightarrow U_2}$$

$$(\sigma\lambda)\ \frac{\Gamma^{\forall}, x : \forall\emptyset.T_1 \vdash^{\forall} t \rightsquigarrow t' : U_2}{\Gamma^{\forall} \vdash^{\forall} \lambda x.t \rightsquigarrow \lambda x : T_1.t' : T_1 \rightarrow U_2}$$

$$(\sigma\text{let})\ \frac{\Gamma^{\forall} \vdash^{\forall} (\lambda x.t_2)\ t_1 \rightsquigarrow t' : U \quad t_1 \notin \textsc{Value}}{\Gamma^{\forall} \vdash^{\forall} \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \rightsquigarrow t' : U}$$

$$(\sigma\text{letp})\ \frac{\begin{array}{c}\Gamma^{\forall} \vdash^{\forall} v_1 \rightsquigarrow v_1' : U_1 \quad \Gamma^{\forall}, x : \sigma \vdash^{\forall} t_2 \rightsquigarrow t_2' : U_2 \\ \sigma = \forall\overline{X_i}.\overline{[X_i/A_i]}U_1 \qquad \overline{A_i} \notin \Gamma^{\forall}\end{array}}{\begin{array}{l}\Gamma^{\forall} \vdash^{\forall} \mathsf{let}\ x = v_1\ \mathsf{in}\ t_2 \rightsquigarrow \\ \quad \mathsf{let}\ x = (\overline{\Lambda X_i}.\overline{[X_i/A_i]}v_1')\ \mathsf{in}\ t_2' : U_2\end{array}}$$

$$(\sigma{::})\ \frac{\Gamma^{\forall} \vdash^{\forall} t \rightsquigarrow t' : U \quad U \sim U_1}{\Gamma^{\forall} \vdash^{\forall} (t :: U_1) \rightsquigarrow \langle\!\langle U_1 \Leftarrow U \rangle\!\rangle t' : U_1}$$

$$\langle\!\langle U \Leftarrow U \rangle\!\rangle t = t$$

$$\langle\!\langle U_2 \Leftarrow U_1 \rangle\!\rangle t = \langle U_2 \Leftarrow U_1 \rangle\, t \text{ if } U_1 \neq U_2$$

Figure 15. Gradual Language: Schematic Translation

To motivate their design, Siek and Vachharajani consider how best to answer the following type inference problem:

$$\lambda x : ?.(\lambda y : X.y)\ x$$

Clearly this program is typeable, but what type should X get? Our type inferencer makes it a type parameter, producing the type $? \rightarrow A$, but Siek and Vachharajani's type inference algorithm produces $? \rightarrow ?$. Their reasoning is that the type $? \rightarrow A$ implies parametric polymorphism, and with it the idea that the type can be changed arbitrarily without affecting the program's behavior. However, this simple program gets the value of y from a value of type ?, which, assuming a naïve casting model, could trigger a cast error depending on which type is assigned to X.[8] In their view, type signatures with type parameters should indicate static parametric polymorphism, and this type does not.

This observation suggests a distinction that we currently suppress. Sometimes type parameters indicate parametric polymorphism in the traditional sense: irrelevant to program execution.

[8] Their system precedes PBC and its dynamics had no runtime sealing.

$$\begin{array}{l}B \in \textsc{STParam}, \quad T \in \textsc{Type}, \quad U \in \textsc{GType}, \\ W \in \textsc{Castable}, \quad Y \in \textsc{GCastable} \\ V \in \textsc{StaticType} = \textsc{Type} \setminus \textsc{Castable} \\ Z \in \textsc{GStaticType} = \textsc{GType} \setminus \textsc{GCastable} \\ T ::= A \mid B \mid \mathsf{Int} \mid \mathsf{Bool} \mid T \rightarrow T \\ U ::= ? \mid A \mid B \mid \mathsf{Int} \mid \mathsf{Bool} \mid U \rightarrow U \\ W ::= A \mid \mathsf{Int} \mid \mathsf{Bool} \mid W \rightarrow W \\ Y ::= ? \mid A \mid \mathsf{Int} \mid \mathsf{Bool} \mid Y \rightarrow Y\end{array}$$

Figure 16. Gradual and Static Type Parameters: Syntax

Other times, type parameters indicate that a value's only constraint is that it may be cast to and from ?, which may introduce casts. We distinguish this latter circumstance as an instance of *gradual polymorphism*, the exact circumstance that the Polymorphic Blame Calculus was designed to address.

To clearly delineate this type-level distinction, we extend our languages with another class of type parameters: in addition to the *gradual type parameters* A, we introduce *static type parameters* B (Fig. 16).

In ITSL, these type parameters would all be equivalent. However, when we transition to gradual typing, these types are distinguished from their forebears.

The gradual types extend the static types as usual, thereby creating the universe of types for this language. The type distinction, however, leads to several families of static and gradual types. Castable types W are those static types that can be passed to the dynamic world. Their two salient properties are the absence of the unknown type ? and the static type parameter B, which may not pass to the dynamic world. Indeed there is a gradual counterpart Y. The static and gradual types that do not appear in these sets make up the *static types* V and *gradual static types* Z. These latter two refer to those types that represent pure parametric polymorphism.

In this new model, the unknown type ? no longer represents all possible static types, but only the types in GCastable. In particular, they do not include the *static types*: any type that contains a static type parameter.

The most immediate change that results from this interpretation is that not all types are consistent with ?. In particular, $Z \not\sim ?$ for any $Z \in \textsc{GStaticType}$. In fact, it hereditarily follows that any type containing a static type parameter B is not consistent with ?. The relevant changes to consistency are as follows.

$$\frac{}{B \sim B} \qquad \frac{}{? \sim Y} \qquad \frac{}{Y \sim ?}$$

Static type parameters are consistent to themselves, as is standard, but now ? is consistent to only the *castable* gradual types Y, not all gradual types U.

In ITSL, the consequence is that the language has more type variables, which to date has not been a problem in practice. For instance, the program $\lambda x.x$ may be given type $A \rightarrow A$, using the gradual type parameters, but our earlier program can at best be given the type $B \rightarrow B$ using static type parameters.

Furthermore, this extension changes the nature of our type polymorphism theorem:

Definition 14.

1. *Let* $S^A \in \textsc{ASubst} = \textsc{TParam} \rightarrow \textsc{Castable}$ *denote the gradual type parameter substitutions, or* A*-substitutions.*
2. *Let* $S^B \in \textsc{BSubst} = \textsc{STParam} \rightarrow \textsc{Type}$ *denote the static type parameter substitutions, or* B*-substitutions.*
3. *Let* $S^P \in \textsc{PSubst} =$ $\{\, S^A \cup S^B \mid S^A \in \textsc{ASubst}, S^B \in \textsc{BSubst} \,\}$ *denote the type parameter substitutions, or* P*-substitutions.*

These two substitutions distill the semantics of polymorphism. Gradual type parameters A represent castable types W; conversely, static type parameters B represent all static types, including the gradual type parameters A. In practice, we are interested in substitutions $S^P = S^A \cup S^B$ which simultaneously map gradual and static type parameters to castable types and static types, respectively. These P-substitutions capture our new notion of type polymorphism.

Proposition 17.

1. *If* $\Gamma \vdash t : U$ *then* $\widehat{S^P}(\Gamma) \vdash \widehat{S^P}(t) : \widehat{S^P}(U)$ *for any* $S^P \in \text{PSUBST}$.

Proof. Straightforward induction on derivations $\Gamma \vdash t : U$. □

12.1 Type Inference for Pure Parametric Polymorphism

Now that we have described an implicitly typed language with two tiers of parametric polymorphism, how do we infer these two types? Conveniently, doing so requires only a small extension to our type inference approach. Our type inferencer up to now has simply discarded constraints that require a type to be consistent with ?, since this was true of every type. Now we must consider them, because they make the difference between a static parametric type and a gradual one.

To account for the type distinction, we extend the unification relation to be a ternary relation

$$\boldsymbol{\mathcal{U}} \in \mathcal{P}(\text{GCASTABLEEXP}) \times \mathcal{P}(\text{CONSTRAINT}) \times \text{XSUBST},$$

where $\mathcal{T} \mid C^\star \; \boldsymbol{\mathcal{U}} \; S^{\mathtt{X}}$ means that the constraints $C^\star$ plus $(U^{\mathtt{X}} \dot{\sim} ?)$ for each $U^{\mathtt{X}} \in \mathcal{T}$ are unified by $S^{\mathtt{X}}$. We modify the previous unification relation as follows:

$$\frac{}{\mathcal{T} \mid \emptyset \; \boldsymbol{\mathcal{U}} \; \widehat{S_{\mathcal{B}}} \circ [\overline{\text{Vars}(\mathcal{T}) \mapsto A_i}]} \qquad \frac{\mathcal{T} \cup \{\, U^{\mathtt{X}} \,\} \mid C^\star \; \boldsymbol{\mathcal{U}} \; S^{\mathtt{X}}}{\mathcal{T} \mid C^\star \cup \{\, ? \dot{\sim} U^{\mathtt{X}} \,\} \; \boldsymbol{\mathcal{U}} \; S^{\mathtt{X}}}$$

$$\frac{\mathcal{T} \cup \{\, U^{\mathtt{X}} \,\} \mid C^\star \; \boldsymbol{\mathcal{U}} \; S^{\mathtt{X}}}{\mathcal{T} \mid C^\star \cup \{\, U^{\mathtt{X}} \dot{\sim} ? \,\} \; \boldsymbol{\mathcal{U}} \; S^{\mathtt{X}}}$$

$$\frac{[\widehat{X \mapsto T^{\mathtt{X}}}](\mathcal{T}) \mid [\widehat{X \mapsto T^{\mathtt{X}}}](C^\star) \; \boldsymbol{\mathcal{U}} \; S^{\mathtt{X}} \qquad X \notin \text{Vars}(T^{\mathtt{X}})}{\mathcal{T} \mid C^\star \cup \{\, X \doteq T^{\mathtt{X}} \,\} \; \boldsymbol{\mathcal{U}} \; \widehat{S^{\mathtt{X}}} \circ [X \mapsto T^{\mathtt{X}}]}$$

First, we no longer discard constraints that $U^{\mathtt{X}} \sim ?$ because we must now ensure that $\widehat{S^{\mathtt{X}}}(U^{\mathtt{X}}) \in \text{GCASTABLE}$ to secure the type distinction. Note that $\mathcal{T} \in \text{GCASTABLEEXP}$ imposes implicit side-conditions on each of the rules. Second, to satisfy an empty constraint set, the corresponding substitution $S^{\mathtt{X}}$ must map all type variables in $\mathcal{T}$ to gradual type parameters A_i. The remaining type variables have no constraints, so they can safely be mapped to static type parameters B_i.

With these few extensions, the corresponding notions of soundness and completeness of unification, as well as principal types, hold.

Proposition 18 (Principal Types)**.** *Suppose* $\Gamma^{\mathcal{X}} \vdash t^{\mathcal{X}} : U^{\mathcal{X}} \mid C^\star \mid \mathcal{X}$. *Then if* $S_1^{\mathcal{X}} \models C^\star$ *for some* $S_1^{\mathcal{X}}$*, it follows that* $\emptyset \mid C^\star \; \boldsymbol{\mathcal{U}} \; S_2^{\mathcal{X}}$*, and* $(S_2^{\mathcal{X}}, \widehat{S_2^{\mathcal{X}}}(U^{\mathcal{X}}))$ *is a principal solution for* $\Gamma^{\mathcal{X}}$ *and* $t^{\mathcal{X}}$.

This proposition establishes that our type inference procedure produces the best type possible, and along the way proves that the type system has a well-defined notion of "best" types.

12.2 Dynamic Semantics

This new type distinction suggests two more interpretations of ITGL. These interpretations differ with respect to whether or not they enforce representation-independence at runtime, and conversely with respect to the complexity of the necessary runtime support.

Our language from Sec. 11 corresponds to interpreting both static and gradual type parameters using runtime sealing. A more refined language could use static parametric polymorphism for static type parameters and reserve runtime sealing only for gradual type parameters.

Viewed from this perspective, Siek and Vachharajani interpret all gradual type parameters as ?, leaving only the static type parameters. This language model can be translated to the (monomorphic) Blame Calculus, erasing static type parameters in the standard way. Note, however, that this translation of Bs to ?s implies a syntactic reinterpretation of our programs. In particular, since our type system does not infer gradual types, this reinterpretation corresponds to a refinement process on programs that introduces new type annotations to a program that change some parameters to gradual types. The $\lambda_{\rightarrow}^{?\alpha}$ language interprets these types without incident since gradual types were always inferrable, but at the cost of a complex type system and inferencer.

13. Discussion

Construction of the type system and inference was systematic and driven by underlying principles of the static type system. The structure of ITSL sheds light on the structure of the Simply Typed Gradual Calculus' type system, in particular the origin of the two typing rules for function application.

The constraint typing judgment does not hold for every context–term pair. This property falls out of our decision to reduce all gradual predicates and functions to consistency constraints on gradual types and equality constraints on static types. Alternatively, we could have produced constraints that directly correspond to those operators. Doing so would force us to also support gradual type variables, at least to account for the meet operator when typing if expressions. This design simplifies the constraint typing judgment, at the expense of the unification judgment, which would have to process more constraints, and be mindful of the order in which they are processed. Reducing constraints up-front imposes fewer extensions on the corresponding static unification judgment.

The ITGL type system can be easily shown to conservatively extend the ITSL type system. In particular, the gradual operators all reduce to their static counterparts when applied to static types. In contrast, Siek and Vachharajani (2008) requires explicit reasoning to establish this property.

Distinguishing type parameters from type variables was central to understanding the distinction between static polymorphism and gradual polymorphism, both of which are orthogonal to the type inference problem. This distinction explains Siek and Vachharajani's observation about assigning the unknown type. Such types are not principal according to their type system, but they arise naturally from interpreting gradual type parameters in our system as ?.

The ease with which ITGL extends with let-polymorphism suggests that the language may be amenable to other common features of Hindley-Milner type systems, like row polymorphism (Wand 1987b) and restricted first-class polymorphism (Peyton Jones et al. 2007). We intend to explore some of these more advanced features in the future. Furthermore, we believe that the connection between this system and its underlying static language suggests that it may be reasonable to extend an existing language implementation to support gradual type inference.

Although ITGL can be safely translated to either the Blame Calculus (BC) or the Polymorphic Blame Calculus (PBC), we do not yet understand all the tradeoffs involved. The BC imposes less runtime overhead since it does not require runtime sealing, but the PBC enforces representation-independence. Extending the language with first-class polymorphism, and implementing gradual type inference in a real language, may provide further insight.

Related Work Aside from (Siek and Vachharajani 2008), the most closely related work to ours is Rastogi et al. (2012). It presents an approach to flow-based type inference for a variant of the ActionScript language that combines static and dynamic checking. Although both works involve graduality and inference, the two have essential differences that make them complementary.

First, the two approaches start with different kinds of languages. We begin with an implicitly typed functional language, for which types are typically inferred using unification. Rastogi et al. start from a subtyping-based object-oriented language, for which unification is insufficient to infer types.

Second, the two approaches start from different conceptual foundations. Rastogi et al. begin with a gradually typed language in the style of (Siek and Taha 2006), where missing type annotations are syntactic sugar for ? annotations, and extend it with support for type inference. To do so, they reinterpret missing type annotations as type variables and solve for gradual types using a flow-directed algorithm. We begin with an implicitly typed language and extend it with gradual typing. In doing so, we infer the same static types as the underlying language. It is interesting to observe, however, that Siek and Vachharajani (2008) start from similar foundations to Rastogi et al., but arrive at a type system like ours.

Third, the two approaches have distinct goals. Rastogi et al. are interested in reducing the performance overhead of gradual typing by removing dynamic checks where possible without changing the runtime error behaviour of programs. We are interested in consistency-based reasoning about programs in terms of the underlying pre-existing type system.

Ultimately, the two approaches are complementary. In particular, we believe that techniques similar to Rastogi et al. could be applied to ITGL to reduce runtime overhead. One open problem in this direction is the interaction between flow-based inference and runtime sealing.

14. Conclusion

Dynamic checking is often characterized as the practice of checking at runtime those expressions that lack type annotations. Static checking, on the other hand, is associated with introducing type annotations to achieve more safety. This paper investigates the combination of dynamic and static checking, and finds that the tables reverse in the foundations of gradual type inference: there can be no dynamism without annotation.

15. Acknowledgments

The authors would like to thank Suzanna Crage, Jeremy Siek, Éric Tanter, Sam Tobin-Hochstadt, Amal Ahmed, Bob Harper, and the anonymous reviewers for comments and suggestions.

We thank Khurram A. Jafery and Atsushi Igarashi for contributing to updates and corrections after publication.

References

A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proc. 1st Workshop on Script to Program Evolution*, STOP '09, pages 1–13, New York, NY, USA, 2009. ACM. .

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proc. Symposium on Principles of Programming Languages*, POPL '11, pages 201–214, New York, NY, USA, 2011. ACM. .

G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C$^{\#}$. In T. D'Hondt, editor, *Proc. 24th European Conference on Object-oriented Programming (ECOOP 2010)*, number 6183 in Lecture Notes in Computer Science, pages 76–100, Maribor, Slovenia, June 2010. Springer-Verlag.

R. Cartwright and M. Fagan. Soft typing. In *Proc. Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM.

D. Crystal. *A dictionary of linguistics and phonetics*. Blackwell, 2008.

S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, Sept. 1988.

J.-Y. Girard. *Interprtation fonctionelle et limination des coupures de l'arithmtique d'ordre suprieur.* PhD thesis, Universit Paris VII, 1972.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Proc. Scheme and Functional Programming Workshop*, pages 93–104, 2006.

J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Proc. European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, pages 16–31, Berlin, Heidelberg, 2008. Springer-Verlag.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):12:1–12:44, Apr. 2009.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.

A. Ohori. A simple semantics for ml polymorphism. In *Proc. International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 281–292, New York, NY, USA, 1989. ACM. .

S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, Jan. 2007. .

B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proc. Symposium on Principles of Programming Languages*, pages 481–494, New York, NY, USA, 2012. ACM. .

J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, London, UK, UK, 1974. Springer-Verlag.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965. ISSN 0004-5411. .

I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *Proc. European Symposium on Programming Languages and Systems*, volume 7211 of *ESOP '12*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.

J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proc. European Conference on Object-oriented Programming*, number 4609 in ECOOP '07, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.

J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *Proc. European Symposium on Programming Languages*, ESOP '09, pages 17–31, Berlin, 2009. Springer-Verlag.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proc. Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proc. 2008 Symposium on Dynamic Languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM. .

N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in JavaScript. In *Proc. 41st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2014)*, pages 425–437, San Diego, CA, USA, Jan. 2014. ACM Press.

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. European Symposium on Programming Languages*, ESOP '09, pages 1–16, Berlin, 2009. Springer-Verlag.

M. Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987a.

M. Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987b.

R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *Proc. European Conference on Object-oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.

A. K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4): 343–355, Dec. 1995. ISSN 0892-4635. .

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Journal of Information and Computation*, 115(1):38–94, Nov. 1994.

T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proc. 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, pages 377–388, Madrid, Spain, Jan. 2010. ACM Press.