Abstracting Gradual Typing

Ronald Garcia * Alison M. Clark †

Software Practices Lab
Department of Computer Science
University of British Columbia, Canada
{rxg,amclark1}@cs.ubc.ca

Éric Tanter ‡

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Chile
etanter@dcc.uchile.cl

Abstract

Language researchers and designers have extended a wide variety of type systems to support *gradual typing*, which enables languages to seamlessly combine dynamic and static checking. These efforts consistently demonstrate that designing a satisfactory gradual counterpart to a static type system is challenging, and this challenge only increases with the sophistication of the type system. Gradual type system designers need more formal tools to help them conceptualize, structure, and evaluate their designs.

In this paper, we propose a new formal foundation for gradual typing, drawing on principles from abstract interpretation to give gradual types a semantics in terms of pre-existing static types. Abstracting Gradual Typing (AGT for short) yields a formal account of *consistency*—one of the cornerstones of the gradual typing approach—that subsumes existing notions of consistency, which were developed through intuition and ad hoc reasoning.

Given a syntax-directed static typing judgment, the AGT approach induces a corresponding gradual typing judgment. Then the type safety proof for the underlying static discipline induces a dynamic semantics for gradual programs defined over source-language typing derivations. The AGT approach does not resort to an externally justified cast calculus: instead, run-time checks naturally arise by deducing evidence for consistent judgments during proof reduction.

To illustrate the approach, we develop a novel gradually-typed counterpart for a language with record subtyping. Gradual languages designed with the AGT approach satisfy *by construction* the refined criteria for gradual typing set forth by Siek and colleagues.

Categories and Subject Descriptors D.3.1 [*Software*]: Programming Languages—Formal Definitions and Theory

Keywords gradual typing; abstract interpretation; subtyping

* Partially funded by an NSERC Discovery Grant

† Funded by an NSERC Undergraduate Student Research Award

‡ Partially funded by Fondecyt Project 1150017

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837670

1. Introduction

Software developers and researchers alike see great promise in programming languages that seamlessly combine static and dynamic checking of program properties. One particularly promising and vibrant line of work in this vein is *gradual typing* (Siek and Taha 2006). Gradual typing integrates an *unknown type* ? and a notion of *type consistency* into a pre-existing static type system. These concepts lay the groundwork for designing languages that support fully dynamic checking, fully static checking, and any point on the continuum, while supporting incremental migration in either direction. Programmers using such languages enjoy a number of properties that are central to this language design discipline including complete control over how much checking is done statically versus dynamically.

Researchers have developed gradually typed variants for a substantial breadth of languages, including object-oriented (Siek and Taha 2007; Takikawa et al. 2012), typestates (Garcia et al. 2014; Wolff et al. 2011), ownership types (Sergey and Clarke 2012), security typing (Disney and Flanagan 2011; Fennell and Thiemann 2013), and effects (Bañados Schwerter et al. 2014). Despite these successes, gradual typing faces important challenges. One key challenge is that its foundations are still somewhat murky. The above wide-ranging efforts to adapt gradual typing to new typing disciplines consistently demonstrate that designing a satisfactory gradually typed language becomes more difficult as the type system increases in sophistication.

One consistent challenge that arises during efforts to gradualize new type systems is how to adapt Siek and Taha's notion of *consistency* to new rich typing disciplines. The original work motivates consistency based on intuitions that ring true in the simple cases, but become more difficult to adapt as type systems get more complex. Some of the aforementioned papers develop new notions of consistency that can only be justified by appealing to intuitions about the broader meaning of consistency (Sergey and Clarke 2012; Siek and Taha 2007). Others avoid the issue altogether by focusing on user-inserted casts (Disney and Flanagan 2011; Fennell and Thiemann 2013). Furthermore, developing dynamic semantics for gradually typed languages has typically involved the design of an independent *cast calculus* that is peripherally related to the source language, but its dynamics are also driven mostly by intuition and inspiration from other checking disciplines, *e.g.*, (Findler and Felleisen 2002).

This paper addresses these challenges directly by developing a new foundation for gradual typing. A particularly promising approach to such a foundation is hinted at by Bañados Schwerter et al. (2014) who justify the design of a gradual effect framework by appealing to abstract interpretation (Cousot and Cousot 1977) to give a direct interpretation to unknown effects, and from there derive definitions for requisite notions of consistency. Inspired by that suc-

cess, we adapt this technique from gradual effects to gradual types, and find that it replaces insight-driven developments with routine calculations.

Our contributions are as follows:

- A new foundation for gradual typing that we call the *Abstracting Gradual Typing* (AGT) approach (Sec. 4). Following Bañados Schwerter et al. (2014), this approach derives consistent predicates and gradual typing judgments by using abstract interpretation to give meaning to gradual types in terms of static types. The core idea is to interpret gradual types as sets of static types. Predicates and functions on static types can then be lifted to apply to gradual types. This work adapts their techniques from effects to apply to arbitrary type information. The AGT approach subsumes and generalizes traditional consistency, showing that it is but one of many consistent predicates.
- Bañados Schwerter et al. (2014) develop dynamic semantics in the usual fashion: by devising a cast calculus using ad hoc techniques. AGT includes a systematic approach to developing dynamic semantics as proof reductions over source language typing derivations (Sec. 6).
- We use AGT to develop a novel gradually typed language that features width and depth record subtyping (Sec. 5), conditional expressions, and *gradual rows*, a new kind of gradual record type reminiscent of row polymorphism (Rémy 1989).

A gradual language built with the AGT approach fulfills, *by construction*, the design goals of gradual typing and satisfies the refined criteria for gradual typing formulated by Siek et al. (2015b) (Secs. 4.6, 6.7).

2. Background on Gradual Typing

Gradual typing is in essence about plausible reasoning in the presence of unknown type information. The key external feature of a gradual type system is the addition of the *unknown type*, noted ?. The goals of gradual typing are summarized as follows: *(a)* A gradually-typed language supports both fully static and fully dynamic checking of program properties, as well as the continuum between these two extremes. *(b)* The programmer has fine grained control over the static-to-dynamic spectrum, and can manually evolve a program by introducing more or less precise types (*i.e.*, less or more unknown type information) as needed. *(c)* A gradual type system statically rejects implausible programs. Programs that are accepted based on plausibility (due to imprecise types) are checked at runtime to safeguard static assumptions. In short, the underlying type discipline is preserved, but relegated to dynamic checks in some cases.

Consistency. Central to gradual typing is the notion of *consistency* between gradual types, which are types that may involve the unknown type ? (Anderson and Drossopoulou 2003; Siek and Taha 2006). The consistency relation weakens type equality to account for unknown positions in a gradual type. It is defined as follows:

$$\frac{}{T \sim T} \qquad \frac{}{T \sim ?} \qquad \frac{}{? \sim T}$$

$$\frac{T_{11} \sim T_{21} \quad T_{12} \sim T_{22}}{T_{11} \to T_{12} \sim T_{21} \to T_{22}}$$

Consistency conservatively extends static type equality. For instance, $\mathsf{Int} \to \mathsf{Bool} \sim \mathsf{Int} \to \mathsf{Bool}$ and $\mathsf{Int} \to \mathsf{Bool} \nsim \mathsf{Int} \to \mathsf{Int}$, but in the presence of the unknown type it is more permissive, for instance, $\mathsf{Int} \to ? \sim ? \to \mathsf{Bool}$.

A simple typing judgment can be lifted to a gradual typing judgment by replacing static types with gradual types and replacing static type equality with type consistency. The resulting type system accepts and rejects the same statically typed programs, but accepts many more gradually typed programs.

Dynamic checking. Since a gradual type system accepts programs with unequal types, due to unknown type information, such programs must be checked at runtime to ensure that those cases do not compromise type safety.

To address this, the dynamic semantics of the gradual language is defined by type-directed translation to a separate internal language called a *cast calculus*. A cast calculus has static judgments over gradual types and cast operations that evolve during program execution. The translation process introduces runtime casts whenever type consistency holds but type equality is not assured. Therefore cast calculi, of which many varieties exist (*e.g.*, Siek and Wadler (2010); Siek et al. (2009)), together with the cast insertion translation, are normally seen as essential to gradual typing.

3. Gradual Growing Pains?

The unknown type, consistency, and a cast-based intermediate language are elegant and effective notions for the development of gradual typing over simple type systems. However, as this concept and its techniques have been applied to more sophisticated type disciplines, the effort involved in adapting these notions, especially consistency, has increased.

What exactly is consistency? The first challenge to adapting gradual typing involves the process of developing a suitable gradual type system. The resulting type system must *conservatively extend* the corresponding static type system: any program that the static type system accepts (or rejects) must also be accepted (or rejected) by the gradual type system. This goal can be elusive.

Part of the challenge can be attributed to the current understanding of consistency. Consistency was originally pitched with a post-hoc justification: what relation over types could possibly allow the specification of a type system that fulfills the requirements? In particular, that consistency must not be transitive is justified by the observation that if it were, then any two types would be consistent with each other, which would prevent the type system from rejecting programs statically (Siek and Taha 2006).

Our understanding of consistency has changed over time, with no conception proving definitive. Consistency was originally conceived as analogous to consistency among partial functions (Siek and Taha 2006). Later, in support of languages with subtyping, it was conceived in terms of a restriction operator $T_1|_{T_2}$ that masks off the parts of a type T_1 that are unknown in a type T_2. Under this regime, two types are consistent if $T_1|_{T_2} = T_2|_{T_1}$ (Siek and Taha 2007). From this view, a notion of *consistent subtyping* is devised by updating the masking operator in ad hoc ways and replacing equality $=$ with subtyping $<:$. In addition, the paper presents several extensionally equivalent formulations of consistent subtyping, none of which intuitively seems primary. In the end, consistent subtyping is viewed as merely an implementation device: the language is designed by extending a gradual language with subsumption and then seeking a viable implementation. Later, in support of unification-based type inference, consistency is connected to a partial order on gradual types, such that the partial order and consistency itself can be defined in terms of one another (Siek and Vachharajani 2008). This observation raises the question: which conception is primary, consistency or ordering? The resulting gradual language with inference is devised by adding unification to the language, observing that the resulting language accepts too many statically typed programs, and then imposing an ingenious restriction mechanism on the gradual type judgment. Garcia and Cimini (2015) propose an alternative but equivalent approach that involves designing a new gradual language on top of a pre-existing inference-based static language.

Each of the above systems achieves the goals of gradual typing, but at the expense of much cleverness, engineering, and renegotiation of the nature of consistency, which seems more elusive as type systems get more sophisticated. If gradual typing is to grow, then its foundations need to become stable and broadly applicable.

Dynamic semantics. The static semantics of gradual typing exhibits challenges, but even so, "It turns out that the dynamic semantics of gradually-typed languages is more complex than the static semantics..." (Siek and Garcia 2012).

One of the key challenges lies in the fact that the cast calculi used to define the dynamic semantics of gradual languages are designed wholly separate from the source gradual language. A cast calculus has its own type system, its own operational semantics, and its own type safety argument. The tie that binds these two languages together is the type-directed translation from the source language to the cast calculus and a proof that well-typed source programs translate to well-typed cast calculus programs. Bear in mind, though, that the cast calculus typically admits far more programs than those that lie in the image of the translation procedure. Thus the correctness of the cast calculus with respect to the intentions of the source language is typically argued based on intuition and comparison to other pre-existing cast calculus designs.

For this reason, many cast calculi have arisen in the literature, and in some cases a number of them can serve as the target of type-directed translation for the same source language (Siek et al. 2009). How are we to judge that any given dynamic semantics is in some way correct with respect to the source language?

Criteria for gradual typing. In an effort to address some questions about the nature of gradual typing, Siek et al. (2015b) propose a set of refined criteria for what makes a language gradually typed. These criteria constrain both the static and dynamic properties of gradual languages. They are conveyed both informally, with human justification, as well as in the form of formal proof obligations. Indeed a number of languages are analyzed with respect to these criteria and some are found wanting. Such criteria are a useful and pragmatic means of identifying core concepts, but foundational grounding would endow them with more prescriptive force.

Abstracting gradual typing. This paper addresses fundamental challenges to the gradual typing programme by identifying underlying principles for gradual typing, applicable independently of what is being gradualized. Once a formal meaning is given to unknown type information, all the definitions and properties naturally follow by construction, without appeal to intuition.

Inspired by the use of abstract interpretation to design static semantics for gradual effects (Bañados Schwerter et al. 2014), we devise a general approach based on abstract interpretation that subsumes and generalizes much previous work on gradual typing. We call this approach *Abstracting Gradual Typing* (AGT). Given a statically typed programming language, and an interpretation for gradual types couched in terms of that static discipline, the AGT approach enables us to systematically derive static and dynamic semantics for a language that satisfy, *by construction*, all the criteria for gradual typing proposed by Siek et al. (2015b).

4. Abstracting Static Semantics

This section introduces the AGT approach to developing static semantics for a gradually typed language. Along the way it introduces the necessary concepts from abstract interpretation.

4.1 A Language and its Static Typing Discipline

We start with a simply-typed functional language, called STFL, whose syntax and static type system are given in Fig. 1. The presentation follows the style of Garcia and Cimini (2015), where the

$$T \in \textsc{Type}, \quad x \in \textsc{Var}, \quad b \in \textsc{Bool},$$
$$n \in \mathbb{Z}, \quad t \in \textsc{Term}, \quad \Gamma \in \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$$

$$\begin{array}{rcll} T & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid T \to T & \text{(types)} \\ t & ::= & n \mid b \mid x \mid \lambda x : T.t \mid t\ t \mid t + t & \\ & \mid & \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t :: T & \text{(terms)} \end{array}$$

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

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

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

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

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

$dom : \textsc{Type} \rightharpoonup \textsc{Type}$
$dom(T_1 \to T_2) = T_1$
$dom(T)$ 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 1. STFL: Syntax and Type System

types of subterms are kept opaque, relationships among types are expressed as relations, and some result types are expressed using partial functions. Note that there is nothing surprising about the static type system: it is equivalent to a typical presentation (Pierce 2002). The dom and cod functions abstract the acquisition of function type information; the $equate$ function "equates" the types of the two conditional branches, ensuring that the typing rule is only defined when T_2 and T_3 are in fact the same type.

Of particular interest is the presence of an ascription expression $t :: T$, which lets a program assert a static type for a particular expression. In STFL, the ascription acts as no more than machine-checked documentation, since the ascribed type must be exactly the synthesized type of the underlying term.

As we see below, opaque types, partial type functions, and type ascription serve as key hooks for abstracting a static type system to form its gradual counterpart. Furthermore, ascription is critical to defining the dynamic semantics of the gradual language (Sec. 6.6).

4.2 Defining the Unknown

As is standard, we define the syntax of gradual types by extending static types with a notion of *unknown type* ?.

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

We naturally lift terms from the static language to gradual terms $\widetilde{t} \in \text{GTERM}$, *i.e.*, terms with gradual types:

$$\widetilde{t} ::= \ldots \mid \lambda x : \widetilde{T}.\widetilde{t} \mid \widetilde{t} :: \widetilde{T} \mid \ldots$$

Note that a type T is both a TYPE and GTYPE by inclusion. Similarly, a term t is both a TERM and GTERM.

Consider the fully unknown gradual type ?. It is completely unknown: we do not know what type it represents; in other words, it represents *any type*. Alternatively, the type $\mathsf{Int} \to$? represents

a function from Int to some unknown type. In other words, this could be any type of function as long as it maps from integers to some other type. Finally, the gradual type Int by itself represents only the static type Int.

These observations have been used before to intuitively justify the definition of consistency; here we use it to give a direct interpretation to gradual types in terms of static types. Formally, we define a *concretization* γ from gradual types $\widetilde{T}$ to sets of static types $\widehat{T}$:[a]

Definition 1 (Concretization)**.** *Let* $\gamma : \text{GTYPE} \to \mathcal{P}(\text{TYPE})$ *be defined as follows:*

$$\gamma(\mathsf{Int}) = \{\,\mathsf{Int}\,\} \qquad \gamma(\mathsf{Bool}) = \{\,\mathsf{Bool}\,\}$$
$$\gamma(\widetilde{T}_1 \to \widetilde{T}_2) = \gamma(\widetilde{T}_1) \mathbin{\widehat{\to}} \gamma(\widetilde{T}_2) \qquad \gamma(?) = \text{TYPE}$$

The definition uses an operator that lifts the function type constructor to apply to sets of static types.

$$\cdot \mathbin{\widehat{\to}} \cdot : \mathcal{P}(\text{TYPE}) \times \mathcal{P}(\text{TYPE}) \to \mathcal{P}(\text{TYPE})$$
$$\widehat{T}_1 \mathbin{\widehat{\to}} \widehat{T}_2 = \{\, T_1 \to T_2 \mid T_1 \in \widehat{T}_1, T_2 \in \widehat{T}_2 \,\}$$

This flavour of interpretation of gradual types is the primary driver of the AGT approach.

Returning to the examples, we find that $\gamma(?) = \text{TYPE}$, $\gamma(\mathsf{Int} \to ?) = \{\,\mathsf{Int} \to T \mid T \in \text{TYPE}\,\}$, and $\gamma(\mathsf{Int}) = \{\,\mathsf{Int}\,\}$, each of which directly embodies our intuitions.

Design space of unknown type information. The definition of concretization captures an important design choice in gradual type systems, namely the scope of unknown type information. Following the typical exposition of gradual typing, we defined ? to represent any type by defining $\gamma(?) = \text{TYPE}$. We could instead restrict ? to represent selected types, for instance restricting it to base types by defining $\gamma(?) = \{\,\mathsf{Int}, \mathsf{Bool}\,\}$ instead. This would yield a gradually typed language that statically guarantees that only expressions with function type can be applied, which in turn guarantees that at runtime the resulting operator will indeed be a function. In a language with subtyping, we could introduce *bounded* unknown types $?_T$, by defining $\gamma(?_T) = \{\, T' \in \text{TYPE} \mid T' <: T \,\}$, which is similar to the work of Ina and Igarashi (2011) on gradual generics.

AGT can also be used to introduce unknown information selectively on parts of types. For instance, Bañados Schwerter et al. (2014) use an *unknown effect* ¿ to inject imprecision on effect annotations in types. The novel gradual rows presented in Sec. 5.4 are another example of selective unknown type information.

Precision. Consider again the interpretation of gradual types set forth by Defn. 1. One straightforward but important property of the interpretation is that $\gamma(T) = \{\,T\,\}$ for any static type T, so any gradual type that is also a static type represents only itself. At the other end of the scale, the unknown type ? represents all static types. In this sense, the fully static type denotes more precise information about the identity of a type than its fully unknown counterpart, since it rules out certain types. This intuition motivates a formal notion of *precision* between gradual types.

Definition 2 (Type Precision)**.** $\widetilde{T}_1$ *is less imprecise (*i.e.*, more precise) than* $\widetilde{T}_2$*, notation* $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$*, if and only if* $\gamma(\widetilde{T}_1) \subseteq \gamma(\widetilde{T}_2)$.

This definition of type precision *exactly* coincides with the naïve subtyping relation $<:_n$ of Wadler and Findler (2009), which indicates that one type is less unknown than another:[b]

$$\frac{}{\widetilde{T} <:_n ?} \qquad \frac{}{\widetilde{T} <:_n \widetilde{T}} \qquad \frac{\widetilde{T}_{11} <:_n \widetilde{T}_{21} \quad \widetilde{T}_{12} <:_n \widetilde{T}_{22}}{\widetilde{T}_{11} \to \widetilde{T}_{12} <:_n \widetilde{T}_{21} \to \widetilde{T}_{22}}$$

Proposition 1. $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$ *if and only if* $\widetilde{T}_1 <:_n \widetilde{T}_2$.

Precision arises naturally from the meaning of gradual types.

4.3 Lifting Predicates to Gradual Types

Having conceptually reconstructed gradual type precision, we now turn to the notion of consistency, which has been central to the gradual typing approach. Recall that consistency in particular means that two gradual types could *plausibly* be identical, or *equal*, types. Just as we did with precision, we recast consistency in terms of our compositional interpretation of gradual types. Instead of focusing on consistent equality in particular, though, we recast consistency as a general notion that applies to any relationship among types.

Type equality $=$ is a particular binary relation between types. Consider instead some arbitrary predicate $P(T_1, T_2, \ldots, T_n)$ on types. We are interested in using this predicate to induce a new *consistent predicate* $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2, \ldots, \widetilde{T}_n)$ on gradual types, in a manner that formalizes the intuitions underlying gradual typing. The consistent predicate $\widetilde{P}$ should hold when the gradual types in question could plausibly be types for which P holds. To formalize this optimistic notion of consistency, we define the *consistent lifting* of a type predicate in terms of the meanings of gradual types.

Definition 3 (Predicate Lifting)**.** *Let* $P \subseteq \text{TYPE}^2$ *be some binary predicate on types.*

1. *We define its* (consistent) collecting lifting $\widehat{P} \subseteq \mathcal{P}(\text{TYPE})^2$ *as follows:*
   $\widehat{P}(\widehat{T}_1, \widehat{T}_2)$ *iff* $P(T_1, T_2)$ *for* some $\langle T_1, T_2 \rangle \in \widehat{T}_1 \times \widehat{T}_2$.
2. *We then define its* (consistent) gradual lifting $\widetilde{P} \subseteq \text{GTYPE}^2$ *as follows:* $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$ *iff* $\widehat{P}(\gamma(\widetilde{T}_1), \gamma(\widetilde{T}_2))$.

We generalize these to finite arity predicates $P(T_1, T_2, \ldots, T_n)$.

The key insight of this definition is that gradual types are consistent with the predicate if *some* static types in their interpretation satisfy the static predicate. We take this existential lifting as the definition of generalized consistency. As type systems become more sophisticated and use properties beyond type equality, this general notion becomes crucial for developing their gradual counterparts.

Consistency revealed. Consider the simple predicate of type equality $T = T$. We lift this predicate to form *consistent equality* on gradual types $\widetilde{T} \mathrel{\widetilde{=}} \widetilde{T}$. This predicate is not new whatsoever.

Proposition 2. $\widetilde{T}_1 \mathrel{\widetilde{=}} \widetilde{T}_2$ *if and only if* $\widetilde{T}_1 \sim \widetilde{T}_2$.

Thus the consistency relation of Siek and Taha arises naturally as the consistent lifting of type equality. From now on we use the symbol $\sim$ but refer to it as *consistent equality* since it is but one consistent relation on gradual types.

4.4 Lifting Functions on Gradual Types

Just as we have lifted predicates on types to consistent predicates on gradual types, we must also lift our partial functions on static types to corresponding partial functions on gradual types. We begin by defining a collecting semantics for functions.

Definition 4 (Collecting Function)**.** *Let* $F : \text{TYPE}^2 \rightharpoonup \text{TYPE}$ *be some partial function on static types. Then we define its* collecting

[a] Throughout the paper, we use wide parens $\widehat{x}$ (resp. wide tildes $\widetilde{x}$) on collecting (resp. consistent) metavariables, predicates and functions.

[b] Wadler and Findler called this relation "naïve subtyping" because the formal rules coincide with an old incorrect specification of substitutability for higher-order types. We use the name "precision" because it accurately reflects the conceptual meaning of the judgment, which is not concerned with sound substitutability, and so not about subtyping.

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

$$(\widetilde{T}\text{app})\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \quad \widetilde{T}_2 \sim \widetilde{dom}(\widetilde{T}_1)}{\Gamma \vdash \widetilde{t}_1\ \widetilde{t}_2 : \widetilde{cod}(\widetilde{T}_1)}$$

$$(\widetilde{T}\text{+})\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \quad \widetilde{T}_1 \sim \mathsf{Int} \quad \widetilde{T}_2 \sim \mathsf{Int}}{\Gamma \vdash \widetilde{t}_1 + \widetilde{t}_2 : \mathsf{Int}}$$

$$(\widetilde{T}\text{if})\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \quad \Gamma \vdash \widetilde{t}_3 : \widetilde{T}_3 \quad \widetilde{T}_1 \sim \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ \widetilde{t}_1\ \mathsf{then}\ \widetilde{t}_2\ \mathsf{else}\ \widetilde{t}_3 : \widetilde{T}_2 \sqcap \widetilde{T}_3}$$

$$(\widetilde{T}\lambda)\frac{\Gamma, x : \widetilde{T}_1 \vdash \widetilde{t} : \widetilde{T}_2}{\Gamma \vdash (\lambda x : \widetilde{T}_1.\widetilde{t}) : \widetilde{T}_1 \to \widetilde{T}_2} \qquad (\widetilde{T}::)\frac{\Gamma \vdash \widetilde{t} : \widetilde{T} \quad \widetilde{T} \sim \widetilde{T}_1}{\Gamma \vdash (\widetilde{t} :: \widetilde{T}_1) : \widetilde{T}_1}$$

$\widetilde{dom} : \text{GTYPE} \rightharpoonup \text{GTYPE}$	$\widetilde{cod} : \text{GTYPE} \rightharpoonup \text{GTYPE}$
$\widetilde{dom}(\widetilde{T}_1 \to \widetilde{T}_2) = \widetilde{T}_1$	$\widetilde{cod}(\widetilde{T}_1 \to \widetilde{T}_2) = \widetilde{T}_2$
$\widetilde{dom}(?) = ?$	$\widetilde{cod}(?) = ?$
$\widetilde{dom}(\widetilde{T})$ undefined otherwise	$\widetilde{cod}(\widetilde{T})$ undefined otherwise

Figure 2. GTFL: Type System

lifting $\widehat{F} : \mathcal{P}(\text{TYPE})^2 \to \mathcal{P}(\text{TYPE})$ *as follows:*

$$\widehat{F}(\widehat{T}_1, \widehat{T}_2) = \{\, F(T_1, T_2) \mid \langle T_1, T_2 \rangle \in \widehat{T}_1 \times \widehat{T}_2 \,\}.$$

We generalize to finite arity functions $F : \text{TYPE}^n \rightharpoonup \text{TYPE}$.

We apply the original partial function pointwise to the collection of corresponding static types, ignoring cases where the function is not defined, to produce the set of possible results. If the function is undefined for the entire collection, then the result is the empty set.

As a first example, the collecting lifting of the codomain function *cod* is defined as $\widehat{cod}(\widehat{T}) = \{\, T_2 \mid T_1 \to T_2 \in \widehat{T} \,\}$. This means that, for instance, $\widehat{cod}(\text{TYPE}) = \text{TYPE}$ since every type appears in the codomain of some type.

From here we wish to produce the corresponding function for gradual types. We should compose the concretization function with the collecting semantics, *i.e.*, $\widehat{F} \circ \gamma$, but the result of this is some arbitrary collection of static types. How do we get from here back to gradual types? For this, we define an *abstraction function*, which represents the collection of static types as precisely as it can in terms of a gradual type.

Definition 5 (Abstraction)**.** *Let the abstraction (partial) function* $\alpha : \mathcal{P}(\text{TYPE}) \rightharpoonup \text{GTYPE}$ *be defined as follows:*

$$\begin{aligned}
\alpha(\{\,\mathsf{Int}\,\}) &= \mathsf{Int} \\
\alpha(\{\,\mathsf{Bool}\,\}) &= \mathsf{Bool} \\
\alpha(\{\,\overline{T_{i1} \to T_{i2}}\,\}) &= \alpha(\{\,\overline{T_{i1}}\,\}) \to \alpha(\{\,\overline{T_{i2}}\,\}) \\
\alpha(\emptyset) &\textit{ is undefined} \\
\alpha(\widehat{T}) &= ? \textit{ otherwise}
\end{aligned}$$

Not just any function from sets of static types to gradual types will do; the abstraction function satisfies two correctness criteria.

Proposition 3 (α is Sound)**.** *If* $\widehat{T}$ *is not empty, then* $\widehat{T} \subseteq \gamma(\alpha(\widehat{T}))$.

Soundness ensures that α retains all of the types that are represented by the collection $\widehat{T}$. A constant function that maps every collection $\widehat{T}$ to ? meets this criterion, but it is not satisfying because it unnecessarily loses precision: it forgets everything we knew about $\widehat{T}$. At the least we want $\alpha(\gamma(\widetilde{T})) = \widetilde{T}$ for every gradual type $\widetilde{T}$. The function α achieves this and more.

Proposition 4 (α is Optimal)**.** *If* $\widehat{T}$ *is not empty and* $\widehat{T} \subseteq \gamma(\widetilde{T})$ *then* $\alpha(\widehat{T}) \sqsubseteq \widetilde{T}$.

Optimality ensures that α always retains as much precision as possible, given the gradual types that we have available and their meaning via γ.[c] In fact, the optimal α is uniquely determined by γ.

Using the abstraction function, we specify the gradual lifting of partial functions on types.[d]

Definition 6 (Gradual Function)**.** *Let* $F : \text{TYPE}^2 \rightharpoonup \text{TYPE}$ *be some partial function on static types. Then we define its* gradual lifting $\widetilde{F} : \text{GTYPE}^2 \rightharpoonup \text{GTYPE}$ *as follows:*

$$\widetilde{F}(\widetilde{T}_1, \widetilde{T}_2) = \alpha(\widehat{F}(\gamma(\widetilde{T}_1), \gamma(\widetilde{T}_2))).$$

We generalize to finite arity partial functions.

Lifting domains and codomains. As a first example, we use the collecting codomain function to show that

$$\widetilde{cod}(?) = \alpha(\widehat{cod}(\gamma(?))) = \alpha(\widehat{cod}(\text{TYPE})) = \alpha(\text{TYPE}) = ?$$

Also, just like its static counterpart, $\widetilde{cod}$ is partial:

$$\widetilde{cod}(\mathsf{Int}) = \alpha(\widehat{cod}(\gamma(\mathsf{Int}))) = \alpha(\widehat{cod}(\{\,\mathsf{Int}\,\})) = \alpha(\emptyset), \text{undefined}$$

More generally, *cod* and $\widetilde{cod}$ coincide on static types.

Proposition 5.

1. $\widetilde{dom}(T) = \widetilde{T}$ *iff* $T = T_1 \to T_2$ *and* $\widetilde{T} = T_1$ *for some* T_1, T_2.
2. $\widetilde{cod}(T) = \widetilde{T}$ *iff* $T = T_1 \to T_2$ *and* $\widetilde{T} = T_2$ *for some* T_1, T_2.

Furthermore, $\widetilde{dom}(\widetilde{T})$ and $\widetilde{cod}(\widetilde{T})$ are defined if and only if $\widetilde{T} \sim (\widetilde{dom}(\widetilde{T}) \to \widetilde{cod}(\widetilde{T}))$, which is what we expect from a gradual type that can be used as a function.

Lifting *equate***.** The *equate* partial function forces static types to be equal, but how shall we "equate" imprecise gradual types? Garcia and Cimini (2015) informally argue that $\widetilde{equate}$ should be the meet operator $\sqcap$ on gradual types:

$$\widetilde{T}_1 \sqcap \widetilde{T}_2 = \alpha(\gamma(\widetilde{T}_1) \cap \gamma(\widetilde{T}_2))$$

Using the AGT approach, we replace this intuition with a proof:

Proposition 6. $\widetilde{equate}(\widetilde{T}_1, \widetilde{T}_2) = \widetilde{T}_1 \sqcap \widetilde{T}_2$.

Note that $\widetilde{T}_1 \sqcap \widetilde{T}_2$ is defined if and only if $\widetilde{T}_1 \sim \widetilde{T}_2$. Thus, our use of $\sqcap$ captures the expected relationship between the types of the branches: they must be consistently equal. The above α-γ-based definition of meet would be hard to implement since it involves generating infinite sets of types. However, we can calculate a decidable characterization of meet by cases on pairs of gradual types. The result is a subset of the meet of Siek and Wadler (2010): their meet exploits an inconsistent $\bot$ gradual type to lazily propagate failures, while our meet fails eagerly because α is undefined for the empty set.

Proposition 7.

1. $\mathsf{Int} \sqcap \mathsf{Int} = \mathsf{Int}$;
2. $\mathsf{Bool} \sqcap \mathsf{Bool} = \mathsf{Bool}$;

[c] In abstract interpretation terminology, soundness and optimality together are called a *Galois connection*.

[d] The gradual partial function could be made total by introducing a bottom gradual type $\bot$ which abstracts $\emptyset$; we leave this type undefined to preserve the partiality intuitions from the static typing world.

3. $\widetilde{T} \sqcap ? = ? \sqcap \widetilde{T} = \widetilde{T}$;
4. $(\widetilde{T}_{11} \to \widetilde{T}_{12}) \sqcap (\widetilde{T}_{21} \to \widetilde{T}_{22}) = (\widetilde{T}_{11} \sqcap \widetilde{T}_{21}) \to (\widetilde{T}_{12} \sqcap \widetilde{T}_{22})$;
5. $\widetilde{T}_1 \sqcap \widetilde{T}_2$ *is undefined otherwise.*

The AGT approach often uses calculational reasoning to produce inductive definitions for "semantically" defined concepts.

4.5 Gradual Type System

Fig. 2 presents the type system of GTFL, the gradual counterpart to STFL. It builds directly on the STFL type system. The rules have the same structure as Fig. 1, except that static types have been lifted to gradual types, and predicates and partial functions have been lifted to consistent predicates and gradual partial functions.

Compositional lifting. The ($\widetilde{T}$app) rule presents a subtlety: the gradual lifting $\widetilde{cod}$ is as expected, but rather than lifting the predicate $P(T_1, T_2) \equiv T_1 = dom(T_2)$, the rule uses the independent lifting of the dom partial function and the $=$ predicate. While lifting a complex predicate by lifting its parts seems natural, it does not in general produce the correct lifted predicate. For example, consider the following (contrived) partial function:

$$F(\mathsf{Int}) = \mathsf{Bool} \qquad F(\mathsf{Bool}) = \mathsf{Int}$$

and define a predicate $P(T_1, T_2) \equiv T_1 = F(T_2)$. Using our framework, we find that $\mathsf{Int} \to \mathsf{Int} \sim \widetilde{F}(?)$ holds, but $\widetilde{P}(\mathsf{Int} \to \mathsf{Int}, ?)$ does not. Piecewise lifting loses precision and in doing so misses some inconsistencies. We can prove, however, that piecewise lifting works here.

Proposition 8. *Let* $P(T_1, T_2) \equiv T_1 = dom(T_2)$. *Then* $\widetilde{T}_1 \sim \widetilde{dom}(\widetilde{T}_2)$ *if and only if* $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$.

4.6 Static Criteria for Gradual Typing

The AGT approach allows us to systematically derive a gradual type system that enjoys the expected properties discussed in the literature, including type safety and the criteria of Siek et al. (2015b).

Equivalence for fully-annotated terms. Siek and Taha (2006) prove that the gradual type system is a conservative extension of the static language; in other words, it is equivalent to the STLC on fully-annotated programs. Denoting by $\vdash_S$ the STFL typing relation (Fig. 1), the equivalence is stated as follows:

Proposition 9 (Equivalence for fully-annotated terms)**.** $\vdash_S t : T$ *if and only if* $\vdash t : T$.

This proof is trivial in the AGT approach because each lifted predicate and operator is equivalent to its static counterpart when only static types are involved.

Embedding of dynamic language terms. The dynamic counterpart of STFL—which includes terms $\check{t}$ that are not directly typeable in the gradual language like $(\lambda x.x\ x)$—can be embedded in the gradual language by a simple translation $\lceil \cdot \rceil$ that ascribes the unknown type to every subexpression and annotates every function with ? (Garcia and Cimini 2015; Siek and Taha 2006).

Proposition 10 (Dynamic embedding)**.** *If* $\check{t}$ *is closed then* $\vdash \lceil \check{t} \rceil : ?$.

Gradual guarantee. Siek et al. (2015b) present a set of refined criteria for gradual typing, called the *gradual guarantee*. The gradual guarantee has three parts. The first part addresses only the type system of a gradually-typed language. It states that decreasing the precision of the types in a program does not affect its well-typedness. Precision among terms $t \sqsubseteq t$ is the natural lifting of type precision to terms.

Proposition 11 (Gradual guarantee (static))**.** *If* $\vdash \widetilde{t}_1 : \widetilde{T}_1$ *and* $\widetilde{t}_1 \sqsubseteq \widetilde{t}_2$, *then* $\vdash \widetilde{t}_2 : \widetilde{T}_2$ *and* $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$.

$$\frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \frac{T_{21} <: T_{11} \quad T_{12} <: T_{22}}{T_{11} \to T_{12} <: T_{21} \to T_{22}}$$

$$\frac{}{\mathsf{Bool} <: \mathsf{Bool}} \qquad \frac{\overline{T_{i1} <: T_{i2}}}{\overline{[l_i : T_{i1}}, \overline{l_j : T_j}] <: \overline{[l_i : T_{i2}]}}$$

$$\text{(Trec)}\ \frac{\overline{\Gamma \vdash t_i : T_i}}{\Gamma \vdash \overline{[l_i = t_i]} : \overline{[l_i : T_i]}} \qquad \text{(Tproj)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash t.l : proj(T, l)}$$

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

$$\text{(Tif)}\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_1 <: \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T_2 \mathbin{\ddot{\vee}} T_3}$$

$proj : \textsc{Type} \times \textsc{Label} \rightharpoonup \textsc{Type}$
$proj([l : T, \overline{l_i : T_i}], l) = T$
$proj(T, l)$ undefined otherwise

$\ddot{\vee} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type}$
$\mathsf{Int} \mathbin{\ddot{\vee}} \mathsf{Int} = \mathsf{Int}$
$\mathsf{Bool} \mathbin{\ddot{\vee}} \mathsf{Bool} = \mathsf{Bool}$
$(T_{11} \to T_{12}) \mathbin{\ddot{\vee}} (T_{21} \to T_{22}) = (T_{11} \mathbin{\ddot{\wedge}} T_{21}) \to (T_{12} \mathbin{\ddot{\vee}} T_{22})$
$\overline{[l_i : T_{i1}}, \overline{l_j : T_j}] \mathbin{\ddot{\vee}} \overline{[l_i : T_{i2}}, \overline{l_k : T_k}] = [l_i : T_{i1} \mathbin{\ddot{\vee}} T_{i2}]$
$T \mathbin{\ddot{\vee}} T$ undefined otherwise

$\ddot{\wedge} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type}$
$\mathsf{Int} \mathbin{\ddot{\wedge}} \mathsf{Int} = \mathsf{Int}$
$\mathsf{Bool} \mathbin{\ddot{\wedge}} \mathsf{Bool} = \mathsf{Bool}$
$(T_{11} \to T_{12}) \mathbin{\ddot{\wedge}} (T_{21} \to T_{22}) = (T_{11} \mathbin{\ddot{\vee}} T_{21}) \to (T_{12} \mathbin{\ddot{\wedge}} T_{22})$
$\overline{[l_i : T_{i1}}, \overline{l_j : T_j}] \mathbin{\ddot{\wedge}} \overline{[l_i : T_{i2}}, \overline{l_k : T_k}] =$
$[l_i : T_{i1} \mathbin{\ddot{\wedge}} T_{i2}, \overline{l_j : T_j}, \overline{l_k : T_k}]$
$T \mathbin{\ddot{\wedge}} T$ undefined otherwise

Figure 3. STFL$_{<:}$: Subtyping Extensions to STFL

Since consistent predicates and gradual functions are monotone for precision, using the AGT approach naturally yields a type system that satisfies this guarantee.

5. Case Study: Subtyping

This section extends STFL with records and subtyping to form the STFL$_{<:}$ language, and then applies the AGT approach to produce GTFL$_{\lesssim}$, the corresponding gradual type system. The resulting language bears strong similarity to the $\mathbf{Ob}^{?}_{<:}$ language of Siek and Taha (2007). That language has objects as implicitly recursive record types with width subtyping. Here we focus on pure record types with both width and depth subtyping.

The STFL$_{<:}$ language (Fig. 3) extends STFL with records, projections, and record types:

$$\begin{array}{rcl} l \in \textsc{Label} & & \\ t & ::= & \ldots \mid \overline{[l = t]} \mid t.l \\ T & ::= & \ldots \mid \overline{[l : T]} \end{array}$$

We define the standard subtyping relation, and use subtyping in the typing rules in place of type equality. The (Trec) rule is standard. The (Tproj) rule uses a partial function to ascertain its output type, but is otherwise standard. The (Tapp) rule loosens the requirement on the argument type: it simply needs to be a subtype of the function domain. The (Tif) rule uses subtyping to bound the predicate type. Interestingly, the output type is the least upper bound of the two branch types with respect to subtyping $\ddot{\vee}$. This is standard practice for a language with subtyping (Pierce 2002). In

fact, we can view $=$ and *equate* in Fig. 1 as degenerate instances of $<:$ and $\ddot{\vee}$ respectively.

5.1 Extending Gradual Types

Now that we have extended the set of static types, we must also extend the gradual types and their meanings. First we add record types to the set of gradual types:

$$\widetilde{T} ::= \ldots \mid \overline{[l : \widetilde{T}]}$$

and extend the concretization function to give them meaning:

$$\gamma(\overline{[l_i : \widetilde{T}_i]}) = \widehat{\overline{[l_i : \gamma(\widetilde{T}_i)]}}$$

$$\text{where } \widehat{\overline{[l_i : \widehat{T}_i]}} = \{\, \overline{[l_i : T_i]} \mid \overline{T_i \in \widehat{T}_i} \,\}$$

In turn the abstraction function is extended:

$$\alpha(\{\, \overline{[l_i : T_{ij}]} \,\}) = \overline{[l_i : \alpha(\{\, \overline{T_{ij}} \,\})]}$$

These extended definitions are sound and optimal.

5.2 Consistent Subtyping

Next, we lift subtyping to gradual types. Siek and Taha (2007) call this relation *consistent subtyping*.

Definition 7 (Consistent Subtyping)**.**

$\widetilde{T}_1 \widetilde{<:} \widetilde{T}_2$ *if and only if* $T_1 <: T_2$ *for some* $T_1 \in \gamma(\widetilde{T}_1), T_2 \in \gamma(\widetilde{T}_2)$.

Siek and Taha define consistent subtyping $\lesssim$ in two steps. First they define a static subtyping relation $\widetilde{T} \leq \widetilde{T}$ over gradual types. Guided by the intuition that subtyping and gradual typing are orthogonal, they simply assert $? \leq ?$, *i.e.*, the unknown type is only its own subtype. Then they define an algorithm for consistent subtyping in terms of masking $T_1|_{T_2}$ and characterize it several ways:

Proposition 12 (Siek and Taha (2007))**.**

1. $\widetilde{T}_1 \lesssim \widetilde{T}_2$ *if and only if* $\widetilde{T}_1 \sim \widetilde{T}_1'$ *and* $\widetilde{T}_1' \leq \widetilde{T}_2$ *for some* $\widetilde{T}_1'$.
2. $\widetilde{T}_1 \lesssim \widetilde{T}_2$ *if and only if* $\widetilde{T}_1 \leq \widetilde{T}_2'$ *and* $\widetilde{T}_2' \sim \widetilde{T}_2$ *for some* $\widetilde{T}_2'$.

These are two quite different foundations for consistent subtyping: ours is defined purely in terms of static subtyping; Siek and Taha define static subtyping directly on gradual types, then blend it with consistent equality. Nonetheless, the two notions coincide:

Proposition 13. $\widetilde{T}_1 \lesssim \widetilde{T}_2$ *if and only if* $\widetilde{T}_1 \widetilde{<:} \widetilde{T}_2$.

From now on we use the symbol $\lesssim$ for consistent subtyping.

5.3 Lifting Meet and Join

We define join for gradual types $\widetilde{\ddot{\vee}}$ in the usual fashion:

Definition 8. $\widetilde{T}_1 \,\widetilde{\ddot{\vee}}\, \widetilde{T}_2 = \alpha(\gamma(\widetilde{T}_1) \,\widehat{\ddot{\vee}}\, \gamma(\widetilde{T}_2))$
where $\widehat{T}_1 \,\widehat{\ddot{\vee}}\, \widehat{T}_2 = \{\, T_1 \ddot{\vee} T_2 \mid T_1 \in \widehat{T}_1, T_2 \in \widehat{T}_2 \,\}$.

Here the AGT machinery pays off in spades. We need not appeal to intuition to develop a notion of consistent join. We simply define it as the approach dictates. By construction, the resulting operation soundly, optimally, and conservatively extends its static counterpart.

5.4 Extension: Gradual Rows

The AGT approach not only simplifies gradual type system design, but it can also inspire the design of new gradual typing features. We now consider how AGT inspired an extension to our language.

The unknown type ? is quite useful and general for gradual types, but in the presence of record types and subtyping, it loses a lot of information. To see this, consider the results of abstraction for several cases:

$$\alpha(\{\, [l_1 : \mathsf{Int}] \,\}) = [l_1 : \mathsf{Int}]$$
$$\alpha(\{\, [l_1 : \mathsf{Int}], [l_1 : \mathsf{Bool}] \,\}) = [l_1 : ?]$$
$$\alpha(\{\, [l_1 : \mathsf{Int}], [l_1 : \mathsf{Int}, l_2 : \mathsf{Bool}] \,\}) = ?$$

The first abstraction is perfect, while the second must abstract the type of l_1. The last abstraction, however, loses *all* type information, including the fact that every type in the set is a record. Can we be a bit more precise about graduality? Yes we can. To do so, we introduce a new kind of gradual type, which we call a *gradual row*:

$$\widetilde{T} ::= \ldots \mid \overline{[l : \widetilde{T}}, ?]$$

A gradual row type represents *unknown extra fields* in the record type. We define this notion more precisely via concretization:

$$\gamma(\overline{[l_i : \widetilde{T}_i}, ?]) = \widehat{\overline{[l_i : \gamma(\widetilde{T}_i)}, ?]}$$

$$\text{where } \widehat{\overline{[l_i : \widehat{T}_i}, ?]} = \{\, \overline{[l_i : T_i}, \overline{l_j : T_j}] \mid \overline{T_i \in \widehat{T}_i}, \overline{T_j \in \textsc{Type}} \,\}$$

Records with gradual rows represent any record type that has *at least* the stated labels with some corresponding type.

In turn the abstraction function is extended:

$$\alpha(\{\, \overline{[l_i : T_{ij}, \ldots]} \,\}) = \overline{[l_i : \alpha(\{\, \overline{T_{ij}} \,\})}, ?]$$

Note that the ellipsis above denotes labels that are not common to all records in the set.

Given these extensions, our earlier examples produce strictly more precise results. In particular, the third example produces a record with a gradual row type:

$$\alpha(\{\, [l_1 : \mathsf{Int}], [l_1 : \mathsf{Int}, l_2 : \mathsf{Bool}] \,\}) = [l_1 : \mathsf{Int}, ?]$$

From this extended definition of gradual types, we calculate the following inductive characterization of consistent subtyping:

$$\frac{}{? \lesssim \widetilde{T}} \qquad \frac{}{\widetilde{T} \lesssim ?} \qquad \frac{}{\mathsf{Int} \lesssim \mathsf{Int}} \qquad \frac{}{\mathsf{Bool} \lesssim \mathsf{Bool}}$$

$$\frac{\widetilde{T}_{21} \lesssim \widetilde{T}_{11} \qquad \widetilde{T}_{12} \lesssim \widetilde{T}_{22}}{\widetilde{T}_{11} \to \widetilde{T}_{12} \lesssim \widetilde{T}_{21} \to \widetilde{T}_{22}}$$

$$\frac{\overline{\widetilde{T}_{i1} \lesssim \widetilde{T}_{i2}}}{\overline{[l_i : \widetilde{T}_{i1}}, \overline{l_j : \widetilde{T}_j}] \lesssim \overline{[l_i : \widetilde{T}_{i2}}, *]}$$

$$\frac{\overline{\widetilde{T}_{i1} \lesssim \widetilde{T}_{i2}}}{\overline{[l_i : \widetilde{T}_{i1}}, \overline{l_j : \widetilde{T}_j}, ?] \lesssim \overline{[l_i : \widetilde{T}_{i2}}, \overline{l_k : \widetilde{T}_k}, *]}$$

We use the notation $\overline{[l_i : \widetilde{T}_i}, *]$ to stand for both $\overline{[l_i : \widetilde{T}_i]}$ and $\overline{[l_i : \widetilde{T}_i}, ?]$ when the presence of a gradual row does not matter. Comparing the two rules for records, we find that gradual rows admit consistent supertypes that have extraneous fields, so long as the common fields of the two record types satisfy consistent subtyping.

The abstract interpretation framework motivated us to introduce gradual rows, but they can also be quite convenient for programming. A record type can be gradual with respect to which fields it has: known fields enjoy the assurances of static checking, while unknown fields afford the flexibility of dynamic checking.

6. Abstracting Dynamic Semantics

The AGT approach to gradual typing induces not only static semantics, but also dynamic semantics. To demonstrate this, we develop a

dynamic semantics for GTFL$_{\lesssim}$, our gradual counterpart to STFL$_{<:}$. Our approach subsumes and generalizes the threesome casts (without blame) of Siek and Wadler (2010). In addition, the resulting dynamic semantics honours, by construction, the dynamic criteria of Siek et al. (2015b).

6.1 An Overview of the Approach

To set the groundwork for the technical details, we begin with a high-level intuition for the approach. Our gradual dynamic semantics is founded directly on the proof of syntactic type safety (*i.e.*, progress and preservation) for the underlying static language. Conceptually, a type safety proof induces reduction rules over static typing derivations that mirror the reduction rules of the operational semantics. However, we typically focus on the evolving program rather than its evolving derivation. AGT shifts the focus to typing derivations in accordance with the proofs-as-programs correspondence, under which subject reduction of natural deduction proofs corresponds to reduction of programs (Howard 1980).

Our gradual typing derivations match the structure of static typing derivations, where the only differences are that static types become gradual types and type relations become consistent relations. Reasoning about these consistent relations is the key to inducing a gradual dynamic semantics.

We first refine each consistent typing judgment in a derivation with *evidence*. A consistent typing judgment (*e.g.*, consistent subtyping) by itself denotes the mere *possibility* that its corresponding static judgment (*e.g.*, subtyping) holds. Our notion of evidence characterizes *how likely* it is that the judgment holds.

We then develop operations to evolve the evidence for consistent judgments in accordance with type safety-induced reductions on derivations. Each new type relationship in a post-reduction derivation must be justified by applying properties of the relevant typing judgment (*e.g.*, transitivity of type equality and subtyping) to the type relations evident in the pre-reduction derivation. Using abstract interpretation, we lift logical deductions about static type relations to deductions about the evidence for consistent typing judgments (*e.g.*, consistent transitivity of consistent subtyping (Sec. 6.4)). However, consistent judgments are merely plausible, not guaranteed, so a consistent deduction may yield *no evidence* for the desired consistent judgment. Failure to deduce evidence *refutes* the consistent deduction, indicates an evident breakdown in the underlying type safety argument, and thereby justifies a runtime error.

In short, AGT yields a dynamic semantics with a crisp connection to the syntactic type safety argument of the underlying static language, which in turn is connected to its dynamic semantics.

6.2 Intrinsic Gradual Terms and Their Reduction

Defining reduction directly on 2-dimensional typing derivations is rather unwieldy. Instead of doing this, we represent gradual typing derivations as *intrinsically typed* terms (Church 1940). Intrinsic terms correspond directly to typing derivations, but are far more convenient to manipulate. In our approach, they fill the role typically played by an independently-designed cast calculus, but their structure and semantics emerge directly from the static and dynamic semantics of the static language. We illustrate these notions below.

Runtime Typing Rules Consider the GTFL$_{\lesssim}$ typing rules for functions and function application:

$$\frac{\boxed{(\widetilde{T}\lambda)} \quad \Gamma, x : \widetilde{T}_1 \vdash \widetilde{t} : \widetilde{T}_2}{\Gamma \vdash (\lambda x : \widetilde{T}_1 . \widetilde{t}) : \widetilde{T}_1 \to \widetilde{T}_2} \qquad \frac{\boxed{(\widetilde{T}\text{app})} \quad \Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \;\; \varepsilon_1 \vdash \widetilde{T}_1 \lesssim \widetilde{T}_{11} \to \widetilde{T}_{12} \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \;\; \varepsilon_2 \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}}{\Gamma \vdash \widetilde{t}_1 \; \widetilde{t}_2 : \widetilde{T}_{12}}$$

The $(\widetilde{T}\lambda)$ rule is the same as in GTFL (Fig. 2), but the $(\widetilde{T}\text{app})$ rule, has three significant changes. First, as expected, consistent equality has been replaced with consistent subtyping. More interesting, though, is that the $\widetilde{dom}(\widetilde{T}_1)$ and $\widetilde{cod}(\widetilde{T}_1)$ partial functions have been replaced with a consistent subtyping judgment on $\widetilde{T}_1$. This change lets the type safety proof smoothly evolve typing derivations. The $\widetilde{dom}(\widetilde{T}_1)$ and $\widetilde{cod}(\widetilde{T}_1)$ functions support the syntax-directed typing of gradual source terms by pinning the source type of the operator expression. In particular, if a source program's operator has type $\widetilde{T}_0$, then the fact that $\widetilde{dom}(\widetilde{T}_0)$ and $\widetilde{cod}(\widetilde{T}_0)$ are defined for the source type implies that $\widetilde{T}_0 \sim \widetilde{dom}(\widetilde{T}_0) \to \widetilde{cod}(\widetilde{T}_0)$, which corresponds to equality in the static system. At runtime, however, the operator expression may reduce to a new expression whose type $\widetilde{T}_1$ is a consistent subtype of $\widetilde{T}_0$, just as the analogous source type may evolve to a subtype in the static language. Thus, since we preserve types exactly, we untether the source type $\widetilde{T}_{11} \to \widetilde{T}_{12} = \widetilde{dom}(\widetilde{T}_0) \to \widetilde{cod}(\widetilde{T}_0)$ from the evolving operator's type $\widetilde{T}_1$. Thus, the type of the application expression remains $\widetilde{T}_{12}$ throughout evaluation. This generalization arises naturally as part of the type safety proof for the static language, especially if the preservation theorem is formulated to strictly preserve types.

Finally, each consistent subtyping judgment is now supported by *evidence* $\varepsilon \in \text{Ev}^{<:}$, which is runtime information that reflects the plausibility of consistent subtyping. When typing source programs, our only concern is *that* consistent subtyping holds, but during evaluation, we are concerned with *why* it (still) holds. Evidence is our representation of why, and it evolves at runtime. We introduce aspects of evidence as needed, and ultimately provide a precise definition in Sec. 6.4.

Intrinsic Terms Having prepared the gradual typing rules to accomodate runtime evolution, we now transform them into term constructors for intrinsically-typed terms. In contrast to the more common extrinsically-typed terms, intrinsically-typed terms define a *family* of type-indexed sets, such that the only terms that are ever defined are well-typed: ill-typed terms do not even exist. Note, however, that the static type information is purely administrative and can be erased in a practical implementation.

We start with intrinsically-typed variables. For each gradual type $\widetilde{T}$, assume a distinct family of variables of that type:

$$x^{\widetilde{T}} \in \text{VAR}_{\widetilde{T}} \quad \text{where} \quad \widetilde{T}_1 \neq \widetilde{T}_2 \implies \text{VAR}_{\widetilde{T}_1} \cap \text{VAR}_{\widetilde{T}_2} = \emptyset.$$

Note that $x^{\widetilde{T}}$ is a metavariable, not a combination of variable and type: variables are atomic elements. Also, since the variable sets are disjoint, it follows that $\widetilde{T}_1 \neq \widetilde{T}_2 \implies x^{\widetilde{T}_1} \not\equiv x^{\widetilde{T}_2}$. Intrinsically typed variables obviate the need for type environments Γ, yielding a self-contained term language.

To define intrinsic terms, we inspect each typing rule and introduce a term constructor that captures all the information needed to reconstruct the rule. For instance, consider the intrinsic term formation rules for functions and application:

$$(\text{I}\widetilde{T}\lambda)\frac{t^{\widetilde{T}_2} \in \text{TERM}_{\widetilde{T}_2}}{\lambda x^{\widetilde{T}_1} . t^{\widetilde{T}_2} \in \text{TERM}_{\widetilde{T}_1 \to \widetilde{T}_2}}$$

$$(\text{I}\widetilde{T}\text{app})\frac{t^{\widetilde{T}_1} \in \text{TERM}_{\widetilde{T}_1} \qquad \varepsilon_1 \vdash \widetilde{T}_1 \lesssim \widetilde{T}_{11} \to \widetilde{T}_{12} \qquad t^{\widetilde{T}_2} \in \text{TERM}_{\widetilde{T}_2} \qquad \varepsilon_2 \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}}{(\varepsilon_1 t^{\widetilde{T}_1}) \; @^{\widetilde{T}_{11} \to \widetilde{T}_{12}} \; (\varepsilon_2 t^{\widetilde{T}_2}) \in \text{TERM}_{\widetilde{T}_{12}}}$$

The ($\mathrm{I}\widetilde{T}\lambda$) rule builds intrinsic functions that closely resemble their extrinsic counterparts. This rule resembles the ($\widetilde{T}\lambda$) rule, but the premise has no assumptions about variable types since all free variables are intrinsically typed.

The ($\mathrm{I}\widetilde{T}$app) rule, on the other hand, produces a much richer term than its extrinsic counterpart. Instead of using traditional juxtaposition, it uses an explicit @ constructor that is indexed on the gradual type $\widetilde{T}_{11} \to \widetilde{T}_{12}$ at which the function application occurs. Without this index, the corresponding extrinsic typing rule (and therefore the corresponding typing derivation) could not be recreated perfectly. Each typing derivation embodies one specific typing, so each intrinsic term can have only one type in turn.

Each subterm of application is qualified with evidence for consistent subtyping. This evidence is the critical information needed at runtime to detect gradual type inconsistencies. In fact each subterm of each elimination rule is qualified with evidence.

Reduction Reducing an intrinsic term corresponds to rewriting a type derivation tree as part of proving type preservation. Many presentations of subtyping allow the type of the term to evolve to a subtype, which by subsumption implies exact type preservation anyway. In the gradual context, however, consistent subtyping arguments depend on the evidence for *why* subtyping might hold, so subsumption for consistent subtyping may fail at runtime. To retain soundness, then, we must go further and require that reductions preserve the exact gradual type of the original intrinsic term.

When an intrinsic term is reduced, evidence for the relevant consistent judgments must evolve to support the resulting term. For example, consider the following function application:

$$(\varepsilon_1 \lambda x^{\widetilde{T}'_{11}}.t^{\widetilde{T}'_{12}})\ @^{\widetilde{T}_{11}\to\widetilde{T}_{12}}\ (\varepsilon_2 v^{\widetilde{T}_2})$$

Here, $v^{\widetilde{T}} \in \textsc{Value}_{\widetilde{T}}$ indicates some intrinsically typed value. The term formation rules tell us that $\varepsilon_1 \vdash \widetilde{T}'_{11} \to \widetilde{T}'_{12} \lesssim \widetilde{T}_{11} \to \widetilde{T}_{12}$ and $\varepsilon_2 \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}$. How shall we reduce this expression? Conceptually, we want to substitute $v^{\widetilde{T}_2}$ into the body of the function, but this value has type $\widetilde{T}_2$, while the variable $x^{\widetilde{T}'_{11}}$ has type $\widetilde{T}'_{11}$. We must bridge the gap.

The insight here is that we already have evidence $\varepsilon_2 \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}$ and by reasoning about inversion on consistent subtyping, we can extract from ε_1 evidence $idom(\varepsilon_1) \vdash \widetilde{T}_{11} \lesssim \widetilde{T}'_{11}$ and $icod(\varepsilon_1) \vdash \widetilde{T}'_{12} \lesssim \widetilde{T}_{12}$. To substitute $v^{\widetilde{T}_2}$ for $x^{\widetilde{T}'_{11}}$, we must deduce that $\widetilde{T}_2 \lesssim \widetilde{T}'_{11}$. Recall that consistent subtyping is *not* in general transitive. Nonetheless, we can combine the evidence ε_2 and $idom(\varepsilon_1)$ to (possibly) produce evidence $(\varepsilon_2 \circ^{<:} idom(\varepsilon_1)) \vdash \widetilde{T}_2 \lesssim \widetilde{T}'_{11}$. The *consistent transitivity* operation $\circ^{<:}$ attempts to combine two pieces of evidence to form evidence for the transitive case. Note that this *plausible* argument exactly mirrors the *definite* argument that arises when proving type safety for the underlying static language. However, consistent transitivity is a partial function: it may fail, in which case we signal a runtime error.

Leveraging consistent transitivity and inversion, we reduce the term as follows:

$$(\varepsilon_1 \lambda x^{\widetilde{T}'_{11}}.t^{\widetilde{T}'_{12}})\ @^{\widetilde{T}_{11}\to\widetilde{T}_{12}}\ (\varepsilon_2 v^{\widetilde{T}_2}) \longrightarrow icod(\varepsilon_1)\left([(\varepsilon_2 \circ^{<:} idom(\varepsilon_1))v^{\widetilde{T}_2} :: \widetilde{T}'_{11}/x^{\widetilde{T}'_{11}}]t^{\widetilde{T}'_{12}}\right) :: \widetilde{T}_{12}.$$

Since consistent transitivity is partial, this rule is only defined if $\varepsilon_2 \circ^{<:} idom(\varepsilon_1)$ is defined; otherwise we reduce to an error (Sec. 6.5).[e]

[e] Alternatively substitution could be defined to incorporate evidence at variable occurrences.

In summary, to develop the runtime semantics for gradual types we 1) qualify all consistent judgments with evidence; 2) define evaluation in terms of type-preserving rewrites on type derivations; and 3) reason explicitly about evidence for consistent subtyping judgments. We develop evidence and consistent transitivity below.

6.3 Initial Evidence: Interiors

We now elaborate our notion of evidence for consistent judgments. We start by extracting initial evidence from the bare knowledge that a consistent judgment holds.

A consistent predicate judges whether some gradual types represent some static types that satisfy the corresponding static judgment. As predicates, they convey only Boolean information: either yes, the gradual types represent some satisfying collection, or no they do not.

However, this coarse-grained information is not all that can be deduced from a consistent judgment. For instance, consider the judgment $[x : \mathsf{Int} \to ?, y : ?] \lesssim [x : ? \to \mathsf{Bool}]$. Indeed *some* types T_1 and T_2 in the concretization of each respective gradual type are subtypes. However, we can immediately deduce more precise information about T_1 and T_2 than this.

Proposition 14.

1. *If* $T_1 \in \gamma([x : \mathsf{Int} \to ?, y : ?])$, $T_2 \in \gamma([x : ? \to \mathsf{Bool}])$ *and* $T_1 <: T_2$ *then* $T_1 \in \gamma([x : \mathsf{Int} \to \mathsf{Bool}, y : ?])$ *and* $T_2 \in \gamma([x : \mathsf{Int} \to \mathsf{Bool}])$.
2. $[x : \mathsf{Int} \to \mathsf{Bool}, y : ?]$ *and* $[x : \mathsf{Int} \to \mathsf{Bool}]$ *are the most precise gradual types that satisfy property 1.*

Prop. 14 deduces *precise gradual type bounds* on each static type involved in the consistent subtyping judgment, based solely on the knowledge that the judgment holds. We generalize this notion to arbitrary consistent predicates. First, since consistent judgments involve multiple gradual types, we extend our gradual type abstraction to tuples of gradual types.

Definition 9 (Coordinate Concretization)**.**
Let $\gamma^2 : \textsc{GType} \times \textsc{GType} \to \mathcal{P}(\textsc{Type} \times \textsc{Type})$ *be defined by*

$$\gamma^2(\widetilde{T}_1, \widetilde{T}_2) = \gamma(\widetilde{T}_1) \times \gamma(\widetilde{T}_2).$$

This definition induces the following precision ordering: $\langle \widetilde{T}_{11}, \widetilde{T}_{12}\rangle \sqsubseteq^2 \langle \widetilde{T}_{21}, \widetilde{T}_{22}\rangle$ *if and only if* $\widetilde{T}_{11} \sqsubseteq \widetilde{T}_{21}$ *and* $\widetilde{T}_{12} \sqsubseteq \widetilde{T}_{22}$. *We generalize these to finite products:*

$$\gamma^n : \textsc{GType}^n \to \mathcal{P}(\textsc{Type}^n),$$
$$\sqsubseteq^n \subseteq \textsc{GType}^n \times \textsc{GType}^n.$$

We denote by π_n *the* n^{th} *projection function from a tuple.*

Definition 10 (Coordinate Abstraction)**.**
Let $\alpha^2 : \mathcal{P}(\textsc{Type} \times \textsc{Type}) \rightharpoonup \textsc{GType} \times \textsc{GType}$ *be defined by*

$$\alpha^2(\{\, \langle T_{i1}, T_{i2}\rangle, \dots \,\}) = \langle \alpha(\{\, T_{i1}, \dots \,\}), \alpha(\{\, T_{i2}, \dots \,\})\rangle\,.$$

We generalize this to arbitrary finite products $\alpha^n : \mathcal{P}(\textsc{Type}^n) \to \textsc{GType}^n$.

Note that γ^n, α^n pairs satisfy soundness and optimality.

Definition 11 (Interior)**.** *Let* P *be a binary predicate on static types. Then the* interior *of the judgment* $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$*, notation* $\mathcal{I}_P(\widetilde{T}_1, \widetilde{T}_2)$*, is the smallest tuple* $\langle \widetilde{T}'_1, \widetilde{T}'_2\rangle \sqsubseteq^2 \langle \widetilde{T}_1, \widetilde{T}_2\rangle$ *such that for* $\langle T_1, T_2\rangle \in \textsc{Type}^2$*, if* $\langle T_1, T_2\rangle \in \gamma^2(\widetilde{T}_1, \widetilde{T}_2)$ *and* $P(T_1, T_2)$*, then* $\langle T_1, T_2\rangle \in \gamma^2(\widetilde{T}'_1, \widetilde{T}'_2)$*. We generalize this to finite arity predicates.*

In essence, the interior produces the best coordinate-wise information that we can deduce from a consistent judgment. If $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$ does not hold, then $\mathcal{I}_P(\widetilde{T}_1, \widetilde{T}_2)$ is undefined.

The definition of interior is direct, but not easy to use in practice. We recast it in terms of coordinate abstraction.

Proposition 15.

$$\mathcal{I}_P(\widetilde{T}_1, \ldots, \widetilde{T}_n) = \alpha^n(\{\, \langle T_1, \ldots, T_n\rangle \in \gamma^n(\widetilde{T}_1, \ldots, \widetilde{T}_n) \mid P(T_1, \ldots, T_n)\,\}).$$

The interior of consistent subtyping. Now consider the interior of the consistent subtyping judgment in particular.

$$\mathcal{I}_{<:} : \text{GTYPE} \times \text{GTYPE} \rightharpoonup \text{EV}^{<:}$$

$$\begin{aligned}\mathcal{I}_{<:}(\widetilde{T}_1, \widetilde{T}_2) &= \alpha^2(\{\, \langle T_1, T_2\rangle \in \gamma^2(\widetilde{T}_1, \widetilde{T}_2) \mid T_1 <: T_2\,\}) \\ &= \langle \alpha(\{\, T_1 \in \gamma(\widetilde{T}_1) \mid \exists T_2 \in \gamma(\widetilde{T}_2).\, T_1 <: T_2\,\}), \\ &\qquad \alpha(\{\, T_2 \in \gamma(\widetilde{T}_2) \mid \exists T_1 \in \gamma(\widetilde{T}_1).\, T_1 <: T_2\,\})\rangle.\end{aligned}$$

As the signature of $\mathcal{I}_{<:}$ indicates, these precise tuples are instances of evidence for consistent subtyping. This characterization of $\mathcal{I}_{<:}$ is concise, but does not immediately suggest an algorithm. However, we use case-based reasoning to calculate a set of syntax-directed, invertible proof rules that imply a proof-search algorithm.[f] Fig. 4 presents the algorithmic rules for $\mathcal{I}_{<:}$. The rules exploit the fact that $\pi_1(\mathcal{I}_{<:}(\widetilde{T}, ?)) = \widetilde{T}$ and $\pi_2(\mathcal{I}_{<:}(?, \widetilde{T})) = \widetilde{T}$.

For an example, consider the following instance:

$$\mathcal{I}_{<:}(?, [g : \mathsf{Bool}]) = \langle [g : \mathsf{Bool}, ?], [g : \mathsf{Bool}]\rangle$$

Intuitively, this tells us that any subtype of $[g : \mathsf{Bool}]$ must be a record that has at least one field $g : \mathsf{Bool}$. Note that if we remove gradual rows from the language, we have no way to represent this precise information, and then the result remains $\langle ?, [g : \mathsf{Bool}]\rangle$. Though less precise, this evidence is still a sufficient basis for the dynamic semantics of a gradual language.

From consistency to threesomes. Applying the notion of interiors to consistent equality independently validates the cast insertion process for the Threesome Calculus (Siek and Wadler 2010). Since consistent equality lifts static type equality, we calculate its interior.

Proposition 16.
If $\widetilde{T}_1 \sim \widetilde{T}_2$, *then* $\mathcal{I}_{=}(\widetilde{T}_1, \widetilde{T}_2) = \langle \widetilde{T}_1 \sqcap \widetilde{T}_2, \widetilde{T}_1 \sqcap \widetilde{T}_2\rangle$.

In the Threesome Calculus, a cast-insertion procedure $\langle\!\langle \widetilde{T} \Leftarrow \widetilde{T} \rangle\!\rangle$ converts twosome casts into threesome casts. The process introduces a *middle type* that is the meet of the two ends, *e.g.*:

$$\langle\!\langle \mathsf{Int} \to ? \Leftarrow ? \to \mathsf{Bool} \rangle\!\rangle = \langle \mathsf{Int} \to ? \overset{\mathsf{Int}\to\mathsf{Bool}}{\Longleftarrow} ? \to \mathsf{Bool}\rangle$$

The deduced middle type matches the interior calculation, $\mathcal{I}_{=}(\mathsf{Int} \to ?, ? \to \mathsf{Bool}) = \langle \mathsf{Int} \to \mathsf{Bool}, \mathsf{Int} \to \mathsf{Bool}\rangle$. In the case of consistent equality, both type bounds are always identical, so the middle type reflects all of the information from the interior.

6.4 Evolving Evidence: Consistent Transitivity

To generalize evidence beyond the notion of interior, we develop *consistent transitivity* as a means to entail evidence for new consistent judgments from evidence for prior judgments. This form of reasoning generalizes the reduction strategy developed for threesome-style casts.

Consider two consistent subtyping judgments:

$$\varepsilon_{12} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2 \qquad \varepsilon_{23} \vdash \widetilde{T}_2 \lesssim \widetilde{T}_3.$$

When might the transitive case $\widetilde{T}_1 \lesssim \widetilde{T}_3$ hold? We use the AGT framework to formally combine the given evidence to form the most precise justifiable evidence for judging transitivity. Reasoning about transitivity pervades type-theoretic reasoning, so we formalize consistent transitivity for any binary predicate.

[f] Syntax-directed rules for static predicates simplify these calculations.

$$\frac{\widetilde{T} \in \{\, \mathsf{Int}, \mathsf{Bool}, ?\,\}}{\mathcal{I}_{<:}(\widetilde{T}, \widetilde{T}) = \langle \widetilde{T}, \widetilde{T}\rangle} \qquad \frac{\widetilde{T} \in \{\, \mathsf{Int}, \mathsf{Bool}, [\,]\,\}}{\mathcal{I}_{<:}(\widetilde{T}, ?) = \langle \widetilde{T}, \widetilde{T}\rangle}$$

$$\frac{\widetilde{T} \in \{\, \mathsf{Int}, \mathsf{Bool}\,\}}{\mathcal{I}_{<:}(?, \widetilde{T}) = \langle \widetilde{T}, \widetilde{T}\rangle} \qquad \frac{\mathcal{I}_{<:}(\widetilde{T}_{11} \to \widetilde{T}_{12}, ? \to ?) = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle}{\mathcal{I}_{<:}(\widetilde{T}_{11} \to \widetilde{T}_{12}, ?) = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle}$$

$$\frac{\mathcal{I}_{<:}(? \to ?, \widetilde{T}_{21} \to \widetilde{T}_{22}) = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle}{\mathcal{I}_{<:}(?, \widetilde{T}_{21} \to \widetilde{T}_{22}) = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle}$$

$$\frac{\mathcal{I}_{<:}(\widetilde{T}_{21}, \widetilde{T}_{11}) = \langle \widetilde{T}'_{21}, \widetilde{T}'_{11}\rangle \quad \mathcal{I}_{<:}(\widetilde{T}_{12}, \widetilde{T}_{22}) = \langle \widetilde{T}'_{12}, \widetilde{T}'_{22}\rangle}{\mathcal{I}_{<:}(\widetilde{T}_{11} \to \widetilde{T}_{12}, \widetilde{T}_{21} \to \widetilde{T}_{22}) = \langle \widetilde{T}'_{11} \to \widetilde{T}'_{12}, \widetilde{T}'_{21} \to \widetilde{T}'_{22}\rangle}$$

$$\frac{}{\mathcal{I}_{<:}(?, [\overline{l_i : \widetilde{T}_i}, ?]) = \mathcal{I}_{<:}([?], [\overline{l_i : \widetilde{T}_i}?])}$$

$$\frac{}{\mathcal{I}_{<:}([l : \widetilde{T}, \overline{l_i : \widetilde{T}_i}, *], ?) = \langle [l : \widetilde{T}, \overline{l_i : \widetilde{T}_i}, *], [?]\rangle}$$

$$\frac{}{\mathcal{I}_{<:}([\overline{l_i : \widetilde{T}_i}, *], [\,]) = \langle [\overline{l_i : \widetilde{T}_i}, *], [\,]\rangle}$$

$$\frac{}{\mathcal{I}_{<:}([\,], [?]) = \langle [\,], [\,]\rangle} \qquad \frac{}{\mathcal{I}_{<:}([?], [?]) = \langle [?], [?]\rangle}$$

$$\frac{}{\mathcal{I}_{<:}([l : \widetilde{T}, \overline{l_i : \widetilde{T}_i}, *], [?]) = \langle [l : \widetilde{T}, \overline{l_i : \widetilde{T}_i}, *], [?]\rangle}$$

$$\frac{\mathcal{I}_{<:}(\widetilde{T}_1, \widetilde{T}_2) = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle \quad \overline{\mathcal{I}_{<:}(\widetilde{T}_{i1}, \widetilde{T}_{i2}) = \langle \widetilde{T}'_{i1}, \widetilde{T}'_{i2}\rangle}}{\begin{array}{l}\mathcal{I}_{<:}([l : \widetilde{T}_1, \overline{l_i : \widetilde{T}_{i1}}, \overline{l_j : \widetilde{T}_j}, *_1], [l : \widetilde{T}_2, \overline{l_i : \widetilde{T}_{i2}}, *_2]) = \\ \qquad \langle [l : \widetilde{T}'_1, \overline{l_i : \widetilde{T}'_{i1}}, \overline{l_j : \widetilde{T}_j}, *_1], [l : \widetilde{T}'_2, \overline{l_i : \widetilde{T}'_{i2}}, *_2]\rangle\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{I}_{<:}([\overline{l_i : \widetilde{T}_{i1}}, \overline{l_j : \widetilde{T}_j}, l : ?, \overline{l_k : ?}, ?], [\overline{l_i : \widetilde{T}_{i2}}, l : \widetilde{T}, \overline{l_k : \widetilde{T}_k}, *]) \\ = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle\end{array}}{\mathcal{I}_{<:}([\overline{l_i : \widetilde{T}_{i1}}, \overline{l_j : \widetilde{T}_j}, ?], [\overline{l_i : \widetilde{T}_{i2}}, l : \widetilde{T}, \overline{l_k : \widetilde{T}_k}, *]) = \langle \widetilde{T}'_1, \widetilde{T}'_2\rangle}$$

Figure 4. Interior for Consistent Subtyping

Definition 12 (Consistent transitivity)**.** *Let* $P \subseteq \text{TYPE}^2$ *be a binary predicate on static types, and suppose*

$$\varepsilon_{12} \vdash \widetilde{P}(\widetilde{T}_1, \widetilde{T}_2) \quad \textit{and} \quad \varepsilon_{23} \vdash \widetilde{P}(\widetilde{T}_2, \widetilde{T}_3).$$

Then we deduce evidence for consistent transitivity *as* $(\varepsilon_{12} \circ^P \varepsilon_{23}) \vdash \widetilde{P}(\widetilde{T}_1, \widetilde{T}_3)$ *where*
$\circ^P : \text{EV}^P \times \text{EV}^P \rightharpoonup \text{EV}^P$ *is defined by:*

$$\begin{aligned}&\langle \widetilde{T}_1, \widetilde{T}_{21}\rangle \circ^P \langle \widetilde{T}_{22}, \widetilde{T}_3\rangle = \\ &\quad \alpha^2(\{\langle T_1, T_3\rangle \in \gamma^2(\widetilde{T}_1, \widetilde{T}_3) \mid \\ &\quad \exists T_2 \in \gamma(\widetilde{T}_{21}) \cap \gamma(\widetilde{T}_{22}).\, P(T_1, T_2) \wedge P(T_2, T_3)\}).\end{aligned}$$

The consistent transitivity operator for P collects and abstracts all available evidence that transitivity might hold in a particular instance. If this partial function yields no evidence for the transitive consistent judgment, then the judgment has been *refuted* and the operator is undefined for the given input.

To simplify this definition, we observe a useful property of gradual types:

Proposition 17. $\gamma(\widetilde{T}_1 \sqcap \widetilde{T}_2) = \gamma(\widetilde{T}_1) \cap \gamma(\widetilde{T}_2)$.

This means that the meet operator *losslessly* abstracts the intersection of static type sets that arise from gradual types. As such, we

can safely use $\sqcap$ without losing precision.[g] We use this property to recast consistent transitivity:

Proposition 18. $\langle \widetilde{T}_1, \widetilde{T}_{21} \rangle \circ^P \langle \widetilde{T}_{22}, \widetilde{T}_3 \rangle = \triangle^P(\widetilde{T}_1, \widetilde{T}_{21} \sqcap \widetilde{T}_{22}, \widetilde{T}_3)$ *where*

$$\triangle^P(\widetilde{T}_1, \widetilde{T}_2, \widetilde{T}_3) = \alpha^2(\{\langle T_1, T_3 \rangle \in \gamma^2(\widetilde{T}_1, \widetilde{T}_3) \mid \exists T_2 \in \gamma(\widetilde{T}_2).P(T_1, T_2) \wedge P(T_2, T_3)\}).$$

This formulation of consistent transitivity is generally useful.

What is Evidence? As we have seen, the interior of a consistent judgment yields initial evidence for the judgment, and consistent transitivity evolves that evidence as program reduction requires new judgments. Given these properties, we are now equipped to completely define our notion of evidence for consistent judgments.

Definition 13 (Evidence for $\widetilde{P}$)**.** *Let $P \subseteq \text{TYPE}^2$ be a binary type predicate. Then:*

1. $\text{EV}^P = \{\, \langle \widetilde{T}_1, \widetilde{T}_2 \rangle \in \text{GTYPE}^2 \mid \mathcal{I}_P(\widetilde{T}_1, \widetilde{T}_2) = \langle \widetilde{T}_1, \widetilde{T}_2 \rangle \,\}$*;*
2. *The evidence judgment $\cdot \vdash P(\cdot,\cdot) \subseteq \text{EV}^P \times \text{GTYPE}^2$ is defined as follows:*

$$\varepsilon \vdash \widetilde{P}(\widetilde{T}_1, \widetilde{T}_2) \iff \varepsilon \sqsubseteq^2 \mathcal{I}_P(\widetilde{T}_1, \widetilde{T}_2).$$

These notions generalize to finite n-ary predicates.

Evidence for a consistent judgment is represented as a tuple of gradual types that characterize the space of possible static type relations.[h] The tuple is self-interior with respect to the relevant judgment to reflect the most precise information available. Naturally, evidence for a judgment must be at least as precise as the judgment itself, otherwise the judgment itself would be better evidence.

Evidence represents a bound on the plausibility of a consistent judgment. This knowledge gains precision monotonically since consistent deductions are based on prior evidence.

Consistent Subtyping Revisited As defined, consistent transitivity directly embodies the notion that we need for consistent subtyping, but that definition suggests no obvious procedure for computing it. We refine this definition to an algorithmic specification for consistent subtyping. We could use the inductive definition of $<:$ to calculate a recursive definition for $\triangle^{<:}$ over the structure of $\widetilde{T}_2$, but we can also recast $\circ^{<:}$ in particular in terms of interiors and meets.

Proposition 19.

$$\triangle^{<:}(\widetilde{T}_1, \widetilde{T}_2, \widetilde{T}_3) = \langle \pi_1(\mathcal{I}_{<:}(\widetilde{T}_1, \widetilde{T}_2)), \pi_2(\mathcal{I}_{<:}(\widetilde{T}_2, \widetilde{T}_3)) \rangle\,.$$

Corollary 20.
$\langle \widetilde{T}_1, \widetilde{T}_{21} \rangle \circ^{<:} \langle \widetilde{T}_{22}, \widetilde{T}_3 \rangle = \langle \pi_1(\mathcal{I}_{<:}(\widetilde{T}_1, \widetilde{T}_2)), \pi_2(\mathcal{I}_{<:}(\widetilde{T}_2, \widetilde{T}_3)) \rangle\,.$
where $\widetilde{T}_2 = \widetilde{T}_{21} \sqcap \widetilde{T}_{22}$.

Similar reasoning yields the same result for consistent equality:

Proposition 21.

$$\triangle^{=}(\widetilde{T}_1, \widetilde{T}_1 \sqcap \widetilde{T}_2, \widetilde{T}_2) = \langle \widetilde{T}_1 \sqcap \widetilde{T}_2, \widetilde{T}_1 \sqcap \widetilde{T}_2 \rangle\,.$$

This result independently confirms the use of the meet operator to combine middle types in the Threesome Calculus.

[g] In the terminology of abstract interpretation, $\sqcap$ is *forward-complete* (Giacobazzi and Quintarelli 2001) or γ*-complete* (Schmidt 2008).

[h] Abstractions can lift to tuples other ways too (Cousot and Cousot 1994).

The interior is not always enough. In the general case, the $\triangle^P$ operation is not just the projection of two interiors. A relevant counterexample to this is consistent subtyping when gradual rows are omitted from the language of gradual types. Consider the following two evidence judgments:

$$\langle [f : \mathsf{Int}, g : ?], ? \rangle \vdash [f : \mathsf{Int}, g : ?] \lesssim ?$$
$$\langle ?, [g : \mathsf{Bool}] \rangle \vdash ? \lesssim [g : \mathsf{Bool}].$$

Then $\mathcal{I}_{<:}([f : \mathsf{Int}, g : ?], ? \sqcap ?) = \langle [f : \mathsf{Int}, g : ?], ? \rangle$ and $\mathcal{I}_{<:}(? \sqcap ?, [g : \mathsf{Bool}]) = \langle ?, [g : \mathsf{Bool}] \rangle$ but

$$\triangle^{<:}([f : \mathsf{Int}, g : ?], ? \sqcap ?, [g : \mathsf{Bool}]) = \langle [f : \mathsf{Int}, g : \mathsf{Bool}], [g : \mathsf{Bool}] \rangle\,.$$

The issue is that the two ?'s in the middle lose all type information, so taking their meet does not provide the gradual types at each extreme with the missing information from one another. With gradual rows, though, the evidence judgments are:

$$\langle [f : \mathsf{Int}, g : ?], [?] \rangle \vdash [f : \mathsf{Int}, g : ?] \lesssim ?$$
$$\langle [g : \mathsf{Bool}, ?], [g : \mathsf{Bool}] \rangle \vdash ? \lesssim [g : \mathsf{Bool}]$$

and

$$\mathcal{I}_{<:}([f : \mathsf{Int}, g : ?], [?] \sqcap [g : \mathsf{Bool}, ?]) = \langle [f : \mathsf{Int}, g : \mathsf{Bool}], [g : \mathsf{Bool}, ?] \rangle$$
$$\mathcal{I}_{<:}([?] \sqcap [g : \mathsf{Bool}, ?], [g : \mathsf{Bool}]) = \langle [g : \mathsf{Bool}, ?], [g : \mathsf{Bool}] \rangle$$

So gradual rows introduce enough precision to simplify the calculation of consistent transitivity. In practice, this just means that we cannot reduce $\triangle^{<:}$ to $\mathcal{I}_{<:}$ if we do not want gradual rows. We conjecture that many gradual type definitions will yield transitive type predicates that admit the interior-meet based characterization of consistent transitivity. Alternatively, this property can serve as a design guideline for enriching gradual type abstractions.

6.5 Running Gradual Programs

Armed with a run-time representation for gradual type derivations (intrinsic terms), initial evidence for consistent judgments (interiors), and a means to evolve that evidence (consistent transitivity), we can now present the full dynamic semantics of GTFL$_{\lesssim}$.

Fig. 5 presents the formation rules for gradual intrinsic terms. Their structure mirrors the structure of the corresponding extrinsic typing rules, which are straightforward to reconstruct rule-by-rule. As mentioned earlier, the terms that correspond to elimination rules qualify their subexpressions with evidence.

In essence, type-checking an extrinsic gradual term builds an intrinsic term (*i.e.* typing derivation) by introducing interior evidence for each consistent judgment. This corresponds to the translation from a gradual language to a cast calculus from prior approaches, with evidence playing the role of casts, but here the "target language" is derived directly from the source language type system.

To denote terms independently of their types, we define a set of all intrinsic variables and a set of all intrinsic terms:

$$x^* \in \text{VAR}_* = \bigcup \text{VAR}_{\widetilde{T}} \qquad t^* \in \text{TERM}_* = \bigcup \text{TERM}_{\widetilde{T}}$$

Just as term application requires a type index to ensure unicity of typing, so do record-projection $t^{\widetilde{T}}.l^{\widetilde{T}_1}$ and ascription $\varepsilon t^{\widetilde{T}_1} :: \widetilde{T}_2$. These type annotations play no computational role in the language: they support the type safety proof, and in practice can be erased.

6.6 Reduction

Fig. 6 presents a structural operational semantics for intrinsic gradual terms. To support this, several syntactic families are introduced.

$$(\mathrm{I}\widetilde{T}\mathrm{n})\ \frac{}{n \in \textsc{Term}_{\mathsf{Int}}} \qquad (\mathrm{I}\widetilde{T}\mathrm{b})\ \frac{}{b \in \textsc{Term}_{\mathsf{Bool}}}$$

$$(\mathrm{I}\widetilde{T}\mathrm{x})\ \frac{}{x^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}}} \qquad (\mathrm{I}\widetilde{T}\mathrm{+})\ \frac{t^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} \quad \varepsilon_1 \vdash \widetilde{T}_1 \lesssim \mathsf{Int} \quad t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2} \quad \varepsilon_2 \vdash \widetilde{T}_2 \lesssim \mathsf{Int}}{\varepsilon_1 t^{\widetilde{T}_1} + \varepsilon_2 t^{\widetilde{T}_2} \in \textsc{Term}_{\mathsf{Int}}}$$

$$(\mathrm{I}\widetilde{T}\lambda)\ \frac{t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2}}{\lambda x^{\widetilde{T}_1}.t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_1 \to \widetilde{T}_2}} \qquad (\mathrm{I}\widetilde{T}\mathrm{rec})\ \frac{\overline{t^{\widetilde{T}_i} \in \textsc{Term}_{\widetilde{T}_i}}}{\overline{[l_i = t^{\widetilde{T}_i}]} \in \textsc{Term}_{\overline{[l_i : \widetilde{T}_i]}}}$$

$$(\mathrm{I}\widetilde{T}\mathrm{::})\ \frac{t^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} \quad \varepsilon \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2}{\varepsilon t^{\widetilde{T}_1} :: \widetilde{T}_2 \in \textsc{Term}_{\widetilde{T}_2}}$$

$$(\mathrm{I}\widetilde{T}\mathrm{app})\ \frac{t^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} \quad \varepsilon_1 \vdash \widetilde{T}_1 \lesssim \widetilde{T}_{11} \to \widetilde{T}_{12} \quad t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2} \quad \varepsilon_2 \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}}{(\varepsilon_1 t^{\widetilde{T}_1})\ @^{\widetilde{T}_{11} \to \widetilde{T}_{12}}\ (\varepsilon_2 t^{\widetilde{T}_2}) \in \textsc{Term}_{\widetilde{T}_{12}}}$$

$$(\mathrm{I}\widetilde{T}\mathrm{proj})\ \frac{t^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}} \quad \varepsilon \vdash \widetilde{T} \lesssim [l : \widetilde{T}_1, ?]}{\varepsilon t^{\widetilde{T}}.l^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1}}$$

$$(\mathrm{I}\widetilde{T}\mathrm{if})\ \frac{t^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} \quad \varepsilon_1 \vdash \widetilde{T}_1 \lesssim \mathsf{Bool} \quad t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2} \quad \varepsilon_2 \vdash \widetilde{T}_2 \lesssim \widetilde{T}_2 \mathbin{\widetilde{\vee}} \widetilde{T}_3 \quad t^{\widetilde{T}_3} \in \textsc{Term}_{\widetilde{T}_3} \quad \varepsilon_3 \vdash \widetilde{T}_3 \lesssim \widetilde{T}_2 \mathbin{\widetilde{\vee}} \widetilde{T}_3}{\mathsf{if}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{then}\ \varepsilon_2 t^{\widetilde{T}_2}\ \mathsf{else}\ \varepsilon_3 t^{\widetilde{T}_3} \in \textsc{Term}_{\widetilde{T}_2 \mathbin{\widetilde{\vee}} \widetilde{T}_3}}$$

Figure 5. Gradual Intrinsic Terms

The *values* v are either simple values u or simple values that have been ascribed a new type $\varepsilon u :: \widetilde{T}$. We introduce *evidence terms* et and *evidence values* ev to succinctly denote evaluation steps that compose evidence. An evidence term $et = \varepsilon t^{\widetilde{T}}$ is a term of type $\widetilde{T}$ that appears in some larger term that uses it at type $\widetilde{T}'$, justified by the evidence $\varepsilon \vdash \widetilde{T} \lesssim \widetilde{T}'$.

Finally, there are two kinds of evaluation frames: *evidence frames*, whose holes must be filled with an evidence term, and traditional *term frames*, whose holes must be filled with merely a term. Elimination forms in the language yield evidence frames because their subexpressions have associated evidence. Record formation, on the other hand, yields only term frames because the evaluated subexpressions are simply collected as the values of fields.

The reduction rules directly mirror the rules for the statically typed language, except that they must manage evidence at subexpression boundaries and combine evidence to form new evidence. In the addition rule, the only well-formed evidence for the requisite consistent subtypes is $\langle \mathsf{Int}, \mathsf{Int} \rangle$. Similarly, the predicate position of an if redex has evidence $\langle \mathsf{Bool}, \mathsf{Bool} \rangle$.

Sec. 6.2 introduced the function application rule. Here we know that the function evidence must be of function type. This rule appeals to *evidence inversion* functions $idom$ and $icod$, which manifest the evidence for inversion principles on consistent subtyping judgments. By inversion on consistent subtyping, we have the principle: If $\widetilde{T}_{11} \to \widetilde{T}_{12} \lesssim \widetilde{T}_{21} \to \widetilde{T}_{22}$ then $\widetilde{T}_{21} \lesssim \widetilde{T}_{11}$. $idom$ takes the evidence for the former ε_1 and yields the available evidence for the latter $idom(\varepsilon_1)$. Its definition follows:

$$idom(\langle \widetilde{T}_1' \to \widetilde{T}_2', \widetilde{T}_1'' \to \widetilde{T}_2'' \rangle) = \langle \widetilde{T}_1'', \widetilde{T}_1' \rangle$$
$$idom(\varepsilon) \quad \text{undefined otherwise}$$

The $icod$ operator is defined analogously. This reduction step produces an error if consistent transitivity refutes the consistent subtyping argument. The conditional rules are straightforward. Record projection appeals to the $iproj$ evidence inversion function to recover the evidence that the value at label l_j is a consistent subtype of the expected type.

Ascription expressions anchor the residual evidence produced by function application and record projection. Without ascription as a primitive, we could not easily represent each step of evaluation as a legal source typing: we would have to encode ascriptions using functions, which is unwieldy and does not generalize to languages that ascribe properties about computations rather than values, such as effects (Bañados Schwerter et al. 2014).

$et \in \textsc{EvTerm}$, $ev \in \textsc{EvValue}$, $u \in \textsc{SimpleValue}$, $x^* \in \textsc{Var}_*$
$t^* \in \textsc{Term}_*$, $v \in \textsc{Value}$, $g \in \textsc{EvFrame}$, $f \in \textsc{TmFrame}$

$$\begin{array}{rcl} et & ::= & \varepsilon t^* \\ ev & ::= & \varepsilon u \\ u & ::= & x^* \mid n \mid b \mid [\overline{l_i = v_i}] \mid \lambda x^*.t^* \\ v & ::= & u \mid \varepsilon u :: \widetilde{T} \\ g & ::= & \square + et \mid ev + \square \mid \square\ @^{\widetilde{T}}\ et \mid ev\ @^{\widetilde{T}}\ \square \mid \square :: \widetilde{T} \mid \square.l^{\widetilde{T}} \\ & \mid & \mathsf{if}\ \square\ \mathsf{then}\ et\ \mathsf{else}\ et \\ f & ::= & [\overline{l = v}, l = \square, \overline{l = t}] \mid g[\varepsilon\square] \end{array}$$

Notions of Reduction

$$\boxed{\longrightarrow : \textsc{Term}_{\widetilde{T}} \times (\textsc{Term}_{\widetilde{T}} \cup \{\,\mathbf{error}\,\})}$$

$$\varepsilon_1 n_1 + \varepsilon_2 n_2 \longrightarrow n_3 \text{ where } n_3 = n_1 + n_2$$

$$\varepsilon_1(\lambda x^{\widetilde{T}_{11}}.t^*)\ @^{\widetilde{T}_1 \to \widetilde{T}_2}\ \varepsilon_2 u \longrightarrow \begin{cases} icod(\varepsilon_1)([((\varepsilon_2 \circ^{<:} idom(\varepsilon_1))u :: \widetilde{T}_{11})/x^{\widetilde{T}_{11}}]t^*) :: \widetilde{T}_2 \\ \mathbf{error} \quad \text{if } (\varepsilon_2 \circ^{<:} idom(\varepsilon_1)) \text{ not defined} \end{cases}$$

$$\mathsf{if}\ \varepsilon_1 \mathsf{true}\ \mathsf{then}\ \varepsilon_2 t^{\widetilde{T}_2}\ \mathsf{else}\ \varepsilon_3 t^{\widetilde{T}_3} \longrightarrow \varepsilon_2 t^{\widetilde{T}_2} :: (\widetilde{T}_2 \mathbin{\widetilde{\vee}} \widetilde{T}_3)$$
$$\mathsf{if}\ \varepsilon_1 \mathsf{false}\ \mathsf{then}\ \varepsilon_2 t^{\widetilde{T}_2}\ \mathsf{else}\ \varepsilon_3 t^{\widetilde{T}_3} \longrightarrow \varepsilon_3 t^{\widetilde{T}_3} :: (\widetilde{T}_2 \mathbin{\widetilde{\vee}} \widetilde{T}_3)$$

$$\varepsilon[\overline{l_i = v_i}].l_j^{\widetilde{T}} \longrightarrow iproj(\varepsilon, l_j)v_j :: \widetilde{T}$$

$$\boxed{\longrightarrow_c : \textsc{EvTerm} \times (\textsc{EvTerm} \cup \{\,\mathbf{error}\,\})}$$

$$\varepsilon_1(\varepsilon_2 v :: \widetilde{T}) \longrightarrow_c \begin{cases} (\varepsilon_2 \circ^{<:} \varepsilon_1)v \\ \mathbf{error} \quad \text{if } (\varepsilon_2 \circ^{<:} \varepsilon_1) \text{ not defined} \end{cases}$$

$$\boxed{\longmapsto : \textsc{Term}_{\widetilde{T}} \times (\textsc{Term}_{\widetilde{T}} \cup \{\,\mathbf{error}\,\})}\ \textbf{Reduction}$$

$$(\mathrm{R}\longrightarrow)\ \frac{t^{\widetilde{T}} \longrightarrow r \quad r \in (\textsc{Term}_{\widetilde{T}} \cup \{\,\mathbf{error}\,\})}{t^{\widetilde{T}} \longmapsto r}$$

$$(\mathrm{R}g)\ \frac{et \longrightarrow_c et'}{g[et] \longmapsto g[et']} \qquad (\mathrm{R}g\mathrm{err})\ \frac{et \longrightarrow_c \mathbf{error}}{g[et] \longmapsto \mathbf{error}}$$

$$(\mathrm{R}f)\ \frac{t_1^{\widetilde{T}} \longmapsto t_2^{\widetilde{T}}}{f[t_1^{\widetilde{T}}] \longmapsto f[t_2^{\widetilde{T}}]} \qquad (\mathrm{R}f\mathrm{err})\ \frac{t_1^{\widetilde{T}} \longmapsto \mathbf{error}}{f[t_1^{\widetilde{T}}] \longmapsto \mathbf{error}}$$

Figure 6. Intrinsic Reduction

Finally, evidence reduction $\longrightarrow_c$ combines the evidence associated with an ascription with the evidence of a surrounding term. Note that if v has type $\widetilde{T}_2$, and $\varepsilon_1 \vdash \widetilde{T} \lesssim \widetilde{T}_3$ in the surrounding context, then reducing to $(\varepsilon_2 \circ^{<:} \varepsilon_1) \vdash \widetilde{T}_2 \lesssim \widetilde{T}_3$ updates the consistent subtyping judgment in the surrounding context.

6.7 Dynamic Criteria for Gradual Typing

This dynamic semantics satisfies criteria set out for the dynamic semantics of gradual languages by Siek et al. (2015b). In fact, the AGT approach yields semantics that satisfy these criteria *by construction*.

First, a gradual language must be safe: programs do not get stuck, though they may terminate with cast errors.

Proposition 22 (Type Safety)**.** *If* $t^{\widetilde{T}} \in \text{TERM}_{\widetilde{T}}$ *then one of the following is true:*

1. $t^{\widetilde{T}}$ *is a value* v*;*
2. $t^{\widetilde{T}} \longmapsto t'^{\widetilde{T}}$ *for some term* $t'^{\widetilde{T}} \in \text{TERM}_{\widetilde{T}}$*;*
3. $t^{\widetilde{T}} \longmapsto$ **error**.

The type safety proof mirrors the corresponding proof for the underlying static type discipline, replacing static type judgments with consistent judgments, and definite type deductions with consistent deductions.

Even more significant is that the language satisfies the dynamic components of the gradual guarantee, which essentially state that any program that runs without error would continue to do so if it were given less precise types. The key insight behind this is that evidence monotonically increases with the imprecision of the types involved, and $\circ^{<:}$ is monotone with respect to precision. Thus, less precise types produce fewer errors.

Proposition 23 (Dynamic guarantee)**.** *Suppose* $t_1^{\widetilde{T}_1} \sqsubseteq t_1^{\widetilde{T}_2}$*. Then if* $t_1^{\widetilde{T}_1} \longmapsto t_2^{\widetilde{T}_1}$ *then* $t_1^{\widetilde{T}_2} \longmapsto t_2^{\widetilde{T}_2}$ *where* $t_2^{\widetilde{T}_1} \sqsubseteq t_2^{\widetilde{T}_2}$*.*

Proof. Straightforward strong bisimulation. □

The simplicity of this proof is an important asset of the AGT approach. The analogous proof for the $\lambda^?_{\rightarrow}$ language required a number of tedious and non-trivial lemmas (Siek et al. 2015b).

Prop. 23 means not only that more dynamic programs continue to run at least as well as their more static counterparts, but also that *more static programs continue to run at least as poorly*: adding more static types will not fix prior type errors.

7. Related Work

Cimini and Siek (2016) present an approach and system to mechanically transform a static type system into a gradual type system and cast insertion procedure. The transformation introduces the unknown type ? and consistent equality $\sim$ to the gradual type system, and defines a type-directed translation to a place-holder cast calculus, which has static semantics but no dynamic semantics. Their approach has been demonstrated to handle features like fix, sums, and implicit typing. It would be interesting to combine their new automated approach with AGT, which admits richer notions of graduality like gradual rows and gradual effects.

A number of languages have developed typing disciplines that combine static and dynamic typing alongside subtyping.

Takikawa et al. (2012) develop a gradual type system for first-class classes in Racket, tackling mixins and other higher-order patterns. The type system supports inheritance and deals with accidental overriding. To do so, they use row polymorphism instead of standard subtype polymorphism. To match these static typing features, on the dynamic checking side they develop opaque and sealed contracts. It would be interesting to apply the AGT approach to their static typing discipline and compare the resulting dynamics.

Swamy et al. (2014) develop a novel hybrid type discipline for a subset of JavaScript. They use multi-language semantics to sandbox untrusted code in a web environment. In general the sandboxing technique introduces checking overhead, so they introduce a quasi-static type discipline (Thatte 1990) that uses type information to remove overhead. Siek et al. (2015b) demonstrate by counterexample that this language does not satisfy the gradual guarantee. It would be interesting to revisit their type-based sandboxing approach through the AGT lens.

Because information-flow labels form lattices that induce a natural notion of subtyping, work on developing gradual information-flow security languages has had to contend with it. Disney and Flanagan (2011) introduce a novel simply typed language that blends static checking, dynamic checking, and blame tracking of information-flow securities properties, while keeping all traditional type reasoning static. Fennell and Thiemann (2013) extend this work with support for mutable references. Both languages treat the unknown label as the top of the security lattice, which leads to the same difficulties that the quasi-static typing discipline, which treats ? as the top supertype, lends to gradual typing (Siek and Taha 2006). To avoid those issues, both languages require explicit downcasts in the security lattice. Using the AGT approach, one can derive a gradual source language analogous to Disney and Flanagan (2011) that avoids the challenges of quasi-static disciplines (Garcia and Tanter 2015). It would be beneficial to explore using AGT to gradualize information flow in the presence of mutable references.

8. Conclusion

Abstracting Gradual Typing yields compelling static and dynamic semantics grounded in the type discipline of the underlying source language and a designer-selected interpretation for gradual types, without further resort to intuition or speculation. AGT straightforwardly reconstructs notions from previous work on gradual typing, including consistent equality, consistent subtyping, and threesomes. As such, we believe that AGT provides a broad and general foundation for the development of gradually typed languages, old and new. Finally, the resulting static and dynamic semantics satisfy, by construction, all semantic criteria for gradually typed languages that we have found in the literature.

This work opens a number of avenues for further exploration. First, the dynamic semantics that arise from AGT invariably place evidence throughout the program. This design greatly simplifies metatheoretic reasoning about the language, but it is not practical. Much work on gradual typing focuses on its time and space efficiency (Herman et al. 2007; Rastogi et al. 2012, 2015); devising efficient semantics with AGT is future work. We intend to explore these techniques in the context of more comprehensive language features, like mutable references (Sergey and Clarke 2012; Siek et al. 2015c), parametric polymorphism (Ahmed et al. 2011), and polymorphic effects (Toro and Tanter 2015). Finally, we intend to investigate whether our approach sheds light on blame tracking (Siek et al. 2009, 2015a; Wadler and Findler 2009).

Acknowledgments We would like to thank Joshua Dunfield, Matteo Cimini, Jeremy Siek, Khurram A. Jafery, Felipe Bañados Schwerter, Matías Toro, and the anonymous referees for their feedback.

References

A. Ahmed, R. B. Findler, J. Siek, and P. Wadler. Blame for all. In *38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 201–214, Austin, Texas, USA, Jan. 2011. ACM Press.

C. Anderson and S. Drossopoulou. BabyJ: from object based to class based programming via types. *Electronic Notes in Theoretical Computer Science*, 82(8), 2003.

F. Bañados Schwerter, R. Garcia, and É. Tanter. A theory of gradual effect systems. In *19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, Sept. 2014. ACM Press.

A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.

M. Cimini and J. G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, St Petersburg, FL, USA, Jan. 2016. ACM Press.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, Jan. 1977. ACM Press.

P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California.

T. Disney and C. Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

L. Fennell and P. Thiemann. Gradual security typing with references. In *Computer Security Foundations Symposium*, pages 224–239, June 2013.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, Sept. 2002. ACM Press.

R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 303–315. ACM Press, Jan. 2015.

R. Garcia and É. Tanter. Deriving a simple gradual security language. available on arXiv, 2015. URL `http://arxiv.org/abs/1511.01399`.

R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, Oct. 2014.

R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In P. Cousot, editor, *Static Analysis*, volume 2126 of *LNCS*, pages 356–373. Springer-Verlag, 2001.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming*, page XXVIII, April 2007.

W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

L. Ina and A. Igarashi. Gradual typing for generics. In *26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624. ACM Press, Oct. 2011.

B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 481–494, Philadelphia, USA, Jan. 2012. ACM Press.

A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for typescript. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 167–180. ACM Press, Jan. 2015.

D. Rémy. Type checking records and variants as a natural extension of ML. In *16th ACM Symposium on Principles of Programming Languages (POPL 89)*, pages 77–88, Austin, TX, USA, Jan. 1989. ACM Press.

D. Schmidt. Internal and external logics of abstract interpretations. In F. Logozzo, D. Peled, and L. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *LNCS*, pages 263–278. Springer-Verlag, 2008.

I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211 of *LNCS*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.

J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in LNCS, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.

J. Siek and P. Wadler. Threesomes, with and without blame. In *37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 365–376. ACM Press, Jan. 2010.

J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In G. Castagna, editor, *18th European Symposium on Programming Languages and Systems (ESOP 2009)*, volume 5502 of *LNCS*, pages 17–31, York, UK, 2009. Springer-Verlag.

J. Siek, P. Thiemann, and P. Wadler. Blame and coercion: Together again for the first time. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, pages 425–435, Portland, OR, USA, June 2015a. ACM Press.

J. G. Siek and R. Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*, pages 68–80, 2012.

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

J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *4th ACM Dynamic Languages Symposium (DLS 2008)*, pages 7:1–7:12, Paphos, Cyprus, July 2008. ACM Press.

J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, pages 274–293, 2015b.

J. G. Siek, M. M. Vitousek, M. Cimini, S. Tobin-Hochstadt, and R. Garcia. Monotonic references for efficient gradual typing. In J. Vitek, editor, *24th European Symposium on Programming Languages and Systems (ESOP 2015)*, volume 9032 of *LNCS*, pages 432–456, London, UK, Mar. 2015c. Springer-Verlag.

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

A. Takikawa, T. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 793–810, Tucson, AZ, USA, Oct. 2012. ACM Press.

S. Thatte. Quasi-static typing. In *17th ACM Symposium on Principles of Programming Languages (POPL 90)*, pages 367–381, San Francisco, CA, United States, Jan. 1990. ACM Press.

M. Toro and É. Tanter. Customizable gradual polymorphic effects for Scala. In *30th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2015)*, pages 935–953, Pittsburgh, PA, USA, Oct. 2015. ACM Press.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In G. Castagna, editor, *18th European Symposium on Programming Languages and Systems (ESOP 2009)*, volume 5502 of *LNCS*, pages 1–16, York, UK, 2009. Springer-Verlag.

R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813 of *LNCS*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.