Gradual Refinement Types

Nico Lehmann [*]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
nlehmann@dcc.uchile.cl

Éric Tanter [†]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
etanter@dcc.uchile.cl

Abstract

Refinement types are an effective language-based verification technique. However, as any expressive typing discipline, its strength is its weakness, imposing sometimes undesired rigidity. Guided by abstract interpretation, we extend the gradual typing agenda and develop the notion of gradual refinement types, allowing smooth evolution and interoperability between simple types and logically-refined types. In doing so, we address two challenges unexplored in the gradual typing literature: dealing with imprecise logical information, and with dependent function types. The first challenge leads to a crucial notion of locality for refinement formulas, and the second yields novel operators related to type- and term-level substitution, identifying new opportunity for runtime errors in gradual dependently-typed languages. The gradual language we present is type safe, type sound, and satisfies the refined criteria for gradually-typed languages of Siek *et al.* We also explain how to extend our approach to richer refinement logics, anticipating key challenges to consider.

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

Keywords gradual typing; refinement types; abstract interpretation

1. Introduction

Refinement types are a lightweight form of language-based verification, enriching types with logical predicates. For instance, one can assign a refined type to a division operation ($/$), requiring that its second argument be non-zero:

$$\mathsf{Int} \to \{\nu\!:\!\mathsf{Int} \mid \nu \neq 0\} \to \mathsf{Int}$$

Any program that type checks using this operator is guaranteed to be free from division-by-zero errors at runtime. Consider:

$$\texttt{let}\ f\ (x\!: \mathsf{Int})\ (y\!: \mathsf{Int}) = 1/(x-y)$$

Int is seen as a notational shortcut for $\{\nu : \mathsf{Int} \mid \top\}$. Thus, in the definition of f the only information about x and y is that they are Int, which is sufficient to accept the subtraction, but insufficient to prove that the divisor is non-zero, as required by the type of the division operator. Therefore, f is rejected statically.

Refinement type systems also support *dependent* function types, allowing refinement predicates to depend on prior arguments. For instance, we can give f the more expressive type:

$$x\!:\!\mathsf{Int} \to \{\nu\!:\!\mathsf{Int} \mid \nu \neq x\} \to \mathsf{Int}$$

The body of f is now well-typed, because $y \neq x$ implies $x - y \neq 0$.

Refinement types have been used to verify various kinds of properties (Bengtson et al. 2011; Kawaguchi et al. 2009; Rondon et al. 2008; Xi and Pfenning 1998), and several practical implementations have been recently proposed (Chugh et al. 2012a; Swamy et al. 2016; Vazou et al. 2014; Vekris et al. 2016).

Integrating static and dynamic checking. As any static typing discipline, programming with refinement types can be demanding for programmers, hampering their wider adoption. For instance, all callers of f must establish that the two arguments are different.

A prominent line of research for improving the usability of refinement types has been to ensure automatic checking and inference, *e.g.* by restricting refinement formulas to be drawn from an SMT decidable logic (Rondon et al. 2008). But while type inference does alleviate the annotation burden on programmers, it does not alleviate the rigidity of *statically enforcing* the type discipline.

Therefore, several researchers have explored the complementary approach of combining static and dynamic checking of refinements (Flanagan 2006; Greenberg et al. 2010; Ou et al. 2004; Tanter and Tabareau 2015), providing *explicit casts* so that programmers can statically assert a given property and have it checked dynamically. For instance, instead of letting callers of f establish that the two arguments are different, we can use a cast:

$$\texttt{let}\ g\ (x\!: \mathsf{Int})\ (y\!: \mathsf{Int}) = 1/(\langle c\rangle\ (x-y))$$

The cast $\langle c\rangle$ ensures at runtime that the division is only applied if the value of $x - y$ is not 0. Division-by-zero errors are still guaranteed to not occur, but cast errors can.

These casts are essentially the refinement types counterpart of downcasts in a language like Java. As such, they have the same limitation when it comes to navigating between static and dynamic checking—programming in Python feels very different from programming in Java with declared type Object everywhere and explicit casts before every method invocation!

Gradual typing. To support a full slider between static and dynamic checking, without requiring programmers to explicitly deal with casts, Siek and Taha (2006) proposed *gradual typing*. Gradual typing is much more than "auto-cast" between static types, because gradual types can denote *partially-known* type information, yield-

[*] Funded by CONICYT-PCHA/Magíster Nacional/2015-22150894

[†] Partially funded by Fondecyt Project 1150017

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009856

ing a notion of *consistency*. Consider a variable x of gradual type $\mathsf{Int} \to ?$, where ? denotes the unknown type. This type conveys *some* information about x that can be statically exploited to definitely reject programs, such as $x + 1$ and $x(\mathsf{true})$, definitely accept valid applications such as $x(1)$, and optimistically accept any use of the return value, *e.g.* $x(1) + 1$, subject to a runtime check.

In essence, gradual typing is about plausible reasoning in presence of imprecise type information. The notion of *type precision* ($\mathsf{Int} \to ?$ is more precise than $? \to ?$), which can be raised to terms, allows distinguishing gradual typing from other forms of static-dynamic integration: weakening the precision of a term must preserve both its typeability and reduceability (Siek et al. 2015). This *gradual guarantee* captures the smooth evolution along the static-dynamic checking axis that gradual typing supports.

Gradual typing has triggered a lot of research efforts, including how to adapt the approach to expressive type disciplines, such as information flow typing (Disney and Flanagan 2011; Fennell and Thiemann 2013) and effects (Bañados Schwerter et al. 2014, 2016). These approaches show the value of *relativistic* gradual typing, where one end of the spectrum is a static discipline (*i.e.* simple types) and the other end is a *stronger* static discipline (*i.e.* simple types and effects). Similarly, in this work we aim at a gradual language that ranges from simple types to logically-refined types.

Gradual refinement types. We extend refinement types with *gradual formulas*, bringing the usability benefits of gradual typing to refinement types. First, gradual refinement types accommodate flexible idioms that do not fit the static checking discipline. For instance, assume an external, simply-typed function $check :: \mathsf{Int} \to \mathsf{Bool}$ and a refined get function that requires its argument to be positive. The absence of refinements in the signature of $check$ is traditionally interpreted as the absence of static knowledge denoted by the trivial formula $\top$:

$$check :: \{\nu\!:\!\mathsf{Int} \mid \top\} \to \{\nu\!:\!\mathsf{Bool} \mid \top\}$$

Because of the lack of knowledge about $check$ idiomatic expressions like:

$$\textbf{if}\ check(x)\ \textbf{then}\ get(x)\ \textbf{else}\ get(-x)$$

cannot possibly be statically accepted. In general, lack of knowledge can be due to simple/imprecise type annotations, or to the limited expressiveness of the refinement logic. With gradual refinement types we can interpret the absence of refinements as *imprecise knowledge* using the unknown refinement ?:[1]

$$check :: \{\nu\!:\!\mathsf{Int} \mid ?\} \to \{\nu\!:\!\mathsf{Bool} \mid ?\}$$

Using this annotation the system can exploit the imprecision to optimistically accept the previous code subject to dynamic checks.

Second, gradual refinements support a smooth evolution on the way to static refinement checking. For instance, consider the challenge of using an existing library with a refined typed interface:

$$a :: \{\nu\!:\!\mathsf{Int} \mid \nu < 0\} \to \mathsf{Bool} \qquad b :: \{\nu\!:\!\mathsf{Int} \mid \nu < 10\} \to \mathsf{Int}$$

One can start using the library without worrying about refinements:

$$\textbf{let}\ g\ (x\!: \{\nu\!:\!\mathsf{Int} \mid ?\}) = \textbf{if}\ a(x)\ \textbf{then}\ 1/x\ \textbf{else}\ b(x)$$

Based on the unknown refinement of x, all uses of x are statically accepted, but subject to runtime checks. Clients of g have no static requirement beyond passing an Int. The evolution of the program can lead to strengthening the type of x to $\{\nu\!:\!\mathsf{Int} \mid \nu > 0 \wedge ?\}$ forcing clients to statically establish that the argument is positive. In the definition of g, this more precise gradual type pays off: the type system definitely accepts $1/x$, making the associated runtime check superfluous, and it still optimistically accepts $b(x)$, subject to a dynamic check. It now, however, definitely rejects $a(x)$. Replacing $a(x)$ with $a(x - 2)$ again makes the type system optimistically accept the program. Hence, programmers can fine tune the level of static enforcement they are willing to deal with by adjusting the precision of type annotations, and get as much benefits as possible (both statically and dynamically).

Gradualizing refinement types presents a number of novel challenges, due to the presence of both logical information and dependent types. A first challenge is to properly capture the notion of precision between (partially-unknown) formulas. Another challenge is that, conversely to standard gradual types, we do not want an unknown formula to stand for *any* arbitrary formula, otherwise it would be possible to accept too many programs based on the potential for logical contradictions (a value refined by a false formula can be used everywhere). This issue is exacerbated by the fact that, due to dependent types, subtyping between refinements is a *contextual* relation, and therefore contradictions might arise in a non-local fashion. Yet another challenge is to understand the dynamic semantics and the new points of runtime failure due to the stronger requirements on term substitution in a dependently-typed setting.

Contributions. This work is the first development of gradual typing for refinement types, and makes the following contributions:

- Based on a simple static refinement type system (Section 2), and a generic interpretation of gradual refinement types, we derive a gradual refinement type system (Section 3) that is a conservative extension of the static type system, and preserves typeability of less precise programs. This involves developing a notion of consistent subtyping that accounts for *gradual logical environments*, and introducing the novel notion of *consistent type substitution*.
- We identify key challenges in defining a proper interpretation of gradual formulas. We develop a *non-contradicting*, *semantic*, and *local* interpretation of gradual formulas (Section 4), establishing that it fulfills both the practical expectations illustrated above, and the theoretical requirements for the properties of the gradual refinement type system to hold.
- Turning to the dynamic semantics of gradual refinements, we identify, beyond consistent subtyping transitivity, the need for two novel operators that ensure, at runtime, type preservation of the gradual language: *consistent subtyping substitution*, and *consistent term substitution* (Section 5). These operators crystallize the additional points of failure required by gradual dependent types.
- We develop the runtime semantics of gradual refinements as reductions over gradual typing derivations, extending the work of Garcia et al. (2016) to accommodate logical environments and the new operators dictated by dependent typing (Section 6). The gradual language is type safe, satisfies the dynamic gradual guarantee, as well as refinement soundness.
- To address the decidability of gradual refinement type checking, we present an algorithmic characterization of consistent subtyping (Section 7).
- Finally, we explain how to extend our approach to accommodate a more expressive refinement logic (Section 8), by adding *measures* (Vazou et al. 2014) to the logic.

Section 9 elaborates on some aspects of the resulting design, Section 10 discusses related work and Section 11 concludes. The proofs and complete definitions can be found in a companion technical report (Lehmann and Tanter 2016).

[1] In practice a language could provide a way to gradually import statically annotated code or provide a compilation flag to treat the absence of a refinement as the unknown formula ? instead of $\top$.

$$T \in \textsc{Type},\quad x \in \textsc{Var},\quad c \in \textsc{Const},\quad t \in \textsc{Term},$$
$$p \in \textsc{Formula},\quad \Gamma \in \textsc{Env},\quad \Phi \in \textsc{LEnv}$$

$$\begin{array}{rcll}
v & ::= & \lambda x{:}T.t \mid x \mid c & \text{(Values)} \\
t & ::= & v \mid t\ v \mid \mathsf{let}\ x = t\ \mathsf{in}\ t & \text{(Terms)} \\
 & & \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t :: T & \\
B & ::= & \mathsf{Int} \mid \mathsf{Bool} & \text{(Base Types)} \\
T & ::= & \{\nu{:}B \mid p\} \mid x{:}T \to T & \text{(Types)} \\
p & ::= & p = p \mid p < p \mid p + p \mid v \mid \nu & \text{(Formulas)} \\
 & & p \wedge p \mid p \vee p \mid \neg p \mid \top \mid \bot & \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}T & \text{(Type Environments)} \\
\Phi & ::= & \cdot \mid \Phi, x{:}p & \text{(Logical Environments)}
\end{array}$$

$$\boxed{\Gamma;\Phi \vdash t : T} \qquad (\text{Tx-refine})\frac{\Gamma(x) = \{\nu{:}B \mid p\}}{\Gamma;\Phi \vdash x : \{\nu{:}B \mid \nu = x\}}$$

$$(\text{Tx-fun})\frac{\Gamma(x) = y{:}T_1 \to T_2}{\Gamma;\Phi \vdash x : (y{:}T_1 \to T_2)} \qquad (\text{Tc})\frac{}{\Gamma;\Phi \vdash c : ty(c)}$$

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

$$(\text{Tapp})\frac{\Gamma;\Phi \vdash t : (x{:}T_1 \to T_2) \qquad \Gamma;\Phi \vdash v : T \qquad \Phi \vdash T <: T_1}{\Gamma;\Phi \vdash t\ v : T_2[v/x]}$$

$$(\text{Tif})\frac{\begin{array}{c}\Gamma;\Phi \vdash v : \{\nu{:}\mathsf{Bool} \mid p\} \qquad \Phi \vdash T_1 <: T \qquad \Phi \vdash T_2 <: T \\ \Gamma;\Phi, x{:}(v{=}\mathsf{true}) \vdash t_1 : T_1 \qquad \Gamma;\Phi, x{:}(v{=}\mathsf{false}) \vdash t_2 : T_2\end{array}}{\Gamma;\Phi \vdash \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : T}$$

$$(\text{Tlet})\frac{\begin{array}{c}\Gamma, x{:}T_1;\Phi, x{:}(\!|T_1|\!) \vdash t_2 : T_2 \qquad \Gamma;\Phi \vdash t_1 : T_1 \\ \Phi, x{:}(\!|T_1|\!) \vdash T_2 <: T \qquad \Phi \vdash T\end{array}}{\Gamma;\Phi \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T}$$

$$(\text{T::})\frac{\Gamma;\Phi \vdash t : T_1 \qquad \Phi \vdash T_1 <: T_2}{\Gamma;\Phi \vdash t :: T_2 : T_2}$$

$$\boxed{\Phi \vdash T_1 <: T_2} \qquad (<:\text{-refine})\frac{\begin{array}{c}(\!|\Phi|\!) \cup \{\,p\,\} \models q \\ \vdash \Phi \qquad \Phi \vdash p \qquad \Phi \vdash q\end{array}}{\Phi \vdash \{\nu{:}B \mid p\} <: \{\nu{:}B \mid q\}}$$

$$(<:\text{-fun})\frac{\Phi \vdash T_{21} <: T_{11} \qquad \Phi, x{:}(\!|T_{21}|\!) \vdash T_{12} <: T_{22}}{\Phi \vdash x{:}T_{11} \to T_{12} <: x{:}T_{21} \to T_{22}}$$

Figure 1. Syntax and static semantics of the static language.

To design the gradual refinement type system, we exercise the Abstracting Gradual Typing methodology (Garcia et al. 2016), a systematic approach to derive gradual typing disciplines based on abstract interpretation (Cousot and Cousot 1977). We summarize our specific contributions to AGT in Section 10.

2. A Static Refinement Type System

Figure 1 describes the syntax and static semantics of a language with refinement types. Base refinements have the form $\{\nu{:}B \mid p\}$ where p is a formula that can mention the *refinement variable* ν. Here, we consider the simple quantifier free logic of linear arithmetic (QF-LIA), which suffices to capture the key issues in designing gradual refinement types; we discuss extensions of our approach to more expressive logics in Section 8. Base refinements are used to build up *dependent function types* $x : T_1 \to T_2$. In order to restrict logical formulas to values, application arguments and conditions must be syntactic values (Vazou et al. 2014).

The typing judgment $\Gamma;\Phi \vdash t : T$ uses a type environment Γ to track variables in scope and a separate *logical environment* to track logical information. This separation is motivated by our desire to gradualize only the logical parts of types. For example, in Rule (Tλ) the body of a lambda $\lambda x{:}T.t$ is typed as usual extending the type environment and also enriching the logical environment with the logical information extracted from T, denoted $(\!|T|\!)$. Extraction is defined as $(\!|\{\nu{:}B \mid p\}|\!) = p$, and $(\!|x{:}T_1 \to T_2|\!) = \top$ since function types have no logical interpretation per se.

Subtyping is defined based on entailment in the logic. Rule (<:-refine) specifies that $\{\nu{:}B \mid p\}$ is a subtype of $\{\nu{:}B \mid q\}$ in an environment Φ if p, in conjunction with the information in Φ, entails q. Extraction of logical information from an environment, denoted $(\!|\Phi|\!)$, substitutes actual variables for refinement variables, *i.e.* $(\!|x{:}p|\!) = p[x/\nu]$. The judgment $\Delta \models p$ states that the set of formulas Δ entails p modulo the theory from which formulas are drawn. A judgment $\Delta \models p$ can be checked by issuing the query $\mathtt{VALID}(\Delta \to p)$ to an SMT solver. Note that subtyping holds only for well-formed types and environments (<:-refine); a type is well-formed if it only mentions variables that are in scope.

For the dynamic semantics we assume a standard call-by-value evaluation. Note also that Rule (Tapp) allows for arbitrary values in argument position, which are then introduced into the logic upon type substitution. Instead of introducing arbitrary lambdas in the logic, making verification undecidable, we introduce a fresh constant for every lambda (Chugh 2013). This technique is fundamental to prove soundness directly in the system since lambdas can appear in argument position at runtime.

3. A Gradual Refinement Type System

Abstracting Gradual Typing (AGT) is a methodology to systematically derive the gradual counterpart of a static typing discipline (Garcia et al. 2016), by viewing gradual types through the lens of abstract interpretation (Cousot and Cousot 1977). Gradual types are understood as abstractions of possible static types. The *meaning* of a gradual type is hence given by concretization to the set of static types that it represents. For instance, the unknown gradual type ? denotes any static type; the imprecise gradual type $\mathsf{Int} \to ?$ denotes all function types from Int to any type; and the precise gradual type Int only denotes the static type Int.

Once the interpretation of gradual types is defined, the static typing discipline can be promoted to a gradual discipline by *lifting* type predicates (*e.g.* subtyping) and type functions (*e.g.* subtyping join) to their *consistent* counterparts. This lifting is driven by the plausibility interpretation of gradual typing: a consistent predicate between two types holds if and only if the static predicate holds *for some* static types in their respective concretization. Lifting type functions additionally requires a notion of *abstraction* from sets of static types back to gradual types. By choosing the best abstraction that corresponds to the concretization, a Galois connection is established and a coherent gradual language can be derived.

In this work, we develop a gradual language that ranges from simple types to logically-refined types. Therefore we need to specify the syntax of gradual formulas, $\widetilde{p} \in \textsc{GFormula}$, and their meaning through a concretization function $\gamma_p : \textsc{GFormula} \to \mathcal{P}(\textsc{Formula})$. Once γ_p is defined, we need to specify the corresponding best abstraction α_p such that $\langle \gamma_p, \alpha_p \rangle$ is a Galois connection. We discovered that capturing a proper definition of gradual formulas and their interpretation to yield a *practical* gradual refinement type system—*i.e.* one that does not degenerate and accept almost any program—is rather subtle. In order to isolate this crucial design point, we defer the exact definition of $\textsc{GFormula}$, γ_p and α_p to Section 4. In the remainder of this section, we define the gradual refinement type system and establish its properties independently of the specific interpretation of gradual formulas.

The typing rules of the gradual language (Figure 2) directly mimic the static language typing rules, save for the fact that they use gradual refinement types $\widetilde{T}$, built from gradual formulas $\widetilde{p}$, and

$\widetilde{T} \in \text{GType}$, $t \in \text{GTerm}$, $\widetilde{p} \in \text{GFormula}$, $\Gamma \in \text{GEnv}$, $\widetilde{\Phi} \in \text{GLEnv}$

$$\boxed{\Gamma;\widetilde{\Phi} \vdash t : \widetilde{T}} \qquad (\widetilde{\text{T}}\text{x-refine})\,\frac{\Gamma(x) = \{\nu : B \mid \widetilde{p}\}}{\Gamma;\widetilde{\Phi} \vdash x : \{\nu : B \mid \nu = x\}}$$

$$(\widetilde{\text{T}}\text{x-fun})\,\frac{\Gamma(x) = y : \widetilde{T}_1 \rightarrow \widetilde{T}_2}{\Gamma;\widetilde{\Phi} \vdash x : (y : \widetilde{T}_1 \rightarrow \widetilde{T}_2)} \qquad (\widetilde{\text{T}}\text{c})\,\frac{}{\Gamma;\widetilde{\Phi} \vdash c : ty(c)}$$

$$(\widetilde{\text{T}}\lambda)\,\frac{\widetilde{\Phi} \vdash \widetilde{T}_1 \qquad \Gamma, x : \widetilde{T}_1 ; \widetilde{\Phi}, x : \llparenthesis \widetilde{T}_1 \rrparenthesis \vdash t : \widetilde{T}_2}{\Gamma;\widetilde{\Phi} \vdash \lambda x : \widetilde{T}_1 . t : (x : \widetilde{T}_1 \rightarrow \widetilde{T}_2)}$$

$$(\widetilde{\text{T}}\text{app})\,\frac{\Gamma;\widetilde{\Phi} \vdash t : x : \widetilde{T}_1 \rightarrow \widetilde{T}_2 \qquad \Gamma;\widetilde{\Phi} \vdash v : \widetilde{T} \qquad \widetilde{\Phi} \vdash \widetilde{T} \lesssim \widetilde{T}_1}{\Gamma;\widetilde{\Phi} \vdash t\ v : \widetilde{T}_2[\![v/x]\!]}$$

$$(\widetilde{\text{T}}\text{if})\,\frac{\begin{array}{ccc}\Gamma;\widetilde{\Phi} \vdash v : \{\nu : \mathsf{Bool} \mid \widetilde{p}\} & \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T} & \widetilde{\Phi} \vdash \widetilde{T}_2 \lesssim \widetilde{T} \\ \Gamma;\widetilde{\Phi}, x : (v{=}\mathsf{true}) \vdash t_1 : \widetilde{T}_1 & & \Gamma;\widetilde{\Phi}, x : (v{=}\mathsf{false}) \vdash t_2 : \widetilde{T}_2\end{array}}{\Gamma;\widetilde{\Phi} \vdash \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \widetilde{T}}$$

$$(\widetilde{\text{T}}\text{let})\,\frac{\begin{array}{cc}\Gamma, x : \widetilde{T}_1 ; \widetilde{\Phi}, x : \llparenthesis \widetilde{T}_1 \rrparenthesis \vdash t_2 : \widetilde{T}_2 & \Gamma;\widetilde{\Phi} \vdash t_1 : \widetilde{T}_1 \\ \widetilde{\Phi}, x : \llparenthesis \widetilde{T}_1 \rrparenthesis \vdash \widetilde{T}_2 \lesssim \widetilde{T} & \widetilde{\Phi} \vdash \widetilde{T}\end{array}}{\Gamma;\widetilde{\Phi} \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \widetilde{T}}$$

$$(\widetilde{\text{T}}{::})\,\frac{\Gamma;\widetilde{\Phi} \vdash t : \widetilde{T}_1 \qquad \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2}{\Gamma;\widetilde{\Phi} \vdash t :: \widetilde{T}_2 : \widetilde{T}_2}$$

Figure 2. Typing rules of the gradual language.

gradual environments $\widetilde{\Phi}$. Also, the rules use *consistent subtyping* and *consistent type substitution*. We define these notions below.[2]

3.1 Gradual Types and Environments

As we intend the imprecision of gradual types to reflect the imprecision of formulas, the syntax of gradual types $\widetilde{T} \in \text{GType}$ simply contains gradual formulas.

$$\widetilde{T} \quad ::= \quad \{\nu : B \mid \widetilde{p}\} \ \mid \ x : \widetilde{T} \rightarrow \widetilde{T} \qquad \text{(Gradual Types)}$$

Then, the concretization function for gradual formulas γ_p can be compatibly lifted to gradual types.

Definition 1 (Concretization of gradual types)**.** *Let the concretization function $\gamma_T : \text{GType} \rightarrow \mathcal{P}(\text{Type})$ be defined as follows:*

$$\gamma_T(\{\nu : B \mid \widetilde{p}\}) = \{\, \{\nu : B \mid p\} \mid p \in \gamma_p(\widetilde{p}) \,\}$$
$$\gamma_T(x : \widetilde{T}_1 \rightarrow \widetilde{T}_2) = \{\, x : T_1 \rightarrow T_2 \mid T_1 \in \gamma_T(\widetilde{T}_1) \wedge T_2 \in \gamma_T(\widetilde{T}_2) \,\}$$

The notion of *precision* for gradual types captures how much static information is available in a type, and by extension, in a program. As noticed by Garcia et al. (2016), precision is naturally induced by the concretization of gradual types:

Definition 2 (Precision of gradual types)**.** *$\widetilde{T}_1$ is less imprecise than $\widetilde{T}_2$, notation $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$, if and only if $\gamma_T(\widetilde{T}_1) \subseteq \gamma_T(\widetilde{T}_2)$.*

The interpretation of a gradual environment is obtained by pointwise lifting of the concretization of gradual formulas.

[2] Terms t and type environments Γ have occurrences of gradual types in them, but we do not change their notation for readability. In particular, functions that operate over the type environment Γ are unaffected by gradualization, which only affects the meaning of relations over the logical environment Φ, most notably subtyping.

Definition 3 (Concretization of gradual logical environments)**.** *Let $\gamma_\Phi : \text{GLEnv} \rightarrow \mathcal{P}(\text{LEnv})$ be defined as:*

$$\gamma_\Phi(\widetilde{\Phi}) = \{\, \Phi \mid \forall x . \Phi(x) \in \gamma_p(\widetilde{\Phi}(x)) \,\}$$

3.2 Consistent Relations

With the meaning of gradual types and logical environments, we can lift static subtyping to its consistent counterpart: consistent subtyping holds between two gradual types, in a given logical environment, if and only if static subtyping holds *for some* static types and logical environment in the respective concretizations.

Definition 4 (Consistent subtyping)**.** *$\widetilde{\Phi} \vdash \widetilde{T}_1 \mathrel{\widetilde{<:}} \widetilde{T}_2$ if and only if $\Phi \vdash T_1 <: T_2$ for some $\Phi \in \gamma_\Phi(\widetilde{\Phi}), T_1 \in \gamma_T(\widetilde{T}_1)$ and $T_2 \in \gamma_T(\widetilde{T}_2)$.*

We describe an algorithmic characterization of consistent subtyping, noted $\cdot \vdash \cdot \lesssim \cdot$, in Section 7.

The static type system also relies on a type substitution function. Following AGT, lifting type functions to operate on gradual types requires an abstraction function from sets of types to gradual types: the lifted function is defined by abstracting over all the possible results of the static function applied to all the represented static types. Instantiating this principle for type substitution:

Definition 5 (Consistent type substitution)**.**

$$\widetilde{\widetilde{T}[v/x]} = \alpha_T(\{\, T[v/x] \mid T \in \gamma_T(\widetilde{T}) \,\})$$

where α_T is the natural lifting of the abstraction for formulas α_p:

Definition 6 (Abstraction for gradual refinement types)**.** *Let $\alpha_T : \mathcal{P}(\text{Type}) \rightharpoonup \text{GType}$ be defined as:*

$$\alpha_T(\{\, \overline{\{\nu : B \mid p_i\}} \,\}) = \{\nu : B \mid \alpha_p(\{\, \overline{p_i} \,\})\}$$
$$\alpha_T(\{\, \overline{x : T_{i1} \rightarrow T_{i2}} \,\}) = x : \alpha_T(\{\, \overline{T_{i1}} \,\}) \rightarrow \alpha_T(\{\, \overline{T_{i2}} \,\})$$

The algorithmic version of consistent type substitution, noted $\cdot[\![\cdot/\cdot]\!]$, substitutes in the known parts of formulas.

3.3 Properties of the Gradual Refinement Type System

The gradual refinement type system satisfies a number of desirable properties. First, the system is a conservative extension of the underlying static system: for every fully-annotated term both systems coincide (we use $\vdash_S$ to denote the static system).

Proposition 1 (Equivalence for fully-annotated terms)**.** *For any $t \in \text{Term}$, $\Gamma;\Phi \vdash_S t : T$ if and only if $\Gamma;\Phi \vdash t : T$*

More interestingly, the system satisfies the static gradual guarantee of Siek et al. (2015): weakening the precision of a term preserves typeability, at a less precise type.

Proposition 2 (Static gradual guarantee)**.** *If $\cdot;\cdot \vdash t_1 : \widetilde{T}_1$ and $t_1 \sqsubseteq t_2$, then $\cdot;\cdot \vdash t_2 : \widetilde{T}_2$ and $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$.*

We prove both properties parametrically with respect to the actual definitions of GFormula, γ_p and α_p. The proof of Prop 1 only requires that static type information is preserved exactly, *i.e.* $\gamma_T(T) = \{\, T \,\}$ and $\alpha_T(\{\, T \,\}) = T$, which follows directly from the same properties for γ_p and α_p. These hold trivially for the different interpretations of gradual formulas we consider in the next section. The proof of Prop 2 relies on the fact that $\langle \gamma_T, \alpha_T \rangle$ is a Galois connection. Again, this follows from $\langle \gamma_p, \alpha_p \rangle$ being a Galois connection—a result we will establish in due course.

4. Interpreting Gradual Formulas

The definition of the gradual type system of the previous section is parametric over the interpretation of gradual formulas. Starting from a naive interpretation, in this section we progressively build a practical interpretation of gradual formulas. More precisely, we start in Section 4.1 with a definition of the syntax of gradual formulas, GFORMULA, and an associated concretization function γ_p, and then successively *redefine* both until reaching a satisfactory definition in Section 4.4. We then define the corresponding abstraction function α_p in Section 4.5.

We insist on the fact that any interpretation of gradual formulas that respects the conditions stated in Section 3.3 would yield a "coherent" gradual type system. Discriminating between these different possible interpretations is eventually a *design decision*, motivated by the expected behavior of a gradual refinement type system, and is hence driven by considering specific examples.

4.1 Naive Interpretation

Following the abstract interpretation viewpoint on gradual typing, a *gradual logical formula* denotes a set of possible logical formulas. As such, it can contain some statically-known logical information, as well as some additional, unknown assumptions. Syntactically, we can denote a gradual formula as either a *precise* formula (equivalent to a fully-static formula), or as an *imprecise* formula, $p \wedge ?$, where p is called its *known part*.[3]

$$\begin{array}{lll} \widetilde{p} \in \text{GFORMULA},\ p \in \text{FORMULA} & & \\ \widetilde{p} \quad ::= \quad p & & \text{(Precise Formulas)} \\ \qquad\ \mid \quad p \wedge ? & & \text{(Imprecise Formulas)} \end{array}$$

We use a conjunction in the syntax to reflect the intuition of a formula that can be made more precise by adding logical information. Note however that the symbol ? can only appear once and in a conjunction at the top level. That is, $p \vee ?$ and $p \vee (q \wedge ?)$ are not syntactically valid gradual formulas. We pose $? \overset{\text{def}}{=} \top \wedge ?$.

Having defined the syntax of gradual formulas, we must turn to their semantics. Following AGT, we give gradual formulas meaning by concretization to sets of static formulas. Here, the ? in a gradual formula $p \wedge ?$ can be understood as a placeholder for additional logical information that strengthens the known part p. A natural, but naive, definition of concretization follows.

Definition 7 (Naive concretization of gradual formulas)**.** *Let* $\gamma_p : \text{GFORMULA} \to \mathcal{P}(\text{FORMULA})$ *be defined as follows:*

$$\gamma_p(p) = \{\, p \,\} \qquad \gamma_p(p \wedge ?) = \{\, p \wedge q \mid q \in \text{FORMULA} \,\}$$

This definition is problematic. Consider a value v refined with the gradual formula $\nu \geq 2 \wedge ?$. With the above definition, we would accept passing v as argument to a function that expects a negative argument! Indeed, a possible interpretation of the gradual formula would be $\nu \geq 2 \wedge \nu = 1$, which is unsatisfiable[4] and hence trivially entails $\nu < 0$. Therefore, accepting that the unknown part of a formula denotes any arbitrary formula—including ones that contradict the known part of the gradual formula—annihilates one of the benefits of gradual typing, which is to reject such blatant inconsistencies between pieces of static information.

4.2 Non-Contradicting Interpretation

To avoid this extremely permissive behavior, we must develop a *non-contradicting interpretation* of gradual formulas. The key requirement is that when the known part of a gradual formula is satisfiable, the interpretation of the gradual formula should remain satisfiable, as captured by the following definition (we write $\texttt{SAT}(p)$ for a formula p that is satisfiable):

Definition 8 (Non-contradicting concretization of gradual formulas)**.** *Let* $\gamma_p : \text{GFORMULA} \to \mathcal{P}(\text{FORMULA})$ *be defined as:*

$$\gamma_p(p) = \{\, p \,\} \qquad \gamma_p(p \wedge ?) = \{\, p \wedge q \mid \texttt{SAT}(p) \Rightarrow \texttt{SAT}(p \wedge q) \,\}$$

[3] Given a static entity X, $\widetilde{X}$ denotes a gradual entity, and $\widehat{X}$ a set of Xs.

[4] We use the term "(un)satisfiable" instead of "(in)consistent" to avoid confusion with the term "consistency" from the gradual typing literature.

This new definition of concretization is however still problematic. Recall that a given concretization induces a natural notion of precision by relating the concrete sets (Garcia et al. 2016). Precision of gradual formulas is the key notion on top of which precision of gradual types and precision of gradual terms are built.

Definition 9 (Precision of gradual formulas)**.** $\widetilde{p}$ *is less imprecise (more precise) than* $\widetilde{q}$*, noted* $\widetilde{p} \sqsubseteq \widetilde{q}$*, if and only if* $\gamma_p(\widetilde{p}) \subseteq \gamma_p(\widetilde{q})$*.*

The non-contradicting interpretation of gradual formulas is *purely syntactic*. As such, the induced notion of precision fails to capture intuitively useful connections between programs. For instance, the sets of static formulas represented by the gradual formulas $x \geq 0 \wedge ?$ and $x > 0 \wedge ?$ are incomparable, because they are *syntactically* different. However, the gradual formula $x > 0 \wedge ?$ should intuitively refer to a more restrictive set of formulas, because the static information $x > 0$ is more *specific* than $x \geq 0$.

4.3 Semantic Interpretation

To obtain a meaningful notion of precision between gradual formulas, we appeal to the notion of *specificity* of logical formulas, which is related to the actual semantics of formulas, not just their syntax.

Formally, a formula p is more specific than a formula q if $\{\, p \,\} \models q$. Technically, this relation only defines a pre-order, because formulas that differ syntactically can be logically equivalent. As usual we work over the equivalence classes and consider equality up to logical equivalence. Thus, when we write p we actually refer to the equivalence class of p. In particular, the equivalence class of unsatisfiable formulas is represented by $\bot$, which is the bottom element of the specificity pre-order.

In order to preserve non-contradiction in our semantic interpretation of gradual formulas, it suffices to remove (the equivalence class of) $\bot$ from the concretization. Formally, we isolate $\bot$ from the specificity order, and define the order only for the satisfiable fragment of formulas, denoted SFORMULA:

Definition 10 (Specificity of satisfiable formulas)**.** *Given two formulas* $p, q \in \text{SFORMULA}$*, we say that* p *is more specific than* q *in the satisfiable fragment, notation* $p \preceq q$*, if* $\{\, p \,\} \models q$*.*

Then, we define gradual formulas such that the known part of an imprecise formula is required to be satisfiable:

$$\begin{array}{lll} \widetilde{p} \in \text{GFORMULA},\ p \in \text{FORMULA},\ p^{\checkmark} \in \text{SFORMULA} & & \\ \widetilde{p} \quad ::= \quad p & & \text{(Precise Formulas)} \\ \qquad\ \mid \quad p^{\checkmark} \wedge ? & & \text{(Imprecise Formulas)} \end{array}$$

Note that the imprecise formula $x > 0 \wedge x = 0 \wedge ?$, for example, is syntactically rejected because its known part is not satisfiable. However, $x > 0 \wedge x = 0$ is a syntactically valid formula because precise formulas are not required to be satisfiable.

The semantic definition of concretization of gradual formulas captures the fact that an imprecise formula stands for any satisfiable strengthening of its known part:

Definition 11 (Semantic concretization of gradual formulas)**.** *Let* $\gamma_p : \text{GFORMULA} \to \mathcal{P}(\text{FORMULA})$ *be defined as follows:*

$$\gamma_p(p) = \{\, p \,\} \qquad \gamma_p(p^{\checkmark} \wedge ?) = \{\, q^{\checkmark} \mid q^{\checkmark} \preceq p^{\checkmark} \,\}$$

This semantic interpretation yields a practical notion of precision that admits the judgment $x > 0 \wedge ? \sqsubseteq x \geq 0 \wedge ?$, as wanted.

Unfortunately, despite the fact that, taken in isolation, gradual formulas cannot introduce contradictions, the above definition does not yield an interesting gradual type system yet, because it allows other kinds of contradictions to sneak in. Consider the following:

$$\textbf{let}\ g\ (x\colon \{\nu\colon\mathsf{Int} \mid \nu > 0\})\ (y\colon \{\nu\colon\mathsf{Int} \mid \nu = 0 \wedge ?\}) = x/y$$

The static information $y = 0$ should suffice to statically reject this definition. But, at the use site of the division operator, the consistent subtyping judgment that must be proven is:

$$x\!:\!(\nu > 0), y\!:\!(\nu = 0 \wedge ?) \vdash \{\nu\!:\!\mathsf{Int} \mid \nu = y\} \lesssim \{\nu\!:\!\mathsf{Int} \mid \nu \neq 0\}$$

While the interpretation of the imprecise refinement of y cannot contradict $y = 0$, it can stand for $\nu = 0 \wedge x \leq 0$, which contradicts $x > 0$. Hence the definition is statically accepted.

The introduction of contradictions in the presence of gradual formulas can be even more subtle. Consider the following program:

$$\textbf{let}\ h\ (x\!: \{\nu\!:\!\mathsf{Int} \mid ?\})\ (y\!: \{\nu\!:\!\mathsf{Int} \mid ?\})\ (z\!: \{\nu\!:\!\mathsf{Int} \mid \nu = 0\}) \\ = (x + y)/z$$

One would expect this program to be rejected statically, because it is clear that $z = 0$. But, again, one can find an environment that makes consistent subtyping hold: $x : (\nu > 0), y : (\nu = x \wedge \nu < 0), z : (\nu = 0)$. This interpretation introduces a contradiction between the separate interpretations of *different* gradual formulas.

4.4 Local Interpretation

We need to restrict the space of possible static formulas represented by gradual formulas, in order to avoid contradicting already-established static assumptions, and to avoid introducing contradictions between the interpretation of different gradual formulas involved in the same consistent subtyping judgment.

Stepping back: what do refinements refine? Intuitively, the refinement type $\{\nu\!:\!B \mid p\}$ refers to all values of type B that satisfy the formula p. Note that apart from ν, the formula p can refer to other variables in scope. In the following, we use the more explicit syntax $p(\vec{x};\nu)$ to denote a formula p that constrains the refinement variable ν based on the variables in scope $\vec{x}$.

The well-formedness condition in the static system ensures that variables $\vec{x}$ on which a formula depends are in scope, but does not restrict in any way how a formula *uses* these variables. This permissiveness of traditional static refinement type systems admits curious definitions. For example, the first argument of a function can be constrained to be positive by annotating the second argument:

$$x\!:\!\mathsf{Int} \to y\!:\!\{\nu\!:\!\mathsf{Int} \mid x > 0\} \to \mathsf{Int}$$

Applying this function to some negative value is perfectly valid but yields a function that expects $\perp$. A formula can even contradict information already assumed about a prior argument:

$$x\!:\!\{\nu\!:\!\mathsf{Int} \mid \nu > 0\} \to y\!:\!\{\nu\!:\!\mathsf{Int} \mid x < 0\} \to \mathsf{Int}$$

We observe that this unrestricted freedom of refinement formulas is the root cause of the (non-local) contradiction issues that can manifest in the interpretation of gradual formulas.

Local formulas. The problem with contradictions arises from the fact that a formula $p(\vec{x};\nu)$ is allowed to express *restrictions* not just on the refinement variable ν but also on the variables in scope $\vec{x}$. In essence, we want unknown formulas to stand for any *local* restriction on the refinement variable, without allowing for contradictions with prior information on variables in scope.

Intuitively, we say that a formula is *local* if it only restricts the refinement variable ν. Capturing when a formula is local goes beyond a simple syntactic check because formulas should be able to mention variables in scope. For example, the formula $\nu > x$ is local: it restricts ν based on x without further restricting x. The key to identify $\nu > x$ as a local formula is that, for every value of x, there exists a value for ν for which the formula holds.

Definition 12 (Local formula)**.** *A formula $p(\vec{x};\nu)$ is* local *if the formula $\exists\nu.p(\vec{x};\nu)$ is valid.*

We call LFORMULA the set of local formulas. Note that the definition above implies that a local formula is satisfiable, because there must exist some ν for which the formula holds. Hence, LFORMULA $\subset$ SFORMULA $\subset$ FORMULA.

Additionally, a local formula always produces satisfiable assumptions when combined with a satisfiable logical environment:

Proposition 3. *Let Φ be a logical environment, $\vec{x} = \mathtt{dom}(\Phi)$ the vector of variables bound in Φ, and $q(\vec{x},\nu) \in$ LFORMULA. If $(\!|\Phi|\!)$ is satisfiable then $(\!|\Phi|\!) \cup \{\, q(\vec{x},\nu)\,\}$ is satisfiable.*

Moreover, we note that local formulas have the same expressiveness than non-local formulas when taken as a conjunction (we use $\equiv$ to denote logical equivalence).

Proposition 4. *Let Φ be a logical environment. If $(\!|\Phi|\!)$ is satisfiable then there exists an environment Φ' with the same domain such that $(\!|\Phi|\!) \equiv (\!|\Phi'|\!)$ and for all x the formula $\Phi'(x)$ is local.*

Similarly to what we did for the semantic interpretation, we redefine the syntax of gradual formulas such that the known part of an imprecise formula is required to be local:

$$\widetilde{p} \in \text{GFORMULA},\ p \in \text{FORMULA},\ p^\circ \in \text{LFORMULA}$$
$$\begin{array}{rcll} \widetilde{p} & ::= & p & \text{(Precise Formulas)} \\ & \mid & p^\circ \wedge ? & \text{(Imprecise Formulas)} \end{array}$$

The local concretization of gradual formulas allows imprecise formulas to denote any *local* formula strengthening the known part:

Definition 13 (Local concretization of gradual formulas)**.** *Let γ_p : GFORMULA $\to \mathcal{P}($FORMULA$)$ be defined as follows:*

$$\gamma_p(p) = \{\, p\,\} \qquad \gamma_p(p^\circ \wedge ?) = \{\, q^\circ \mid q^\circ \preceq p^\circ\,\}$$

From now on, we simply write $p \wedge ?$ for imprecise formulas, leaving implicit the fact that p is a local formula.

Examples. The local interpretation of imprecise formulas forbids the restriction of previously-defined variables. To illustrate, consider the following definition:

$$\textbf{let}\ f\ (x\!: \mathsf{Int})\ (y\!: \{\nu\!:\!\mathsf{Int} \mid ?\}) = y/x$$

The static information on x is not sufficient to prove the code safe. Because any interpretation of the unknown formula restricting y must be local, x cannot possibly be restricted to be non-zero, and the definition is rejected statically.

The impossibility to restrict previously-defined variables avoids generating contradictions and hence accepting too many programs. Recall the example of contradictions between different interpretations of imprecise formulas:

$$\textbf{let}\ h\ (x\!: \{\nu\!:\!\mathsf{Int} \mid ?\})\ (y\!: \{\nu\!:\!\mathsf{Int} \mid ?\})\ (z\!: \{\nu\!:\!\mathsf{Int} \mid \nu = 0\}) \\ = (x + y)/z$$

This definition is now rejected statically because accepting it would mean finding well-formed local formulas p and q such that the following static subtyping judgment holds:

$$x\!:\!p, y\!:\!q, z\!:\!(\nu = 0) \vdash \{\nu\!:\!\mathsf{Int} \mid \nu = z\} <: \{\nu\!:\!\mathsf{Int} \mid \nu \neq 0\}$$

However, by well-formedness, p and q cannot restrict z; and by locality, p and q cannot contradict each other.

4.5 Abstracting Formulas

Having reached a satisfactory definition of the syntax and concretization function γ_p for gradual formulas, we must now find the corresponding best abstraction α_p in order to construct the required Galois connection. We observe that, due to the definition of γ_p, specificity $\preceq$ is central to the definition of precision $\sqsubseteq$. We exploit this connection to derive a framework for abstract interpretation based on the structure of the specificity order.

The specificity order for the satisfiable fragment of formulas forms a join-semilattice. However, it does not contain a join for arbitrary (possible infinite) non-empty sets. The lack of a join for arbitrary sets, which depends on the expressiveness of the logical language, means that it is not always possible to have a best abstraction. We can however define a *partial* abstraction function, defined whenever it is possible to define a best one.

Definition 14. *Let* $\alpha_p : \mathcal{P}(\text{FORMULA}) \rightharpoonup \text{GFORMULA}$ *be the partial abstraction function defined as follows.*

$$\alpha_p(\{\, p \,\}) = p$$

$$\alpha_p(\widehat{p}\,) = \left(\curlyvee \widehat{p}\right) \wedge ? \textit{ if } \widehat{p} \subseteq \text{LFORMULA} \textit{ and } \curlyvee \widehat{p} \textit{ is defined}$$

$$\alpha_p(\widehat{p}\,) \textit{ is undefined otherwise}$$

($\curlyvee$ is the join for the specificity order in the satisfiable fragment)

The function α_p is well defined because the join of a set of local formulas is necessarily a local formula. In fact, an even stronger property holds: any upper bound of a local formula is local.

We establish that, whenever α_p is defined, it is the *best* possible abstraction that corresponds to γ_p. This characterization validates the use of specificity instead of precision in the definition of α_p.

Proposition 5 (α_p is sound and optimal)**.** *If $\alpha_p(\widehat{p})$ is defined, then $\widehat{p} \subseteq \gamma_p(\widetilde{p})$ if and only if $\alpha_p(\widehat{p}) \sqsubseteq \widetilde{p}$.*

A pair $\langle \alpha, \gamma \rangle$ that satisfies soundness and optimality is a Galois connection. However, Galois connections relate *total* functions. Here α_p is undefined whenever: (1) $\widehat{p}$ is the empty set (the join is undefined since there is no least element), (2) $\widehat{p}$ is non-empty, but contains both local and non-local formulas, or (3) $\widehat{p}$ is non-empty, and only contains local formulas, but $\curlyvee \widehat{p}$ does not exist.

Garcia et al. (2016) also define a partial abstraction function for gradual types, but the only source of partiality is the empty set. Technically, it would be possible to abstract over the empty set by adding a least element. But they justify the decision of leaving abstraction undefined based on the observation that, just as static type functions are partial, consistent functions (which are defined using abstraction) must be too. In essence, statically, abstracting the empty set corresponds to a *type error*, and dynamically, it corresponds to a *cast error*, as we will revisit in Section 5.

The two other sources of partiality of α_p cannot however be justified similarly. Fortunately, both are benign in a very precise sense: whenever we operate on sets of formulas obtained from the concretization of gradual formulas, we *never* obtain a non-empty set that cannot be abstracted. Miné (2004) generalized Galois connections to allow for *partial* abstraction functions that are always defined whenever applying some operator of interest. More precisely, given a set $\mathcal{F}$ of concrete operators, Miné defines the notion of $\langle \alpha, \gamma \rangle$ being an $\mathcal{F}$-partial Galois connection, by requiring, in addition to soundness and optimality, that the composition $\alpha \circ F \circ \gamma$ be defined for every operator $F \in \mathcal{F}$.

Abstraction for gradual types α_T is the natural extension of abstraction for gradual formulas α_p, and hence inherits its partiality. Observe that, in the static semantics of the gradual language, abstraction is only used to define the consistent type substitution operator $\cdot[\cdot/\cdot]$ (Section 3.2). We establish that, despite the partiality of α_p, the pair $\langle \alpha_T, \gamma_T \rangle$ is a partial Galois connection:

Proposition 6 (Partial Galois connection for gradual types)**.** *The pair $\langle \alpha_T, \gamma_T \rangle$ is a $\{\, \widehat{tsubst} \,\}$-partial Galois connection, where $\widehat{tsubst}$ is the collecting lifting of type substitution,* i.e.

$$\widehat{tsubst}(\widehat{T}, v, x) = \{\, T[v/x] \mid T \in \widehat{T} \,\}$$

The runtime semantics described in Sect. 5 rely on another notion of abstraction built over α_p, hence also partial, for which a similar result will be established, considering the relevant operators.

5. Abstracting Dynamic Semantics

Exploiting the correspondence between proof normalization and term reduction (Howard 1980), Garcia et al. (2016) derive the dynamic semantics of a gradual language by reduction of gradual typing derivations. This approach provides the *direct* runtime semantics of gradual programs, instead of the usual approach by translation to some intermediate cast calculus (Siek and Taha 2006).

As a term (*i.e.* and its typing derivation) reduces, it is necessary to justify new judgments for the typing derivation of the new term, such as subtyping. In a type safe static language, these new judgments can always be established, as justified in the type preservation proof, which relies on properties of judgments such as transitivity of subtyping. However, in the case of gradual typing derivations, these properties may not always hold: for instance the two *consistent* subtyping judgments $\mathsf{Int} \lesssim ?$ and $? \lesssim \mathsf{Bool}$ cannot be combined to justify the transitive judgment $\mathsf{Int} \lesssim \mathsf{Bool}$.

More precisely, Garcia et al. (2016) introduce the notion of *evidence* to characterize *why* a consistent judgment holds. A consistent operator, such as *consistent transitivity*, determines when evidences can be combined to produce evidence for a new judgment. The impossibility to combine evidences so as to justify a combined consistent judgment corresponds to a cast error: the realization, at runtime, that the plausibility based on which the program was considered (gradually) well-typed is not tenable anymore.

Compared to the treatment of (record) subtyping by Garcia et al. (2016), deriving the runtime semantics of gradual refinements presents a number of challenges. First, evidence of consistent subtyping has to account for the logical environment in the judgment (Sect. 5.1), yielding a more involved definition of the consistent subtyping transitivity operator (Sect. 5.2). Second, dependent types introduce the need for two additional consistent operators: one corresponding to the subtyping substitution lemma, accounting for substitution in types (Sect. 5.3), and one corresponding to the lemma that substitution in terms preserves typing (Sect. 5.4).

Section 6 presents the resulting runtime semantics and the properties of the gradual refinement language.

5.1 Evidence for Consistent Subtyping

Evidence represents the plausible static types that support some consistent judgment. Consider the valid consistent subtyping judgment $x\!:\!? \vdash \{\nu\!:\!\mathsf{Int} \mid \nu = x\} \lesssim \{\nu\!:\!\mathsf{Int} \mid \nu > 0\}$. In addition to knowing *that* it holds, we know *why* it holds: for any satisfying interpretation of the gradual environment, x should be refined with a formula ensuring that it is positive. That is, we can deduce precise bounds on the set of static entities that supports why the consistent judgment holds. The abstraction of these static entities is what Garcia et al. (2016) call *evidence*.

Because a consistent subtyping judgment involves a gradual environment and two gradual types, we extend the abstract interpretation framework coordinate-wise to *subtyping tuples*:[5]

Definition 15 (Subtyping tuple concretization)**.** *Let* $\gamma_\tau : \text{GTUPLE}^{<:} \to \mathcal{P}(\text{TUPLE}^{<:})$ *be defined as:*

$$\gamma_\tau(\widetilde{\Phi}, \widetilde{T}_1, \widetilde{T}_2) = \gamma_\Phi(\widetilde{\Phi}) \times \gamma_T(\widetilde{T}_1) \times \gamma_T(\widetilde{T}_2)$$

Definition 16 (Subtypting tuple abstraction)**.** *Let* $\alpha_\tau : \mathcal{P}(\text{TUPLE}^{<:}) \rightharpoonup \text{GTUPLE}^{<:}$ *be defined as:*

$$\alpha_\tau(\{\, \overline{\Phi_i, T_{i1}, T_{i2}} \,\}) = \langle \alpha_\Phi(\{\, \overline{\Phi_i} \,\}), \alpha_T(\{\, \overline{T_{i1}} \,\}), \alpha_T(\{\, \overline{T_{i2}} \,\}) \rangle$$

This definition uses abstraction of gradual logical environments.

Definition 17 (Abstraction for gradual logical environments)**.** *Let* $\alpha_\Phi : \mathcal{P}(\text{ENV}) \rightharpoonup \text{GENV}$ *be defined as:*

$$\alpha_\Phi(\widehat{\Phi})(x) = \alpha_p(\{\, \Phi(x) \mid \Phi \in \widehat{\Phi} \,\})$$

[5] We pose $\tau \in \text{TUPLE}^{<:} = \text{LENV} \times \text{TYPE} \times \text{TYPE}$ for subtyping tuples, and $\text{GTUPLE}^{<:} = \text{GLENV} \times \text{GTYPE} \times \text{GTYPE}$ for their gradual lifting.

We can now define the *interior* of a consistent subtyping judgment, which captures the best coordinate-wise information that can be deduced from knowing that such a judgment holds.

Definition 18 (Interior)**.** *The interior of the judgment* $\widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2$*, notation* $\mathcal{I}_{<:}(\widetilde{\Phi}, \widetilde{T}_1, \widetilde{T}_2)$ *is defined by the function* $\mathcal{I}_{<:} : \text{GTUPLE}^{<:} \to \text{GTUPLE}^{<:}$*:*

$$\mathcal{I}_{<:}(\widehat{\tau}) = \alpha_\tau(F_{\mathcal{I}_{<:}}(\gamma_\tau(\widehat{\tau})))$$

where $F_{\mathcal{I}_{<:}} : \mathcal{P}(\text{TUPLE}^{<:}) \to \mathcal{P}(\text{TUPLE}^{<:})$

$$F_{\mathcal{I}_{<:}}(\widehat{\tau}) = \{ \langle \Phi, T_1, T_2 \rangle \in \widehat{\tau} \mid \Phi \vdash T_1 <: T_2 \}$$

Based on interior, we define what counts as *evidence* for consistent subtyping. Evidence is represented as a tuple in $\text{GTUPLE}^{<:}$ that abstracts the possible satisfying static tuples. The tuple is *self-interior* to reflect the most precise information available:

Definition 19 (Evidence for consistent subtyping)**.** $\text{Ev}^{<:} = \{ \langle \widetilde{\Phi}, \widetilde{T}_1, \widetilde{T}_2 \rangle \in \text{GTUPLE}^{<:} \mid \mathcal{I}_{<:}(\widetilde{\Phi}, \widetilde{T}_1, \widetilde{T}_2) = \langle \widetilde{\Phi}, \widetilde{T}_1, \widetilde{T}_2 \rangle \}$

We use metavariable ε to range over $\text{Ev}^{<:}$, and introduce the extended judgment $\varepsilon \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2$, which associates particular runtime evidence to some consistent subtyping judgment. Initially, before a program executes, evidence ε corresponds to the interior of the judgment, also called the *initial evidence* (Garcia et al. 2016).

The abstraction function α_τ inherits the partiality of α_p. We prove that $\langle \alpha_\tau, \gamma_\tau \rangle$ is a partial Galois connection for every operator of interest, starting with $F_{\mathcal{I}_{<:}}$, used in the definition of interior:

Proposition 7 (Partial Galois connection for interior)**.** *The pair* $\langle \alpha_\tau, \gamma_\tau \rangle$ *is a* $\{ F_{\mathcal{I}_{<:}} \}$*-partial Galois connection.*

5.2 Consistent Subtyping Transitivity

The initial gradual typing derivation of a program uses initial evidence for each consistent judgment involved. As the program executes, evidence can be combined to exhibit evidence for other judgments. The way evidence evolves to provide evidence for further judgments mirrors the type safety proof, and justifications supported by properties about the relationship between static entities.

As noted by Garcia et al. (2016), a crucial property used in the proof of preservation is transitivity of subtyping, which may or may not hold in the case of consistent subtyping judgments, because of the imprecision of gradual types. For instance, both $\cdot \vdash \{\nu{:}\mathsf{Int} \mid \nu > 10\} \lesssim \{\nu{:}\mathsf{Int} \mid ?\}$ and $\cdot \vdash \{\nu : \mathsf{Int} \mid ?\} \lesssim \{\nu : \mathsf{Int} \mid \nu < 10\}$ hold, but $\cdot \vdash \{\nu{:}\mathsf{Int} \mid \nu > 10\} \lesssim \{\nu{:}\mathsf{Int} \mid \nu < 10\}$ does not.

Following AGT, we can formally define how to combine evidences to provide justification for *consistent subtyping*.

Definition 20 (Consistent subtyping transitivity)**.** *Suppose:*

$$\varepsilon_1 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2 \qquad \varepsilon_2 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_2 \lesssim \widetilde{T}_3$$

We deduce evidence for consistent subtyping transitivity *as*

$$(\varepsilon_1 \circ^{<:} \varepsilon_2) \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_3$$

where $\circ^{<:} : \text{Ev}^{<:} \to \text{Ev}^{<:} \rightharpoonup \text{Ev}^{<:}$ *is defined by:*

$$\varepsilon_1 \circ^{<:} \varepsilon_2 = \alpha_\tau(F_{\circ^{<:}}(\gamma_\tau(\varepsilon_1), \gamma_\tau(\varepsilon_2)))$$

and $F_{\circ^{<:}} : \mathcal{P}(\text{TUPLE}^{<:}) \to \mathcal{P}(\text{TUPLE}^{<:}) \to \mathcal{P}(\text{TUPLE}^{<:})$ *is:*

$$F_{\circ^{<:}}(\widehat{\tau}_1, \widehat{\tau}_2) = \{ \langle \Phi, T_1, T_3 \rangle \mid \exists T_2.\ \langle \Phi, T_1, T_2 \rangle \in \widehat{\tau}_1 \wedge \\ \langle \Phi, T_2, T_3 \rangle \in \widehat{\tau}_2 \wedge \Phi \vdash T_1 <: T_2 \wedge \Phi \vdash T_2 <: T_3 \}$$

The consistent transitivity operator collects and abstracts all available justifications that transitivity might hold in a particular instance. Consistent transitivity is a partial function: if $F_{\circ^{<:}}$ produces an empty set, α_τ is undefined, and the transitive claim has been refuted. Intuitively this corresponds to a runtime cast error.

Consider, for example, the following evidence judgments:

$$\varepsilon_1 \triangleright \cdot \vdash \{\nu{:}\mathsf{Int} \mid \nu > 0 \wedge ?\} \lesssim \{\nu{:}\mathsf{Int} \mid ?\} \\ \varepsilon_2 \triangleright \cdot \vdash \{\nu{:}\mathsf{Int} \mid ?\} \lesssim \{\nu{:}\mathsf{Int} \mid \nu < 10\}$$

where

$$\varepsilon_1 = \langle \cdot, \{\nu{:}\mathsf{Int} \mid \nu > 0 \wedge ?\}, \{\nu{:}\mathsf{Int} \mid ?\} \rangle \\ \varepsilon_2 = \langle \cdot, \{\nu{:}\mathsf{Int} \mid \nu < 10 \wedge ?\}, \{\nu{:}\mathsf{Int} \mid \nu < 10\} \rangle$$

Using consistent subtyping transitivity we can deduce evidence for the judgment:

$$(\varepsilon_1 \circ^{<:} \varepsilon_2) \triangleright \cdot \vdash \{\nu{:}\mathsf{Int} \mid \nu > 0 \wedge ?\} \lesssim \{\nu{:}\mathsf{Int} \mid \nu < 10\}$$

where

$$\varepsilon_1 \circ^{<:} \varepsilon_2 = \langle \cdot, \{\nu{:}\mathsf{Int} \mid \nu > 0 \wedge \nu < 10 \wedge ?\}, \{\nu{:}\mathsf{Int} \mid \nu < 10\} \rangle$$

As required, $\langle \alpha_\tau, \gamma_\tau \rangle$ is a partial Galois connection for the operator used to define consistent subtyping transitivity.

Proposition 8 (Partial Galois connection for transitivity)**.** *The pair* $\langle \alpha_\tau, \gamma_\tau \rangle$ *is a* $\{ F_{\circ^{<:}} \}$*-partial Galois connection.*

5.3 Consistent Subtyping Substitution

The proof of type preservation for refinement types also relies on a subtyping substitution lemma, stating that a subtyping judgment is preserved after a value is substituted for some variable x, and the binding for x is removed from the logical environment:

$$\frac{\Gamma; \Phi_1 \vdash v : T_{11} \qquad \Phi_1 \vdash T_{11} <: T_{12} \qquad \Phi_1, x{:}(\!|T_{12}|\!), \Phi_2 \vdash T_{21} <: T_{22}}{\Phi_1, \Phi_2[v/x] \vdash T_{21}[v/x] <: T_{22}[v/x]}$$

In order to justify reductions of gradual typing derivations, we need to define an operator of *consistent subtyping substitution* that combines evidences from consistent subtyping judgments in order to derive evidence for the consistent subtyping judgment between types after substitution of v for x.

Definition 21 (Consistent subtyping substitution)**.** *Suppose:*

$$\Gamma; \widetilde{\Phi}_1 \vdash v : \widetilde{T}_{11} \quad \varepsilon_1 \triangleright \widetilde{\Phi}_1 \vdash \widetilde{T}_{11} \lesssim \widetilde{T}_{12} \\ \varepsilon_2 \triangleright \widetilde{\Phi}_1, x{:}(\!|\widetilde{T}_{12}|\!), \widetilde{\Phi}_2 \vdash \widetilde{T}_{21} \lesssim \widetilde{T}_{22}$$

Then we deduce evidence for consistent subtyping substitution *as*

$$(\varepsilon_1 \circ^{[v/x]}_{<:} \varepsilon_2) \triangleright \widetilde{\Phi}_1, \widetilde{\Phi}_2[\![v/x]\!] \vdash \widetilde{T}_{21}[\![v/x]\!] \lesssim \widetilde{T}_{22}[\![v/x]\!]$$

where $\circ^{[v/x]}_{<:} : \text{Ev}^{<:} \to \text{Ev}^{<:} \rightharpoonup \text{Ev}^{<:}$ *is defined by:*

$$\varepsilon_1 \circ^{[v/x]}_{<:} \varepsilon_2 = \alpha_\tau(F_{\circ^{[v/x]}_{<:}}(\gamma_\tau(\varepsilon_1), \gamma_\tau(\varepsilon_2)))$$

and $F_{\circ^{[v/x]}_{<:}} : \mathcal{P}(\text{TUPLE}^{<:}) \to \mathcal{P}(\text{TUPLE}^{<:}) \to \mathcal{P}(\text{TUPLE}^{<:})$ *is:*

$$F_{\circ^{[v/x]}_{<:}}(\widehat{\tau}_1, \widehat{\tau}_2) = \{ \langle \Phi_1 \cdot \Phi_2[v/x], T_{21}[v/x], T_{22}[v/x] \rangle \mid \\ \exists T_{11}, T_{12}.\ \langle \Phi_1, T_{11}, T_{12} \rangle \in \widehat{\tau}_1 \wedge \\ \langle \Phi_1 \cdot x{:}(\!|T_{12}|\!) \cdot \Phi_2, T_{21}, T_{22} \rangle \in \widehat{\tau}_2 \wedge \\ \Phi_1 \vdash T_{11} <: T_{12} \wedge \Phi_1 \cdot x{:}(\!|T_{12}|\!) \cdot \Phi_2 \vdash T_{21} <: T_{22} \}$$

The consistent subtyping substitution operator collects and abstracts all justifications that some consistent subtyping judgment holds after substituting in types with a value, and produces the most precise evidence, if possible. Note that this new operator introduces a new category of runtime errors, made necessary by dependent types, and hence not considered in the simply-typed setting of Garcia et al. (2016).

To illustrate consistent subtyping substitution consider:

$$\cdot; \cdot \vdash 3 : \{\nu{:}\mathsf{Int} \mid \nu = 3\} \\ \varepsilon_1 \triangleright \cdot \vdash \{\nu{:}\mathsf{Int} \mid \nu = 3\} \lesssim \{\nu{:}\mathsf{Int} \mid ?\} \\ \varepsilon_2 \triangleright x{:}?, y{:}? \vdash \{\nu{:}\mathsf{Int} \mid \nu = x + y\} \lesssim \{\nu{:}\mathsf{Int} \mid \nu \geq 0\}$$

where

$$\varepsilon_1 = \langle \cdot, \{\nu:\mathsf{Int} \mid \nu = 3\}, \{\nu:\mathsf{Int} \mid ?\}\rangle$$
$$\varepsilon_2 = \langle x:?\cdot y:?, \{\nu:\mathsf{Int} \mid \nu = x + y\}, \{\nu:\mathsf{Int} \mid \nu \geq 0\}\rangle$$

We can combine ε_1 and ε_2 with the consistent subtyping substitution operator to justify the judgment after substituting 3 for x:

$$(\varepsilon_1 \circ^{[3/x]}_{<:} \varepsilon_2) \triangleright y:? \vdash \{\nu:\mathsf{Int} \mid \nu = 3 + y\} \lesssim \{\nu:\mathsf{Int} \mid \nu \geq 0\}$$

where

$$\varepsilon_1 \circ^{[3/x]}_{<:} \varepsilon_2 = \langle y:\nu \geq -3 \land ?, \{\nu:\mathsf{Int} \mid \nu = 3 + y\}, \{\nu:\mathsf{Int} \mid \nu \geq 0\}\rangle$$

Proposition 9 (Partial Galois connection for subtyping substitution)**.** *The pair $\langle \alpha_\tau, \gamma_\tau \rangle$ is a $\{\, F_{\circ^{[v/x]}_{<:}} \,\}$-partial Galois connection.*

5.4 Consistent Term Substitution

Another important aspect of the proof of preservation is the use of a term substitution lemma, *i.e.* substituting in an open term preserves typing. Even in the simply-typed setting considered by Garcia et al. (2016), the term substitution lemma does not hold for the gradual language because it relies on subtyping transitivity. Without further discussion, they adopt a simple technique: instead of substituting a plain value v for the variable x, they substitute an *ascribed* value $v :: \widetilde{T}$, where $\widetilde{T}$ is the expected type of x. This technique ensures that the substitution lemma always holds.

With dependent types, the term substitution lemma is more challenging. A subtyping judgment can rely on the plausibility that a gradually-typed variable is replaced with the right value, which may not be the case at runtime. Consider the following example:

$$\begin{array}{l}\mathbf{let}\ f\ (x\!: \{\nu:\mathsf{Int} \mid \nu > 0\}) = x \\ \mathbf{let}\ g\ (x\!: \{\nu:\mathsf{Int} \mid ?\})\ (y\!: \{\nu:\mathsf{Int} \mid \nu \geq x\}) = \\ \quad \mathbf{let}\ z = f\ y\ \mathbf{in}\ z + x\end{array}$$

This code is accepted statically due to the possibility of x being positive inside the body of g. If we call g with -1 the application $f\ y$ can no longer be proven possibly safe. Precisely, the application $f\ y$ relies on the consistent subtyping judgment $x:?\cdot y:\nu \geq x \vdash \{\nu:\mathsf{Int} \mid \nu = y\} \lesssim \{\nu:\mathsf{Int} \mid \nu > 0\}$ supported by the evidence $\langle x:\nu > 0 \land ?\cdot y:\nu \geq x, \{\nu:\mathsf{Int} \mid \nu = y\}, \{\nu:\mathsf{Int} \mid \nu > 0\}\rangle$. After substituting by -1 the following judgment must be justified: $y:\nu \geq -1 \vdash \{\nu:\mathsf{Int} \mid \nu = y\} \lesssim \{\nu:\mathsf{Int} \mid \nu > 0\}$. This (fully precise) judgment cannot however be supported by any evidence.

Note that replacing by an ascribed value does not help in the dependently-typed setting because, as illustrated by the previous example, judgments that must be proven after substitution may not even correspond to syntactic occurrences of the replaced variable. Moreover, substitution also pervades types, and consequently formulas, but the logical language has no notion of ascription.

Stepping back, the key characteristic of the ascription technique used by Garcia et al. (2016) is that the resulting substitution operator on gradual terms preserves exact types. Noting that after substitution some consistent subtyping judgments may fail, we define a *consistent term substitution* operator that preserves typeability, but is undefined if it cannot produce evidence for some judgment. This yields yet another category of runtime failure, occurring at substitution time. In the above example, the error manifests as soon as the application $g\ {-1}$ beta-reduces, before reducing the body of g.

Consistent term substitution relies on the consistent subtyping substitution operator defined in Section 5.2 to produce evidence for consistent subtyping judgments that result from substitution. We defer its exact definition to Section 6.3 below.

6. Dynamic Semantics and Properties

We now turn to the actual reduction rules of the gradual language with refinement types. Following AGT, reduction is expressed over gradual *typing derivations*, using the consistent operators mentioned in the previous section. Because writing down reduction rules over (bidimensional) derivation trees is unwieldy, we use *instrincally-typed terms* (Church 1940) as a convenient unidimensional notation for derivation trees (Garcia et al. 2016).

We expose this notational device in Section 6.1, and then use it to present the reduction rules (Section 6.2) and the definition of the consistent term substitution operator (Section 6.3). Finally, we state the meta-theoretic properties of the resulting language: type safety, gradual guarantee, and refinement soundness (Section 6.4).

$$\text{(Ix-refine)}\ \frac{}{\widetilde{\Phi}\ ;\ x^{\{\nu:B \mid \widetilde{p}\}} \in \mathrm{TERM}_{\{\nu:B \mid \nu = x\}}}$$

$$\text{(Ix-fun)}\ \frac{}{\widetilde{\Phi}\ ;\ x^{y:\widetilde{T}_1 \to \widetilde{T}_2} \in \mathrm{TERM}_{y:\widetilde{T}_1 \to \widetilde{T}_2}}$$

$$\text{(I}\lambda\text{)}\ \frac{\widetilde{\Phi}, x:(\!|\widetilde{T}_1|\!)\ ;\ t^{\widetilde{T}_2} \in \mathrm{TERM}_{\widetilde{T}_2}}{\widetilde{\Phi}\ ;\ \lambda x^{\widetilde{T}_1}.t^{\widetilde{T}_2} \in \mathrm{TERM}_{x:\widetilde{T}_1 \to \widetilde{T}_2}}$$

$$\text{(Iapp)}\ \frac{\begin{array}{cc}\widetilde{\Phi}\ ;\ t^{\widetilde{T}_1} \in \mathrm{TERM}_{\widetilde{T}_1} & \varepsilon_1 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim (x:\widetilde{T}_{11} \to \widetilde{T}_{12}) \\ \widetilde{\Phi}\ ;\ v \in \mathrm{TERM}_{\widetilde{T}_2} & \varepsilon_2 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}\end{array}}{\widetilde{\Phi}\ ;\ (\varepsilon_1 t^{\widetilde{T}_1})@^{x:\widetilde{T}_{11} \to \widetilde{T}_{12}}\ (\varepsilon_2 v) \in \mathrm{TERM}_{\widetilde{T}_{12}[\![v/x]\!]}}$$

Figure 3. Gradual intrinsic terms (selected rules)

6.1 Intrinsic Terms

We first develop gradual intrinsically-typed terms, or gradual intrinsic terms for short. Intrinsic terms are isomorphic to typing derivation trees, so their structure corresponds to the gradual typing judgment $\Gamma; \widetilde{\Phi} \vdash t : \widetilde{T}$—a term is given a type in a specific type environment and gradual logical environment. Intrinsic terms are built up from disjoint families of intrinsically-typed variables $x^{\widetilde{T}} \in \mathrm{VAR}_{\widetilde{T}}$. Because these variables carry type information, type environments Γ are not needed in intrinsic terms. Because typeability of a term depends on its logical context, we define a family $\mathrm{TERM}^{\widetilde{\Phi}}_{\widetilde{T}}$ of sets indexed by both types and gradual logical environments. For readability, we use the notation $\widetilde{\Phi}\ ;\ t^{\widetilde{T}} \in \mathrm{TERM}_{\widetilde{T}}$, allowing us to view an intrinsic term as made up of a logical environment and a term (when $\widetilde{\Phi}$ is empty we stick to $\mathrm{TERM}^{\cdot}_{\widetilde{T}}$).

Figure 3 presents selected formation rules of intrinsic terms. Rules (Ix-refine) and (Ix-fun) are straightforward. Rule (Iλ) requires the body of the lambda to be typed in an extended logical environment. Note that because gradual typing derivations include evidence for consistent judgments, gradual intrinsic terms carry over evidences as well, which can be seen in rule (Iapp). The rule for application additionally features a type annotation with the @ notation. As observed by Garcia et al. (2016), this annotation is necessary because intrinsic terms represent typing derivations *at different steps of reduction*. Therefore, they must account for the fact that runtime values can have more precise types than the ones determined statically. For example, a term t in function position of an application may reduce to some term whose type is a subtype of the type given to t statically. An intrinsic application term hence carries the type given statically to the subterm in function position.

6.2 Reduction

Figure 4 presents the syntax of the intrinsic term language and its evaluation frames, in the style of Garcia et al. (2016). Values v are either raw values u or ascribed values $\varepsilon u :: \widetilde{T}$, where ε is the

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

$$
\begin{array}{rcl}
et & ::= & \varepsilon t\\
ev & ::= & \varepsilon u\\
u & ::= & x^* \mid n \mid b \mid \lambda x^*.t^*\\
v & ::= & u \mid \varepsilon u :: \widetilde{T}\\
g & ::= & \square \mathbin{@^{\widetilde{T}}} et \mid ev \mathbin{@^{\widetilde{T}}} \square \mid \square :: \widetilde{T} \mid (\mathsf{let}\ x = \square \mathbin{@^{\widetilde{T}}} \mathsf{in}\ et) @^{\widetilde{T}}\\
f & ::= & g[\varepsilon\square]
\end{array}
$$

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

$$
\varepsilon_1(\lambda x^{\widetilde{T}_{11}}.t) @^{x:\widetilde{T}_1 \rightarrow \widetilde{T}_2} \varepsilon_2 u \longrightarrow
$$

$$
\begin{cases}
icod_u(\varepsilon_2, \varepsilon_1) t[(\varepsilon_2 \circ^{<:} idom(\varepsilon_1))u/x^{\widetilde{T}_{11}}] :: \widetilde{T}_2\llbracket u/x \rrbracket\\
\mathbf{error} \quad \text{if } (\varepsilon_2 \circ^{<:} idom(\varepsilon_1)), icod_u(\varepsilon_2, \varepsilon_1) \text{ or}\\
\qquad\quad t[\varepsilon_u u/x^{\widetilde{T}_{11}}] \text{ is not defined}
\end{cases}
$$

$$
(\mathsf{let}\ x^{\widetilde{T}_1} = \varepsilon_1 u\ \mathsf{in}\ \varepsilon_2 t) @^{\widetilde{T}} \longrightarrow
\begin{cases}
(\varepsilon_1 \circ^{[v/x]}_{<:} \varepsilon_2) t[\varepsilon_1 u/x^{\widetilde{T}_1}] :: \widetilde{T}\\
\mathbf{error} \quad \text{if } t[\varepsilon_1 u/x^{\widetilde{T}_1}] \text{ or}\\
\quad (\varepsilon_1 \circ^{[v/x]}_{<:} \varepsilon_2) \text{ is not defined}
\end{cases}
$$

$$
(\mathsf{if\ true\ then}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{else}\ \varepsilon_2 t^{\widetilde{T}_2}) @^{\widetilde{T}} \longrightarrow \varepsilon_1 t^{\widetilde{T}_1} :: \widetilde{T}
$$

$$
(\mathsf{if\ false\ then}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{else}\ \varepsilon_2 t^{\widetilde{T}_2}) @^{\widetilde{T}} \longrightarrow \varepsilon_2 t^{\widetilde{T}_2} :: \widetilde{T}
$$

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

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

$$\boxed{\longmapsto : \textsc{Term}^{\boldsymbol{\cdot}}_{\widetilde{T}} \times (\textsc{Term}^{\boldsymbol{\cdot}}_{\widetilde{T}} \cup \{\, \mathbf{error} \,\})}$$

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

$$
(\mathrm{R}g)\ \frac{et \longrightarrow_c et'}{g[et] \longmapsto g[et']}
\qquad\qquad
(\mathrm{R}f)\ \frac{t_1^{\widetilde{T}} \longmapsto t_2^{\widetilde{T}}}{f[t_1^{\widetilde{T}}] \longmapsto f[t_2^{\widetilde{T}}]}
$$

Figure 4. Intrinsic reduction (error propagation rules omitted)

evidence that u is of a subtype of $\widetilde{T}$. Such a pair $\varepsilon u \in \textsc{EvValue}$ is called an *evidence value*. Similarly, an *evidence term* $\varepsilon t \in \textsc{EvTerm}$ is a term augmented with evidence. We use $\textsc{Var}_*$ (resp. $\textsc{Term}^*_*$) to denote the set of all intrinsic variables (resp. terms).

Figure 4 presents the reduction relation $\longmapsto$ and the two notions of reductions $\longrightarrow$ and $\longrightarrow_c$. Reduction rules preserve the exact type of a term and explicit ascriptions are used whenever a reduction may implicitly affect type precision. The rules handle evidences, combining them with consistent operators to derive new evidence to form new intrinsic terms. Whenever combining evidences fails, the program ends with an **error**. An application may produce an error because it cannot produce evidence using consistent transitivity to justify that the actual argument is subtype of the formal argument. Additionally, the rules for application and let expression use consistent term substitution, which fails whenever consistent subtyping substitution cannot combine evidences to justify all substituted occurrences.

6.3 Consistent Term Substitution

The consistent term substitution operator described in Section 5.4 is defined on intrinsic terms (Figure 5). To substitute a variable $x^{\widetilde{T}}$ by a value u we must have evidence justifying that the type of u

$$\boxed{(\cdot)[\cdot/\cdot] : \textsc{Term}^*_* \rightarrow \textsc{EvValue} \rightarrow \textsc{Var}_* \rightharpoonup \textsc{Term}^*_*}$$

$$
x^{\{\nu:B \,|\, \widetilde{p}\}}[\varepsilon u/x^{\{\nu:B \,|\, \widetilde{p}\}}] = u \qquad y^{\widetilde{T}_2}[\varepsilon u/x^{\widetilde{T}_1}] = y^{\widetilde{T}\llbracket u/x \rrbracket} \quad \text{if } x^{\widetilde{T}_1} \neq y^{\widetilde{T}_2}
$$

$$
x^{y:\widetilde{T}_1 \rightarrow \widetilde{T}_2}[\varepsilon u/x^{y:\widetilde{T}_1 \rightarrow \widetilde{T}_2}] = \varepsilon u :: (y : \widetilde{T}_1 \rightarrow \widetilde{T}_2)
$$

$$
(\lambda y^{\widetilde{T}}.t)[\varepsilon u/x^{\widetilde{T}}] = \lambda y^{\widetilde{T}\llbracket u/x \rrbracket}.t[\varepsilon u/x^{\widetilde{T}}]
$$

$$
\begin{array}{l}
((\varepsilon_1 t^{\widetilde{T}_1}) @^{x:\widetilde{T}_{11} \rightarrow \widetilde{T}_{12}} (\varepsilon_2 v))[\varepsilon u/x^{\widetilde{T}}] =\\
\qquad (\varepsilon_1 t^{\widetilde{T}_1})[\varepsilon u/x^{\widetilde{T}}] @^{(x:\widetilde{T}_{11} \rightarrow \widetilde{T}_{12})\llbracket u/x \rrbracket} (\varepsilon_2 v)[\varepsilon u/x^{\widetilde{T}}]
\end{array}
$$

$$\boxed{(\cdot)[\cdot/\cdot] : \textsc{EvTerm} \rightarrow \textsc{EvValue} \rightarrow \textsc{Var}_* \rightharpoonup \textsc{EvTerm}}$$

$$
(\varepsilon_1 t^{\widetilde{T}_2})[\varepsilon u/x^{\widetilde{T}_1}] = (\varepsilon \circ^{[u/x]}_{<:} \varepsilon_1) t^{\widetilde{T}_2}[\varepsilon u/x^{\widetilde{T}_1}]
$$

Figure 5. Consistent term substitution (selected cases)

is a subtype of $\widetilde{T}$, supporting that substituting by u may be safe. Therefore, consistent term substitution is defined for evidence values. The consistent term substitution operator recursively traverses the structure of an intrinsic term applying consistent subtyping substitution to every evidence, using an auxiliary definition for substitution into evidence terms. When substituting by an evidence value $\varepsilon_1 u$ in an evidence term $\varepsilon_2 t$, we first combine ε_1 and ε_2 using consistent subtyping substitution and then substitute recursively into t. Note that substitution is undefined whenever consistent subtyping substitution is undefined.

When reaching a variable, there is a subtle difference between substituting by a lambda and a base constant. Because variables with base types are given the exact type $\{\nu : B \mid \nu = x\}$, after substituting x by a value u the type becomes $\{\nu : B \mid \nu = u\}$, which exactly corresponds to the type for a base constant. For higher order variables an explicit ascription is needed to preserve the same type. Another subtlety is that types appearing in annotations above @ must be replaced by the same type, but substituting for the variable x being replaced. This is necessary for the resulting term to be well-typed in an environment where the binding for the substituted variable has been removed from the logical environment. Similarly an intrinsic variable $y^{\widetilde{T}}$ other than the one being replaced must be replaced by a variable $y^{\widetilde{T}\llbracket u/x \rrbracket}$.

The key property is that consistent term substitution preserves typeability whenever it is defined.

Proposition 10 (Consistent substitution preserves types)**.** *Suppose* $\widetilde{\Phi}_1$; $u \in \textsc{Term}_{\widetilde{T}_u}$, $\varepsilon \triangleright \widetilde{\Phi}_1 \vdash \widetilde{T}_u \lesssim \widetilde{T}_x$, *and* $\widetilde{\Phi}_1 \cdot x : (\!|\widetilde{T}_x|\!) \cdot \widetilde{\Phi}_2$; $t \in \textsc{Term}_{\widetilde{T}}$ *then* $\widetilde{\Phi}_1 \cdot \widetilde{\Phi}_2 \llbracket u/x \rrbracket$; $t[\varepsilon u/x^{\widetilde{T}_x}] \in \textsc{Term}_{\widetilde{T}\llbracket u/x \rrbracket}$ *or* $t[\varepsilon u/x^{\widetilde{T}_x}]$ *is undefined.*

6.4 Properties of the Gradual Refinement Types Language

We establish three fundamental properties based on the dynamic semantics. First, the gradual language is type safe by construction.

Proposition 11 (Type Safety)**.** *If* $t_1^{\widetilde{T}} \in \textsc{Term}^{\boldsymbol{\cdot}}_{\widetilde{T}}$ *then either* $t_1^{\widetilde{T}}$ *is a value* v, $t_1^{\widetilde{T}} \longmapsto t_2^{\widetilde{T}}$ *for some term* $t_2^{\widetilde{T}} \in \textsc{Term}^{\boldsymbol{\cdot}}_{\widetilde{T}}$, *or* $t_1^{\widetilde{T}} \longmapsto$ **error**.

More interestingly, the language satisfies the dynamic gradual guarantee of Siek et al. (2015): a well-typed gradual program that runs without errors still does with less precise type annotations.

Proposition 12 (Dynamic gradual guarantee)**.** *Suppose* $t_1^{\widetilde{T}_1} \sqsubseteq t_1^{\widetilde{T}_2}$. *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* $t_2^{\widetilde{T}_1} \sqsubseteq t_2^{\widetilde{T}_2}$.

We also establish refinement soundness: the result of evaluating a term yields a value that complies with its refinement. This property is a direct consequence of type preservation.

Proposition 13 (Refinement soundness)**.**

If $t^{\{\nu:B\,|\,\widetilde{p}\}} \in \text{TERM}^{\cdot}_{\{\nu:B\,|\,\widetilde{p}\}}$ *and* $t^{\{\nu:B\,|\,\widetilde{p}\}} \longmapsto^* v$ *then:*

1. *If* $v = u$ *then* $(\!|\widetilde{p}|\!)_!\,[u/\nu]$ *is valid*
2. *If* $v = \varepsilon u :: \{\nu : B \mid \widetilde{p}\}$ *then* $(\!|\widetilde{p}|\!)_!\,[u/\nu]$ *is valid*

where $(\!|\widetilde{p}|\!)_!$ extracts the static part of $\widetilde{p}$.

7. Algorithmic Consistent Subtyping

Having defined a gradually-typed language with refinements that satisfies the expected meta-theoretic properties (Sect. 3.3 and 6.4), we turn to its decidability. The abstract interpretation framework does not immediately yield algorithmic definitions. While some definitions can be easily characterized algorithmically, consistent subtyping (Sect. 3.2) is both central and particularly challenging. We now present a syntax-directed characterization of consistent subtyping, which is a decision procedure when refinements are drawn from the theory of linear arithmetic.

The algorithmic characterization is based on solving *consistent entailment constraints* of the form $\widetilde{\Phi} \approx\!\!\models \widetilde{q}$. Solving such a constraint consists in finding a well-formed environment $\Phi \in \gamma_\Phi(\widetilde{\Phi})$ and a formula $q \in \gamma_p(\widetilde{q})$ such that $(\!|\Phi|\!) \models q$. We use the notation $\approx\!\!\models$ to mirror $\models$ in a consistent fashion. However, note that $\approx\!\!\models$ does not correspond to the consistent lifting of $\models$, because entailment is defined for sets of formulas while consistent entailment is defined for (ordered) gradual logical environments. This is important to ensure well-formedness of logical environments.

As an example consider the consistent entailment constraint:

$$x\!:?, y\!:?, z\!:(\nu \geq 0) \;\approx\!\!\models\; x + y + z \geq 0 \wedge x \geq 0 \wedge ? \qquad (1)$$

First, note that the unknown on the right hand side can be obviated, so to solve the constraint we must find formulas that restrict the possible values of x and y such that $x + y + z \geq 0 \wedge x \geq 0$ is always true. There are many ways to achieve this; we are only concerned about the *existence* of such an environment.

We describe a canonical approach to determine whether a consistent entailment is valid, by reducing it to a fully static judgment.[6] Let us illustrate how to reduce constraint (1) above. We first focus on the rightmost gradual formula in the environment, for y, and consider a static formula that guarantees the goal, using the static information further right. Here, this means binding y to $\forall z.z \geq 0 \to (x + \nu + z \geq 0 \wedge x \geq 0)$. After quantifier elimination, this formula is equivalent to $x + \nu \geq 0 \wedge x \geq 0$. Because this formula is not local, we retain the strongest possible local formula that corresponds to it. In general, given a formula $q(\nu)$, the formula $\exists\nu.q(\nu)$ captures the non-local part of $q(\nu)$, so the formula $(\exists\nu.q(\nu)) \to q(\nu)$ is local. Here, the non-local information is $\exists\nu.x+\nu \geq 0 \wedge x \geq 0$, which is equivalent to $x \geq 0$, so the local formula for y is $x \geq 0 \to x + \nu \geq 0$. Constraint (1) is reduced to:

$$x\!:?, y\!:(x \geq 0 \to x + \nu \geq 0), z\!:(\nu \geq 0) \;\approx\!\!\models\; x + y + z \geq 0 \wedge x \geq 0$$

Applying the same reduction approach focusing on x, we obtain (after extraction) the following *static* entailment, which is valid:

$$\{\, x \geq 0, x \geq 0 \to x + y \geq 0, z \geq 0 \,\} \models x + y + z \geq 0 \wedge x \geq 0$$

Thus the consistent entailment constraint (1) can be satisfied.

With function types, subtyping conveys many consistent entailment constraints that must be handled *together*, because the same interpretation for an unknown formula must be maintained between different constraints. The reduction approach above can be extended to the higher-order case noting that constraints involved in subtyping form a tree structure, sharing common prefixes.

Proposition 14 (Constraint reduction)**.** *Consider a set of consistent entailment constrains sharing a common prefix* $(\widetilde{\Phi}_1, y\!:?)$*:*

$$\overline{\{\widetilde{\Phi}_1, y\!:?, \Phi_2^i \approx\!\!\models r_i(\vec{x}, y, \vec{z_i})\}}$$

Where $\vec{x} = \mathtt{dom}(\widetilde{\Phi}_1)$ *(resp.* $\vec{z_i} = \mathtt{dom}(\Phi_2^i)$*) is the set of variables bound in* $\widetilde{\Phi}_1$ *(resp.* Φ_2^i*). Let* $\vec{z} = \bigcup_i \vec{z_i}$ *and define the canonical formula* $q(\vec{x}, \nu)$ *and its local restriction* $q'(\vec{x}, \nu)$ *as follows:*

$$q(\vec{x}, \nu) = \forall \vec{z}. \bigwedge_i ((\!|\Phi_2^i|\!) \to r_i(\vec{x}, \nu, \vec{z}))$$

$$q'(\vec{x}, \nu) = (\exists \nu. q(\vec{x}, \nu)) \to q(\vec{x}, \nu)$$

Let $\Phi_1 \in \gamma_\Phi(\widetilde{\Phi}_1)$ *be any logical environment in the concretization of* $\widetilde{\Phi}_1$*. Then the following proposition holds: there exists* $p(\vec{x}, \nu) \in \gamma_p(?)$ *such that* $(\!|\Phi_1, y\!:p(\vec{x}, \nu), \Phi_2^i|\!) \models r_i(\vec{x}, y, \vec{z_i})$ *for every* i *if and only if* $(\!|\Phi_1, y\!:q'(\vec{x}, \nu), \Phi_2^i|\!) \models r_i(\vec{x}, y, \vec{z_i})$ *for every* i*.*

In words, when a set of consistent entailment constraints share the same prefix, we can replace the rightmost gradual formula by a canonical *static* formula that justifies the satisfiability of the constraints.[7] This reduction preserves the set of interpretations of the prefix $\widetilde{\Phi}_1$ that justify the satisfaction of the constraints.

The algorithmic subtyping judgment $\widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2$ is calculated in two steps. First, we recursively traverse the structure of types to collect a set of constraints C^*, made static by reduction. Second, we check that these constraints, prepended with $\widetilde{\Phi}$, again reduced to static constraints, can be satisfied. The algorithmic definition of consistent subtyping coincides with Definition 4 (Sect. 3.2), considering the local interpretation of gradual formulas.

Proposition 15. $\widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2$ *if and only if* $\widetilde{\Phi} \vdash \widetilde{T}_1 \mathrel{\widetilde{<:}} \widetilde{T}_2$*.*

8. Extension: Measures

The derivation of the gradual refinement language is largely independent from the refinement logic. We now explain how to extend our approach to support a more expressive refinement logic, by considering *measures* (Vazou et al. 2014), *i.e.* inductively-defined functions that axiomatize properties of data types.

Suppose for example a data type $\mathsf{IntList}$ of lists of integers. The measure len determines the length of a list.

$$\begin{array}{ll} \mathtt{measure}\; len : \mathsf{IntList} \to \mathsf{Int} & \\ len(\mathtt{[]}) & = 0 \\ len(x :: xs) & = 1 + len(xs) \end{array}$$

Measures can be encoded in the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA): a fresh uninterpreted function symbol is defined for every measure, and each measure equation is translated into a refined type for the corresponding data constructor (Vazou et al. 2014). For example, the definition of len yields refined types for the constructors of $\mathsf{IntList}$, namely $\{\nu : \mathsf{IntList} \mid len(\nu) = 0\}$ for empty list, and $x\!:\!\mathsf{Int} \to l\!:\!\mathsf{IntList} \to \{\nu\!:\!\mathsf{IntList} \mid len(\nu) = 1 + len(l)\}$ for cons.

Appropriately extending the syntax and interpretation of gradual formulas with measures requires some care. Suppose a function

[6] Our approach relies on the theory of linear arithmetic being full first order (including quantifiers) decidable—see discussion at the end of Section 8.

[7] Proposition 14 states the equivalence only when the bound for the gradual formula is $\top$—recall that ? stands for $\top \wedge ?$. Dealing with arbitrary imprecise formulas $p \wedge ?$ requires ensuring that the generated formula is more specific than p, but the reasoning is similar.

get to obtain the n-th element of a list, with type:

$$l\!:\!\mathsf{IntList} \rightarrow n\!:\!\{\nu\!:\!\mathsf{Int} \mid 0 \leq \nu < len(l)\} \rightarrow \mathsf{Int}$$

Consider now a function that checks whether the n-th element of a list is less than a given number:

```
let $f$ $(l\colon \{\nu\!:\!\mathsf{IntList} \mid ?\})$ $(n\colon \{\nu\!:\!\mathsf{Int} \mid ?\})$ $(m\colon \{\nu\!:\!\mathsf{Int} \mid ?\})$ =
    $(get\ l\ n) < m$
```

We expect this code to be accepted statically because n could stand for some valid index. We could naively consider that the unknown refinement of n stands for $0 \leq \nu < len(l)$. This interpretation is however *non-local*, because it restricts $len(l)$ to be strictly greater than zero; a non-local interpretation would then also allow the refinement for m to stand for some formula that contradicts this restriction on l. We must therefore adhere to locality to avoid contradictions (Sect. 4.4). Note that we *can* accept the definition of f based on a local interpretation of gradual formulas: the unknown refinement of l could stand for $len(l) > 0$, and the refinement of n could stand for a local constraint on n based on the fact that $len(l) > 0$ holds, *i.e.* $len(l) > 0 \rightarrow 0 \leq \nu < len(l)$.

To easily capture the notion of locality we leverage the fact that measures can be encoded in a restricted fragment of QF-EUFLIA that contains only unary function symbols, and does not allow for nested uninterpreted function applications. We accordingly extend the syntax of formulas in the static language, with $f \in$ MEASURE:

$$p \quad ::= \quad \ldots \mid f\ v \mid f\ \nu$$

For this logic, locality can be defined syntactically, mirroring Definition 12. It suffices to notice that, in addition to restricting the refinement variable ν, formulas are also allowed to restrict a measure applied to ν. To check locality of a formula, we consider each syntactic occurrence of an application $f(\nu)$ as an atomic constant.

Definition 22 (Local formula for measures)**.** *Let p be a formula in the restricted fragment of QF-EUFLIA. Let p' be the formula resulting by substituting every occurrence of $f(\nu)$ for some function f by a fresh symbol $c_{f(\nu)}$. Then, let X be the set of all symbols $c_{f(\nu)}$. We say that p is local if $\exists X.\exists\nu.p'$ is valid.*

The critical property for local formulas is that they always preserves satisfiability (recall Proposition 3).

Proposition 16. *Let Φ be a logical environment with formulas in the restricted fragment of QF-EUFLIA, $\vec{x} = \mathtt{dom}(\Phi)$ the vector of variables bound in Φ, and $q(\vec{x}, \nu)$ a local formula. If $(\!|\Phi|\!)$ is satisfiable then $(\!|\Phi|\!) \cup \{\, q(\vec{x}, \nu)\,\}$ is satisfiable.*

The definition of the syntax and interpretation of gradual formulas follows exactly the definition from Section 4.4, using the new definition of locality. Then, the concretization function for formulas is naturally lifted to refinement types, gradual logical environment and subtyping triples, and the gradual language is derived as described in previous sections. Recall that the derived semantics relies on $\langle \alpha_T, \gamma_T \rangle$ and $\langle \alpha_\tau, \gamma_\tau \rangle$ being partial Galois connections. The abstraction function for formulas with measures is again partial, thus α_T and α_τ are also partial. Therefore, we must establish that $\langle \alpha_T, \gamma_T \rangle$ and $\langle \alpha_\tau, \gamma_\tau \rangle$ are still partial Galois connections for the operators used in the static and dynamic semantics.

Lemma 17 (Partial Galois connections for measures)**.** *The pair $\langle \alpha_T, \gamma_T \rangle$ is a $\{\, \widetilde{tsubst}\,\}$-partial Galois connection. The pair $\langle \alpha_\tau, \gamma_\tau \rangle$ is a $\{\, F_{\mathcal{I}_{<:}}, F_{\circ<:}, F_{\underset{<:}{\circ[v/x]}}\,\}$-partial Galois connection.*

To sum up, adapting our approach to accommodate a given refinement logic requires extending the notion of locality (preserving satisfiability), and establishing the partial Galois connections for the relevant operators. This is enough to derive a gradual language that satisfies the properties of Sections 3.3 and 6.4.

Additionally, care must be taken to maintain decidable checking. For example, our algorithmic approach to consistent subtyping (Section 7) relies on the theory of linear arithmetic accepting quantifier elimination, which is of course not true in all theories. The syntactic restriction for measures allows us to exploit the same approach for algorithmic consistent subtyping, since we can always see a formula in the restricted fragment of QF-EUFLIA as an "equivalent" formula in QF-LIA. But extensions to other refinement logics may require devising other techniques, or may turn out to be undecidable; this opens interesting venues for future work.

9. Discussion

We now discuss two interesting aspects of the language design for gradual refinement types.

Flow sensitive imprecision. An important characteristic of the static refinement type system is that it is flow sensitive. Flow sensitivity interacts with graduality in interesting ways. To illustrate, recall the following example from the introduction:

$$check :: \mathsf{Int} \rightarrow \{\nu\!:\!\mathsf{Bool} \mid ?\}$$
$$get :: \{\nu\!:\!\mathsf{Int} \mid \nu \geq 0\} \rightarrow \mathsf{Int}$$

The gradual type system can leverage the imprecision in the return type of check and *transfer* it to branches of a conditional. This allows us to statically accept the example from the introduction, rewritten here in normal form:

```
let $b = check(x)$ in
  if $b$ then $get(x)$
  else (let $y = -x$ in $get(y)$)
```

Assuming no extra knowledge about x, in the **then** branch the following consistent entailment constraint must be satisfied:

$$x\!:\!\top, b\!:\!?, b = \mathsf{true}, z\!:\!(\nu = x) \mathrel{|\!\approx} z \geq 0$$

Similarly, for the **else** branch, the following consistent entailment constraint must be satisfied:

$$x\!:\!\top, b\!:\!?, b = \mathsf{false}, y\!:\!(\nu = -x), z\!:\!(\nu = y) \mathrel{|\!\approx} z \geq 0$$

Note that the assumption $b = \mathtt{true}$ in the first constraint and $b = \mathsf{false}$ in the second are inserted by the type system to allow flow sensitivity. The first (resp. second) constraint can be trivially satisfied by choosing ? to stand for $\nu = \mathsf{false}$ (resp. $\nu = \mathsf{true}$). This choice introduces a contradiction in each branch, but is not a violation of locality: the contradiction results from the static formula inserted by the flow-sensitive type system. Intuitively, the gradual type system accepts the program because—without any static information on the value returned by *check*—there is always the possibility for each branch *not* to be executed.

The gradual type system also enables the smooth transition to more precise refinements. For instance, consider a different signature for *check*, which specifies that if it returns true, then the input must be positive:[8]

$$check :: x\!:\!\mathsf{Int} \rightarrow \{\nu\!:\!\mathsf{Bool} \mid (\nu = \mathsf{true} \rightarrow x \geq 0) \wedge ?\}$$

In this case the **then** branch can be definitively accepted, with no need for dynamic checks. However, the static information is not sufficient to definitely accept the **else** branch. In this case, the type system can no longer rely on the possibility that the branch is never executed, because we know that, at least for negative inputs, *check* will return false. Nevertheless, the type system can optimistically assume that *check* returns false only for negative inputs. The program is therefore still accepted statically, and subject to a dynamic check in the **else** branch.

[8] The known part of the annotation may appear to be non-local; its locality becomes evident when writing the contrapositive $x < 0 \rightarrow \nu = \mathsf{false}$.

Eager vs. lazy failures. AGT allows us to systematically derive the dynamic semantics of the gradual language. This dynamic semantics is intended to serve as a reference, and not as an efficient implementation technique. Therefore, defining an efficient cast calculus and a correct translation from gradual source programs is an open challenge.

A peculiarity of the dynamic semantics of gradual refinement types derived with AGT is the consistent term substitution operator (Section 6.3), which detects inconsistencies at the time of beta reduction. This in turn requires verifying consistency relations on open terms, hence resorting to SMT-based reasoning at runtime; a clear source of inefficiency.

We observe that AGT has been originally formulated to derive a runtime semantics that fails *as soon as is justifiable*. Eager failures in the context of gradual refinements incurs a particularly high cost. Therefore, it becomes interesting to study postponing the detection of inconsistencies *as late as possible*, *i.e.* while preserving soundness. If justifications can be delayed until closed terms are reached, runtime checks boil down to direct evaluations of refinement formulas, with no need to appeal to the SMT solver. To the best of our knowledge, capturing different eagerness failure regimes within the AGT methodology has not yet been studied, even in a simply-typed setting; this is an interesting venue for future work.

10. Related Work

A lot of work on refining types with properties has focused on maintaining statically decidable checking (*e.g.* through SMT solvers) via restricted refinement logics (Bengtson et al. 2011; Freeman and Pfenning 1991; Rondon et al. 2008; Xi and Pfenning 1998). The challenge is then to augment the expressiveness of the refinement language to cover more interesting programs without giving up on automatic verification and inference (Chugh et al. 2012b; Kawaguchi et al. 2009; Vazou et al. 2013, 2015). Despite these advances, refinements are necessarily less expressive than using higher-order logics such as Coq and Agda. For instance, subset types in Coq are very expressive but require manual proofs (Sozeau 2007). $F^\star$ hits an interesting middle point between both worlds by supporting an expressive higher-order logic with a powerful SMT-backed type checker and inferencer based on heuristics, which falls back on manual proving when needed (Swamy et al. 2016).

Hybrid type checking (Knowles and Flanagan 2010) addresses the decidability challenge differently: whenever the external prover is not statically able to either verify or refute an implication, a cast is inserted, deferring the check to runtime. Refinements are arbitrary boolean expressions that can be evaluated at runtime. Refinements are however not guaranteed to terminate, jeopardizing soundness (Greenberg et al. 2010).

Earlier, Ou et al. (2004) developed a core language with refinement types, featuring three constructs: **simple**$\{e\}$, to denote that expression e is simply well-typed, **dependent**$\{e\}$, to denote that the type checker should statically check all refinements in e, and **assert**(e, τ) to check at runtime that e produces a value of type τ. The semantics of the source language is given by translation to an internal language, inserting runtime checks where needed.

Manifest contracts (Greenberg et al. 2010) capture the general idea of allowing for explicit typecasts for refinements, shedding light on the relation with dynamic contract checking (Findler and Felleisen 2002) that was initiated by Gronski and Flanagan (2007). More recently, Tanter and Tabareau (2015) provide a mechanism for casting to subset types in Coq with arbitrary decidable propositions. Combining their cast mechanism with the implicit coercions of Coq allows refinements to be implicitly asserted where required.

None of these approaches classify as gradual typing per se (Siek and Taha 2006; Siek et al. 2015), since they either require programmers to explicitly insert casts, or they do not mediate between various levels of type precision. For instance, Ou et al. (2004) only support either simple types or fully-specified refinements, while a gradual refinement type system allows for, and exploits, partially-specified refinements such as $\nu > 0 \wedge ?$.

Finally, this work relates in two ways to the gradual typing literature. First, our development is in line with the relativistic view of gradual typing already explored by others (Bañados Schwerter et al. 2014; Disney and Flanagan 2011; Fennell and Thiemann 2013; Thiemann and Fennell 2014), whereby the "dynamic" end of the spectrum is a simpler static discipline. We extend the state-of-the-art by gradualizing refinement types for the first time, including dependent function types. Notably, we prove that our language satisfies the gradual guarantee (Siek et al. 2015), a result that has not been established for any of the above-mentioned work.

Second, this work builds upon and extends the Abstracting Gradual Typing (AGT) methodology of Garcia et al. (2016). It confirms the effectiveness of AGT to streamline most aspects of gradual language design, while raising the focus on the key issues. For gradual refinements, one of the main challenges was to devise a practical interpretation of gradual formulas, coming up with the notion of locality of formulas. To support the local interpretation of gradual formulas, we had to appeal to *partial* Galois connections (Miné 2004). This approach should be helpful for future applications of AGT in which the interpretation of gradual types is not as straightforward as in prior work. Also, while Garcia et al. (2016) focus exclusively on consistent subtyping transitivity as the locus of runtime checking, dealing with refinement types requires other meta-theoretic properties used for type preservation—lemmas related to substitution in both terms and types—to be backed by evidence in the gradual setting, yielding new consistent operators that raise new opportunities for runtime failure.

11. Conclusion

Gradual refinement types support a smooth evolution between simple types and logically-refined types. Supporting this continuous slider led us to analyze how to deal with imprecise logical information. We developed a novel *semantic* and *local* interpretation of gradual formulas that is key to practical gradual refinements. This specific interpretation of gradual formulas is the main challenge in extending the refinement logic, as illustrated with measures. We also demonstrate the impact of dependent function types in a gradual language, requiring new notions of term and type substitutions with runtime checks. This work should inform the gradualization of other advanced type disciplines, both regarding logical assertions (*e.g.* Hoare logic) and full-fledged dependent types.

A most pressing perspective is to combine gradual refinement types with type inference, following the principled approach of Garcia and Cimini (2015). This would allow progress towards a practical implementation. Such an implementation should also target a cast calculus, such as a manifest contract system, respecting the reference dynamic semantics induced by the AGT methodology. Finally, while we have explained how to extend the refinement logic with measures, reconciling locality and decidability with more expressive logics—or arbitrary terms in refinements—might be challenging.

Acknowledgments

We would like to thank Ronald Garcia, Jorge Pérez, Niki Vazou and the anonymous referees for their feedback, and Nick, Jesse and Jordan for their precious inspiration.

References

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.

F. Bañados Schwerter, R. Garcia, and É. Tanter. Gradual type-and-effect systems. *Journal of Functional Programming*, 26:19:1–19:69, Sept. 2016.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8:1–8:45, Jan. 2011.

R. Chugh. *Nested Refinement Types for JavaScript*. PhD thesis, University of California, Sept. 2013.

R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 587–606, Tucson, AZ, USA, Oct. 2012a. ACM Press.

R. Chugh, P. M. Rondon, A. Bakst, and R. Jhala. Nested refinements: a logic for duck typing. In *39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 231–244, Philadelphia, USA, Jan. 2012b. ACM Press.

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

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.

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.

C. Flanagan. Hybrid type checking. In *33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 245–256, Charleston, SC, USA, Jan. 2006. ACM Press.

T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, pages 268–277. ACM Press, 1991.

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, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, St Petersburg, FL, USA, Jan. 2016. ACM Press.

M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 353–364. ACM Press, Jan. 2010.

J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming*, pages 54–70, 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.

M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 304–315. ACM Press, June 2009.

K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2), 2010.

N. Lehmann and É. Tanter. Gradual refinement types – extended version with proofs. Technical Report TR/DCC-2016-1, University of Chile, Nov. 2016.

A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, L'École Polythechnique, Dec. 2004.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169, Tucson, AZ, USA, June 2008. ACM Press.

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, 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, 2015.

M. Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 237–252. Springer-Verlag, 2007.

N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Zanella Béguelin. Dependent types and multi-effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 256–270, St Petersburg, FL, USA, Jan. 2016. ACM Press.

É. Tanter and N. Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*, pages 26–40, Pittsburgh, PA, USA, Oct. 2015. ACM Press.

P. Thiemann and L. Fennell. Gradual typing for annotated type systems. In Z. Shao, editor, *23rd European Symposium on Programming Languages and Systems (ESOP 2014)*, volume 8410 of *LNCS*, pages 47–66, Grenoble, France, 2014. Springer-Verlag.

N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming Languages and Systems (ESOP 2013)*, volume 7792 of *LNCS*, pages 209–228, Rome, Italy, Mar. 2013. Springer-Verlag.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 269–282, Gothenburg, Sweden, Sept. 2014. ACM Press.

N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In *20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*, pages 48–61, Vancouver, Canada, Sept. 2015. ACM Press.

P. Vekris, B. Cosman, and R. Jhala. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 310–325, Santa Barbara, CA, USA, June 2016. ACM Press.

H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 249–257. ACM Press, 1998.