A Gradual Probabilistic Lambda Calculus*

WENJIA YE, The University of Hong Kong, China
MATÍAS TORO, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile
FEDERICO OLMEDO, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile

Probabilistic programming languages have recently gained a lot of attention, in particular due to their applications in domains such as machine learning and differential privacy. To establish invariants of interest, many such languages include some form of static checking in the form of type systems. However, adopting such a type discipline can be cumbersome or overly conservative.

Gradual typing addresses this problem by supporting a smooth transition between static and dynamic checking, and has been successfully applied for languages with different constructs and type abstractions. Nevertheless, its benefits have never been explored in the context of probabilistic languages.

In this work, we present and formalize GPLC, a gradual source probabilistic lambda calculus. GPLC includes a binary probabilistic choice operator and allows programmers to gradually introduce/remove static type –and probability– annotations. The static semantics of GPLC heavily relies on the notion of probabilistic couplings, as required for defining several relations, such as consistency, precision, and consistent transitivity. The dynamic semantics of GPLC is given via elaboration to the target language TPLC, which features a distribution-based semantics interpreting programs as probability distributions over final values. Regarding the language metatheory, we establish that TPLC–and therefore also GPLC– is *type safe* and satisfies two of the so-called *refined criteria* for gradual languages, namely, that it is a conservative extension of a fully static variant and that it satisfies the gradual guarantee, behaving smoothly with respect to type precision.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; **Type structures**; *Operational semantics*; **Program reasoning**; **Probabilistic computation**.

Additional Key Words and Phrases: Type Systems, Gradual Typing, Probabilistic Lambda Calculus

ACM Reference Format:
Wenjia Ye, Matías Toro, and Federico Olmedo. 2023. A Gradual Probabilistic Lambda Calculus. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 84 (April 2023), 30 pages. https://doi.org/10.1145/3586036

1 INTRODUCTION

In a nutshell, *probabilistic programming languages* are traditional programming languages that, on top of their regular constructs, offer the possibility of sampling values from probability distributions [Gordon et al. 2014; van de Meent et al. 2018]. They find applications in a wealth of different areas, ranging from more traditional application domains such randomized algorithms [Motwani and Raghavan 1995] and cryptography [Goldwasser and Micali 1984] to more novel application domains such as differential privacy [Dwork and Roth 2014] and machine learnings [Claret et al. 2013;

*This work has been partially sponsored by Hong Kong Research Grants Council projects number 17209520 and 17209821, and ANID FONDECYT project 3200583, Chile.

Authors' addresses: Wenjia Ye, The University of Hong Kong, Hong Kong, China, yewenjia@connect.hku.hk; Matías Toro, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago, Chile, mtoro@dcc.uchile.cl; Federico Olmedo, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago, Chile, folmedo@dcc.uchile.cl.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/4-ART84
https://doi.org/10.1145/3586036

Ghahramani 2015]. These latter have led to a remarkable resurgence of probabilistic programming in the past years, with the development of a growing number of new probabilistic programming systems [Goodman et al. 2008; Goodman and Stuhlmüller 2014; Kiselyov 2016; Le et al. 2017; Pfeffer 2010; Tran et al. 2017].

To establish certain invariants of interest, programming languages traditionally incorporate some form of *typing*, backed up by a type checking phase. Depending on the moment in which type checking occurs, it is classified either as *static* —when taking place during compilation—, or as *dynamic* —when it takes place during runtime—, each of them having their own strengths and weaknesses. Concretely, programming languages with static typing allows detecting errors (i.e. invariant violations) at an early stage, but are not flexible enough for rapid prototyping. On the other hand, programming languages with dynamic typing accommodate better to changes, but present slower runtimes.

Gradual typing [Siek and Taha 2006] represents an effective alternative for integrating the benefits of static and dynamic typing at the same time, by allowing a smooth transition all along the spectrum. To do so, it introduces *imprecise* (a.k.a. *gradual*) types, which represent types possibly partially known at compile time. Imprecise types can range from fully precise static types (such as $\mathsf{Real} \to \mathsf{Bool}$), to the fully unknown (or imprecise) type, written ?, with partially precise types (such as $\mathsf{Real} \to ?$) in-between. At compile time, a gradual language typechecks programs optimistically, based on the notion of type consistency, (e.g., accepting the application of a function expecting an argument of type $\mathsf{Real} \to ?$ and receiving an argument of type $? \to \mathsf{Bool}$,) while the *runtime* is responsible for detecting (and reporting) any violation of such assumptions (e.g., if the received argument happens to have concrete type $\mathsf{Bool} \to \mathsf{Bool}$).

Gradual typing has been successfully applied to programming languages with diverse constructs and typing disciplines. Some relevant features include first-class classes [Takikawa et al. 2012], mutable references [Herman et al. 2010; Siek and Taha 2006; Siek et al. 2015c; Toro and Tanter 2020], effects as primitives [Bañados Schwerter et al. 2016], tagged and untagged unions [Toro and Tanter 2017], labeling operations (for reasoning about information flow) [Azevedo de Amorim et al. 2020; Disney and Flanagan 2011; Fennell and Thiemann 2013; Toro et al. 2018], and algebraic data types [Malewski et al. 2021]. However, it is an open question whether the benefits of gradual typing carry over to *probabilistic* programming languages.

In this work, we give a positive answer to this question by, on the one hand, designing, to the best of our knowledge, the first gradual probabilistic language and, on the other hand, establishing a set of metatheorethic results, natural to all gradual languages.

First, we present SPLC, a probabilistic λ-calculus that extends ordinary λ-calculus with a (binary) probabilistic choice operator and acts as the static end of our gradual language. It features a big-step semantics relating programs to the probability distribution of final values and to better accommodate the derivation of the gradual variant, its type system presents some distinguished features such as the presence of ascriptions, partial functions *dom* and *cod* over types, and explicit type equality. Furthermore, equality over types is semantic (instead of syntactic).

Second, we introduce GPLC, our source gradual language, whose derivation from SPLC is justified using the Abstracting Gradual Typing (AGT) methodology, a systematic approach for deriving gradual languages based on abstract interpretation [Garcia et al. 2016]. For the so-derived notions of type consistency and type and term precision, we also provide alternative —more amenable to automation— characterizations, based on the notion of probabilistic couplings [Deng and Du 2011]. In effect, probabilistic couplings are a fundamental ingredient, behind all our technical development.

Notably, GPLC allows unknown probabilities not only at the type level, but also at the term level (in probabilistic choices). This yields an increased expressivity and flexibility —characteristic of all gradual languages— and also the opportunity of leveraging the language for program refinement.

Third, we define the dynamic semantics of our gradual language by translating GPLC into the target gradual language TPLC. The runtime semantics of TPLC incorporates the required evidence to confirm or discard the optimistic assumptions made by GPLC type system. In turn, this requires adapting gradual types to encode unknown probabilities through symbolic variables, which are constrained by well-formedness conditions.

Finally, to formally validate our language design, we establish three fundamental properties of GPLC. First, we prove that it is a conservative extension of SPLC. Second, we show that it satisfies type safety. Lastly, we show that it behaves smoothly with respect to precision, a property known as the *gradual guarantee* [Siek et al. 2015b].

Altogether, this provides the first steps into the theoretical foundations of gradual probabilistic programming, and serves as starting point for developing gradual variants of more specialized, domain specific, probabilistic languages, e.g., as used for differential privacy [Reed and Pierce 2010].

Paper Organization. The rest of the paper is organized as follows. Section 2 discusses the motivation and some key design decisions and challenges behind our probabilistic gradual language. Section 3 presents the probabilistic lambda calculus (SPLC) acting as the static end of the gradualization. Section 4 develops the source gradual language (GPLC) and Section 5 the target language (TPLC), together with the metatheory. Section 6 overviews the related work and Section 7 concludes. Full definitions and proofs of the main results can be found in the supplementary material.

2 OVERVIEW

We next discuss the motivation behind a *gradual* probabilistic language through a concrete use case and summarize some key aspects and challenges behind the design of our gradual probabilistic language.

2.1 A Gradual Probabilistic Language: Why?

Assume we must develop a web application for a company, in particular, the login endpoint. To authenticate a user, we must verify that the user remains active in the company, information that is provided by an external web service (exposed by a foreign library). As usual, we support both production and development modes, where in development mode we replace the external web service with a local function, conveniently defined for developing and testing purposes.

Under these requirements, we quickly prototype the following (untyped) program, written in a Scala-like language:

```
1  def isUserActive(user) = if (prod) externalCall(user) else localCall(user)
2  def login(user, pass) = if (isUserActive(user)) /* test password */ else false
```

Probabilistic modeling. The company now requires that the `login` endpoint have a 95% uptime (availability). However, after some research, we learn that the external web service `externalCall` has only a 90% uptime, returning a `503 Service Unavailable` error when down. Written in such an untyped language, the above program is unable to capture this uptime information, let alone detect the impossibility to comply with the login requirements.

As a first step to address this problem, we can adopt a typed language that includes *distribution types*. Loosely speaking, distribution types represent probability distributions of "simpler" types, and can crisply model uptime information. For instance, the return type of `externalCall` shall now be represented by $\{\texttt{Bool}^{\frac{90}{100}}, \texttt{Error503}^{\frac{10}{100}}\}$, and the return type of `login` by $\{\texttt{Bool}^{\frac{95}{100}}, \texttt{Error503}^{\frac{5}{100}}\}$. Furthermore, we can implement a `localCall` function compatible with the uptime requirement of the `login` endpoint, as follows:

```
3  def localCall(user: String): {Bool^95/100, Error503^5/100} = true ⊕_95/100 new Error503()
```

A program of the form `m` $\oplus_p$ `n` is known as a *probabilistic choice* between `m` and `n`, and behaves like `m` with probability `p` and like `n` with probability `1-p`.[1]

Limitations of static typing. Adopting a static typing for our probabilistic language would be cumbersome, as it would require inserting type annotations everywhere, or else extending the language with a type inference mechanism. In either case, the static typing can be overly conservative, rejecting (at compile time) programs that may indeed go right at runtime. For example, declaring the return types of functions `externalCall` and `localCall` as argued above ({`Bool` $^{\frac{90}{100}}$, `Error503` $^{\frac{10}{100}}$} and {`Bool` $^{\frac{95}{100}}$, `Error503` $^{\frac{5}{100}}$}, respectively), would render function `isUserActive` ill-typed as the two branches of the conditional in the function body would have different types. Note that, even though the uptime of the external web service is incompatible with the uptime requirements of the login endpoint, we still would like to have a program able to execute in development mode (and in production mode, with minor modifications in typing annotation, if uptime requirements are reconciled).

Gradual typing at rescue. Gradual typing addresses this problem by supporting a smooth transition between static and dynamic typing, introducing imprecision on static types via the unknown annotation ?.[2] Intuitively, an unknown type (resp. probability) ? represents any type (resp. probability). For example, using gradual (distribution) types we can partially annotate the program to assert only a subset of function uptimes:

```
4  val externalCall: ? -> {Bool^90/100, Error503^10/100} = ...
5  def isUserActive(user: ?): ? = if (prod) externalCall(user) :: ? else
       ↪ localCall(user) :: ?
6  def login(user: ?, pass: ?): {Bool^95/100, Error503^5/100} = if (isUserActive(user)) ...
```

To render `isUserActive` well-typed, we also had to ascribe both its conditional branches to the unknown type (written `:: ?`), since the conditional branches have different (fully static) types.

The type checker of a gradual language treats type equality optimistically, through the notion of *consistency*. Consistency between gradual types tests the plausibility of equality between any of the static types they represent. For instance, gradual type `? -> Bool` is consistent with `Int -> ?`, written `? -> Bool ~ Int -> ?`, because (during runtime) they can both represent, e.g., the fully static type `Int -> Bool`.

In view of this optimistic treatment of equality, the above program is accepted statically as the unknown type ? is (trivially) consistent with every other type. If the application is in development mode, then the `login` endpoint runs successfully. On the contrary, if the application is in production mode, a runtime error is raised. This is because, even though {`Bool` $^{\frac{90}{100}}$, `Error503` $^{\frac{10}{100}}$} $\sim$ `?` and `?` $\sim$ {`Bool` $^{\frac{95}{100}}$, `Error503` $^{\frac{5}{100}}$}, {`Bool` $^{\frac{90}{100}}$, `Error503` $^{\frac{10}{100}}$} $\not\sim$ {`Bool` $^{\frac{95}{100}}$, `Error503` $^{\frac{5}{100}}$}. Said otherwise, consistency is not transitive. Therefore, gradual languages incorporate runtime checks to detect any potential violation of the optimistic assumptions performed statically during type checking.

Finally, note that we can increase the program precision, e.g., changing the return type of `isUserActive` from `?` to {`Bool` $^{\frac{95}{100}}$, `Error503` $^{\frac{5}{100}}$}, which would make the program ill-typed, failing thus at compile time.

Besides for this enhanced expressivity, one can also employ our *gradual* probabilistic language for program refinement purposes. To illustrate this application, assume that the external service `externalCall` is now required to have an uptime of *at least* 95%. This can be modelled by declaring

[1] A probabilistic choice `m` $\oplus_p$ `n` can be readily simulated (in an approximate manner) by all programming languages that include the commonplace primitive `random()`, returning an (approximately) uniform value in the [0, 1] interval. It suffices to take program `if (random() <= p) m else n`.

[2] A fully untyped program is considered to have unknown annotations everywhere.

its return type as $\{\texttt{Bool}^{\frac{95}{100}}, \texttt{Bool}^{?}, \texttt{Error503}^{?}\}$. In contrast to the above example where ? represented unknown *types*, here, both occurrences of ? represent unknown (possibly different) *probabilities*, which together with $\frac{95}{100}$ must sum up to 1.

Furthermore, assume that the external service originally relied on a single server of 90% uptime (`server1`) to keep track of active users. To reach the desired uptime of (at least) 95%, the service provider decides to buy a new —very costly– server of 98% uptime (`server2`). A naive implementation of the service would simply dispense with `server1` and rely only on `server2` to respond queries. However, this would negatively impact on `server2` liftime, diminishing the return of the performed investment. To avoid this problem and still benefit from `server1`, a possible solution consists in, upon each query, *probabilistically* choosing either server to respond the query. The fundamental question left to answer is whether this design would result in an overall (expected) uptime of at least 95%. To answer this question, we can consider the following program:

```
7  val server1: ? -> {Bool^(90/100), Error503^(10/100)} = ...
8  val server2: ? -> {Bool^(98/100), Error503^(2/100)} = ...
9  def externalCall(user: ?): {Bool^(95/100), Bool^?, Error503^?} = server1(user) ⊕?
       ↪ server2(user)
```

where symbol ? in the probabilistic choice $\oplus_?$ also represents an unknown probability. Our gradual language correctly typechecks this program and its runtime informs us about the feasibility of the proposed `externalCall` design. For concreteness, assume that for any user, `server1` (resp. `server2`) responds `true` with probability $\frac{90}{100}$ (resp. $\frac{98}{100}$) and `error503` with the complementary probability. The instrumentation of the language runtime introduces symbolic variables to represent unknown probabilities; say $\omega_{\texttt{Bool}}$, $\omega_{\texttt{Error503}}$ and $\omega_{\texttt{choice}}$ represent the unknown probabilities encoded by ? respectively in $\texttt{Bool}^{?}$, $\texttt{Error503}^{?}$ and $\oplus_?$. The runtime semantics tells us that invoking `externallCall` with any user returns `true` with probability $\frac{95}{100} + \omega_{\texttt{Bool}}$ and `error503` with probability $\frac{2}{100} + \left(\frac{98}{100} - \frac{90}{100}\right)\omega_{\texttt{choice}}$, where the symbolic variables are constrained by formulas $\frac{95}{100} + \omega_{\texttt{Bool}} + \omega_{\texttt{Error503}} = 1$ and $\frac{90}{100}\omega_{\texttt{choice}} + \frac{98}{100}(1 - \omega_{\texttt{choice}}) = \frac{95}{100} + \omega_{\texttt{Bool}}$.[3] Any probability $\omega_{\texttt{choice}} \leq \frac{37.5}{100}$ yields a valid solution of the equation system, yielding a valid refinement of `externallCall` and validating the proposed design.

2.2 Design Decisions and Challenges

When designing our source (GPLC) and target (TPLC) gradual probabilistic languages, we faced several design decisions and challenges.

Where to introduce imprecision. In most traditional gradual typing calculi, imprecision is introduced via the unknown type ?. To gain expressivity and flexibility, in this work we allow imprecision at the type level as well as the probability level. For example, given the fully static distribution type $\{\texttt{Bool}^{\frac{9}{10}}, \texttt{Error503}^{\frac{1}{10}}\}$, we can introduce imprecision either in probabilities, e.g. $\{\texttt{Bool}^{?}, \texttt{Error503}^{\frac{1}{10}}\}$, in the underlying types, e.g. $\{?^{\frac{9}{10}}, \texttt{Error503}^{\frac{1}{10}}\}$, or in both. Note that there is no need to introduce the unknown distribution as it can be represented by the gradual distribution type $\{?^{?}\}$. Interestingly, unknown probabilities are particularly useful for expressing *probability bounds*. As hinted above, we can use type $\{\texttt{Bool}^{\frac{95}{100}}, \texttt{Bool}^{?}, \texttt{Error503}^{?}\}$ to represent a service with an uptime of *at least* 95%. On the other hand, type $\{\texttt{Bool}^{?}, \texttt{Error503}^{\frac{5}{100}}, \texttt{Error503}^{?}\}$ models an uptime of *at most* 95%.

Tracking dependencies of probability annotations. When dealing with unknown probabilities, as in type $\{\texttt{Bool}^{\frac{9}{10}}, \texttt{Bool}^{?}, \texttt{Error503}^{?}\}$, the gradual language must ensure that the concrete probabilities

[3] Formally, the instrumentation of the runtime semantics yields a handful of further constraints, but altogether they are equivalent to the considered subset.

$$r \in \mathbb{R}, \quad b \in \mathbb{B}, \quad x \in \text{Var}, \quad p \in [0,1], \quad \tau \in \text{Type}, \quad T \in \text{DType}$$

$$\begin{array}{rll} \tau ::= & \text{Real} \mid \text{Bool} \mid \tau \to T & \text{(simple types)} \\ T ::= & \{\!\{ \tau_i^{p_i} \mid i \in \mathcal{I} \}\!\} & \text{(distribution types)} \\ m, n ::= & v \mid v\ w \mid \text{let } x = m \text{ in } n \mid m \oplus_p n & \text{(terms)} \\ & m :: T \mid v :: \tau \mid \text{if } v \text{ then } m \text{ else } n \mid v + w & \\ v, w ::= & x \mid r \mid b \mid (\lambda x : \tau . m) & \text{(values)} \end{array}$$

Fig. 1. Syntax of SPLC.

they represent induce only well-defined static distribution types, with a total probability of 1. This requirement induces implicit dependencies and gradual probabilities are thus elaborated to fresh variables (ω) constrained by formulas, e.g. of the form $\frac{9}{10} + \omega_1 + \omega_2 = 1$.

Ascribing to distribution types. One of the fundamental features of GPLC is the possibility of ascribing programs to distribution types. For example, a program $f = (\lambda x : ?.x) :: \{(\text{Real} \to ?)^{\frac{1}{2}}, (? \to \text{Bool})^{\frac{1}{2}}\}$ behaves as a function that takes a number as argument with probability $\frac{1}{2}$, and as a function that returns a boolean also with probability $\frac{1}{2}$. Reducing an application to f and correctly propagating the respective type information is not a trivial task. Intuitively, our approach consists in "pushing" the real argument into each (compatible) type in the distribution. For instance, the reduction of program $f\ 1$ proceeds, informally, as follows:

$$f\ 1 \ \mapsto^* \ \{(\lambda x : ?.x) :: (\text{Real} \to ?)^{\frac{1}{2}}\ 1, (\lambda x : ?.x) :: (? \to \text{Bool})^{\frac{1}{2}}\ 1\} \ \mapsto^* \ \{1 :: ?^{\frac{1}{2}}, \mathbf{error}^{\frac{1}{2}}\}$$

Couplings as a central tool. Defining some key relations between distribution types is another technical challenge. For instance, should we consider distribution type $\{(\text{Real} \to ?)^{\frac{1}{2}}, (? \to \text{Real})^{\frac{1}{2}}\}$ consistent with $\{(? \to \text{Bool})^{\frac{1}{3}}, (\text{Real} \to ?)^{\frac{2}{3}}\}$? Is $\{\text{Real}^{\frac{1}{2}}, ?^{\frac{1}{2}}\}$ more precise than $\{\text{Bool}^{\frac{2}{3}}, ?^{\frac{1}{3}}\}$? To define these (and other) relations over distribution types we heavily rely on the notion of probabilistic coupling, which yields a canonical lifting from relations over pair of sets to probability distributions over the sets.

3 SPLC: STATIC LANGUAGE

In this section, we present SPLC, a statically-typed lambda calculus, extended with a probabilistic choice operator, which is the starting point —static end— of our gradualization effort. The static semantics of SPLC is based on that of $\lambda_\oplus$ from [Lago and Grellois 2017], with two major differences: SPLC features a semantic (rather than syntactic) equality between types and also allows type ascriptions. As for the dynamic semantics, programs are interpreted as probability distributions over final values.

3.1 Syntax

The syntax of SPLC is presented in Figure 1, comprising its type and term languages.

Type language. The type language contains two (mutually defined) syntactic categories: simple types and distributions types. A *simple type*, ranged over by τ, can be the type Real of real numbers, the type Bool of Boolean values, or a function type of the form $\tau \to T$, where T is a distribution type. A *distribution type*, ranged over by T, is a multi-set of pairs comprised by a simple type τ and a probability p in the interval $[0, 1]$. Intuitively, we use $\{\!\{ \tau_i^{p_i} \mid i \in \mathcal{I} \}\!\}$ to denote a distribution type in which simple type τ_i occurs with probability p_i, for each i in the (non-empty and finite) subset $\mathcal{I}$ of the natural numbers. For instance, distribution type $\{\!\{\text{Real}^{\frac{1}{4}}, \text{Real}^{\frac{1}{4}}, \text{Bool}^{\frac{1}{2}}\}\!\}$ represents Real with probability $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ and Bool with probability $\frac{1}{2}$. Notationwise, we sometimes omit the index set

$$\boxed{\Gamma \vdash_s v : \tau, \quad \Gamma \vdash_s m : T}$$

$$\text{(Tr)}\frac{}{\Gamma \vdash_s r : \text{Real}} \qquad \text{(Tb)}\frac{}{\Gamma \vdash_s b : \text{Bool}} \qquad \text{(Tx)}\frac{\Gamma(x) = \tau}{\Gamma \vdash_s x : \tau} \qquad \text{(Tv)}\frac{\Gamma \vdash_s v : \tau}{\Gamma \vdash_s v : \{\!|\tau^1|\!\}}$$

$$\text{(T}\lambda\text{)}\frac{\Gamma, x : \tau \vdash_s m : T \qquad \vdash \tau}{\Gamma \vdash_s \lambda x : \tau . m : \tau \to T} \qquad \text{(T::}\ \tau\text{)}\frac{\Gamma \vdash_s v : \tau' \qquad \tau' =_s \tau \qquad \vdash \tau}{\Gamma \vdash_s v :: \tau : \{\!|\tau^1|\!\}}$$

$$\text{(Tapp)}\frac{\Gamma \vdash_s v : \tau_1 \qquad \Gamma \vdash_s w : \tau_2 \qquad dom(\tau_1) =_s \tau_2}{\Gamma \vdash_s v\ w : cod(\tau_1)} \qquad \text{(T}\oplus\text{)}\frac{\Gamma \vdash_s m : T_1 \qquad \Gamma \vdash_s n : T_2}{\Gamma \vdash_s m \oplus_p n : p \cdot T_1 + (1-p) \cdot T_2}$$

$$\text{(Tlet)}\frac{\Gamma \vdash_s m : \{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\} \qquad \forall i \in \mathcal{I}.\ \Gamma, x : \tau_i \vdash_s n : T_i}{\Gamma \vdash_s \text{let } x = m \text{ in } n : \sum_{i \in \mathcal{I}} p_i \cdot T_i} \qquad \text{(T::}\ T\text{)}\frac{\Gamma \vdash_s m : T' \qquad T' =_s T \qquad \vdash T}{\Gamma \vdash_s m :: T : T}$$

$$dom(\tau \to T) = \tau \qquad cod(\tau \to T) = T \qquad p \cdot \{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\} = \{\!|\tau_i^{p \cdot p_i} \mid i \in \mathcal{I}|\!\}$$
$$dom(\tau) \text{ undef. otherwise} \qquad cod(\tau) \text{ undef. otherwise}$$

$$\{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\} + \{\!|\tau_j^{p_j} \mid j \in \mathcal{J}|\!\} = \{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\} \cup \{\!|\tau_j^{p_j} \mid j \in \mathcal{J}|\!\} \quad \text{if } \sum_{i \in \mathcal{I}} p_i + \sum_{j \in \mathcal{J}} p_j \leq 1$$

Fig. 2. Type system of SPLC (excerpt).

$\mathcal{I}$ and simple write $\{\!|\tau_i^{p_i}|\!\}$. Finally, note that distribution types —as the name suggests— represent *probability distributions* (over simple types) and therefore, well-typed programs are associated distribution types whose probabilities sum up to 1 (this restriction is formally captured by the notion of *type well-formedness* defined in Section 3.1).

Term language. Terms, ranged over by m, n, and values, ranged over by v, w, are mutually defined. A *term* can be a value v, an applications $v\ w$ between two values, a let expression let $x = m$ in n, a probabilistic choice $m \oplus_p n$, a term ascription $m :: T$, a value ascription $v :: \tau$, a conditional if v then m else n, or an addition $v + w$ between two values. Note that terms are defined in A-normal form [Sabry and Felleisen 1993], which pushes all the reasoning about probabilities to the let construct. Randomization is introduced though probabilistic choices: program $m \oplus_p n$ behaves like (i.e. reduce to) m with probability p and like n with probability $1 - p$.

3.2 Type System

Figure 2 presents the type system of SPLC. Type rules are defined using a pair of mutually-defined judgments: one for values and another for computations. Judgment $\Gamma \vdash_s v : \tau$ (resp. $\Gamma \vdash_s m : T$) for values (resp. computations) denotes that value v (resp. term m) has simple type τ (resp. distribution type T) under type environment Γ, which maps variables to simple types.

Type rules for values are standard, only a few rules deserving special attention. For example, rule (Tv) allows assigning a value of simple type τ also distribution type $\{\!|\tau^1|\!\}$ (e.g. program 1 can be typed as $\{\!|\text{Int}^1|\!\}$). Also, note that rules (Tλ), (T:: τ) and (T:: T) require all program type annotations to be well-formed. We say that a distribution type $\{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\}$ is *well-formed*, written $\vdash \{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\}$, if $\sum_{i \in \mathcal{I}} p_i = 1$ and simple type τ_i is well-formed for every $i \in \mathcal{I}$. A simple type τ is *well-formed*, written $\vdash \tau$, if it is either a base type (Real or Bool) or a function type $\tau \to T$, where τ and T are well-formed.

A particularity of SPLC's type system is that it relies on a semantic —rather than syntactic— notion of type equality ($=_s$), as used in rules (T:: τ), (Tapp) and (T:: T). For example, $\{\!|\text{Real}^{\frac{1}{2}}, \text{Bool}^{\frac{1}{4}}, \text{Bool}^{\frac{1}{4}}|\!\} =_s \{\!|\text{Real}^{\frac{1}{3}}, \text{Real}^{\frac{1}{6}}, \text{Bool}^{\frac{1}{2}}|\!\}$ because $\frac{1}{2} = \frac{1}{3} + \frac{1}{6}$ and $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$. Formally, type equality is given by rules:

$\boxed{m \Downarrow_s \mathscr{V}}$ $\qquad \mathscr{V} ::= \ \{\!\{ v_i^{p_i} \mid i \in \mathcal{I} \}\!\}$ (distribution values)

$$\frac{}{v \Downarrow_s \{\!\{v^1\}\!\}} \qquad \frac{m[v/x] \Downarrow_s \mathscr{V}}{(\lambda x : \tau . m)\ v \Downarrow_s \mathscr{V}} \qquad \frac{m \Downarrow_s \mathscr{V}_1 \quad n \Downarrow_s \mathscr{V}_2}{m \oplus_p n \Downarrow_s p \cdot \mathscr{V}_1 + (1-p) \cdot \mathscr{V}_2}$$

$$\frac{m \Downarrow_s \{\!\{ v_i^{p_i} \mid i \in \mathcal{I} \}\!\} \qquad \forall i \in \mathcal{I}.\ n[v_i/x] \Downarrow_s \mathscr{V}_i}{\mathsf{let}\ x = m\ \mathsf{in}\ n \Downarrow_s \sum_{i \in \mathcal{I}} p_i \cdot \mathscr{V}_i} \qquad \frac{}{v :: \tau \Downarrow_s \{\!\{v^1\}\!\}} \qquad \frac{m \Downarrow_s \mathscr{V}}{m :: T \Downarrow_s \mathscr{V}}$$

Fig. 3. Runtime semantics of SPLC (excerpt).

$$\frac{}{\mathsf{Real} =_s \mathsf{Real}} \qquad \frac{}{\mathsf{Bool} =_s \mathsf{Bool}} \qquad \frac{\tau_1 =_s \tau_2 \qquad T_1 =_s T_2}{\tau_1 \to T_1 =_s \tau_2 \to T_2} \qquad \frac{\forall \tau \in \mathsf{supp}(T_1).\ T_1(\tau) = T_2(\tau)}{T_1 =_s T_2}$$

where $\mathsf{supp}(T)$ represents the *support* of distribution type T defined by $\mathsf{supp}(T) = \{\tau \mid \tau^p \in T \wedge p > 0\}$ and $T(\tau)$ represents the *probability* that T assigns to a simple type τ, defined by $\{\!\{\tau_i^{p_i} \mid i \in \mathcal{I}\}\!\}(\tau) = \sum_{i \in \mathcal{I} \mid \tau_i =_s \tau} p_i$.

Following the approach of Garcia et al. [2016], to ease the gradualization process we make all type relations and type functions explicit. For (Tapp) rule, we use partial functions *dom* and *cod* to extract the domain and codomain of a function type, respectively. Also we make explicit the fact that the type of the argument should be equal to the domain type of the function. Rule (T⊕) combines the distribution types T_1 and T_2 of sub-expressions m and n, by first scaling T_1 by p and T_2 by $1-p$, and then adding the resulting scaled distributions types together. Scaling $p \cdot T$ is defined pointwise, i.e. by scaling all the probabilities in the distribution type by p; the addition of two (sub) distribution types $T_1 + T_2$ is defined as the union of the two multi-sets, provided that the sum of the resulting probabilities do not exceed 1. For instance, consider program $1 \oplus_{\frac{1}{3}} \mathsf{true}$. Expression 1 is typed as $\{\!\{\mathsf{Real}^1\}\!\}$ and true as $\{\!\{\mathsf{Bool}^1\}\!\}$. After scaling both distribution types and adding them together, the resulting distribution type is $\frac{1}{3} \cdot \{\!\{\mathsf{Real}^1\}\!\} + \frac{2}{3} \cdot \{\!\{\mathsf{Bool}^1\}\!\} = \{\!\{\mathsf{Real}^{\frac{1}{3}}, \mathsf{Bool}^{\frac{2}{3}}\}\!\}$. Rule (Tlet) propagates the type of m to n as follows. If m has a distribution type T, then for each type and probability $\tau^p \in T$, n is type checked under an extended environment where x is typed as τ. The resulting type of the let expression is computed by adding each distribution type of n scaled by its corresponding p.

Note that, as expected, well-typed terms are assigned well-formed types, only.

Lemma 3.1 (Type well-formedness). *For every value v, every term m, every simple type $\tau \in \mathit{Type}$, every distribution type $T \in \mathit{DType}$ and every environment Γ,*

(1) If $\Gamma \vdash_s v : \tau$, then $\vdash \tau$ $\qquad$ *(2) If $\Gamma \vdash_s m : T$, then $\vdash T$*

3.3 Dynamic Semantics

We endow SPLC with a big-step *distribution-based* semantics that relates programs to probability distributions over final values [Lago and Zorzi 2012a], following a call-by-value reduction strategy. Concretely, judgment $m \Downarrow_s \mathscr{V}$ denotes that expression m reduces to a *distribution value* $\mathscr{V}$, i.e. a probability distribution over values. The reduction relation is formally defined in Figure 3.

A value v reduces to a Dirac distribution, i.e. a distribution that assigns probability 1 to v (and 0 to any other value). A function application reduces by substituting the argument for the variable binder in the function body. A probabilistic choice first reduces its pair of branches and then returns the weighted sum of the so obtained distribution values. The scaling and addition operators for

$$\mathsf{r} \in \mathbb{R}, \quad \mathsf{b} \in \mathbb{B}, \quad \mathsf{x} \in \mathrm{Var}, \quad p \in [0,1], \quad \rho \in \textsc{GProb}, \quad \sigma \in \textsc{GType}, \quad \mu \in \textsc{GDType}$$

$$\begin{array}{rcll}
\rho & ::= & p \mid ? & \text{(gradual probabilities)} \\
\sigma, \delta & ::= & \mathsf{Real} \mid \mathsf{Bool} \mid \sigma \to \mu \mid ? & \text{(gradual simple types)} \\
\mu, \nu & ::= & \{\!\{ \sigma_i^{\rho_i} \mid i \in \mathcal{I} \}\!\} & \text{(gradual distribution types)} \\
\mathsf{m}, \mathsf{n} & ::= & \mathsf{v} \mid \mathsf{v}\ \mathsf{w} \mid \mathsf{let\ x = m\ in\ n} \mid \mathsf{m} \oplus_\rho \mathsf{n} & \text{(terms)} \\
 & & \mathsf{m :: \mu} \mid \mathsf{v :: \sigma} \mid \mathsf{if\ v\ then\ m\ else\ n} \mid \mathsf{v + w} & \\
\mathsf{v}, \mathsf{w} & ::= & \mathsf{x} \mid \mathsf{r} \mid \mathsf{b} \mid \lambda \mathsf{x} : \sigma . \mathsf{m} & \text{(values)}
\end{array}$$

Fig. 4. Syntax of GPLC.

distribution values are defined analogously to those for type distributions (Fig. 2). For example, program $(1 \oplus_{\frac{1}{2}} 2) \oplus_{\frac{2}{3}} \mathsf{true}$ reduces to distribution value $\{\!\{ 1^{\frac{1}{3}}, 2^{\frac{1}{3}}, \mathsf{true}^{\frac{1}{3}} \}\!\}$. The reduction of a let–expression $\mathsf{let}\ x = m\ \mathsf{in}\ n$ is more involved and proceeds as follows. First, subterm m is reduced to a distribution value $\{\!\{ v_i^{p_i} \mid i \in \mathcal{I} \}\!\}$. Second, subterm n is reduced by substituting each v_i (i.e. each possible outcome of m) for x, resulting in distribution values $\mathscr{V}_i$. The entire let–expression then reduces to the weighted sum $\sum_{i \in \mathcal{I}} p_i \cdot \mathscr{V}_i$. Finally, ascribed terms reduce by removing type ascription.

SPLC is *type safe*, meaning that every well-typed closed expression reduces to a distribution value. Formally, this follows from three results of GPLC that we establish in Section 4 (Theorem 4.13) and Section 5 (Theorems 5.11 and 5.15).

4 GPLC: GRADUAL SOURCE LANGUAGE

We now present GPLC, our gradual source probabilistic language. GPLC is derived from SPLC and its design is justified by the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016]. The section is structured as follows. First, we introduce GPLC syntax, specifying, in particular, where we support (im)precision. Second, we present GPLC type system and definine consistency, discussing why a naive approach to consistency is bound to fail. Third, we define type and term precision, proving that GPLC satisfies the gradual guarantee and that its type system conservatively extends that of SPLC. The dynamic semantics of GPLC is defined through an elaboration to a target language, introduced in Section 5.[4]

4.1 Syntax

The syntax of GPLC is presented in Figure 4. We introduce imprecision in the language by extending probabilities and simple types with the unknown annotation ?. The unknown probability ? represents any probability in the interval $[0, 1]$, and similarly, the unknown simple type ? represents any static simple type. We do not need an (explicit) unknown distribution type as it can already be encoded by the singleton distribution type $\{\!\{ ?^? \}\!\}$ (of unknown simple type, with unknown probability). Notationwise, we use ρ to range over gradual probabilities (GProb), σ, δ to range over simple gradual types (GType), and μ, ν to range over gradual distribution types (GDType).

Design driven by AGT. To justify some of the design decisions behind GPLC, we follow, in parallel, the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016]. In short, the idea behind AGT is that starting from a specification of the meaning of gradual types in terms of sets of static types, we can systematically derive all relevant notions of the gradual language, which by construction, will enjoy a set of desired properties (to be discussed later). Unfortunately, some of the so-obtained definitions turn out not to be very amenable to implementation. To address this limitation, we also derive alternative (equivalent) definitions, with a more operational nature.

[4]We use the blue color for source languages and the red color for target languages.

$$\boxed{\Gamma \vdash \mathsf{v} : \sigma, \quad \Gamma \vdash \mathsf{m} : \mu}$$

$$\frac{\Gamma \vdash \mathsf{v} : \sigma}{\Gamma \vdash \mathsf{v} : \{\!|\sigma^1|\!\}} \qquad \frac{\Gamma, \mathsf{x} : \sigma \vdash \mathsf{m} : \mu \qquad \vdash \sigma}{\Gamma \vdash \lambda \mathsf{x} : \sigma . \mathsf{m} : \sigma \to \mu} \qquad \frac{\Gamma \vdash \mathsf{v} : \sigma \qquad \sigma \sim \delta \qquad \vdash \delta}{\Gamma \vdash \mathsf{v} :: \delta : \{\!|\delta^1|\!\}}$$

$$\frac{\Gamma \vdash \mathsf{v} : \sigma \qquad \Gamma \vdash \mathsf{w} : \delta \qquad \delta \sim \widetilde{dom}(\sigma)}{\Gamma \vdash \mathsf{v}\ \mathsf{w} : \widetilde{cod}(\sigma)} \qquad \frac{\Gamma \vdash \mathsf{m} : \mu \qquad \Gamma \vdash \mathsf{n} : \nu}{\Gamma \vdash \mathsf{m} \oplus_\rho \mathsf{n} : \rho \cdot \mu + (1-\rho) \cdot \nu}$$

$$\frac{\Gamma \vdash \mathsf{m} : \{\!|\sigma_i^{\rho_i} \mid i \in \mathcal{I}|\!\} \qquad \forall i \in \mathcal{I}.\, \Gamma, \mathsf{x} : \sigma_i \vdash \mathsf{n} : \mu_i}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{m}\ \mathsf{in}\ \mathsf{n} : \sum_{i \in \mathcal{I}} \rho_i \cdot \mu_i} \qquad \frac{\Gamma \vdash \mathsf{m} : \mu \qquad \mu \sim \nu \qquad \vdash \nu}{\Gamma \vdash \mathsf{m} :: \nu : \nu}$$

Fig. 5. Type system of GPLC (excerpt).

As just hinted, we start providing the meaning of gradual types and probabilities via concretization functions that map *gradual* simple types, distribution types and probabilities to non-empty sets of *static* simple types, distribution types and probabilities, respectively.

$$\gamma_p : [0,1] \cup \{?\} \to \mathscr{P}([0,1]) \qquad \gamma_p(?) = [0,1] \qquad \gamma_p(p) = \{p\}$$

$$\gamma_\tau : \textsc{GType} \to \mathscr{P}(\textsc{Type}) \qquad \gamma_\tau(\sigma \to \mu) = \{\tau \to T \mid \tau \in \gamma_\tau(\sigma) \wedge T \in \gamma_T(\mu)\} \qquad \gamma_\tau(?) = \textsc{Type} \qquad \gamma_\tau(\mathsf{Real}) = \{\mathsf{Real}\} \qquad \gamma_\tau(\mathsf{Bool}) = \{\mathsf{Bool}\}$$

$$\gamma_T : \textsc{GDType} \to \mathscr{P}(\textsc{DType}) \qquad \gamma_T(\{\!|\sigma_i^{\rho_i} \mid i \in \mathcal{I}|\!\}) = \{\{\!|\tau_i^{p_i} \mid i \in \mathcal{I}|\!\} \mid \forall i \in \mathcal{I}.\, \tau_i \in \gamma_\tau(\sigma_i) \wedge p \in \gamma_p(\rho_i)\}$$

The concretization functions crisply captures the intuition behind imprecision: The meaning of the unknown gradual probability is any probability in the interval $[0,1]$ and the meaning of the unknown gradual simple type is any static simple type. The meaning of a gradual distribution type is computed inductively, by computing the meaning of both gradual simple types and gradual probabilities.

4.2 Type System

Figure 5 shows the type system of GPLC, which is obtained from the type system of the static language (Figure 2) by replacing static elements with their gradual counterpart. Let us briefly describe these liftings. The lifting $\widetilde{dom}$: $\textsc{GType} \rightharpoonup \textsc{GType}$ (resp. $\widetilde{cod}$: $\textsc{GType} \rightharpoonup \textsc{GDType}$) of type function *dom* is standard: $\widetilde{dom}(\sigma \to \mu) = \sigma$, $\widetilde{dom}(?) = ?$, and $\widetilde{dom}$ is undefined elsewhere. Function *cod* is defined analogously. The lifting of the minus (resp. product) operation between probabilities, also denoted by $-$ (resp. $\cdot$), returns ? if either of the operands is ?:

$$\rho_1\ op\ \rho_2 = \begin{cases} \rho_1\ op\ \rho_2 & \text{if } \rho_1, \rho_2 \in [0,1] \\ ? & \text{otherwise} \end{cases} \qquad op \in \{\cdot, -\}$$

The lifting of the scaling of distribution types, also denoted by $\cdot$, is defined pointwise, in terms of the lifting of the product between probabilities: $\rho \cdot \{\!|\sigma_i^{\rho_i} \mid i \in \mathcal{I}|\!\} = \{\!|\sigma_i^{\rho \cdot \rho_i} \mid i \in \mathcal{I}|\!\}$. The lifting of the sum between distribution types, also denoted by $+$, coincides with the original operation (see Fig. 2).[5]

The lifting of type equality, called type *consistency* and denoted by $\sim$ in GPLC, plays a fundamental role in gradual languages. It allows soundly handling the notion of (im)precision, which is conveniently introduced via type ascriptions. For example, program $1 \oplus_{\frac{1}{2}} \mathsf{true} :: \{\!|?^?|\!\} :: \{\!|\mathsf{Real}^{\frac{2}{3}}, \mathsf{Bool}^{\frac{1}{3}}|\!\}$ is (optimistically) accepted by the gradual type system of GPLC because $\{\!|\mathsf{Real}^{\frac{1}{2}}, \mathsf{Bool}^{\frac{1}{2}}|\!\} \sim \{\!|?^?|\!\}$

[5]Formally, side condition $\sum_{i \in \mathcal{I}} \rho_i + \sum_{j \in \mathcal{J}} \rho_j \leq 1$ is defined following the AGT approach, i.e. it holds if there exist concretizations $p_i \in \gamma_p(\rho_i)$ and $p_j \in \gamma_p(\rho_j)$ such that $\sum_{i \in \mathcal{I}} p_i + \sum_{j \in \mathcal{J}} p_j \leq 1$.

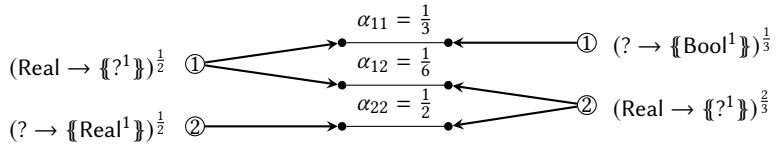

Fig. 6. Probability splitting to justify consistency between gradual distribution types.

and $\{\!|?^?|\!\} \sim \{\!|\text{Real}^{\frac{2}{3}}, \text{Bool}^{\frac{1}{3}}|\!\}$. Following AGT, we define type consistency by the existential lifting of type equality $=_s$:

Definition 4.1 (Type consistency, by AGT). For any pair of gradual simple types $\sigma, \delta \in \text{GType}$ and any pair of gradual distribution types $\mu, \nu \in \text{GDType}$, we define:

$$\begin{array}{lll} \sigma \sim_{\text{AGT}} \delta & \text{if and only if} & \exists \tau_1 \in \gamma_\tau(\sigma), \tau_2 \in \gamma_\tau(\delta).\ \tau_1 =_s \tau_2 \\ \mu \sim_{\text{AGT}} \nu & \text{if and only if} & \exists T_1 \in \gamma_T(\mu), T_2 \in \gamma_T(\nu).\ T_1 =_s T_2 \end{array}$$

In words, two gradual types are consistent if there exist static simple types in their concretizations that are equal. The problem with this definition is that is not practical, as it can depend on sets of infinitely many types. For gradual simple types, this can be partially addressed by stating that ? is consistent with every other gradual simple type, but for gradual distribution types the problem is more challenging as probabilities must also be taken into account.

4.3 Consistency, Refined

We are thus interested in an *inductive* definition of consistency. To illustrate the main idea behind our alternative characterization, consider the pair of gradual distribution types

$$\mu = \{\!|(\text{Real} \to \{\!|?^1|\!\})^{\frac{1}{2}}, (? \to \{\!|\text{Real}^1|\!\})^{\frac{1}{2}}|\!\} \qquad \text{and} \qquad \nu = \{\!|(? \to \{\!|\text{Bool}^1|\!\})^{\frac{1}{3}}, (\text{Real} \to \{\!|?^1|\!\})^{\frac{2}{3}}|\!\},$$

represented on the left and right hand side of Figure 6 (for concreteness, we assume that the elements of μ and ν are enumerated by index set $\mathcal{I} = \{1, 2\}$, thus, e.g., $(\text{Real} \to \{\!|?|\!\})^{\frac{1}{2}}$ corresponds to the simple type of index 1 in μ). Intuitively, μ and ν will be consistent if (and only if) there exists a *splitting* of the probabilities $\frac{1}{2}, \frac{1}{2}$ from μ and $\frac{1}{3}, \frac{2}{3}$ from ν that relates the simple types in μ with the simple types in and ν as follows:

(1) Type $\text{Real} \to \{\!|?^1|\!\}$ in μ is consistent with both $? \to \{\!|\text{Bool}^1|\!\}$ and $\text{Real} \to \{\!|?^1|\!\}$ in ν. This means that $\frac{1}{2}$, the probability of $\text{Real} \to \{\!|?^1|\!\}$ in μ, must be split into two, i.e. $\frac{1}{2} = \alpha_{11} + \alpha_{12}$, where α_{11} (resp. α_{12}) represents the probability of relating $\text{Real} \to \{\!|?^1|\!\}$, the first simple type in μ, with $? \to \{\!|\text{Bool}^1|\!\}$ (resp. $\text{Real} \to \{\!|?^1|\!\}$), the first (resp. second) simple type in ν.
(2) Type $? \to \{\!|\text{Real}^1|\!\}$ in μ is consistent only with $\text{Real} \to \{\!|?^1|\!\}$ in ν. Therefore, the probability of $? \to \{\!|\text{Real}^1|\!\}$ in μ need not be split, leading to $\frac{1}{2} = \alpha_{22}$.
(3) Similarly, type $? \to \{\!|\text{Bool}^1|\!\}$ in ν is consistent only with $\text{Real} \to \{\!|?^1|\!\}$ in μ, so $\frac{1}{3} = \alpha_{11}$.
(4) Finally, type $\text{Real} \to \{\!|?^1|\!\}$ in ν is consistent with $\text{Real} \to \{\!|?^1|\!\}$ and $? \to \{\!|\text{Real}^1|\!\}$ in μ, resulting in $\frac{2}{3} = \alpha_{12} + \alpha_{22}$.

Since the system of four equations so derived is feasible, witnessed e.g. by solution $\alpha_{11} = \frac{1}{3}$, $\alpha_{12} = \frac{1}{6}$ and $\alpha_{22} = \frac{1}{2}$, we can conclude that μ and ν are consistent. This thought process constitutes a lifting of the consistency relation between simple types to distribution types through *couplings* [Deng and Du 2011], tool that has already been exploited e.g. in the context of probabilistic bisimulation [Segala and Lynch 1995] and verification of cryptographic properties [Barthe et al. 2009].

Definition 4.2. (Relation lifting) Assume that $\mathscr{A} = \{\!|a_i^{p_i} \mid i \in \mathcal{I}|\!\}$ and $\mathscr{B} = \{\!|b_j^{q_j} \mid j \in \mathcal{J}|\!\}$ are multi-set representations of discrete probability distributions over sets A and B, respectively (that is,

$a_i \in A$ and $p_i \in [0,1]$ for all $i \in \mathcal{I}$, $b_j \in B$ and $q_j \in [0,1]$ for all $j \in \mathcal{J}$, and $\sum_{i\in\mathcal{I}} p_j = \sum_{j\in\mathcal{J}} q_j = 1$). Moreover, let $R \subseteq A \times B$ be a relation between A and B. We say that $\mathscr{A}$ and $\mathscr{B}$ are related by the *lifting* of R, written $L_R(\mathscr{A}, \mathscr{B})$, iff there exist $\mathscr{C} = \{\!\{\alpha_{ij} \in [0,1] \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}\!\}$ such that for all $i \in \mathcal{I}$ and all $j \in \mathcal{J}$,

$$(1)\ \ p_i = \textstyle\sum_{j\in\mathcal{J}} \alpha_{ij} \wedge p_j = \sum_{i\in\mathcal{I}} \alpha_{ij}, \qquad \text{and} \qquad (2)\ \ \alpha_{ij} > 0 \Rightarrow a_i\ R\ b_j$$

We write $\mathscr{C} \vdash \mathscr{A}\ R\ \mathscr{B}$ to denote that $\mathscr{C}$ is a witness of the relation $L_R(\mathscr{A}, \mathscr{B})$, i.e. to denote the conjunction between conditions 1 and 2 above. Moreover, any $\mathscr{C}$ satisfying (only) condition 1 is called a *coupling* between $\mathscr{A}$ and $\mathscr{B}$.

Type equality via couplings. To make a uniform treatment of type equality ($=_s$) in SPLC and type consistency ($\sim$) in GPLC, we start by redefining the type equality in SPLC in terms of couplings, via relation $=$:

$$\frac{}{\mathsf{Real} = \mathsf{Real}} \qquad \frac{}{\mathsf{Bool} = \mathsf{Bool}} \qquad \frac{\tau_1 = \tau_2 \quad T_1 = T_2}{\tau_1 \to T_1 = \tau_2 \to T_2} \qquad \frac{L_=(T_1, T_2)}{T_1 = T_2}$$

The last rule above says that two distribution types are equal if there exists a coupling that justifies the lifting of equality on simple types to distribution types (note that the $=$ symbol in the rule premise refers to equality over simple types, while the $=$ symbol in the conclusion refers to equality over distribution types). Using this rule we can, e.g., derive that $\{\!\{\mathsf{Real}^{\frac{1}{2}}, \mathsf{Bool}^{\frac{1}{2}}\}\!\} = \{\!\{\mathsf{Real}^{\frac{1}{3}}, \mathsf{Real}^{\frac{1}{6}}, \mathsf{Bool}^{\frac{1}{2}}\}\!\}$ because the set of formulas $\frac{1}{2} = \alpha_{11} + \alpha_{12}$, $\frac{1}{2} = \alpha_{23}$, $\frac{1}{3} = \alpha_{11}$, $\frac{1}{6} = \alpha_{12}$ and $\frac{1}{2} = \alpha_{23}$ is satisfiable (by solution $\alpha_{11} = \frac{1}{3}$, $\alpha_{12} = \frac{1}{6}$, $\alpha_{23} = \frac{1}{2}$).

As expected, this alternative definition of equality is equivalent to the original from Section 3.

Lemma 4.3 (Alt. characterization of equality). *For all pairs of simple types $\tau_1, \tau_2 \in$ Type and distributions types $T_1, T_2 \in$ DType,*

$$\tau_1 =_s \tau_2 \ \ \textit{iff}\ \ \tau_1 = \tau_2 \qquad \textit{and} \qquad T_1 =_s T_2 \ \ \textit{iff}\ \ T_1 = T_2$$

Armed with this new definition of equality based on couplings we proceed to define consistency.

Type consistency, a straightforward approach. A straightforward approach to define consistency in GPLC consists in (rule-wise) lifting the definition of type equality $=$ in SPLC, and extending it with rules stating that ? is consistent with any gradual simple type. An excerpt of the resulting set of rules would be:

$$\frac{}{? \sim \sigma} \qquad \frac{}{\mathsf{Real} \sim \mathsf{Real}} \qquad \frac{\sigma_1 \sim \sigma_2 \quad \mu_1 \sim \mu_2}{\sigma_1 \to \mu_1 \sim \sigma_2 \to \mu_2} \qquad \frac{L_\sim(\mu_1, \mu_2)}{\mu_1 \sim \mu_2}$$

According to this definition (in particular, by the last rule), establishing the consistency, e.g., between gradual distribution types $\{\!\{\sigma_i^{\rho_i} \mid i \in \mathcal{I}\}\!\}$ and $\{\!\{\sigma_j^{\rho_j} \mid j \in \mathcal{J}\}\!\}$ requires exhibiting a coupling between them. The problem here is that probabilities in either of the distribution types can be only partially known, that is, ρ_i could be ? for some $i \in \mathcal{I}$, rendering formula $\sum_{j\in\mathcal{J}} \alpha_{ij} = ?$ even ill-defined. A first approach to tackle this problem consists in lifting these formulas (from the static setting) to the gradual setting. Note, however, that this lifting cannot be done for each formula independently because the "same" ? will probably occur in multiple formulas. We should then lift all related formulas at the same time, but this still suffers from scoping problems because unknown probabilities (represented by ?) must remain visible outside the formulas lifting: At runtime, we need to carry witness information about consistency, where gradual probabilities "flow" across reductions (see the let and probabilistic choice reduction rules in Fig. 3). To tackle this problem, we introduce (fresh) symbolic variables representing unknown probabilities, and analyze the existence of couplings for this symbolic representation of distribution types.

$$\sigma, \delta \in \text{FSType}, \quad \mu, \nu \in \text{FDType}, \quad \omega \in \text{TVar}, \quad \Phi \in \text{Formula}$$

$$\begin{array}{rcll}
\omega & ::= & \langle \alpha, \ell, \kappa \rangle & \text{(tagged variables)} \\
\varrho & ::= & \omega \mid r & \text{(symbolic probabilities)} \\
\sigma, \delta & ::= & \text{Real} \mid \text{Bool} \mid \sigma \to \mu \mid ? & \text{(formula simple types)} \\
\mu, \nu & ::= & \Phi \triangleright \{\!\{ \sigma_i^{\varrho_i} \mid i \in \mathcal{I} \}\!\} & \text{(formula distribution types)} \\
\Phi & ::= & \varphi = \varphi \mid \varphi \leq \varphi \mid \Phi \wedge \Phi & \text{(formulas)} \\
\varphi & ::= & \varrho \mid \varphi + \varphi \mid \varphi - \varphi \mid \varphi \cdot \varphi \mid \varphi / \varphi & \text{(expressions)}
\end{array}$$

Fig. 7. Formula types.

Type consistency via symbolic liftings. To represent unknown probabilities as (free) variables in the set of formulas defining the lifting of consistency (from gradual simple types to gradual distribution types), we extend the syntax of GPLC with *formula simple types* (FSType) and *formula distribution types* (FDType) as shown in Figure 7. Intuitively, a formula type is the same as an ordinary gradual type, except that (1) unknown probabilities are replaced by variables, and (2) distribution types are guarded by formulas (like in refinement types). Formally, a *symbolic probability* ϱ is either a constant r or a tagged variable. A tagged variable ω represents a symbolic variable α (to be interpreted over the $[0, 1]$ interval), which for convenience is tagged by a pair of natural numbers ℓ and κ. For simplicity, we adopt the following notation conventions. First, we use $\omega.\alpha$, $\omega.\ell$ and $\omega.\kappa$ to access the first, second and third component of ω, respectively. Second, given $\omega = \langle \alpha, i, j \rangle$, we use $\omega(i, j)$ as a shorthand for α. Third, in order not to clutter formulas, we sometime write ω for $\omega.\alpha$ (e.g., $\omega_1 + \omega_2 = 1$ for $\omega_1.\alpha + \omega_2.\alpha = 1$). Finally, when clear from the context, we refer to tagged variables simply as variables. An *expression* φ represents either a symbolic probability ϱ or algebraic operations (addition, subtraction, multiplication or division) between symbolic probabilities. A *formula* Φ is either a comparison between two expressions or the conjunction of other two formulas. *Formula simple types* are defined similarly to gradual simple types (including the ? type), except that the codomain of function types are formula distribution types. Formula distribution types are now multi-sets of pairs of formula simple types and symbolic probabilities, closed under a formula Φ.

Let us introduce some handy notation for the rest of the presentation. First, given formula Φ, we use $FV(\Phi)$ to denote the set of (free) tagged variables occurring in Φ. Second, given a set of symbolic probabilities $\{\varrho_i \mid i \in \mathcal{I}\}$, we use $TV(\{\varrho_i \mid i \in \mathcal{I}\})$ to denote the subset of tagged variables, only (i.e. the result of filtering out static probabilities). Lastly, given formula Φ over tagged variables $\omega_1, \ldots, \omega_n$, we use $sat(\Phi)$ to denote that Φ is satisfiable, i.e. as a shorthand for $\exists \omega_1, \ldots, \exists \omega_n.\, \Phi$.

There is a canonical lifting from gradual types to formula gradual types. For example, the gradual distribution type $\{\!\{ \text{Int}^{\frac{1}{3}}, \text{Bool}^{?}, ?^{?} \}\!\}$ can be represented by the formula distribution type $\Phi \triangleright \{\!\{ \text{Int}^{\omega_1}, \text{Bool}^{\omega_2}, ?^{\omega_3} \}\!\}$, where $\Phi = \omega_1 = \frac{1}{3} \wedge \omega_2 \in [0, 1] \wedge \omega_3 \in [0, 1] \wedge \omega_1 + \omega_2 + \omega_3 = 1$, and $\varrho \in [r_1, r_2]$ is syntactic sugar for $r_1 \leq \varrho \wedge \varrho \leq r_2$. Formally, the lifting is captured by three mutually recursive functions that act over gradual simple types, gradual probabilities and gradual distribution types respectively as follows:

$$\lceil \cdot \rceil : \text{GType} \to \text{FSType}$$
$$\lceil \text{Real} \rceil = \text{Real} \qquad \lceil \text{Bool} \rceil = \text{Bool} \qquad \lceil ? \rceil = ? \qquad \lceil \sigma \to \mu \rceil = \lceil \sigma \rceil \to \lceil \mu \rceil$$

$$\lceil \cdot \rceil_{\cdot} : \text{GProb} \times \text{TVar} \to \text{Formula}$$
$$\lceil p \rceil_{\omega} = (\omega = p) \qquad \lceil ? \rceil_{\omega} = (\omega \in [0, 1])$$

$$\lceil \cdot \rceil : \text{GDType} \to \text{FDType}$$
$$\lceil \{\!\{ \sigma_i^{\rho_i} \mid i \in \mathcal{I} \}\!\} \rceil = \left(\textstyle\bigwedge_{i \in \mathcal{I}} \lceil \rho_i \rceil_{\omega_i} \wedge \sum_{i \in \mathcal{I}} \omega_i = 1 \right) \triangleright \{\!\{ \lceil \sigma_i \rceil^{\omega_i} \mid i \in \mathcal{I} \}\!\} \qquad \omega_i = \langle \alpha_i, i, i \rangle, \, \alpha_i \text{ is fresh}$$

To give the inductive definition of type consistency (and other forthcoming notions), we require a variant of the traditional notion of coupling. This variant differs from the traditional definition (see Def. 4.2) in that it operates over *symbolic* probability distributions, where probabilities are given by logical variables rather than contrete numbers, and these variables are subject to given constraints.

Definition 4.4. (Coupling over symbolic distributions) Assume that $\mathscr{A} = \{\!\{a_i^{p_i} \mid i \in \mathcal{I}\}\!\}$ and $\mathscr{B} = \{\!\{b_j^{q_j} \mid j \in \mathcal{J}\}\!\}$ are (multi-set representations of) symbolic discrete probability distributions over sets A and B. Moreover, let $R \subseteq A \times B$ be a relation between A and B. Given $\mathscr{C} = \{\!\{\alpha_{ij} \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}\!\}$, constraint ψ_1 over p_i, and constraint ψ_2 over p_j, we use $\mathscr{C} \vdash \mathscr{A}^{\psi_1}\ R\ \mathscr{B}^{\psi_2}$ to denote the conjunction of conditions 1 and 2 from Def. 4.2 (i.e. that $\mathscr{C}$ is a "traditional" coupling between $\mathscr{A}$ and $\mathscr{B}$), together with $\psi_1 \wedge \psi_2$. We also use $L_R(\mathscr{A}^{\psi_1}, \mathscr{B}^{\psi_2})$ to denote formula $\exists\{p_i \mid i \in \mathcal{I}\} \cup \{p_j \mid j \in \mathcal{J}\} \cup \{\alpha_{ij} \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}.\ \mathscr{C} \vdash \mathscr{A}^{\psi_1}\ R\ \mathscr{B}^{\psi_2}$.

Note that $L_R(\mathscr{A}^{\psi_1}, \mathscr{B}^{\psi_2})$ requires the existence not only of a coupling $\mathscr{C}$, but also of concretizations of (symbolic distributions) $\mathscr{A}$ and $\mathscr{B}$, respectively satisfying ψ_1 and ψ_2. Typically, ψ_1 and ψ_2 will require that probabilities sum up to 1. Like in Definition 4.4, in the rest of the presentation we allow ourselves some abuse of notation and write $\exists\{x_1, x_2, \ldots, x_n\}$ as a shorthand for $\exists x_1.\ \exists x_2.\ \ldots \exists x_n$, and similarly for $\forall\{x_1, x_2, \ldots, x_n\}$.

Armed with the above notion of relation lifting, we can define the lifting of any relation R over gradual simple types to gradual distribution types as:

$$L_R\Big(\Phi_1 \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\}, \Phi_2 \triangleright \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathcal{J}\}\!\}\Big) \quad \text{iff} \quad L_R\Big(\{\!\{\sigma_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\}^{\Phi_1}, \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathcal{J}\}\!\}^{\Phi_2}\Big)$$

Now, we can readily provide an inductive characterization of consistency, by simply lifting the definition of equality:

Definition 4.5 (Type consistency, inductively). The consistency relation $\sim$ between gradual types and formula distribution types ($\mu, \nu \in \text{FDType}$) is defined as follows:

$$\frac{}{\text{Real} \sim \text{Real}} \qquad \frac{}{\text{Bool} \sim \text{Bool}} \qquad \frac{}{\sigma \sim ?} \qquad \frac{}{? \sim \sigma}$$

$$\frac{\sigma_1 \sim \sigma_2 \quad \mu_1 \sim \mu_2}{\sigma_1 \to \mu_1 \sim \sigma_2 \to \mu_2} \qquad \frac{\lceil \mu_1 \rceil \sim \lceil \mu_2 \rceil}{\mu_1 \sim \mu_2} \qquad \frac{L_\sim(\mu, \nu)}{\mu \sim \nu}$$

As expected, the inductive definition of consistency (Def. 4.5) coincides with the one yielded by AGT (Def. 4.1):

Lemma 4.6 (Equivalence of consistencies). *For any pair of gradual simple types $\sigma, \delta \in GType$ and any pair of gradual distribution types $\mu, \nu \in GDType$,*

$$\sigma \sim_{AGT} \delta \ \textit{ iff } \ \sigma \sim \delta \qquad \textit{and} \qquad \mu \sim_{AGT} \nu \ \textit{ iff } \ \mu \sim \nu$$

Type well-formedness. Another relevant aspect of GPLC type system is that, like SPLC type system, programs are assigned well-formed types, only. The definition of well-formedness for gradual types is similar to that of static types, except that a gradual distribution type is well-formed iff it is *plausible* (rather than certain) that its underlying probabilities sum up to 1, and moreover, all its tagged variables occur in the closing formula:

Definition 4.7 (Type well-formedness). The well-formedness of gradual and formula types (denoted by symbol $\vdash$) is defined as follows:

$$\frac{}{\vdash \text{Real}} \qquad \frac{}{\vdash \text{Bool}} \qquad \frac{}{\vdash ?} \qquad \frac{\vdash \sigma \qquad \vdash \mu}{\vdash \sigma \to \mu} \qquad \frac{\vdash \lceil \mu \rceil}{\vdash \mu}$$

$$\frac{TV(\{\varrho_i \mid i \in \mathcal{I}\}) \subseteq FV(\Phi) \qquad sat\big(\Phi \wedge \textstyle\sum_{i\in\mathcal{I}} \varrho_i = 1\big) \qquad \forall i \in \mathcal{I}.\ \vdash \sigma_i}{\vdash \Phi \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\}}$$

Note that while the first line of rules defines well-formedness for both gradual simple and gradual distribution types, the second line defines well-formedness for formula distribution types, only. Well-formedness for formula simple types follows the same rules as for gradual simple types (first four rules above).

Lemma 4.8 (Type well-formedness). *For any value* v*, any term* m*, any gradual simple type* $\sigma \in GType$ *and gradual distribution type* $\mu \in GDType$ *from GPLC, and any environment* Γ*,*

(1) If $\Gamma \vdash \mathsf{v} : \sigma$*, then* $\vdash \sigma$ *(2) If* $\Gamma \vdash \mathsf{m} : \mu$*, then* $\vdash \mu$

An appealing property of the operator $\lceil \cdot \rceil$ lifting gradual distribution types to formula distribution types is that it preserves well-formedness:

Lemma 4.9 (Preservation of type well-formedness). *For any gradual simple type* $\sigma \in GType$*, and any gradual distribution type* $\mu \in GDType$*,*

(1) If $\vdash \sigma$*, then* $\vdash \lceil \sigma \rceil$ *(2) If* $\vdash \mu$*, then* $\vdash \lceil \mu \rceil$

4.4 Refined Criteria

The refined criteria for gradual languages [Siek et al. 2015b] establish a set of distinguishing properties for such class of languages, where (only) two such properties are related to the static semantics: the *static gradual guarantee*, which guarantees that typing is monotone with respect to imprecision, and the *conservative extension of the static discipline*, which guarantees that every fully-statically-annotated well-typed term in the gradual language is also typeable in the static language (and vice versa). To establish the first property, the static gradual guarantee for GPLC, we first need to define a notion of precision between types, and subsequently between terms.

Type precision. AGT casts the definition of type precision in terms of set containment on the concretization of the gradual types, i.e. $G_1 \sqsubseteq G_2$ (meaning that gradual type G_1 is at least as precise as gradual type G_2) if and only if $\gamma(G_1) \subseteq \gamma(G_2)$. Nevertheless, in the presence of gradual distribution types, the definition based on set containment is not satisfactory as it assumes a *syntactic* equality between set elements. For instance, while $\{\!\{\text{Real}^1\}\!\} \sqsubseteq \{\!\{\text{Real}^{\frac{1}{2}}, \text{Real}^{\frac{1}{2}}\}\!\}$ is expected to hold since the involved pair of types are equal (under our semantic view of equality), a naive definition of precision would reject this relation. Therefore, we adopt an alternative definition of precision by [Lennon-Bertrand et al. 2022], which can be successfully applied when equality is not syntactic.

Definition 4.10 (Type precision). For any pair of gradual simple types $\sigma, \delta \in \text{GType}$ and any pair of gradual distribution types $\mu, \nu \in \text{GDType}$,

(1) $\sigma \sqsubseteq_{AGT} \delta$ if and only if $\forall \tau_1 \in \gamma_\tau(\sigma).\ \exists \tau_2 \in \gamma_\tau(\delta).\ \tau_1 = \tau_2$.
(2) $\mu \sqsubseteq_{AGT} \nu$ if and only if $\forall T_1 \in \gamma_T(\mu).\ \exists T_2 \in \gamma_T(\nu).\ T_1 = T_2$.

Like for consistency, the definition of precision above, despite being sound, is unpractical. We thus present an alternative, inductive characterization. This inductive characterization is rather standard, only the case of (gradual and formula) distribution types deserving special attention;

$$\frac{\lceil \mu_1 \rceil \sqsubseteq \lceil \mu_2 \rceil}{\mu_1 \sqsubseteq \mu_2} \qquad \frac{\begin{array}{c}\forall\, FV(\Phi_1).\ \Phi_1 \implies \exists\, FV(\Phi_2) \cup \{\omega_{ij} \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}. \\ \{\!\{\omega_{ij} \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}\!\} \vdash \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\}^{\Phi_1} \sqsubseteq \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathcal{J}\}\!\}^{\Phi_2}\end{array}}{\Phi_1 \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\} \sqsubseteq \Phi_2 \triangleright \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathcal{J}\}\!\}}$$

Fig. 8. Type precision in GPLC (excerpt).

$$\frac{\mathsf{m} \sqsubseteq \mathsf{n} \qquad \mu \sqsubseteq \nu}{\mathsf{m} :: \mu \sqsubseteq \mathsf{n} :: \nu} \qquad \frac{}{p \sqsubseteq p} \qquad \frac{}{\rho \sqsubseteq ?} \qquad \frac{\mathsf{m} \sqsubseteq \mathsf{m}' \qquad \mathsf{n} \sqsubseteq \mathsf{n}' \qquad \rho \sqsubseteq \rho'}{\mathsf{m} \oplus_\rho \mathsf{n} \sqsubseteq \mathsf{m}' \oplus_{\rho'} \mathsf{n}'}$$

Fig. 9. Term precision in GPLC (excerpt).

see Figure 8. Two gradual distribution types are in precision if their lifting to formula distribution types are in precision. Precision for formula distribution types is slightly different from consistency. Loosely speaking, formula distribution types μ_1 and μ_2 are related by precision iff every solution that makes the probabilities of μ_1 sum up to 1 can be "completed" to form a coupling between μ_1 and μ_2 that witnesses the lifting of precision. Intuitively, the definition is designed to reproduce the quantifier structure of Definition 4.10.

To illustrate how this new definition of type precision works, consider the following examples:

- $\{\!\{\mathsf{Real}^{\frac{1}{2}}, ?^{\frac{1}{2}}\}\!\} \sqsubseteq \{\!\{\mathsf{Real}^{\frac{1}{3}}, \mathsf{Real}^{\frac{1}{6}}, ?^{\frac{1}{2}}\}\!\}$ holds because $\omega_{11} + \omega_{12} + \omega_{13} = \frac{1}{2} \wedge \omega_{21} + \omega_{22} + \omega_{23} = \frac{1}{2} \wedge \omega_{11} + \omega_{21} = \frac{1}{3} \wedge \omega_{12} + \omega_{22} = \frac{1}{6} \wedge \omega_{13} + \omega_{23} = \frac{1}{2}$ is satisfiable by the solution set $\{0 \leq \omega_{11} \leq \frac{1}{3}, 0 \leq \omega_{12} \leq \frac{1}{6}, 0 \leq \omega_{13} \leq \frac{1}{2}, 0 \leq \omega_{21} \leq \frac{1}{3}, 0 \leq \omega_{22} \leq \frac{1}{6}\}$.
- $\{\!\{\mathsf{Real}^{\frac{1}{2}}, ?^{\frac{1}{2}}\}\!\} \not\sqsubseteq \{\!\{\mathsf{Bool}^{\frac{2}{3}}, ?^{\frac{1}{3}}\}\!\}$ does not hold because $\omega_{12} = \frac{1}{2} \wedge \omega_{21} + \omega_{22} = \frac{1}{2} \wedge \omega_{21} = \frac{2}{3} \wedge \omega_{12} + \omega_{22} = \frac{1}{3}$ is not satisfiable.

As already hinted, this inductive definition of precision is equivalent to Definition 4.10:

Lemma 4.11 (Equivalence of type precision). *For any pair of gradual simple types $\sigma, \delta \in GType$ and any pair of gradual distribution types $\mu, \nu \in GDType$,*

$$\sigma \sqsubseteq_{AGT} \delta \text{ iff } \sigma \sqsubseteq \delta \qquad \text{and} \qquad \mu \sqsubseteq_{AGT} \nu \text{ iff } \mu \sqsubseteq \nu$$

Term precision. Term precision is the natural lifting of type precision to the space of terms. Its definition is rather standard, by induction in the term structure, as presented in Figure 9.

Metatheory. Armed with the definition of precision, we can now state the two fundamental properties that hold for the static semantics of GPLC. First, typeability is monotone w.r.t. imprecision:

Theorem 4.12 (Static Gradual Guarantee for GPLC). *For every value v, every term m, every gradual simple type σ and every gradual distribution type μ from GPLC,*

(1) If $\vdash \mathsf{v} : \sigma$ and $\mathsf{m} \sqsubseteq \mathsf{n}$, then there exists δ such that $\vdash \mathsf{n} : \delta$ and $\sigma \sqsubseteq \delta$.
(2) If $\vdash \mathsf{m} : \mu$ and $\mathsf{m} \sqsubseteq \mathsf{n}$, then there exists ν such that $\vdash \mathsf{n} : \nu$ and $\mu \sqsubseteq \nu$.

Second, the static semantics of SPLC and GPLC are equivalent for fully-statically-annotated terms:

Theorem 4.13 (Conservative extension of the static semantic). *For every value v, every term m, every simple type τ and every distribution type T from SPLC,*

$$\vdash_s v : \tau \text{ iff } \vdash m : \tau \qquad \text{and} \qquad \vdash_s m : T \text{ iff } \vdash m : \mathsf{T}$$

$$r \in \mathbb{R}, \quad b \in \mathbb{B}, \quad x \in \text{Var}, \quad \sigma \in \text{FSType}, \quad \mu \in \text{FDType}$$

$$\begin{array}{rcll} m, n & ::= & v \mid v\ w \mid \text{let } x = m \text{ in } n \mid m\, {}_{\varrho}\oplus^{\Phi}_{\varrho} n \mid \xi m :: \mu & \text{(terms)} \\ & & \varepsilon v :: \sigma \mid \text{if } v \text{ then } m \text{ else } n \mid v + w \mid \mathbf{error}_{\mu} & \\ u & := & r \mid b \mid (\lambda x : \sigma . m) & \text{(raw values)} \\ v, w & ::= & x \mid \varepsilon u :: \sigma \mid \mathbf{error}_{\sigma} & \text{(values)} \\ \mathscr{V} & ::= & \{\!\{ v_i^{\varrho_i} \mid i \in \mathcal{I} \}\!\} & \text{(distribution values)} \end{array}$$

Fig. 10. Syntax of TPLC (excerpt).

4.5 Dynamic Semantics

Traditionally, when designing gradual languages, the runtime semantics are not defined directly over the gradual source language. The program is translated or elaborated into a *cast calculus* program, inserting casts at the boundaries between static and dynamic typing, ensuring at runtime that no static assumptions are violated. If a static assumption is violated, then a runtime error is raised. This cast calculus is usually called the gradual target language. The dynamic semantics of GPLC are no exception: taking inspiration from AGT, we elaborate GPLC into an evidence-based gradual target language, where evidence play the role of casts that justify consistency judgments. The gradual target language for GPLC, dubbed TPLC, is presented next.

5 TPLC: GRADUAL TARGET LANGUAGE

In this section, we introduce TPLC, an evidence-based target language for GPLC. We start by presenting the static semantics, followed by the dynamic semantics, which relates programs to probability distributions over values. Finally, we establish type safety and two refine criteria for TPLC (dynamic counterparts of the refined criteria already established for GPLC): the gradual guarantee, and that the language is a conservative extension of SPLC, its static counterpart.

5.1 Static Semantics

The static semantics of TPLC differs from GPLC in five key aspects: (1) we use formula distribution types from the beginning, (2) consistency judgment are augmented with concrete type information (called *evidence*) that justifies judgment validity, (3) explicit ascriptions are incorporated along type derivations to push all consistency judgments to the ascription type rules, (4) ascriptions carry their underlying evidence to justify consistency transitivity at runtime, and (5) to simplify the reduction rules and proofs, all values are ascribed.

Syntax. Figure 10 presents the syntax of TPLC. Types are the formula types from GPLC (see Fig. 7). Terms are now annotated with formula simple types and formula distribution types introduced in previous section. The probabilistic choice operator $m\, {}_{\varrho_1}\oplus^{\Phi}_{\varrho_2} n$ is now annotated with variables ϱ_1 and ϱ_2 closed by formula Φ, corresponding to the probability of taking the left or right branch respectively. Ascriptions $\varepsilon v :: \sigma$ and $\xi m :: \mu$ are augmented with evidences, where ε is an evidence for a formula simple type consistency judgment, and ξ for a formula distribution type consistency judgment (both kind of evidences, to be defined later). A raw value is either a real number r, a constant b or a lambda abstraction $\lambda x : \sigma . m$. As previously mentioned, all values in GPLC become ascribed values in TPLC. Therefore, a value v is either a variable x, an ascribed raw value u, or a tagged error $\mathbf{error}_{\sigma}$. Note that in constrast to classical gradual approaches, in TPLC error is also a term, and can be either a redex ($\mathbf{error}_{\mu}$) or a value ($\mathbf{error}_{\sigma}$). The main reason for this is to simplify the metatheory when accounting for probabilistic branches that may fail during runtime. Errors also carry type information related to the expected type of the expression in order to establish type

$$\boxed{\Gamma \vdash v : \sigma, \quad \Gamma \vdash m : \mu, \quad \Gamma \vdash \Phi \triangleright \mathscr{V} : \mu}$$

$$(\text{Gerr}_\sigma)\frac{\vdash \sigma}{\Gamma \vdash \mathbf{error}_\sigma : \sigma} \qquad (\text{Gerr}_\mu)\frac{\vdash \mu}{\Gamma \vdash \mathbf{error}_\mu : \mu} \qquad (\text{G::}\sigma)\frac{\Gamma \vdash v : \sigma \quad \varepsilon \vdash \sigma \sim \delta \quad \vdash \delta}{\Gamma \vdash \varepsilon v :: \delta : \{\!\{\delta^1\}\!\}}$$

$$(\text{Gapp})\frac{\Gamma \vdash v : \sigma \to \mu \quad \Gamma \vdash w : \sigma}{\Gamma \vdash v\ w : \mu} \qquad (\text{Glet})\frac{\Gamma \vdash m : \Phi \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathscr{I}\}\!\} \quad \forall i \in \mathscr{I}.\, \Gamma, x : \sigma_i \vdash n : \mu_i}{\Gamma \vdash \text{let } x = m \text{ in } n : \Phi \vdash \sum_{i\in\mathscr{I}} \varrho_i \cdot \mu_i}$$

$$(\text{G::}\mu)\frac{\Gamma \vdash m : \mu \quad \xi \vdash \mu \sim \nu \quad \vdash \nu}{\Gamma \vdash \xi m :: \nu : \nu} \qquad (\text{G}\oplus)\frac{\Gamma \vdash m : \mu \quad \Gamma \vdash n : \nu \quad sat(\Phi \Rightarrow \varrho_1 + \varrho_2 = 1)}{\Gamma \vdash m\ {}^{\Phi}_{\varrho_1}\!\oplus_{\varrho_2} n : \Phi \vdash \varrho_1 \cdot \mu + \varrho_2 \cdot \nu}$$

$$(\text{GV})\frac{\forall i \in \mathscr{I}.\, \vdash v_i : \sigma_i}{\Gamma \vdash \Phi \triangleright \{\!\{v_i^{\varrho_i} \mid i \in \mathscr{I}\}\!\} : \Phi \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathscr{I}\}\!\}}$$

$$\varrho \cdot \Phi \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathscr{I}\}\!\} = \Phi \wedge \left(\textstyle\bigwedge_{i\in\mathscr{I}} \omega_i = \varrho \cdot \varrho_i\right) \triangleright \{\!\{\sigma_i^{\omega_i} \mid i \in \mathscr{I}\}\!\} \qquad \omega_i = \langle \alpha_i, \omega_i.\ell, \omega_i.r \rangle,\ \alpha_i \text{ fresh}$$

$$\Phi \vdash \textstyle\sum_{i\in\mathscr{I}} \Phi_i \triangleright \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathscr{J}_i\}\!\} = \Phi \wedge \left(\textstyle\bigwedge_{i\in\mathscr{I}} \Phi_i\right) \wedge \left(\textstyle\sum_{i\in\mathscr{I}} \sum_{j\in\mathscr{J}_i} \varrho_j = 1\right) \triangleright \bigcup_{i\in\mathscr{I}} \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathscr{J}_i\}\!\}$$

Fig. 11. Type system of TPLC (excerpt).

safety, and can be removed in a real implementation. Finally, a *distribution value* $\mathscr{V}$ stands for a distribution over values v.

Type System. The type system of TPLC is presented in Figure 11. Compared to GPLC, the only rules that use consistency are the ascription rules, making all top-level constructors match in the remaining type rules. Rule (G⊕) requires that the fact that formula Φ entails that probabilities ϱ_1 and ϱ_2 sum up to 1 be plausible. Note that formula Φ is also pushed as part of the constraints of the resulting type —the weighted sum between the branch types. Similary, rule (Glet) scales each μ_i with variable ϱ_i, so Φ is pushed to the resulting distribution type to close the type. Consistency judgments are now justified by some evidence, written $\varepsilon \vdash \sigma \sim \delta$ (for simple types) and $\xi \vdash \mu \sim \nu$ (for distribution types). Intuitively, evidences ε and ξ correspond to the most precise type information that support the respective consistency judgment; we elaborate on this in Section 5.2.

The notion of consistency of formula types is defined in the same way as in GPLC:

Definition 5.1 (Type consistency of formula types). Type consistency over simple (FSType) and distribution (FDType) formula types is defines as follows:

$$\frac{}{\text{Real} \sim \text{Real}} \qquad \frac{}{\text{Bool} \sim \text{Bool}} \qquad \frac{}{\sigma \sim ?} \qquad \frac{}{? \sim \sigma} \qquad \frac{\sigma_1 \sim \sigma_2 \quad \mu_1 \sim \mu_2}{\sigma_1 \to \mu_1 \sim \sigma_2 \to \mu_2} \qquad \frac{L_\sim(\mu, \nu)}{\mu \sim \nu}$$

The definition of well-formedness is defined identical to Def. 4.7, and omitted for brevity. Like in SPLC and GPLC, all well-typed terms type check to well-formed formula types:

Lemma 5.2. *For any value v, any term m, any formula simple type $\sigma \in$ FSType and formula distribution type $\mu \in$ FDType from TPLC, and any environment Γ,*

(1) If $\Gamma \vdash v : \sigma$, then $\vdash \sigma$ *(2) If $\Gamma \vdash m : \mu$, then $\vdash \mu$*

5.2 Evidence

Following AGT, evidences are encoded as pairs of (gradual) types of the form $\langle G_1, G_2 \rangle$. Intuitively, each type of the pair corresponds to a type in the consistent judgment that the evidence shall

justify, e.g. in $\langle G_1, G_2\rangle \vdash G_1' \sim G_2'$, G_1 corresponds to G_1' and G_2 to G_2'. Furthermore, each type in the evidence is at least as precise as its corresponding type in the consistent judgment, i.e. $G_1 \sqsubseteq G_1'$ and $G_2 \sqsubseteq G_2'$. When dealing with consistency (the gradual counterpart of equality), both types in evidence coincide, and therefore evidence is represented by single types, namely

$$\varepsilon ::= \sigma \quad \text{(simple evidences)} \qquad\qquad \xi ::= \mu \quad \text{(distribution evidences)}$$

A simple evidence ε (resp. distribution evidence ξ) is just a formula simple type (resp. formula distribution type) that justifies a consistency judgment between two formula simple types (resp. two formula distribution types). Formally, an evidence justifies a consistency judgment if and only if the evidence is more precise than both types:

Definition 5.3 (Evidence). For all formula simple types $\varepsilon, \sigma, \delta \in \text{FSType}$ and all formula distribution types $\xi, \mu, \nu \in \text{FDType}$, we define

$$\varepsilon \vdash \sigma \sim \delta \iff \varepsilon \sqsubseteq \sigma \wedge \varepsilon \sqsubseteq \delta \qquad \text{and} \qquad \xi \vdash \mu \sim \nu \iff \xi \sqsubseteq \mu \wedge \xi \sqsubseteq \nu$$

For instance, $\text{Real} \to \text{Bool} \vdash \text{Real} \to ? \sim ? \to \text{Bool}$. Now we can justify the role of tags in tagged variables. Consider judgment $\Phi \triangleright \{\!\{\sigma_k^{\omega_k}\}\!\} \vdash \Phi_1 \triangleright \{\!\{\sigma_i^{\varrho_i}\}\!\} \sim \Phi_2 \triangleright \{\!\{\delta_j^{\varrho_j}\}\!\}$. Tagged variables connect evidence with their underlying types, i.e. type σ_k justifies that σ_i is consistent with δ_j, because $\omega_k.\ell = i$ and $\omega_k.\prec = j$, where $\omega_k.\alpha$ is the *weight* of the connection between σ_i and δ_j. Note that a pair of simple types can be connected through multiple evidences.

As usual in gradual languages, consistency is not transitive, e.g. $\{\!\{\text{Real}^{\frac{2}{3}}, \text{Bool}^{\frac{1}{3}}\}\!\} \sim \{\!\{?^?\}\!\}$ and $\{\!\{?^?\}\!\} \sim \{\!\{\text{Real}^{\frac{1}{3}}, \text{Bool}^{\frac{2}{3}}\}\!\}$, but $\{\!\{\text{Real}^{\frac{2}{3}}, \text{Bool}^{\frac{1}{3}}\}\!\} \not\sim \{\!\{\text{Real}^{\frac{1}{3}}, \text{Bool}^{\frac{2}{3}}\}\!\}$. Therefore, during runtime, evidence is combined to try to justify transitivity. If the combination succeeds, the resulting (and possible more precise) evidence justifies the resulting judgment from transitivity, otherwise a runtime error is raised. The combination of evidence is formalized using the *consistent transitivity* operator, which coincides with the meet (least upper bound) operator w.r.t. the precision order, i.e.

$$\varepsilon_1 \circ \varepsilon_2 = \varepsilon_1 \sqcap \varepsilon_2 \qquad \text{and} \qquad \xi_1 \circ \xi_2 = \xi_1 \sqcap \xi_2$$

The meet operator is partial. For simple types, it is defined by the following clauses:

$$\text{Real} \sqcap \text{Real} = \text{Real} \qquad \text{Bool} \sqcap \text{Bool} = \text{Bool} \qquad ? \sqcap \sigma = \sigma \qquad \sigma \sqcap ? = \sigma$$

$$\sigma_1 \to \mu_1 \sqcap \sigma_2 \to \mu_2 = \sigma_1 \sqcap \sigma_2 \to \mu_1 \sqcap \mu_2$$

For distribution types, the definition follows the same approach as used for defining consistency and precision, in terms of the existence of couplings that justifies the lifting, but explicitly capturing all witness couplings. Formally,

$$\mu_1 \sqcap \mu_2 = W_\sqcap(\mu_1, \mu_2)$$

where $W\colon (\text{FSType}\times\text{FSType} \rightharpoonup \text{FSType})\times\text{FDType}\times\text{FDType} \rightharpoonup \text{FDType}$ returns (a characterization of) all couplings that witness the lifting, and is defined as:

$$W_f(\Phi_1 \triangleright \{\!\{\sigma_i^{\omega_i} \mid i \in \mathcal{I}\}\!\}, \Phi_2 \triangleright \{\!\{\delta_j^{\omega_j} \mid j \in \mathcal{J}\}\!\}) = \\ \Phi \triangleright \{\!\{f(\sigma_i, \delta_j)^{\omega_{ij}} \mid (i,j) \in \mathcal{I}\times\mathcal{J} \wedge (\sigma_i, \delta_j) \in dom(f)\}\!\} \qquad \omega_{ij} \text{ fresh}$$

provided $\exists FV(\Phi_1) \cup FV(\Phi_2) \cup \{\omega_{ij} \mid (i,j) \in \mathcal{I}\times\mathcal{J}\}.\ \Phi' \wedge \Phi$, where $\Phi' = \forall i \in \mathcal{I}.\ \forall j \in \mathcal{J}.\ \omega_{ij}.\ell = \omega_i.\ell \wedge \omega_{ij}.\prec = \omega_j.\prec$, $\Phi = \{\!\{\omega_{ij} \mid (i,j) \in \mathcal{I}\times\mathcal{J}\}\!\} \vdash \{\!\{\sigma_i^{\omega_i} \mid i \in \mathcal{I}\}\!\}^{\Phi_1}\ R\ \{\!\{\delta_j^{\omega_j} \mid j \in \mathcal{J}\}\!\}^{\Phi_2}$ and $\sigma_i\, R\, \delta_j$ iff $(\sigma_i, \delta_j) \in dom(f)$.[6]

[6]Note that Φ can be cast as a Formula by taking $\mathcal{K} = \{(i,j) \mid (\sigma_i, \delta_j) \in dom(f)\}$ as the index set of the witness couplings.

As an example of the meet between formula distribution types, observe that $(\omega_1 = \frac{1}{2} \wedge \omega_2 = \frac{1}{2}) \triangleright \{\!\{(\text{Real} \to ?)^{\omega_1}, (? \to \text{Real})^{\omega_2}\}\!\} \sqcap (\omega_3 = \frac{1}{3} \wedge \omega_4 = \frac{2}{3}) \triangleright \{\!\{(\text{Real} \to ?)^{\omega_3}, (? \to \text{Real})^{\omega_4}\}\!\} = (\omega_{11} + \omega_{21} = \omega_1 \wedge \omega_{22} = \omega_2 \wedge \omega_{11} = \omega_3 \wedge \omega_{21} + \omega_{22} = \omega_4) \triangleright \{\!\{(\text{Real} \to ?)^{\omega_{11}}, (\text{Real} \to \text{Real})^{\omega_{12}}, (\text{Real} \to \text{Real})^{\omega_{21}}, (? \to \text{Real})^{\omega_{22}}\}\!\}$ because $\omega_{11} + \omega_{21} = \omega_1 \wedge \omega_{22} = \omega_2 \wedge \omega_{11} = \omega_3 \wedge \omega_{21} + \omega_{22} = \omega_4$ is satisfiable by the solution set $\{\omega_{11} = \frac{1}{2}, \omega_{21} = \frac{1}{6}, \omega_{22} = \frac{1}{3}\}$.

Importantly, the meet between a pair of types is at least as precise as either of them (and therefore, a valid evidence for their consistency).

Lemma 5.4 (Monotonicity of the meet operator). *For all formula simple types $\sigma_1, \sigma_2, \sigma_3 \in$ FSType and all formula distribution types $\mu_1, \mu_2, \mu_3 \in$ FDType,*

(1) If $\sigma_3 = \sigma_1 \sqcap \sigma_2$, then $\sigma_3 \sqsubseteq \sigma_1 \wedge \sigma_3 \sqsubseteq \sigma_2$ *(2) If $\mu_3 = \mu_1 \sqcap \mu_2$, then $\mu_3 \sqsubseteq \mu_1 \wedge \mu_3 \sqsubseteq \mu_2$*

Armed with the definition of evidence and the meet operator, we can now state the following invariant for distribution evidences that crisply captures when an evidence is well-defined with respect to a judgment.

$$\text{(wdB)}\ \frac{\varepsilon \in \{\text{Real}, \text{Bool}, ?\} \qquad \varepsilon \sqsubseteq \sigma \qquad \varepsilon \sqsubseteq \delta}{\varepsilon \vdash \sigma \sim \delta} \qquad \text{(wd}\to\text{)}\ \frac{\varepsilon \vdash \sigma \sim \delta \qquad \xi \vdash \mu \sim \nu}{\varepsilon \to \xi \vdash \sigma \to \mu \sim \delta \to \nu}$$

$$\text{(wd}\xi\text{)}\ \frac{\begin{array}{c}\exists\, FV(\Phi) \cup FV(\Phi_1) \cup FV(\Phi_2).\ \Phi \wedge \Phi_1 \wedge \Phi_2 \wedge \Phi_L \wedge \Phi_R \wedge \Phi_\sim \\ \Phi_L = \forall i \in \mathscr{J}.\ \textstyle\sum_{k \mid \omega_k.\ell = i} \omega_k = \varrho_i \qquad \Phi_R = \forall j \in \mathscr{J}, \textstyle\sum_{k \mid \omega_k.r = j} \omega_k = \varrho_j \\ \Phi_\sim = \forall k \in \mathscr{K}.\omega_k > 0 \Rightarrow \sigma_k \vdash \sigma_{\omega_k.\ell} \sim \sigma_{\omega_k.r}\end{array}}{\Phi \triangleright \{\!\{\sigma_k^{\omega_k} \mid k \in \mathscr{K}\}\!\} \vdash \Phi_1 \triangleright \{\!\{\sigma_i^{\varrho_i} \mid i \in \mathscr{J}\}\!\} \sim \Phi_2 \triangleright \{\!\{\sigma_j^{\varrho_j} \mid j \in \mathscr{J}\}\!\}}$$

The invariant for distribution evidences ensures that (1) the sum of all the weights connected to a simple type must be equal to the probability of that type, and (2) each evidence in the distribution evidence of weight larger than zero must be well-defined (and thus more precise than both types involved in the consistency judgment). Finally, we use this invariant, to validate that the consistent transitivity operator preserves the invariant.

Lemma 5.5 (Invariant preservation). *For all formula simple types $\varepsilon_1, \varepsilon_2, \sigma_1, \sigma_2, \delta \in$ FSType and all formula distribution types $\xi_1, \xi_2, \mu_1, \mu_2, \nu \in$ FDType,*

(1) Let $\varepsilon_1 \vdash \sigma_1 \sim \delta$ and $\varepsilon_2 \vdash \delta \sim \sigma_2$. If $\varepsilon_1 \circ \varepsilon_2$ is defined, then $\varepsilon_1 \circ \varepsilon_2 \vdash \sigma_1 \sim \sigma_2$
(2) Let $\xi_1 \vdash \mu_1 \sim \nu$ and $\xi_2 \vdash \nu \sim \mu_2$. If $\xi_1 \circ \xi_2$ is defined, then $\xi_1 \circ \xi_2 \vdash \mu_1 \sim \mu_2$.

5.3 Dynamic Semantics

We now present the dynamic semantics for TPLC, which relates programs to probability distributions over final values, through a big-step reduction relation (like in SPLC). The adoption of a distribution semantics is key to establish the dynamic gradual guarantee (DGG), which requires that reduction be monotone with respect to imprecision. To see this, assume that we adopt a (e.g. sampling) semantics that relates programs to *individual* final values. Under this semantics, program $(\lambda x : \text{Bool}.(x :: ? + 1) \oplus_{\frac{1}{2}} \text{true})\ \text{false}$ reduces to true (with probability $\frac{1}{2}$), while the less precise program $(\lambda x : ?\,.(x :: ? + 1) \oplus_{\frac{1}{2}} \text{true})\ \text{false}$ reduces to an error (also with probability $\frac{1}{2}$), which contradicts the DGG. Nevertheless, we can recover the DGG by considering all possible program outcomes at the same time, via a distribution semantics.

The distribution semantics is presented in Figure 12. Reduction judgment $m \Downarrow_k \Phi \rhd \mathscr{V}$ denotes that term m reduces to *distribution configuration* $\Phi \rhd \mathscr{V}$ in k steps. The number of steps of reduction judgments are only required to establish the metatheory, and can be removed in a real implementation. Several rules are defined similarly to SPLC, but accounting for the fact that values

$\boxed{m \Downarrow_k \Phi \rhd \mathscr{V}}$ $\qquad \mathscr{V} ::= \ \{\!\{ v_i^{\varrho_i} \mid i \in \mathscr{I} \}\!\}$ (distribution values)

$$(Dapp)\ \frac{\widetilde{dom}(\varepsilon) v :: \delta \Downarrow_1 \cdot \rhd \{\!\{ w^1 \}\!\} \quad sub(\widetilde{cod}(\varepsilon) m :: \mu), w, x) \Downarrow_k \Phi \rhd \mathscr{V}}{(\varepsilon(\lambda x : \delta . m) :: \sigma \to \mu)\ v \Downarrow_{k+1} \Phi \rhd \mathscr{V}} \qquad (D\oplus)\ \frac{m \Downarrow_{k_1} \Phi_1 \rhd \mathscr{V}_1 \quad n \Downarrow_{k_2} \Phi_2 \rhd \mathscr{V}_2 \quad \Phi' = \Phi_1 \wedge \Phi_2 \wedge \Phi}{m \,{}^{\Phi}_{\varrho_1}\!\oplus_{\varrho_2}\! n \Downarrow_{k_1+k_2+1} \Phi' \rhd \varrho_1 \cdot \mathscr{V}_1 + \varrho_2 \cdot \mathscr{V}_2}$$

$$(Dlet)\ \frac{m \Downarrow_{k_1} \{\!\{ v_i^{\varrho_i} \mid i \in \mathscr{I} \}\!\} \quad \forall i \in \mathscr{I}.\ sub(n, v_i, x) \Downarrow_{k_2} \Phi_i \rhd \mathscr{V}_i}{\text{let } x = m \text{ in } n \Downarrow_{k_1+k_2+1} (\bigwedge_{i\in\mathscr{I}} \Phi_i) \rhd \sum_{i\in\mathscr{I}} \varrho_i \cdot \mathscr{V}_i} \qquad (Dv)\ \frac{}{v \Downarrow_1 \cdot \rhd \{\!\{ v^1 \}\!\}}$$

$$(Derr)\ \frac{\mu = \Phi \triangleright \{\!\{ \sigma_i^{\varrho_i} \mid i \in \mathscr{I} \}\!\}}{\mathbf{error}_\mu \Downarrow_1 \Phi \rhd \{\!\{ \mathbf{error}_{\sigma_i}^{\varrho_i} \mid i \in \mathscr{I} \}\!\}} \qquad (Dmon)\ \frac{m \Downarrow_k \mathscr{V}}{m \Downarrow_{k+1} \mathscr{V}}$$

$$(D::\sigma)\ \frac{}{\varepsilon_2(\varepsilon_1 u :: \sigma) :: \delta \Downarrow_1 \cdot \rhd \begin{cases} \{\!\{ (\varepsilon_3 u :: \delta)^1 \}\!\} & \text{If } \varepsilon_1 \circ \varepsilon_2 = \varepsilon_3 \\ \{\!\{ \mathbf{error}_\sigma^1 \}\!\} & \text{otherwise} \end{cases}}$$

$$(D::\mu)\ \frac{m \Downarrow_{k'} \Phi_1 \rhd \{\!\{ v_i^{\varrho_i} \mid i \in \mathscr{I} \}\!\} \quad \vdash \Phi_1 \triangleright \{\!\{ v_i^{\varrho_i} \mid i \in \mathscr{I} \}\!\} : \mu' \quad \xi \vdash \mu \sim \nu \quad \nu = \Phi_3 \triangleright \{\!\{ \delta_j^{\varrho_j} \mid j \in \mathscr{J} \}\!\}}{(\xi m :: \nu) \Downarrow_{k'+1} \begin{cases} \Phi_2 \rhd \sum_{k\in\mathscr{K}} \omega_k \cdot \mathscr{V}_k & \text{If } (\mu' \parallel \mu) \circ \xi = \Phi_2 \triangleright \{\!\{ \varepsilon_k^{\omega_k} \mid k \in \mathscr{K} \}\!\}, \text{ where } \forall k \in \mathscr{K}, \\ & i = \omega_k.\ell \wedge j = \omega_k.r \implies (\varepsilon_k v_i :: \delta_j) \Downarrow_1 \cdot \rhd \mathscr{V}_k \\ \{\!\{ \mathbf{error}_\nu^1 \}\!\} & \text{otherwise} \end{cases}}$$

$m[v/x] : \textsc{Term} \times \textsc{Value} \times \textsc{Var} \rightharpoonup \textsc{Term}$

$sub(m, \varepsilon u :: \sigma, x) = m[\varepsilon u :: \sigma / x] \qquad sub(m, \mathbf{error}_\sigma, x) = \mathbf{error}_\mu \quad \text{where } x : \sigma \vdash m : \mu$

Fig. 12. Distribution semantics of TPLC (excerpt).

are always ascribed. Rule (*Dapp*) first ascribes argument v to δ, appealing to transitivity with the domain of ε as evidence. After its reduction, the obtained value w is substituted for x in the body of the function using the auxiliary function *sub*. If during v reduction transitivity does not hold (and w is thus $\mathbf{error}_\delta$), *sub* yields term $\mathbf{error}_\mu$, μ being the expected distribution type of the application. Rule (*Dlet*) reduces subterm n by substituting x by all the possible outcomes of m, using also function *sub* to properly handle the case where one such outcome is an error. The so obtained distribution configurations are combined (distribution values via their weighted sum and formulas via their conjunction) to form the final outcome of the let–expression. Rule ($D\oplus$) reduces the pair of branches and combines their results like the (*Dlet*) rule, the major difference being that formula Φ is also included in the resulting distribution configuration. Rule (*Dv*) lifts values to (Dirac) distribution values. Rule (*Derr*) reduces an error over distribution type μ to a distribution of errors over simple types σ_i. Rule (*Dmon*) establishes the monotonicity of the reduction relation with respect to the step index. Rule ($D::\sigma$) analyzes whether type of u is consistent with δ, combining the respective evidences through the consistent transitivity operator. The rule yields either a Dirac distribution of a newly ascribed value (if consistent transitivity succeeds), or (else) an error.

Rule ($D::\mu$) is the most challenging. Intuitively, it "pushes" simple evidences within ξ into the outcomes of m. However, note that pushing every simple evidence within ξ into every possible outcome of m is not what we want. For instance, given the (informal) program $\{\!\{ \text{Real}^{\frac{1}{2}}, \text{Bool}^{\frac{1}{2}} \}\!\}(1 \,{}_{\frac{1}{2}}\!\oplus_{\frac{1}{2}}\! \text{true}) :: \{\!\{ \text{Real}^{\frac{1}{2}}, \text{Bool}^{\frac{1}{2}} \}\!\}$, it would be futile to push evidence Real into true, or Bool into 1. Here, we have two

problems to address. First, to determine what evidences must be pushed into what values. Second, to determine the probability of each such combination. We address both problems simultaneously, taking advantage of the consistent transitivity operator, and a subsidiary relation over formula types we introduce next.

Observe that rule $(D{::}\mu)$ proceeds by first reducing m to a value distribution of type μ'. However, this μ' can be (syntactically) different from μ, the actual type of m. What we require here is that μ' be a *reordering* of μ. We thus introduce the reordering relation $\overset{\mathrm{r}}{=}$ over formula types, which (for distribution types) is nothing more than the coupling lifting of the syntactic equality.

Definition 5.6 (Reordering). The reordering relation $\overset{\mathrm{r}}{=}$ over formula simple and distribution types is defined by the following clauses:

$$\frac{}{\mathsf{Real} \overset{\mathrm{r}}{=} \mathsf{Real}} \qquad \frac{}{\mathsf{Bool} \overset{\mathrm{r}}{=} \mathsf{Bool}} \qquad \frac{}{? \overset{\mathrm{r}}{=} ?} \qquad \frac{\sigma_2 \overset{\mathrm{r}}{=} \sigma_1 \quad \mu_1 \overset{\mathrm{r}}{=} \mu_2}{\sigma_1 \to \mu_1 \overset{\mathrm{r}}{=} \sigma_2 \to \mu_2} \qquad \frac{L_{\overset{\mathrm{r}}{=}}(\mu, \nu)}{\mu \overset{\mathrm{r}}{=} \nu}$$

We can construct an initial evidence for reordering judgments similarly to the meet operator:

Definition 5.7 (Reordering initial evidence). The partial operator $\mid\mid$ over formula simple and distribution types is defined as follows:

$$\mathsf{Real} \mid\mid \mathsf{Real} = \mathsf{Real} \qquad \mathsf{Bool} \mid\mid \mathsf{Bool} = \mathsf{Bool} \qquad ? \mid\mid ? = ?$$

$$(\sigma_1 \to \mu_1 \mid\mid \sigma_2 \to \mu_2) = (\sigma_1 \mid\mid \sigma_2) \to (\mu_1 \mid\mid \mu_2) \qquad \mu_1 \mid\mid \mu_2 = W_{\mid\mid}(\mu_1, \mu_2)$$

Lemma 5.8. *For all formula simple types $\sigma_1, \sigma_2 \in \mathit{FSType}$ and all formula distribution types $\mu_1, \mu_2 \in \mathit{FDType}$,*

(1) If $\sigma_1 \overset{\mathrm{r}}{=} \sigma_2$, then $\sigma_1 \mid\mid \sigma_2$ is defined, and $\sigma_1 \mid\mid \sigma_2 \vdash \sigma_1 \overset{\mathrm{r}}{=} \sigma_2$.
(2) If $\mu_1 \overset{\mathrm{r}}{=} \mu_2$, then $\mu_1 \mid\mid \mu_2$ is defined, and $\mu_1 \mid\mid \mu_2 \vdash \mu_1 \overset{\mathrm{r}}{=} \mu_2$.

Here, $\varepsilon \vdash \sigma_1 \overset{\mathrm{r}}{=} \sigma_2$ means that evidence ε justifies reordering $\sigma_1 \overset{\mathrm{r}}{=} \sigma_2$ and holds if $\varepsilon \sqsubseteq \sigma_1 \mid\mid \sigma_2$. The notion of evidence for the reordering between distribution types is defined analogously.

Reordering and consistency interact nicely, in that their evidences can be soundly combined:

Lemma 5.9. *For all formula simple types $\varepsilon, \varepsilon', \sigma, \sigma', \delta \in \mathit{FSType}$ and all formula distribution types $\xi, \xi', \mu, \mu', \nu \in \mathit{FDType}$,*

(1) If $\varepsilon \vdash \sigma \overset{\mathrm{r}}{=} \sigma'$, $\varepsilon' \vdash \sigma' \sim \delta$ and $\varepsilon \circ \varepsilon'$ is defined, then $\varepsilon \circ \varepsilon' \vdash \sigma \sim \delta$.
(2) If $\xi \vdash \mu \overset{\mathrm{r}}{=} \mu'$, $\xi' \vdash \mu' \sim \nu$ and $\xi \circ \xi'$ is defined, then $\varepsilon \circ \varepsilon' \vdash \mu \sim \nu$.

Returning to rule $(D{::}\mu)$, observe that evidence $(\mu' \mid\mid \mu) \circ \xi$ addresses both of the mentioned problems: every simple evidence ε_k in $(\mu' \mid\mid \mu) \circ \xi$ connects a simple type from μ' with a simple type from ν, via its corresponding weight ω_k. To form the final distribution value, evidence ε_k is pushed to the ascription of value $v_{\omega_k.\ell}$ with type $\delta_{\omega_k.r}$. After reducing these terms, the obtained distribution values are combined (through a weighted sume) to yield the final distribution value.

To illustrate this process, consider expression $\xi m{::}\{\!\{?^{\frac{2}{3}}, \mathsf{Real}^{\frac{1}{3}}\}\!\}$, where $\xi = (\omega_1(1,1) = \frac{1}{6} \wedge \omega_2(1,2) = \frac{1}{3} \wedge \omega_3(2,1) = \frac{1}{2}) \triangleright \{\!\{\mathsf{Real}^{\omega_1}, \mathsf{Real}^{\omega_2}, \mathsf{Bool}^{\omega_3}\}\!\}$ and $\xi \vdash \{\!\{\mathsf{Real}^{\frac{1}{2}}, \mathsf{Bool}^{\frac{1}{2}}\}\!\} \sim \{\!\{?^{\frac{2}{3}}, \mathsf{Real}^{\frac{1}{3}}\}\!\}$.[7] If $m \Downarrow_1 \{\!\{((\mathsf{Bool})\mathsf{true} :: \mathsf{Bool})^{\frac{1}{2}}, ((\mathsf{Real})1 :: \mathsf{Real})^{\frac{1}{2}}\}\!\}$, then the $\xi' = (\omega'_1(1,2) = \frac{1}{2} \wedge \omega'_2(2,1) = \frac{1}{2}) \triangleright \{\!\{\mathsf{Real}^{\omega'_1}, \mathsf{Bool}^{\omega'_2}\}\!\}$. Notice that $\xi' \circ \xi = (\omega''_1(1,1) = \frac{1}{2} \wedge \omega''_2(2,1) = \frac{1}{6} \wedge \omega''_3(2,2) = \frac{1}{3}) \triangleright$

[7] We have simplified the notation by using real numbers instead of variables (thus omitting formulas), and omitting some trivial ascriptions.

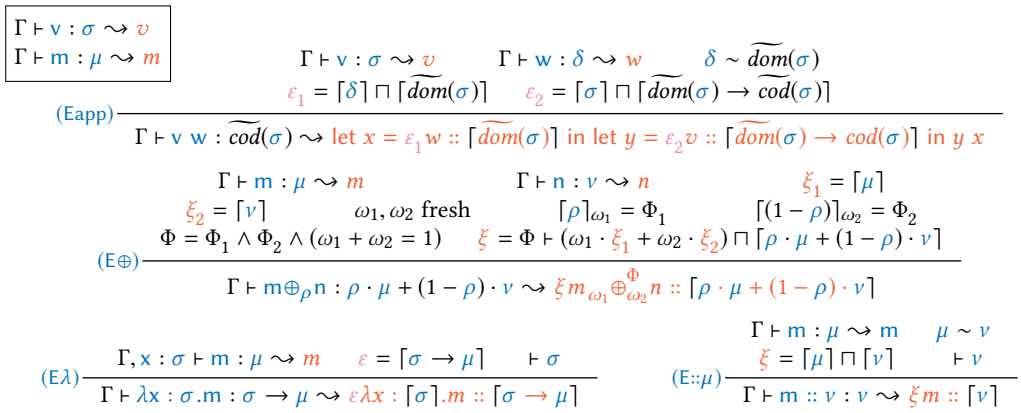

Fig. 13. Elaboration from GPLC to TPLC (excerpt).

$\{\!\{\text{Bool}^{\omega_1''}, \text{Real}^{\omega_2''}, \text{Real}^{\omega_3''}\}\!\}$. Finally, the whole expression reduces to $\{\!\{((\text{Bool})\text{true} :: ?)^{\frac{1}{2}}, ((\text{Real})1 :: \text{Real})^{\frac{1}{6}}, ((\text{Real})1 :: \text{Real})^{\frac{1}{3}}\}\!\}$.

5.4 Elaboration

As previously mentioned, the runtime semantics of GPLC is given via translation to the TPLC target language. Figure 13 presents the type-driven elaboration rules from GPLC to TPLC. Judgment $\Gamma \vdash \mathsf{v} : \sigma \rightsquigarrow v$ (resp. $\Gamma \vdash \mathsf{m} : \mu \rightsquigarrow m$), denotes the elaboration of value v (resp. term m) from value v (resp. tern m), where v (resp. m) is typed σ (resp. μ) under environment Γ. For simplicity, we write $\mathsf{v} : \sigma \rightsquigarrow v$ (resp. $\mathsf{m} : \mu \rightsquigarrow m$) as a shorthand for $\cdot \vdash \mathsf{v} : \sigma \rightsquigarrow v$ (resp. $\cdot \vdash \mathsf{m} : \mu \rightsquigarrow m$). Rule (E$\lambda$) and the ones for elaborating other values, elaborate by inserting ascriptions to their types. The initial evidence between two gradual types is computed using the meet of the lifted types. Rule (Eapp) insert ascriptions in both the function and argument to make top-level constructor match using A-normal form. Rule (E::μ) produces initial evidence to justify the consistency judgment between μ and ν. Rule (E$\oplus$) is designed to carefully deal with the probability annotations. First, it generates two fresh variables: ω_1 for annotation ρ, and ω_2 for the complement $1 - \rho$.[8] Second, these two fresh variables are used to generate two formulas Φ_1 and Φ_2 by lifting ρ and $1 - \rho$. Third, the annotation formula Φ is computed by combining Φ_1 and Φ_2, with the extra requirement that the sum of the probability variables must be one (in case $\rho = ?$). Finally, we insert an extra ascription to relate variables ω_1 and ω_2 with the fresh variables obtained from lifting the type of the expression (as they will be different).

To conclude, we establish that the elaboration rules preserve typing:

Theorem 5.10 (Elaboration Preserve Types). *For every value v, term m, simple type σ and distribution type μ from GPLC,*

(1) If $\Gamma \vdash \mathsf{v} : \sigma$, then there exists value v in TPLC such that $\Gamma \vdash \mathsf{v} : \sigma \rightsquigarrow v$ and $\lceil\Gamma\rceil \vdash v : \lceil\sigma\rceil$.
(2) If $\Gamma \vdash \mathsf{m} : \mu$, then there exists term m in TPLC such that $\Gamma \vdash \mathsf{v} : \mu \rightsquigarrow m$ and $\lceil\Gamma\rceil \vdash m : \lceil\mu\rceil$.

Here, $\lceil\Gamma\rceil$ is the pointwise lifting of Γ.

[8] For ω_1 and ω_2 we do not care about the indexes because they do not flow into evidences. For this reason we can arbitrary choose 0 and 0 as default values.

$$\frac{\forall\, FV(\Phi_1).\ \Phi_1 \implies \exists\, FV(\Phi_2) \cup \{\omega_{ij} \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}.\ \{\!\{\omega_{ij} \mid i \in \mathcal{I} \wedge j \in \mathcal{J}\}\!\} \vdash \{\!\{v_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\}^{\Phi_1} \sqsubseteq \{\!\{v_j^{\varrho_j} \mid j \in \mathcal{J}\}\!\}^{\Phi_2}}{\Phi_1 \triangleright \{\!\{v_i^{\varrho_i} \mid i \in \mathcal{I}\}\!\} \sqsubseteq \Phi_2 \triangleright \{\!\{v_j^{\varrho_j} \mid j \in \mathcal{J}\}\!\}}\ (\sqsubseteq \mathcal{V})$$

$$\frac{\vdash m : \delta \qquad \sigma \sqsubseteq \delta}{\mathbf{error}_\sigma \sqsubseteq m} \qquad\qquad \frac{m \sqsubseteq m' \qquad n \sqsubseteq n' \qquad \forall FV(\Phi_1).\Phi_1 \Rightarrow \Phi_2}{m \,{}_{\varrho_1}\!\oplus^{\Phi_1}_{\varrho_2} n \sqsubseteq m' \,{}_{\varrho_1}\!\oplus^{\Phi_2}_{\varrho_2} n'}\ (\sqsubseteq \oplus)$$

Fig. 14. Term precision of TPLC (excerpt).

5.5 Type Safety and Gradual Guarantee

We can now establish several properties about GPLC, based on the elaboration to TPLC. To this end, we start highlighting that even though our static language (SPLC) is terminating, when introducing unknown types, one can encode statically well-typed programs that diverge, rendering our gradual languages (GPLC and TPLC) non-terminating. The emblematic program illustrating this phenomenon is the omega term $\Omega = (\lambda x :?.x\ x)(\lambda x :?.x\ x)$.

In the remaining of this section, for simplicity, given m we write $\mathsf{m} \Downarrow \Phi \triangleright \mathscr{V}$ if $\vdash \mathsf{m} : \mu \rightsquigarrow m$ (for some μ and m) and there exists k such that $m \Downarrow_k \Phi \triangleright \mathscr{V}$ (for some Φ and $\mathscr{V}$), and $\mathsf{m} \Uparrow$ if $\vdash \mathsf{m} : \mu \rightsquigarrow m$ (for some μ and m) and there exists no such k. In the former case we say that m (similarly, m) *terminates*, reducing to a distribution value, while in the latter case we say that m (similarly, m) *diverges* (also denoted by $m \Uparrow$). Formally, this terminology corresponds to the notion of *certain* termination, where, intuitively, a program is considered terminating if there is a bound on the length of all its executions.

We note that probabilistic programs aslo support more general notions of termination, such as *almost-sure* termination, which intuitively allows diverging executions, but requires them to have an overall null probability. Support for reasoning about this class of programs is left as future work.

Type Safety. Following [Lennon-Bertrand et al. 2022], we model errors ($\mathbf{error}_\sigma$ and $\mathbf{error}_\mu$) as expressions, simplifying this way the statement of type safety, as we do not need to reason about error separately. Type safety for GPLC then states that if a term m is well-typed, then it either reduces to a distribution value of an equivalent type, or diverges:

Theorem 5.11 (Type safety for GPLC). *For every term m and gradual distribution type μ from GPLC, if $\vdash \mathsf{m} : \mu$ then either*

(1) $\mathsf{m} \Downarrow \Phi \triangleright \mathscr{V}$, $\vdash \Phi \triangleright \mathscr{V} : \mu$ *and* $\mu \overset{\mathrm{r}}{=} \lceil \mu \rceil$ *for some* Φ, $\mathscr{V}$ *and* μ, *or* *(2)* $\mathsf{m} \Uparrow$.

Dynamic Gradual Guarantee. To establish the dynamic gradual guarantee (DGG) for GPLC, we start by establishing the DGG for TPLC. This requires defining the notion of type and term precision for TPLC. Type precision is defined in the same was as for GPLC (see Figure. 8). Term precision is the natural lifting of type precision to terms, and is defined in Figure. 14. Rule $(\sqsubseteq \mathcal{V})$ relates two value distribution by lifting the precision relation on values to distributions via couplings, similarly to type precision. Like in [Lennon-Bertrand et al. 2022; New et al. 2019], an $\mathbf{error}_\sigma$ is more precise than any term provided σ is more precise than the term type. Rule $(\sqsubseteq \oplus)$ relates two probabilistic choices if the corresponding subterms are in precision relation, and most importantly, the formula in the more precise probabilistic choice entails the formula in the less precise precise probabilistic choice. The notion of entailment over Formula is rather standard, and thus omitted. Finally, note that the pair of probabilistic choices share the same variable names after alpha renaming. To illustrate this rule, assume that we want to show that a probabilsitic choice with a static probability

$$\mathscr{V}[\![\mathsf{Real}]\!] = \{(r, \varepsilon r :: \mathsf{Real}) \in \mathsf{Atom}[\mathsf{Real}] \mid r = r\}$$
$$\mathscr{V}[\![\tau \to T]\!] = \{(v_1, v_2) \in \mathsf{Atom}[\tau \to T] \mid \forall (v_1', v_2') \in \mathscr{V}[\![\tau]\!].\ (v_1\ v_1', v_2\ v_2') \in \mathscr{T}[\![T]\!]\}$$
$$\mathscr{V}[\![T]\!] = \{(\mathscr{V}, \mathscr{V}) \mid \mathscr{V} = \{\!\{v_i^{p_i} \mid i \in \mathscr{I}\}\!\} \wedge \mathscr{V} = \{\!\{v_j^{p_j} \mid j \in \mathscr{J}\}\!\} \wedge \exists \xi = \{\!\{\tau_k^{\omega_k} \mid k \in \mathscr{K}\}\!\}.$$
$$(\xi, \mathscr{V}, \mathscr{V}) \in \mathsf{Atom}[T] \wedge \forall \omega_k > 0, i = \omega_k.\ell, j = \omega_k.r.\ (v_i, v_j) \in \mathscr{V}[\![\tau_k]\!]\})$$
$$\mathscr{T}[\![T]\!] = \{(m_1, m_2) \mid m_1 \Downarrow_s^* \mathscr{V}_1 \wedge m_2 \Downarrow_* \mathscr{V}_2 \wedge (\mathscr{V}_1, \mathscr{V}_2) \in \mathscr{V}[\![T]\!]\}$$
$$\mathscr{G}[\![\Gamma, x : \tau]\!] = \{\gamma[(v, v')/x] \mid \gamma \in \mathscr{G}[\![\Gamma]\!] \wedge (v, v') \in \mathscr{V}[\![\tau]\!]\} \qquad \mathscr{G}[\![\cdot]\!] = \{\emptyset\}$$
$$\mathsf{Atom}[\tau] = \{(v, v') \mid \vdash_s v : \tau \wedge \vdash v' : \tau\}$$
$$\mathsf{Atom}[T] = \{(\xi, \mathscr{V}, \mathscr{V}) \mid \vdash_s \mathscr{V} : T_1 \wedge \vdash \mathscr{V} : T_2 \wedge \xi \vdash T_1 \overset{\mathsf{r}}{=} T_2 \wedge T \overset{\mathsf{r}}{=} T_1 \wedge T \overset{\mathsf{r}}{=} T_2\}$$
$$\Gamma \vdash m_1 \approx m_2 : T \iff \forall (\gamma_1, \gamma_2) \in \mathscr{G}[\![\Gamma]\!].\ (\gamma_1(m_1), \gamma_2(m_2)) \in \mathscr{T}[\![T]\!]$$

Fig. 15. Logical relation between SPLC and TPLC (excerpt).

$\frac{1}{2}$ is more precise than the one that we obtain replacing the static probability with ?. The rule application would then generate, as premise, the entailment $(\omega_1 = \frac{1}{2} \wedge \omega_2 = \frac{1}{2} \wedge \omega_1 + \omega_2 = 1) \Rightarrow (\omega_1 \in [0,1] \wedge \omega_2 \in [0,1] \wedge \omega_1 + \omega_2 = 1)$, with ω_1, ω_1 universally quantified. The remaining rules are standard.

Having defined the notion of precision, the major pending challenge to establish the DGG is to prove that evidence combination is monotone with respect to imprecision:

Lemma 5.12 (Monotonicity of evidence combination). *For all formula simple types $\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4 \in$ FSType and all formula distribution types $\xi_1, \xi_2, \xi_3, \xi_4 \in$ FDType,*

(1) If $\varepsilon_1 \sqsubseteq \varepsilon_2, \varepsilon_3 \sqsubseteq \varepsilon_4$ and $\varepsilon_1 \circ \varepsilon_3$ is defined then $\varepsilon_1 \circ \varepsilon_3 \sqsubseteq \varepsilon_2 \circ \varepsilon_4$
(2) If $\xi_1 \sqsubseteq \xi_2, \xi_3 \sqsubseteq \xi_4$ and $\xi_1 \circ \xi_3$ is defined then $\xi_1 \circ \xi_3 \sqsubseteq \xi_2 \circ \xi_4$

Proof sketch. For simple evidences, the proof proceeds by routine induction. For distribution evidences, the proof requires some coupling combinations. Assume $\mathscr{C}_{12}^{\sqsubseteq} \vdash \xi_1 \sqsubseteq \xi_2, \mathscr{C}_{34}^{\sqsubseteq} \vdash \xi_3 \sqsubseteq \xi_4$, $\mathscr{C}_{13}^{\circ} \vdash \xi_1 \circ \xi_3$. To prove that $\xi_2 \circ \xi_4$ is defined, from $\mathscr{C}_{12}^{\sqsubseteq}$ we build a coupling $\mathscr{C}_{21}^{\sqsupseteq} \vdash \xi_2 \sqsupseteq \xi_1$ and then use $\mathscr{C}_{21}^{\sqsupseteq} \odot \mathscr{C}_{13}^{\circ} \odot \mathscr{C}_{34}^{\sqsubseteq}$ as witness coupling, where the coupling composition operator $\odot$ is defined as:

$$\mathscr{C}_1 \odot \mathscr{C}_2 = \{\!\{\omega_{k_1k_2}(i,j) \mid \omega_{k_1k_2}(i,j) = \frac{\omega_{k_1}(i,h)\omega_{k_2}(h,j)}{\sum_{i,k_1 \mid \omega_{k_1}(i,h)} \omega_{k_1}(i,h)}, \omega_{k_1}(i,h) \in \mathscr{C}_1, \omega_{k_2}(h,j) \in \mathscr{C}_2\}\!\}$$

To justify that $\xi_1 \circ \xi_3 \sqsubseteq \xi_2 \circ \xi_4$, we start from coupling $\mathscr{C}_{13}^{\circ}$ and transform its first dimension (the one typically iterated by index $\mathscr{I}$) following the associations stated by coupling $\mathscr{C}_{12}^{\sqsubseteq}$ and its second dimension (typically iterated by index $\mathscr{J}$) following the associations stated by coupling $\mathscr{C}_{34}^{\sqsubseteq}$.

Now we can establish the DGG for TPLC: reduction is monotone with respect to imprecision.

Theorem 5.13 (Dynamic gradual guarantee for TPLC). *Suppose $m \sqsubseteq n$, $\vdash m : \mu$ and $\vdash n : \nu$.*
(1) If $m \Downarrow_{k_1} \Phi_1 \rhd \mathscr{V}_1$, then $n \Downarrow_{k_2} \Phi_2 \rhd \mathscr{V}_2$, and $\Phi_1 \triangleright \mathscr{V}_1 \sqsubseteq \Phi_2 \triangleright \mathscr{V}_2$. *(2) If $m \Uparrow$, then $n \Uparrow$.*

The DGG for GPLC is given by first elaborating the source terms to TPLC and then reducing the TPLC terms.

Theorem 5.14 (Dynamic gradual guarantee for GPLC). *Suppose $\mathsf{m} \sqsubseteq \mathsf{n}$, $\vdash \mathsf{m} : \mu$ and $\vdash \mathsf{n} : \nu$.*
(1) If $\mathsf{m} \Downarrow \Phi_1 \triangleright \mathscr{V}_1$, then $\mathsf{n} \Downarrow \Phi_2 \triangleright \mathscr{V}_2$ and $\Phi_1 \triangleright \mathscr{V}_1 \sqsubseteq \Phi_2 \triangleright \mathscr{V}_2$. *(2) If $\mathsf{m} \Uparrow$, then $\mathsf{n} \Uparrow$.*

Conservative extension of the dynamic semantics. In Section 4, we establish the equivalence between the static semantics of SPLC and GPLC for fully-statically-annotated terms. Here —due to the syntactic differences between both languages— to establish the equivalence between the dynamic semantics, we use logical relations. The logical relation between SPLC and TPLC is presented in Figure 15, and states that two related terms reduce to related distributions. Formally, it is defined using three mutually-defined interpretations: one for values ($\mathscr{V}[\![\tau]\!]$), one for distribution values ($\mathscr{V}[\![T]\!]$), and another one for terms or computations ($\mathscr{T}[\![T]\!]$).

We write $(v_1, v_2) \in \mathscr{V}[\![\tau]\!]$ to denote that values v_1 and v_2 are related at simple type τ. Two values are related at type τ if, first, they type check to τ, written $(v, v) \in \mathrm{Atom}[\tau]$. Two booleans (resp. real numbers) are related when the underlying values are the same. Two functions are related if their application to related argument yields related computations.

Two distribution values $\mathscr{V}, \mathscr{V}$ are related at a distribution type T if, first, there exists a distribution evidence (of fully-static types) ξ that justifies that their types[9] are equivalent to (i.e. a reorder of) T, written $(\xi, \mathscr{V}, \mathscr{V}) \in \mathrm{Atom}[T]$.[10] Second, for all positive probabilities in the evidence (the coupling), the corresponding values must be related at the corresponding type.

Two computations are related if both reduce to related distribution values. Two value substitutions are related at some type environment, if every variable in the domain of the environments is bound to related values. Finally, two open terms are related if the substitution to any two related value environment yield related computations.

We can now establish the conservative extension of the dynamic semantics of TPLC with respect to SPLC for fully-annotated terms.

Theorem 5.15 (Dynamic conservative extension of TPLC w.r.t. SPLC).

(1) $\vdash_s m : \tau, m \rightsquigarrow m' : \tau$, *then* $\vdash m \approx m' : \tau$ *(2)* $\vdash_s m : \tau, m \rightsquigarrow m' : T$, *then* $\vdash m \approx m' : T$

The proof of Theorem 5.15 relies on the fact that the composition of static evidences is always defined:

Lemma 5.16.

(1) If $\varepsilon_1 \vdash \tau_1 \sim \tau_2$ *and* $\varepsilon_2 \vdash \tau_2 \sim \tau_3$, *then* $\varepsilon_1 \circ \varepsilon_2$ *is defined, and* $\varepsilon_1 \circ \varepsilon_2 \vdash \tau_1 \sim \tau_3$.
(2) If $\xi_1 \vdash T_1 \sim T_2$ *and* $\xi_2 \vdash T_2 \sim T_3$, *then* $\xi_1 \circ \xi_2$ *is defined, and* $\xi_1 \circ \xi_2 \vdash T_1 \sim T_3$.

6 RELATED WORK

Gradual typing. As previously mentioned, gradual typing has been applied to many type discipline and language constructs. To the best of our knowledge, gradual typing has not been applied to probabilistic languages, neither to non-deterministic languages.

Lehmann and Tanter [2017] presented gradual refinement types, which allow the smoothly transition –and interoperability– between simple types and logically-refined types. In this work, we use statically-typed refinement types to implement cast/evidence, but do not support for gradual refinement types at the source level. Phipps-Costin et al. [2021] present TypeWhich, an approach for automatic type migration, which tries to infer additional or improved type annotations in gradually typed languages. Similarly to this work, TypeWhich also generates constrains (formulas) during type checking, and relies in an SMT solver to find solutions to their objectives.

There exist many flavors to define the runtime semantics of gradual languages. The classical approach is via a translation to a cast calculus [Garcia 2013; Herman et al. 2007, 2010; Siek et al.

[9] The type rule for $\mathscr{V}$ is defined analogously to $\mathscr{V}$, and can be found in the supplementary material.
[10] In TPLC, as the lifting of static types T always yields distribution types with one-to-one equality formulas, e.g. $\{\!\{\mathrm{Real}^{\frac{1}{2}}, \mathrm{Bool}^{\frac{1}{2}}\}\!\}$ is lifted as $(\omega_1 = \frac{1}{2} \wedge \omega_2 = \frac{1}{2}) \vdash \{\!\{\mathrm{Real}^{\omega_1}, \mathrm{Bool}^{\omega_2}\}\!\}$, for simplicity, to avoid writing variables, in the rule definition we annotate probabilities as numbers instead (i.e. $v_i^{p_i}$ and $v_j^{p_j}$).

2015a; Siek and Wadler 2010; Siek et al. 2009; Wadler and Findler 2009]; Garcia et al. [2016] defined the runtime semantics directly in the source language, by mimicking the proof normalization steps done in type safety; and recently, Ye et al. [2021] also presented direct dynamic semantics by using type-directed operational semantics (TDOS) [Huang and Oliveira 2020]. In this work, we follow the classical approach –defining a source and target language (cast calculus)–, where casts are implemented by using evidences from the AGT methodology.

There has been active work on designing gradual languages that allows the combination/collection of types. Castagna and Lanvin [2017]; Castagna et al. [2019] proposed a theory for gradual set-theoretic types, supporting union, intersection and the unknown type. In parallel, Toro and Tanter [2017] explored tagged and untagged union types, and Jafery and Dunfield [2017] sums types. Beside many fundamental differences, this work could be seen as a generalization of gradual union types with gradual weights.

Probabilistic λ-calculus. We can trace the origin of probabilistic λ-calculus to the work of [Saheb-Djahromi 1978], who present a typed, higher-order calculus. They develop a denotational semantics based on Plotkin's probabilistic powerdomain [Jones and Plotkin 1989] and an operational semantics in terms of Markov chains. [Lago and Zorzi 2012b] surveys a variety of operational sematics for a λ-calculus with a probabilistic choice operator including small/big-step, inductive/coinductive and call-by-value/name variants. All fell under the category of distribution-based semantics, relating programs to probability distribution of values. [Ramsey and Pfeffer 2002] develop a denotational semantics for a stochastic λ-calculus exploiting the monadic structure of probability distributions. More recently, [Danos and Ehrhard 2011] and [Ehrhard et al. 2017] study a denotational semantics for higher-order programs in terms of coherence spaces.

Different type systems have been developed for probabilistic λ-calculi, aimed at establishing different program invariants. [Lago and Grellois 2017] use sized types to reason about almost-sure termination of higher-order programs. [Avanzini et al. 2021] develop a type system based on refinement types, to perform complexity analysis of higher-order functional programs. [Reed and Pierce 2010] (and many subsequent extensions) present a type system for reasoning about program sensitivity, used for established differential privacy properties of programs.

7 CONCLUSION

In this work, we provide a first step into the theoretical foundation of gradual probabilistic programming. We develop GPLC, to the best of our knowledge, the first gradual probabilistic language. The language enables an increased flexibility and expressivity, allowing some form of probabilistic specifications and also of program refinement via unknown probabilities in probabilistic choices. The development of GPLC is justified using the AGT methodology. The dynamic semantics of GPLC is given via translation to an evidence-based calculus, called TPLC, which features a distribution-based dynamic semantics. The development of GPLC and TPLC heavily relies in the notion of probabilistic coupling, as required for defining several relations and functions, such as type consistency, precision and consistent transitivity. As for the metatheory, GPLC satisfies type safety as well as the refined criteria for gradual languages.

As future work, we plan to explore the addition of more features to the language, such as subtyping and polymorphism. Introducing subtyping may bring several challenges, such as the use of sub-distributions. Another possible line of future work are the practical aspects of the gradual language, such as efficient handling of evidences in runtime, and space-efficient reduction rules.

REFERENCES

Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2021. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '19)*. IEEE Press, Article 41, 13 pages.

Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling Noninterference and Gradual Typing. In *Proceedings of the 2020 Symposium on Logic in Computer Science (LICS 2020)*.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual Type-and-Effect Systems. *Journal of Functional Programming* 26 (Sept. 2016), 19:1–19:69.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. ACM, New York, 90–101.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (Sept. 2017), 41:1–41:28.

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: a new perspective. See[POPL 2019], 16:1–16:32.

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference using Data Flow Analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 92–102.

Vincent Danos and Thomas Ehrhard. 2011. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Information and Computation* 209, 6 (2011), 966–991. https://doi.org/10.1016/j.ic.2011.02.001

Yuxin Deng and Wenjie Du. 2011. Logical, Metric, and Algorithmic Characterisations of Probabilistic Bisimulation. *CoRR* abs/1103.4577 (2011). arXiv:1103.4577 http://arxiv.org/abs/1103.4577

Tim Disney and Cormac Flanagan. 2011. Gradual information flow typing. In *International Workshop on Scripts to Programs*.

Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3-4 (2014), 211–407. https://doi.org/10.1561/0400000042

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2017. Measurable Cones and Stable, Measurable Functions: A Model for Probabilistic Higher-Order Programming. *Proc. ACM Program. Lang.* 2, POPL, Article 59 (dec 2017), 28 pages. https://doi.org/10.1145/3158147

Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*. 224–239.

Ronald Garcia. 2013. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*. 417–428.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM Press, St Petersburg, FL, USA, 429–442. See erratum: https://www.cs.ubc.ca/ rxg/agt-erratum.pdf.

Zoubin Ghahramani. 2015. Probabilistic Machine Learning and Artificial Intelligence. *Nature* 521, 7553 (2015), 452–459.

Shafi Goldwasser and Silvio Micali. 1984. Probabilistic Encryption. *J. Comput. Sys. Sci.* 28, 2 (1984), 270–299.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'08)*. AUAI Press, 220–229.

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2022-10-17.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE 2014*. ACM, 167–181.

David Herman, Aaron Tomb, and Cormac Flanagan. 2007. Space-efficient gradual typing. In *In Trends in Functional Programming (TFP*.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Sympolic Computation* 23, 2 (June 2010), 167–189.

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *ECOOP*.

Khurram A. Jafery and Jana Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual, See [POPL 2017], 804–817.

C. Jones and Gordon D. Plotkin. 1989. A probabilistic powerdomain of evaluations. *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science* (1989), 186–195.

Oleg Kiselyov. 2016. Probabilistic Programming Language and its Incremental Evaluation. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.), Vol. 10017. 357–376. https://doi.org/10.1007/978-3-319-47958-3_19

Ugo Dal Lago and Charles Grellois. 2017. Probabilistic Termination by Monadic Affine Sized Typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41 (2017), 1 – 65.

Ugo Dal Lago and Margherita Zorzi. 2012a. Probabilistic operational semantics for the lambda calculus. *RAIRO Theor. Informatics Appl.* 46, 3 (2012), 413–450. https://doi.org/10.1051/ita/2012012

Ugo Dal Lago and Margherita Zorzi. 2012b. Probabilistic operational semantics for the lambda calculus. *ArXiv* abs/1104.0195 (2012).

Tuan Anh Le, Atilim Gunes Baydin, and Frank D. Wood. 2017. Inference Compilation and Universal Probabilistic Programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.), Vol. 54. PMLR, 1338–1348. http://proceedings.mlr.press/v54/le17a.html

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types, See [POPL 2017], 775–788.

Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. *ACM Transactions on Programming Languages and Systems* (2022). To appear. To be presented at POPL'22.

Stefan Malewski, Michael Greenberg, and Éric Tanter. 2021. Gradually Structured Data. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Nov. 2021), 126:1–126:28.

Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.

Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual Type Theory. See[POPL 2019], 15:1–15:31.

Avi Pfeffer. 2010. Practical Probabilistic Programming. In *Inductive Logic Programming - 20th International Conference, ILP 2010, Florence, Italy, June 27-30, 2010. Revised Papers (Lecture Notes in Computer Science)*, Paolo Frasconi and Francesca A. Lisi (Eds.), Vol. 6489. Springer, 2–3.

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based gradual type migration. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485488

POPL 2017. *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France.

POPL 2019. Proceedings of the 46th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2019).

Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *POPL '02*.

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*. ACM, 157–168.

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. In *LISP AND SYMBOLIC COMPUTATION*. 288–298.

Nasser Saheb-Djahromi. 1978. Probabilistic LCF. In *MFCS*.

Roberto Segala and Nancy A. Lynch. 1995. Probabilistic Simulations for Probabilistic Processes. *Nord. J. Comput.* 2, 2 (1995), 250–273.

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

Jeremy Siek, Peter Thiemann, and Phil Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM Press, Portland, OR, USA, 425–435.

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

Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *ESOP*.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Asilomar, California, USA, 274–293.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015c. Monotonic References for Efficient Gradual Typing. In *ESOP*.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*. ACM Press, Tucson, AZ, USA, 793–810.

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Transactions on Programming Languages and Systems* 40, 4 (Nov. 2018), 16:1–16:55.

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017) (Lecture Notes in Computer Science)*, Vol. 10422. Springer-Verlag, New York City, NY, USA, 382–404.

Matías Toro and Éric Tanter. 2020. Abstracting Gradual References. *Science of Computer Programming* 197 (Oct. 2020), 1–65.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017,*

Conference Track Proceedings. OpenReview.net. https://openreview.net/forum?id=Hy6b4Pqee

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018). arXiv:1809.10756 http://arxiv.org/abs/1809.10756

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009) (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer-Verlag, York, UK, 1–16.

Wenjia Ye, Bruno C. d. S. Oliveira, and Xuejing Huang. 2021. Type-Directed Operational Semantics for Gradual Typing. In *ECOOP*.

Received 2022-10-28; accepted 2023-02-25