

Gradual Sensitivity Typing

DAMIÁN ARQUEZ, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile
 MATÍAS TORO, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile
 ÉRIC TANTER, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile

Reasoning about the sensitivity of functions with respect to their inputs has interesting applications in various areas. In order to check and enforce sensitivity, several approaches have been developed, notably sensitivity type systems. In these systems, sensitivity can be seen as an effect in the sense of type-and-effects systems as originally proposed by Gifford and Lucassen. Because type-and-effect systems can make certain useful programming patterns tedious or overly conservative, there is value in bringing the benefits of gradual typing to these disciplines in order to ease their adoption. In this work, we motivate, formalize, and prototype gradual sensitivity typing. The language GSoul supports both the unrestricted unknown sensitivity and bounded imprecision in the form of intervals. Gradual sensitivity typing allows programmers to smoothly evolve typed programs without any static sensitivity information towards hardened programs with a mix of static and dynamic sensitivity checking. In particular, we show that gradual sensitivity supports recursive functions for which fully static checking would be overly conservative, seamlessly enabling exact runtime sensitivity checks. GSoul satisfies both the gradual guarantees and sensitivity type soundness, known as metric preservation. We establish that, in general, gradual metric preservation is termination insensitive, and that one can achieve termination-sensitive gradual metric preservation by hardening specifications to bounded imprecision. We implement a prototype that provides an interactive test bed for gradual sensitivity typing. This work opens the door to gradualizing other typing disciplines that rely on function sensitivity.

1 INTRODUCTION

Function sensitivity, also called Lipschitz continuity, is an upper bound of how much the output may change given an input perturbation. More formally, a function f is s -sensitive if for all x and y , $|f(x) - f(y)| \leq s|x - y|$. Sensitivity has a very important role in different computer science fields, such as control theory [Zames 1996], dynamic systems [Bournez et al. 2010], program analysis [Chaudhuri et al. 2011] and differential privacy [Dwork and Roth 2014]. For example, in the latter, a function f can be forced to comply with privacy requirements by adding random noise to the results. The added noise has to be sufficient to guarantee privacy but also tight enough to avoid compromising the utility of the secured result. This is usually achieved by calibrating the amount of noise using the sensitivity of the function.

Many approaches have been proposed to reason about sensitivity, either statically [Abuah et al. 2022; D’Antoni et al. 2013; Gaboardi et al. 2013; Near et al. 2019; Reed and Pierce 2010; Toro et al. 2023; Winograd-Cort et al. 2017; Zhang et al. 2019], or dynamically [Abuah et al. 2021]. Handling sensitivity statically as an effect in the sense of type-and-effects systems [Gifford and Lucassen 1986] has the advantage of providing strong guarantees statically. However, type-and-effect systems can make certain useful programming patterns tedious or overly conservative. Combining the advantages of static and dynamic typechecking is a very active area with a long history, but, to the best of our knowledge, sensitivity has not been explored under this perspective. One prominent approach for combining static and dynamic typechecking is gradual typing [Siek and Taha 2006]. Gradual typing supports the smooth transition between dynamic and static checking within

Authors’ addresses: Damián Arquez, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago, Chile, darquez@dcc.uchile.cl; Matías Toro, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago, Chile, mtoro@dcc.uchile.cl; Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago, Chile, etanter@dcc.uchile.cl.

the same language by introducing imprecise types and consistent relations. Imprecision is handled optimistically during static typechecking, and is backed by runtime checks in order to detect potential violations of static assumptions. Hence, the programmer is able to choose at a fine-grained level which portions of the program are dynamically checked and which ones are statically checked. Gradual typing has been studied in many settings such as subtyping [Garcia et al. 2016; Siek and Taha 2007], references [Herman et al. 2010; Siek et al. 2015b], ownership [Sergey and Clarke 2012], information-flow typing [Disney and Flanagan 2011; Fennell and Thiemann 2013] and refinement types [Lehmann and Tanter 2017], among (many) others.

Additionally, the fact that sensitivity is a quantity makes static reasoning even more challenging, and forces simply-typed approaches to over-approximate the sensitivity of some recursive functions as being infinite, specially when the sensitivity depends on the number of recursive calls. This can make writing some recursive functions very hard, unless one considers dependent typing [Gaborardi et al. 2013]. Also, the potential for divergence in a language introduces the possibility of different interpretations of sensitivity type soundness, known as *metric preservation* [Reed and Pierce 2010]. Metric preservation is a hyperproperty that captures the bound on how much the result of two similar computations may change given an input variation. In the presence of possible divergence, the situation is similar to that of information flow security, where both termination-sensitive and termination-insensitive notions of noninterference have been studied [Goguen and Meseguer 1982; Heintze and Riecke 1998; Zdancewic 2002]. For metric preservation, a *termination-insensitive* interpretation says that, if the function terminates on both inputs, then the output differences is bounded [Abuah et al. 2021; Azevedo de Amorim et al. 2017]. A *termination-sensitive* interpretation says that, if the function terminates on the first input, then it also terminates on the second, and the output differences is bounded [Gaborardi et al. 2013; Reed and Pierce 2010]. Azevedo de Amorim et al. [2017] study a termination-insensitive characterization of metric preservation as well as the necessary conditions to establish when two programs behave the same in terms of termination.

Contributions. This work studies the integration of gradual typing with sensitivity typing in GSoul, and presents the following contributions:

- We introduce GSoul, a gradually-typed sensitivity lambda calculus with a latent type-and-effect discipline, featuring novel support for explicit sensitivity polymorphism. Notably, GSoul not only supports the *unknown* sensitivity—which stands for any sensitivity, possibly infinite—but also a *bounded* form of imprecision in the form of *sensitivity intervals*. Bounded imprecision provides programmers with a fine-grained mechanism to balance flexibility and static checking.
- We establish that GSoul satisfies type safety and the gradual guarantee [Siek et al. 2015a].
- We study the sensitivity type soundness of GSoul proving that in general it satisfies a termination-insensitive notion of metric preservation. In particular, we illustrate the novelty of how the design of gradual metric preservation must be driven by pessimistic reasoning in order to soundly account for the inherent optimism of gradual typing. Furthermore, we prove that when imprecision is restricted to finite imprecision, a GSoul program satisfies termination-sensitive metric preservation. These novel results highlight how the progressive hardening of sensitivity information strengthens the property satisfied by a program.

After motivating gradual sensitivity typing with examples that illustrate the benefits of exploiting imprecise sensitivity information (§2), we present a core subset of GSoul, GSmini, capturing the key elements of gradual sensitivity typing (§3). We state the meta-theoretical properties of GSmini, including type safety and the gradual guarantee (§4). We then study the sensitivity type

soundness of G_{SMINI}, proving that in general it satisfies a termination-insensitive notion of metric preservation (§5). In particular, We show that when imprecision is restricted to finite imprecision, a G_{SMINI} program satisfies termination-sensitive metric preservation (§5.2). We then present G_{SOUL}, which extends G_{SMINI} with sum, product and recursive types (§6.1) and we revisit gradual sensitivity soundness in the presence of these new features (§6.2). Finally, we discuss related work (§7) and conclude (§8). We provide proofs of all the results in the supplementary material. Also, a prototype implementation of G_{SOUL} is available (url omitted for anonymity).

2 GRADUAL SENSITIVITY TYPING IN ACTION

We begin with a few examples that illustrate the benefits of gradual sensitivity typing in G_{SOUL}. Just like how gradual typing supports the whole spectrum between fully dynamic checking and fully static checking, a language with gradual sensitivity typing supports the range between types without any sensitivity information to fully static checking of sensitivities. We consider three stages of sensitivity typing information: first, a program without sensitivity checking or guarantees; second, a static variant of the program written in G_{SOUL}, highlighting a tension due to the conservative nature of static sensitivity checking; third, a gradual version where some sensitivity guarantees are established statically, and others are deferred to runtime checking, fully exploiting the graduality of G_{SOUL}.

Privacy without any sensitivity typing. Suppose that an ORM (Object-Relational Mapping) automatically generates utility functions from a static schema provided by the programmer: given a list of persons of some database represented by a table `Person(name, age, height, grade)`, each function computes the sum of some attribute, such as `sumAges` and `sumGrades` for summing all ages and grades, respectively. The programmer wants to publish the averages of different (countable) columns of `Person`. Given that the sum needs to be computed for multiple columns, the programmer abstracts the different sum functions into a map from column names to their respective functions. Each column is represented by a particular symbol, written as an alpha-numerical identifier prefixed by `:`, such as `:age` and `:grade`.

```
let db = loadDB();
let targetCols = List(:grade, ..., :age);
let sumsMap = Map(:grade => sumGrades, ..., :age => sumAges);
let sums = targetCols.map(fn (col) => sumsMap[col](db));
```

After testing the behavior of the program, the programmer worries about differential privacy [Dwork and Roth 2014] and notes that the privacy of some individuals may be violated if this information is publicly released. For this reason, to make the generic sum an differentially-private computation, the programmer decides to use the `laplace` function to add random noise to the result. If a function `f` is s -sensitive in x , then the noise has to be at least `Laplace(s/ε)` to formally guarantee ϵ -differential privacy, where ϵ is known as the privacy budget. The programmer then creates the (curried) function `private`, which applies its first argument `f` to its second argument, adding random noise to the result. In this case the programmer is optimistically assuming that `f` is at most 10-sensitive:¹

```
let epsilon = 0.1;
def private(f: DB -> Number)(db: DB): Number {
  f(db) + laplace(10/epsilon);
};
let sums = targetCols.map(fn (col) => private(sumsMap[col])(db));
```

¹The distance between two DBs is the number of records on which they differ [Dwork and Roth 2014; Reed and Pierce 2010].

The programmer now publishes the results, but unfortunately it is later revealed that privacy is violated when `:age` is in the list `targetCols`. Indeed, while a sensitivity of 10 is a sound upper bound for `sumGrades`, when grades range between 1 and 10, it is unsound for ages. To further explain, adding or removing a person from the database can change the result of `sumGrades` at most by 10, but the same operation can change the result of `sumAges` by far more. Assuming that people do not live beyond 120 years, `sumAges` could be soundly treated as a 120-sensitive function, but no less. Therefore, `private` adds too little noise in that case, but the programming language does not provide any help in detecting this problem.

Static sensitivity typing. Using sensitivity types, as provided in languages such as Fuzz [Reed and Pierce 2010], Duet [Near et al. 2019] and Jazz [Toro et al. 2023], the problem described above can be statically detected. Sensitivity typing extends types with sensitivity information in order to verify that a program respects a given sensitivity specification. A sensitivity s is represented by a positive real number (including zero) or infinity ∞ . Sensitivities can be used in types; for instance, the type $\mathbb{R}[1r_1 + 2r_2]$ characterizes numerical expressions that are 1-sensitive in the resource r_1 , and 2-sensitive in r_2 . The polynomial notation $1r_1 + 2r_2$ is syntactic sugar for a *sensitivity effect* that describes a computation that is 1-sensitive and 2-sensitive in the resources r_1 and r_2 , respectively. Ordering between sensitivities, denoted \leq , induces a covariant notion of subtyping $<:$. For example, $\mathbb{R}[1r_1 + 3r_2] <: \mathbb{R}[2r_1 + 4r_2]$, because $1 \leq 2$ and $3 \leq 4$.

In GSoul, variables can be declared as *resources* by using the **res** modifier in let-bindings. This makes available a resource with the same name as the program variable, which can be used in the sensitivity annotations of types. As an example, the database schema can be decorated so the generated functions are given a sensitivity type: the types of `sumGrades` and `sumAges` are now $[r](DB[r] \rightarrow \text{Number}[10r])$ and $[r](DB[r] \rightarrow \text{Number}[120r])$, respectively. Notice that both functions are *polymorphic* in the resource, denoted by the $[r]$ annotation before the parentheses. Finally, the programmer can make the sensitivity specification of `private` manifest by requiring that the passed function be at most 10-sensitive, since the noise $\text{Laplace}(10/\epsilon)$ only guarantees privacy for 10-sensitive computations. Notice that we may also want to make `private` polymorphic on the resource; the syntax **def** $f[r](\dots)$ is used to parametrize the function f on a resource.

```
let res db = loadDB();
...
def private[r](f: DB[r] -> Number[10r])(db: DB[r]): Number {
  f(db) + laplace(10/epsilon);
};
```

Using these sensitivity-typed functions, directly applying `private` to `sumAges` is a static type error because $DB[db] \rightarrow \text{Number}[120db]$ —the type of `sumAges` instantiated with the resource `db`—is not a subtype of $DB[db] \rightarrow \text{Number}[10db]$, the expected type of the argument of `private` (because $120 \not\leq 10$). Therefore, using static sensitivity types has turned the (silent) privacy issue of the program into a static type error.

Issues with fully-static checking. Static typing is necessarily conservative in order to be sound, and at times this precaution conflicts with useful programming patterns. In particular, the map of sum functions, `sumsMap`, has a generic type that specifies that it maps symbols to functions from $DB[\text{Person}]$ to Number : this uniform characterization of the map is extremely useful, but can be too stringent when function types also include sensitivity information. Assuming that `sumAges` is the most sensitive function in the map, then the only sound type to give to `sumsMap` is $\text{Map}<\text{Column}, [r](DB[r] \rightarrow \text{Number}[120r])>$. Doing so means that `private(sumsMap[col])(db)`

from our example program is now statically rejected. But this is so even if `targetCols` does not include `:age`, despite the fact that, semantically, privacy would not be violated. A possible fix would be to change the expected argument type and the amount of noise introduced in `private` to consider 120-sensitive functions. But then the noise applied to `sumsMap(:grade)` would be excessive, making the result useless.

Another problematic restriction of static sensitivity typing manifests when considering *recursive* functions. For instance, the Fuzz typing rule for fixpoints is [Reed and Pierce 2010]:

$$\frac{\Gamma, f :_{\infty} \tau_1 \multimap \tau_2 \vdash e : \tau_1 \multimap \tau_2}{\infty \Gamma \vdash \text{fix } f.e : \tau_1 \multimap \tau_2}$$

For recursive definitions that happen to capture external variables, i.e. not provided as arguments, this rule reports the fixpoint as infinitely sensitive in such variables, as observable in the conclusion of the rule where Γ is scaled by ∞ . Azevedo de Amorim et al. [2017] also identify this issue and design a fixpoint rule that provides a better bound for a particular class of recursive constructs, such as exponentially decaying lists, but their rule does not cover the general case and the practical usefulness of such types is left as future work.

A second limitation arises for functions on which the sensitivity with respect to an argument depends on the number of recursive calls. As a simple example, consider the following definition of a recursive `scale` function (using concrete Fuzz syntax), that receives a factor and a value to be scaled:

```
scale 0 v = 0
scale n v = v + scale (n-1) v
```

The only possible Fuzz type for this function is `scale : $\mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$` , deeming the function infinitely sensitive in both arguments (as expressed by the use of arrows \rightarrow , instead of lollipops \multimap). Notice that the sensitivity on the scaled value — the second argument — depends on the value of the factor argument. Therefore, the static sensitivity type system has to over-approximate the sensitivity of the function as infinite. Whereas some of these issues could be tackled by using dynamic typing [Abuah et al. 2021] or dependent typing [Gaborardi et al. 2013], such solutions come with their own set of drawbacks, such as the loss of static guarantees or the complexity of the type system, respectively.

Gradual sensitivities. GSoul supports gradual sensitivity typing, which can elegantly address the limitations of static sensitivity typing, simply relying on runtime checking whenever desired.

First, using gradual sensitivities, one can give `sumsMap` an *imprecise* sensitivity type, satisfying type checking and deferring necessary checks to runtime. To support gradual sensitivity typing, sensitivities in GSoul are extended with an unknown sensitivity `?`, akin to the unknown type of gradual typing [Siek and Taha 2006]. Intuitively, `?` represents *any* sensitivity and allows the programmer to introduce imprecision in the sensitivity information, relying on optimistic static checking, backed by runtime checks.

Gradual sensitivity ordering $\tilde{\leq}$ is defined by plausibility: an unknown sensitivity `?` is plausibly both smaller and greater than any other sensitivity, i.e. $?\tilde{\leq}s$ and $s\tilde{\leq}?$, for any sensitivity s . Notice that $\tilde{\leq}$ is not transitive: $2\tilde{\leq}?$ and $?\tilde{\leq}1$, but $2\not\tilde{\leq}1$. Analogous to the simple sensitivities setting, plausible ordering induces a notion of *consistent subtyping* $\tilde{<}$, e.g. `Number[r] -> Number[?r]` is a consistent subtype of `Number[r] -> Number[r]`, where r is a resource in scope (because $?\tilde{\leq}1$). This notion of consistent subtyping is akin to that studied for other gradual typing disciplines [Bañados Schwerter et al. 2014; Garcia et al. 2016; Lehmann and Tanter 2017; Siek and Taha 2007], but here focused only on the sensitivity information conveyed in types.

The programmer can introduce imprecision by using the unknown sensitivity to describe the type of `sumsMap` (using the `::<_>` notation), preserving the fact that it maps symbols to functions:

```
let sumsMap = Map::<Column, [r](DB[r] -> Number[?r])>(...);
```

As `?` is plausibly smaller than any other gradual sensitivity, the type of `sumsMap(col)` is now compatible with the type of `f` in the signature of `private`, because `?db` \lesssim `10db`, and so the program is well-typed. In gradual typing, the flexibility afforded by consistent relations such as consistent subtyping is backed by runtime checks at the boundaries between types of different precision, ensuring at runtime that no static assumptions are silently violated. In the example, if `col` is `:age`, a runtime error is raised when `private` is applied, signalling that a function that is more than 10-sensitive has been passed as argument. If `col` is `:grade`, no runtime error occurs.

Notice that in the example, some sensitivity types are fully precise (`private`, `sumAges`, `sumGrades`) while others are fully imprecise (`sumsMap`). Gradual sensitivity information can also be *partially* precise, such as `?db + 2r`. This means that programmers can selectively and progressively consider abiding to a static sensitivity discipline as needed, and obtain feedback and guarantees accordingly, both statically and dynamically.

Additionally, compared to fully static typing, gradual sensitivity typing offers a simple and effective alternative to handle recursive functions in an exact, albeit dynamically-checked, manner. For recursive functions that capture external variables, the programmer can use the unknown sensitivity to defer the sensitivity check to runtime, where the number of recursive calls can be observed. Additionally, for functions on which the sensitivity with respect to an argument depends on the number of recursive calls, consider the following example of a recursive definition of the `scale` function described before, now using GSoul syntax:

```
def scale[r](n: Number, v: Number[r]):... = (n == 0) ? 0 : v + scale(n - 1, v);
```

As explained above, the only sound static type for the return value of `scale` is `Number[∞r]` (written in the dots), unless we are willing to adopt the complexity of dependent typing [Gabori et al. 2013]. With gradual sensitivities, we can simply declare `scale` to have *unknown* sensitivity in the resource `r` by annotating the return value as `Number[?r]`. Then, assuming `f` requires an argument that is at most 10-sensitive in `x`, we obtain the following behavior:

```
f(scale(10, x)) // typechecks, runs successfully
f(scale(11, x)) // typechecks, fails at runtime
```

Therefore, gradual sensitivity typing not only allows programmers to choose between static and dynamic checking as they see fit, it can also accommodate features that are too conservatively handled by the static discipline. We revisit this example providing the necessary technical details in §6.1.

Bounded imprecision. In standard gradual typing, the unknown type `?` can stand for any type whatsoever. For instance, even if `Int → ?` is only partially imprecise, it denotes infinitely many function types, i.e. those with `Int` as domain. Likewise, gradual sensitivities need not be restricted to unbounded imprecision: while `?` denotes any sensitivity, we also support gradual sensitivities as bounded intervals such as `[1, 2]`. For instance, revisiting our previous example, the values of the map `sumsMap` can be given the more precise type `DB[db] -> Number[10..120db]`, without losing as much information as when using the unknown sensitivity.² For instance, consider two functions `f5` and `f10` that require their argument to be at most 5-sensitive and 10-sensitive with respect to `db`, respectively. Then, depending on the declared sensitivity of the values in `sumsMap`, we have:

²Note that for concrete GSoul syntax we use the `a..b` notation for intervals, instead of the `[a, b]` notation used in the paper.

Declared sensitivity	$f5(\text{sumsMap}(:\text{grade}))$	$f10(\text{sumsMap}(:\text{grade}))$
$0..120\text{db}$	fails at runtime	runs successfully
$10..120\text{db}$	fails statically	runs successfully
$120\text{db} (\triangleq 120..120\text{db})$	fails statically	fails statically

In the second column, we can observe that the use of $10..120\text{db}$ provides the programmer earlier feedback than when using $0..120\text{db}$. In particular, with such a bounded type, we know statically that the argument passed to $f5$ is *definitely* not less than 10-sensitive. Therefore, the type system can reject the program during typechecking. Additionally, the third column shows how the use of a supertype, instead of an imprecise type, needlessly rejects a valid program. Indeed, the use of $10..120\text{db}$ in the type of sumsMap provides the best balance between flexibility and static guarantees.

Although in the table we are only varying the lower bound of the interval, the same reasoning applies to the upper bound. For example, consider a function $f120$ that requires its argument to be at most 120-sensitive with respect to db . Then, one benefit of using $10..120\text{db}$ in $f120(\text{sumsMap}(:\text{grade}))$ would be that there is no need for a runtime check, as we know statically that the passed argument to $f120$ is *definitely* not more than 120-sensitive, in contrast to the upper bound being ∞ (which is the case when using the unknown sensitivity).

Intervals as a form of imprecision have already been explored by Toro et al. [2018] for the runtime semantics of a gradual security language. The difference here is that we use them not only in the runtime semantics but also in the surface syntax, both being valid design choices, as pointed out by Toro et al. Focusing on the runtime semantics of GSoul, the use of intervals is crucial for (1) enforcing modular type-based invariants (§3.3); (2) designing a gradual interpretation of sensitivity soundness, specially when dealing with optimistic static assumptions where the lower bound of intervals is used as the worst-case scenario for the main theorem (§5.1); and (3) providing a stronger soundness property for gradual sensitivity typing, whenever the upper bound of sensitivity intervals can be bounded to not be infinite (§5.2).

3 GSMini: GRADUAL SENSITIVITY TYPING

We now present GSMini, a core language with gradual sensitivity typing, which we later extend to GSoul (§6). We use **green** for sensitivity related variables, and **blue** for the rest of the math notation. The semantics of the language are initially based on SAX [Toro et al. 2023] and its latent type-and-effect discipline, here extended with ascriptions and explicit resource polymorphism. GSMini (and its extension, GSoul) is crafted by following the principles of the Abstracting Gradual Typing methodology [Garcia et al. 2016]. However, this paper only discusses the interesting aspects of the language, instead of detailing the gradualization process, as the application of AGT to various settings is already well-documented in the literature [Garcia et al. 2016; Lehmann and Tanter 2017; Malewski et al. 2021; Toro et al. 2018, 2019; Toro and Tanter 2017, 2020]. Nevertheless, for the curious reader, we present a static sensitivity language, Soul, and the technical details of its AGT-driven gradualization in the supplementary material.

3.1 Syntax

The syntax of GSMini is presented in Figure 1. We aim to support imprecision only in the sensitivity parts of the type system. Therefore, the “dynamic” end of the spectrum is simply typed. As explained and illustrated in §2, we support both unbounded imprecision via the fully-unknown sensitivity $?$ and bounded imprecision via sensitivity intervals such as $[1, 2]$. A gradual sensitivity is defined as a valid interval of two static sensitivities where the lower bound is less than or equal to the upper bound. A sensitivity s is a positive real number or infinity, where $\infty \cdot 0 = 0 \cdot \infty = 0$ and $s \cdot \infty = \infty \cdot s = \infty$ if $s > 0$. This way, a gradual sensitivity captures the plausibility of a

$$\begin{array}{ll}
c \in \mathbb{R} \cup \mathbb{B}, & x \in \text{Var}, \quad e \in \text{Expr}, \quad r \in \text{Res}, \quad i \in \text{GSens}, \quad \Xi \in \text{GSEff}, \quad g \in \text{GType}, \quad G \in \text{GType}_\Sigma \\
\\
i ::= [s, s] & (\text{with sugar } ? \triangleq [0, \infty] \text{ and } s \triangleq [s, s]) \quad (\text{gradual sensitivities}) \\
\Xi ::= ir + \dots + ir & (\text{gradual sensitivity effects}) \\
g ::= \mathbb{R} \mid \mathbb{B} \mid G \rightarrow G \mid \forall r. G & (\text{gradual types}) \\
G ::= g[\Xi] & (\text{gradual type-and-effects}) \\
e ::= c \mid \text{op } \bar{e} \mid x \mid \lambda(x : G).e \mid ee \mid \Lambda r.e \mid e[\Xi] \mid e :: G & (\text{expressions})
\end{array}$$

Fig. 1. Syntax of Gradual Sensitivity Types

sensitivity being any number within the range. Naturally, the unknown sensitivity $?$ is just syntactic sugar for the interval $[0, \infty]$, and a *fully precise* sensitivity s is sugar for the interval $[s, s]$. For example, the gradual sensitivity effect $[0, 5]r_1 + [0, \infty]r_2 + [3, 3]r_3$ is the desugared version of $[0, 5]r_1 + ?r_2 + 3r_3$.

Expressions e include constants, primitive operations, variables, lambda expressions, applications, resource abstractions, sensitivity effect applications, and ascriptions.

A type g can be the real number type \mathbb{R} , the boolean type \mathbb{B} , a function type $G \rightarrow G$, or a resource quantification $\forall r. G$. A type-and-effect G is a tuple $g[\Xi]$ associating a type g and a sensitivity effect Ξ , which is a mapping between resource variables r and gradual sensitivities i . Across the paper, we use meta-variables x, y, z (written in blue) for lambda-bound variables and r (written in green) for resource variables. For simplicity we write effects as first-order polynomials, e.g. $\Xi = r_1 + ?r_2$ is a sensitivity effect where $\Xi(r_1) = 1$, $\Xi(r_2) = ?$, and $\Xi(r) = 0$ for any other resource r in scope.

Note that the homogeneous type-and-effect syntax allows us to encode latent effects without having to annotate arrows or other type constructors. For example, the type-and-effect of an expression that reduces to a function has the form $(g_1[\Xi_1] \rightarrow g_2[\Xi_2])[\Xi]$, where Ξ_1 is the expected effect of the argument, Ξ_2 is called the *latent effect* of the produced function and Ξ is the effect of evaluating the expression. As usual in type-and-effect systems, a latent effect ensures that the effect of the body of the produced function is accounted for upon each function application.

3.2 A Gradual Sensitivity Type System

The typing rules for GSMINI are presented in Figure 2. The judgment $\Gamma; \Omega \vdash_s e : G$ establishes that expression e has type-and-effect G under type-and-effect environment Γ and resource set Ω , where Γ is a mapping from variables to type-and-effects and Ω is a set of the resource variables in scope. Typing rules are standard except for the handling of sensitivity effects. $(G\text{CONST})$ and $(G\Lambda)$ report no effect, \emptyset , as they are introduction forms and thus no resource variables are being used. Constants c are typed using the auxiliary function $ty(c)$, and primitive n-ary operations op are given meaning by the function δ_{op} , which handles the treatment of sensitivity information. The effect of addition δ_+ is computed by adding the effect of each sub-expression. The addition of sensitivity effects is defined by adding the sensitivities of each variable, i.e. $(\Xi_1 + i_1r) + (\Xi_2 + i_2r) = (\Xi_1 + \Xi_2) + (i_1 + i_2)r$. Addition $(+)$, multiplication $(*)$, join (\vee) and meet (\wedge) of gradual sensitivities is defined bound-wise, i.e. for addition: $[s_1, s_2] + [s_3, s_4] = [s_1 + s_3, s_2 + s_4]$. The join and meet of two static sensitivities is the maximum and minimum of both, respectively. The effect of a comparison δ_{\leq} scales to infinite the addition of the sensitivity effects of each sub-expression because a small variation on one of the sub-expression could change the result from *true* to *false*, which are considered ∞ apart [Reed and Pierce 2010]. The scaling of effects is defined naturally: $i(\Xi + i'r) = i\Xi + (i * i')r$. Lastly, the effect of a multiplication δ_* is computed scaling the addition of the sensitivity effects of each sub-expression by ∞ . The choice of syntax for types ensure that

latent effects are naturally captured by introduction rules, while being standard. In contrast to previous work [Near et al. 2019; Reed and Pierce 2010; Toro et al. 2023], resources are explicitly introduced using resource abstraction $\Delta r.e$, where the resource variable r is bound in the body e . As expected, rule (GA) is the only one that extends the resource variables set Ω . The judgment $\Omega \vdash G$ ensures that the type-and-effect G only contains resources contained in Ω and can be found in the supplementary material.

For (GAPP) we follow the presentation of Garcia et al. [2016] of using partial meta-functions `dom` and `cod` to extract the domain and codomain of a function type-and-effect. However, we have to adapt these functions to work on a type-and-effect system. In particular, the `cod` function reports not only the effect of reducing the expression to a value, Ξ , but also the latent effect of the function, contained in G_2 , and the effect of reducing the expression to a function. We do this by leveraging the $(\mid+)$ operator that sums a given effect to a type-and-effect. This way, we account for the latent effect of the body at application time. More generally, we define $g[\Xi] (\text{op}) \Xi' = g[\Xi \text{ op } \Xi']$, where op denotes a binary operation between two sensitivity effects.

Rule (GINST) removes a resource from scope by instantiating it with a sensitivity effect Ξ . We use a meta-function `inst` that works in a similar fashion to `cod`, but also performs a substitution of the resource r with the sensitivity effect passed as argument. A sensitivity effect substitution, $[\Xi_1/r]\Xi$, replaces all occurrences of r in Ξ by Ξ_1 . For example, $[2r_2 + r_3/r_1](3r_1 + ?r_2) = 3(2r_2 + r_3) + ?r_2 = (6 + ?)r_2 + 3r_3 = [6, \infty]r_2 + 3r_3$.

As previously mentioned, GDMINI relies on a relaxed notion of subtyping, consistent subtyping. Instead of checking for exact subtyping, consistent subtyping check if it is *plausible* that a type is a subtype of another. We start by defining consistent subtyping for gradual sensitivities:

Definition 1 (Consistent Sensitivity Subtyping). $[s_1, s_2] \widetilde{\leq} [s_3, s_4]$ if and only if $s_1 \leq s_4$.

The intuition behind this definition is that a gradual sensitivity i_1 is optimistically less or equal to another, i_2 , if there *exists* a sensitivity within i_1 that is less or equal to another one within i_2 . This is easily checkable by comparing the lower bound of i_1 with the upper bound of i_2 , which corresponds to the best-case scenario, an expected behavior for an optimistic subtyping relation. Consistent subtyping for effects, types and type-and-effects, denoted by $\widetilde{\leq}$, is defined naturally from Definition 1 and can be found in the supplementary material.

Plausibility in the Type System. The type system of GDMINI optimistically accepts judgments that *may* hold during runtime. For instance, consider the open expression $e = (x + x) :: \mathbb{R}[?r] :: \mathbb{R}[r]$, a resource set $\Omega = r$ and a type environment $\Gamma = x : \mathbb{R}[r]$. Given that both $\mathbb{R}[2r] \widetilde{\leq} \mathbb{R}[?r]$ and $\mathbb{R}[?r] \widetilde{\leq} \mathbb{R}[r]$ hold, the type derivation of $\Gamma; \Omega \vdash e : \mathbb{R}[r]$ can be easily constructed using the rule (GASCR) twice. This is because, given the removal of precision through the use of $?$, for the typechecker it is *plausible* that they might hold at runtime.

3.3 Accounting for Plausibility

Before moving on to the dynamic semantics of GDMINI, we now present the mechanism for performing runtime checks to ensure that the optimistic judgments hold at runtime. In our language, this is done through the use of *evidences*, which are used to *justify* a single consistent subtyping judgment and can be combined to (try to) justify transitivity of two consistent judgments [Garcia et al. 2016]. As standard for consistent subtyping, an evidence ε is represented as a pair of type-and-effects, written $\langle G, G \rangle$, each of which is at least as *precise* as the types involved in the consistent subtyping relation. Precision in the context of gradual sensitivities corresponds to interval inclusion. Formally, $[s_1, s_2]$ is more precise than $[s_3, s_4]$, written $[s_1, s_2] \sqsubseteq [s_3, s_4]$, if and only if $s_1 \geq s_3$ and $s_2 \leq s_4$. Precision for gradual sensitivity effects, types and type-and-effects

Well-typed gradual expressions

$$\boxed{\Gamma; \Omega \vdash e : G}$$

$$\begin{array}{c}
\text{(GCONST)} \\
\hline
\Gamma; \Omega \vdash_s c : \text{ty}(c)[\emptyset]
\end{array}
\quad
\begin{array}{c}
\text{(GOP)} \\
\hline
\frac{\Gamma; \Omega \vdash_s e_i : G_i \quad \delta_{\text{op}}(\overline{G_i}) = G}{\Gamma; \Omega \vdash_s \text{op } \overline{e_i} : G}
\end{array}
\quad
\begin{array}{c}
\text{(G}\Lambda\text{)} \\
\hline
\frac{\Gamma; \Omega, r \vdash e : G}{\Gamma; \Omega \vdash \Lambda r. e : (\forall r. G)[\emptyset]}
\end{array}$$

$$\begin{array}{c}
\text{(GAPP)} \\
\hline
\frac{\Gamma; \Omega \vdash e_1 : G_1 \quad \Gamma; \Omega \vdash e_2 : G_2 \quad G_2 \prec \text{dom}(G_1)}{\Gamma; \Omega \vdash e_1 e_2 : \text{cod}(G_1)}
\end{array}
\quad
\begin{array}{c}
\text{(GINST)} \\
\hline
\frac{\Gamma; \Omega \vdash e : G \quad \Omega \vdash \Xi}{\Gamma; \Omega \vdash e[\Xi] : \text{inst}(G, \Xi)}
\end{array}
\quad
\begin{array}{c}
\text{(GASCR)} \\
\hline
\frac{\Gamma; \Omega \vdash e : G \quad G \prec G' \quad \Omega \vdash G'}{\Gamma; \Omega \vdash e :: G' : G'}
\end{array}$$

$$\begin{array}{l}
g[\Xi] \text{ (} \text{+)} \Xi' = g[\Xi + \Xi'] \quad \text{dom}((G_1 \rightarrow G_2)[\Xi]) = G_1 \quad \text{cod}((G_1 \rightarrow G_2)[\Xi]) = G_2 \text{ (} \text{+)} \Xi \\
\text{inst}(\Lambda r. G_2[\Xi], \Xi_1) = [\Xi_1/r]G_2 \text{ (} \text{+)} \Xi \quad \delta_+(\mathbb{R}[\Xi_1], \mathbb{R}[\Xi_2]) = \mathbb{R}[\Xi_1 + \Xi_2] \\
\delta_*(\mathbb{R}[\Xi_1], \mathbb{R}[\Xi_2]) = \mathbb{R}[\infty(\Xi_1 + \Xi_2)] \quad \delta_{\leq}(\mathbb{R}[\Xi_1], \mathbb{R}[\Xi_2]) = \mathbb{B}[\infty(\Xi_1 + \Xi_2)]
\end{array}$$

Fig. 2. Type system of G_{SMINI} (excerpt)

is naturally defined from gradual sensitivities precision and can be found in the supplementary material.

Evidence is initially computed using the *interior* operator, which essentially produces a refined pair of gradual types from a consistent judgment, based solely on the knowledge that the judgment holds. Definition 2 shows the interior operator for consistent sensitivity subtyping. The interior operator for other constructs is inductively defined from the definition for gradual sensitivities and can be found in the supplementary material.

Definition 2 (Interior). *If $s_1 \leq s_2 \wedge s_4$ and $s_1 \vee s_3 \leq s_4$:*

$$\mathcal{I}_{<}([s_1, s_2], [s_3, s_4]) = \langle [s_1, s_2 \wedge s_4], [s_1 \vee s_3, s_4] \rangle$$

Otherwise, the interior operation is undefined.

As an example, consider the types $G_1 = \mathbb{R}[?r]$ and $G_2 = \mathbb{R}[10r]$, noting that $G_1 \prec G_2$. Then, we can compute the initial evidence for that judgment as $\varepsilon = \mathcal{I}_{<}(G_1, G_2) = \langle \mathbb{R}[[0, 10]r], \mathbb{R}[[10, 10]r] \rangle$. This evidence is said to *justify* the consistent subtyping judgment, written $\varepsilon \triangleright G_1 \prec G_2$, which is formally defined as $\varepsilon \sqsubseteq^2 \mathcal{I}_{<}(G_1, G_2)$.

Notice that consistent subtyping is not transitive. For instance, although $\mathbb{R}[10r] \prec \mathbb{R}[?r]$ and $\mathbb{R}[?r] \prec \mathbb{R}[5r]$, the transitive judgement of both, $\mathbb{R}[10r] \prec \mathbb{R}[5r]$, does not hold. Therefore, during runtime, evidences need to be *combined* in order to try to justify transitivity of two judgments. In the case where the combination of two evidences is not defined a runtime error is raised. Formally, this notion is captured by the *consistent transitivity operator*, denoted $\circ^{<}$. For instance, suppose $\varepsilon_1 \triangleright \mathbb{R}[10r] \prec \mathbb{R}[?r]$ and $\varepsilon_2 \triangleright \mathbb{R}[?r] \prec \mathbb{R}[5r]$. Since $[10, 10]$ is not plausibly smaller than $[5, 5]$ ($10 \not\leq 5$), then $\varepsilon_1 \circ^{<} \varepsilon_2$ should be undefined and an error should be raised.

Definition 3 shows the consistent transitivity operator for sensitivity subtyping. The consistent transitivity operator for other constructs is inductively defined from the definition for gradual sensitivities and can be found in the supplementary material.

Definition 3 (Consistent Sensitivity Subtyping Transitivity). *If $s_{11} \leq (s_{12} \wedge s_{14} \wedge s_{22})$ and $(s_{13} \vee s_{21} \vee s_{23}) \leq s_{24}$:*

$$\langle [s_{11}, s_{12}], [s_{13}, s_{14}] \rangle \circ^{<} \langle [s_{21}, s_{22}], [s_{23}, s_{24}] \rangle = \langle [s_{11}, s_{12} \wedge s_{14} \wedge s_{22}], [s_{13} \vee s_{21} \vee s_{23}, s_{24}] \rangle$$

$u ::= c \mid \lambda(x : g).t \mid \Delta r.t$	(simple values)	Well-typed terms		$t \in \mathbb{T}[G]$
$v ::= \epsilon u :: G$	(values)	(IGVAR)	(IGASCR)	
$t ::= v \mid \text{op } \bar{t} \mid x \mid \epsilon t :: G \mid t t$		$\frac{}{x^G \in \mathbb{T}[G]}$	$\frac{t \in \mathbb{T}[G] \quad \epsilon \triangleright G \lesssim G'}{\epsilon t :: G' \in \mathbb{T}[G']}$	
$\mid t [\Xi] \mid \epsilon \text{sctx}[r, \Xi](t) :: G$	(terms)			

Fig. 3. Syntax and Type System of G_{SMINI}_ε (excerpt)

Otherwise, the consistent transitivity operation is undefined.

Consider again two evidences such that $\epsilon_1 \triangleright G_1 \lesssim G_2$ and $\epsilon_2 \triangleright G_2 \lesssim G_3$. Then, the evidence $\epsilon_1 \circ^{<} \epsilon_2$, if defined, justifies the consistent subtyping judgment $G_1 \lesssim G_3$, which gives us the ability to perform runtime checks for consistent subtyping. Next, we come back to the choice using intervals as a mechanism for controlling imprecision.

Sensitivity Intervals in the Runtime Semantics. Intervals not only provide programmers with a fine-grained mechanism for controlling imprecision; they are also necessary in the design of the runtime semantics and for proving metric preservation. Fully-precise sensitivities and the unknown sensitivity $?$ fail to retain enough precision to guarantee expected runtime failures, as noted by Toro et al. [2018] and their use of security labels intervals in evidence. Bañados Schwerter et al. [2020] identify this issue as the runtime semantics not enforcing the *modular type-based invariants* expected from the static type discipline. Let us compare how evidence evolves when we do not use intervals. We omit the types for the sake of brevity. Suppose $\epsilon_1 \triangleright 3r \lesssim 5r$, $\epsilon_2 \triangleright 5r \lesssim ?r$ and $\epsilon_3 \triangleright ?r \lesssim 4r$. First of all, the interior operator without intervals would yield $\epsilon_1 = \langle 3r, 5r \rangle$, $\epsilon_2 = \langle 5r, ?r \rangle$ and $\epsilon_3 = \langle ?r, 4r \rangle$. In contrast, with intervals, evidences are computed as $\epsilon_1 = \langle 3r, 5r \rangle$, $\epsilon_2 = \langle 5r, [5, \infty]r \rangle$ and $\epsilon_3 = \langle [0, \infty]r, 4r \rangle$. Then if we compute $(\epsilon_1 \circ^{<} \epsilon_2) \circ^{<} \epsilon_3$:

Without intervals

$$\begin{aligned} & (\langle 3r, 5r \rangle \circ^{<} \langle 5r, ?r \rangle) \circ^{<} \epsilon_3 \\ &= \langle 3r, ?r \rangle \circ^{<} \langle ?r, 4r \rangle \\ &= \langle 3r, 4r \rangle \end{aligned}$$

With intervals

$$\begin{aligned} & (\langle 3r, 5r \rangle \circ^{<} \langle 5r, [5, \infty]r \rangle) \circ^{<} \epsilon_3 \\ &= \langle 3r, [5, \infty]r \rangle \circ^{<} \langle [0, 4]r, 4r \rangle \\ &= \text{undefined} \end{aligned}$$

In the setting without intervals, the operations do not fail because in the combination of ϵ_1 and ϵ_2 information is lost. From the beginning, we can notice that the right-hand side sensitivity of ϵ_2 will never be less than 5. Nevertheless, the result of the combination, $\langle 3r, ?r \rangle$, has no way to encode such information, so later when combined with ϵ_3 the operation can not be refuted. In contrast, the combination using intervals successfully yields an undefined result (since there is no intersection between $[5, \infty]$ and 4). This exemplifies how using only fully-precise sensitivities and the unknown sensitivity $?$ would yield unintended forgetful semantics [Greenberg 2015].

A final key reason as to why use intervals is that they are an expressive enough representation for having associativity of the consistent transitivity operator. Indeed, with intervals we have that $(\epsilon_1 \circ^{<} \epsilon_2) \circ^{<} \epsilon_3$ is equivalent to $\epsilon_1 \circ^{<} (\epsilon_2 \circ^{<} \epsilon_3)$, which is crucial for the soundness proof of GSoul. Furthermore, in the example without intervals, we can observe that associativity is not guaranteed as $\epsilon_2 \circ^{<} \epsilon_3$ is undefined, making the order of the combination of evidences relevant.

Equipped with a formal definition of evidences and operations for initially inferring (interior operator) and combining them (consistent transitivity operator), we can now augment the syntax of our gradual language with evidences that can be refined, at runtime, by combining them.

Notions of reduction

$$\boxed{t \longrightarrow t}$$

$$\begin{array}{ll}
\text{(IGR-OP)} & \text{op } (\overline{\varepsilon_i c_i :: G_i}) \longrightarrow \varepsilon c :: G \\
\text{[where]} & \varepsilon = \delta_{\text{op}}^2(\overline{\varepsilon_i}), \quad c = \llbracket \text{op} \rrbracket \overline{c_i}, \quad G = \delta_{\text{op}}(\overline{G_i}) \\
\text{(IGR-APP)} & (\varepsilon(\lambda(x : G'_1).t) :: G) (\varepsilon_1 u :: G_1) \longrightarrow \begin{cases} \varepsilon_2([\varepsilon'_1 u :: G'_1/x]t) :: \text{cod}(G) \\ \text{error} \end{cases} \quad \begin{array}{l} \text{if not defined} \\ \text{if not defined} \end{array} \\
\text{[where]} & G_1 = \text{dom}(G), \quad \varepsilon_2 = \text{icod}(\varepsilon), \quad \varepsilon'_1 = \varepsilon_1 \circ^{<:} \text{idom}(\varepsilon) \\
\text{(IGR-INST)} & (\varepsilon \lambda r.t :: G) [\Xi] \longrightarrow \text{inst}(\varepsilon, \Xi) \text{sctx}[r, \Xi](t) :: \text{inst}(G, \Xi) \\
\text{(IGR-SCTX)} & \varepsilon \text{sctx}[r, \Xi](v) :: G \longrightarrow \varepsilon[\Xi/r]v :: G \\
\text{(IGR-ASCR)} & \varepsilon(\varepsilon' u :: G') :: G \longrightarrow \begin{cases} (\varepsilon' \circ^{<:} \varepsilon)u :: G \\ \text{error} \end{cases} \quad \begin{array}{l} \text{if not defined} \\ \text{if not defined} \end{array} \\
\text{idom}(\langle G_1, G_2 \rangle) &= \langle \text{dom}(G_1), \text{dom}(G_2) \rangle & \delta_{\text{op}}^2(\langle \overline{G_{i1}}, \overline{G_{i2}} \rangle) &= \langle \delta_{\text{op}}(\overline{G_{i1}}), \delta_{\text{op}}(\overline{G_{i2}}) \rangle \\
\text{icod}(\langle G_1, G_2 \rangle) &= \langle \text{cod}(G_1), \text{cod}(G_2) \rangle & \text{inst}(\langle G_1, G_2 \rangle, \Xi) &= \langle \text{inst}(G_1, \Xi), \text{inst}(G_2, \Xi) \rangle
\end{array}$$

Fig. 4. Dynamic semantics of $\text{GSMINI}_\varepsilon$

3.4 Evidence-based Dynamic Semantics

In order to avoid writing reduction rules on actual (bi-dimensional) derivation trees, [Garcia et al. \[2016\]](#) leverage the use of *intrinsic terms*, a flat representation of terms that are isomorphic to type derivations [[Church 1940](#)]. More specifically, the typing judgment $\Gamma; \Omega \vdash e : G$ is now represented by an intrinsic term $t^G \in \mathbb{T}[G]$, where all the information contained in Γ and Ω is implicitly present in the syntax of t^G . We then introduce $\text{GSMINI}_\varepsilon$, a language with evidence-augmented intrinsic terms, which provides the runtime semantics for GSMINI .

In addition to the use of intrinsic terms, and similar to [[Toro et al. 2019](#)], we heavily rely on a type-directed translation that inserts explicit ascriptions to: (1) allow reduction rules to preserve the original type of an expression, and (2) ensure that all top-level constructor types match in the notions of reduction. The syntax of intrinsic terms is presented in Figure 3. Notice that values are ascribed simple values and introduction forms are always ascribed. We avoid writing the explicit type exponent whenever is not needed or can be inferred from the context, i.e., $t \in \mathbb{T}[G]$.

Dynamic Semantics of $\text{GSMINI}_\varepsilon$. Figure 4 presents the reduction rules for $\text{GSMINI}_\varepsilon$. Rule (IGR-OP) relies on the meta-definition of $\llbracket \text{op} \rrbracket$ and the evidence is computed using the evidence operator δ_{op}^2 . The application rule, (IGR-APP), uses the inversion functions *idom* and *icod* in order to compute the new evidences for the body of the closure and to justify optimistic judgments and fail otherwise. Finally, rule (IGR-ASCR) eliminates ascriptions by keeping only the outer one; consistent transitivity is performed in order to justify the new ascribed (simple) value. Rule (IGR-INST) reduces the application of resource abstraction to an *sctx* context, where the substitution is delayed until the body of the abstraction is reduced, as observable in rule (IGR-SCTX).

Refuting optimistic judgments. In the rule (IGR-INST), if we simply substituted r by Ξ , we may lose information necessary to refute optimistic static assumptions. Consider the following example, where c is some constant and the resource r_1 is in scope: $(\lambda r_2.c :: r_1 :: ?r_1 :: r_2) [r_1]$. Notice that when comparing effects by consistent subtyping, $r_1 \leq ?r_1$ and $?r_1 \leq r_2$, but $r_1 \not\leq r_2$. Although, this program typechecks statically, it should fail at runtime, since clearly $1r_1$ is not a sub-effect of $1r_2$. However, if we immediately substitute r_2 by its instantiation, r_1 , we would get $x :: ?r_1 :: r_1$, which is

Term Precision

$$\boxed{t \sqsubseteq t}$$

$$\begin{array}{c} (IG \sqsubseteq_{\lambda}) \\ \frac{G_{11} \sqsubseteq G_{21} \quad t_{12} \sqsubseteq t_{22}}{\lambda(x : G_{11}).t_{12} \sqsubseteq \lambda(x : G_{21}).t_{22}} \end{array} \quad \begin{array}{c} (IG \sqsubseteq_{\text{INST}}) \\ \frac{t_1 \sqsubseteq t_2 \quad \Xi_1 \sqsubseteq \Xi_2}{t_1 [\Xi_1] \sqsubseteq t_2 [\Xi_2]} \end{array} \quad \begin{array}{c} (IG \sqsubseteq_{::}) \\ \frac{\varepsilon_1 \sqsubseteq^2 \varepsilon_2 \quad t_{11} \sqsubseteq t_{21} \quad G_{12} \sqsubseteq G_{22}}{\varepsilon_1 t_{11} :: G_{12} \sqsubseteq \varepsilon_2 t_{21} :: G_{22}} \end{array}$$

Fig. 5. Precision of terms (excerpt)

going to silently reduce without errors, exposing an inconsistency between the type invariants in the static and dynamic semantics. In order to avoid this problem, we delay the substitution using a `sctx` context to first reduce the body of the abstraction and then perform the substitution.³ This way, evidences preserve the information necessary to refute optimistic judgments when trying to justify transitivity in rule (IGR-ASCR). The example is then instead reduced as follows:

$$\begin{array}{c} (\lambda r_2.c :: r_1 :: ?r_1 :: r_2) [r_1] \quad \xrightarrow{\text{(IGR-INST)}} \quad \text{sctx}[r_2, r_1](c :: r_1 :: ?r_1 :: r_2) \quad \xrightarrow{\text{..., (IGR-ASCR)}} \quad \text{sctx}[r_2, r_1](\text{error}) \mapsto \text{error} \end{array}$$

Elaboration. So far, we have defined the runtime semantics of $\text{GSMINI}_{\varepsilon}$ that, by translation, also defines the runtime semantics for GSMINI . Judgment $\Gamma; \Omega \vdash e : G \rightsquigarrow t^G$ denotes the elaboration of the intrinsic term t^G from the expression e , where e has type G under the type environment Γ and a resource set Ω . The rules of elaboration simply perform the transformations mentioned at the beginning of this section and are presented in the supplementary material. However, as an example, we present the rule for elaborating ascribed expressions, which showcases the use of the interior operator to produce the initial evidence to justify the ascription:

$$\begin{array}{c} (\text{ELASCR}) \\ \frac{\Gamma; \Omega \vdash e : G \rightsquigarrow t \quad \Omega \vdash G' \quad \varepsilon = I_{<}; (G, G')}{\Gamma; \Omega \vdash e :: G' : G' \rightsquigarrow \varepsilon t :: G'} \end{array}$$

Finally, we note that, after typing, elaboration rules only insert trivial ascriptions and enrich derivations with evidence and ascriptions. As this derivations are represented as intrinsic terms, by construction, elaboration of terms trivially preserves typing.

Proposition 1 (Elaboration preserves typing). *If $\Gamma; \Omega \vdash e : G$, then $\Gamma; \Omega \vdash e : G \rightsquigarrow t$ and $t \in \mathbb{T}[G]$.*

4 TYPE SAFETY AND THE GRADUAL GUARANTEE FOR GSMINI

We now study the metatheoretical properties of GSMINI and $\text{GSMINI}_{\varepsilon}$. Recall that we differentiate expressions e of GSMINI from evidence-augmented terms t of $\text{GSMINI}_{\varepsilon}$. We first prove that $\text{GSMINI}_{\varepsilon}$ is type safe: closed terms do not get stuck, but they still can halt with a runtime error.

Proposition 2 (Type safety for $\text{GSMINI}_{\varepsilon}$). *Let t be a closed $\text{GSMINI}_{\varepsilon}$ term. If $t \in \mathbb{T}[G]$, then either t is a value v , or $t \mapsto t'$ for some $t' \in \mathbb{T}[G]$, or $t \mapsto \text{error}$.*

³A similar issue also arises in the context of gradual *parametricity* [Ahmed et al. 2017; Toro et al. 2019]. A stereotypical example is the expression $(\text{AX}.1 :: ? :: X)[\text{Int}]$, which ought to fail, despite the coincidental match between the type of the literal `1` and the instantiated type for X . Notice how: (1) the expression optimistically typechecks, as both $\text{Int} \sim ?$ and $? \sim X$ hold; and (2) a runtime error should be expected, as $\text{Int} \not\approx X$. Gradually-parametric languages usually tackle this issue by adopting *runtime sealing*, i.e. a type application reduces using a sealing substitution with a fresh global type name [Matthews and Ahmed 2008].

The static semantics of GSMINI also satisfy the static gradual guarantee: typeability is monotone with respect to imprecision. Figure 5 presents the precision relation for terms, which correspond to the natural lifting of type precision. Precision on GSMINI expressions is defined analogously to term precision (modulo the evidence parts), and is omitted for brevity.

Proposition 3 (Static gradual guarantee for GSMINI). *Let e_1 and e_2 be two closed GSMINI expressions such that $e_1 \sqsubseteq e_2$ and $\cdot; \cdot \vdash e_1 : G_1$. Then, $\cdot; \cdot \vdash e_2 : G_2$ and $G_1 \sqsubseteq G_2$.*

Finally, GSMINI_ϵ satisfies the dynamic gradual guarantee (DGG): any program that reduces without error will continue to do so if imprecision is increased.

Proposition 4 (Dynamic gradual guarantee for GSMINI_ϵ). *Let t_1 and t_2 be two closed GSMINI_ϵ terms such that $t_{11} \sqsubseteq t_{12}$. If $t_{11} \mapsto t_{21}$ then $t_{12} \mapsto t_{22}$ where $t_{21} \sqsubseteq t_{22}$.*

The proofs for the results of this section are subsumed by the proofs for the full languages, GSoul and GSoul_ϵ (described in Section 6.2), which are provided in the supplementary material.

5 SOUNDNESS OF SENSITIVITY TYPING FOR GSMINI_ϵ

We now study the main metatheoretical property of GSMINI_ϵ , namely the soundness of sensitivity typing known as *metric preservation* [Reed and Pierce 2010]. Metric preservation is a hyperproperty that captures the bound on how much the result of two similar computations may change given an input variation.

In §5.1, we present a notion of gradual metric preservation and illustrate the key ideas of its design. We show that, due to possibly infinite imprecision in sensitivities, it is not possible to establish a uniform behavior for related terms with respect to termination (either divergence or failure): gradual metric preservation is *termination insensitive*. In words, if both terms terminate to values, then the difference between these values is bounded; otherwise the relation is vacuously true.⁴ Then, in §5.2, we observe that when imprecision is restricted to *bounded* imprecision, i.e. not plausibly infinite, GSMINI_ϵ programs satisfy a stronger, *termination-sensitive* gradual metric preservation: if one term terminates to a value, then the other also terminates to a value at a bounded distance.

5.1 Gradual Metric Preservation

Figure 6 presents the logical relation and auxiliary definitions for metric preservation. The relation is defined using two mutually-defined interpretations: one for values $\mathcal{V}_\Delta[[G]]$, and one for terms $\mathcal{T}_\Delta[[G]]$. The interpretations for a type G are indexed by a *distance environment* Δ . A distance environment Δ is just a mapping between sensitivity variables and gradual sensitivities (same as Ξ), and it represents the distance of different inputs across two different executions. We use the meta-variable d to represent gradual sensitivities in the distance environment. This environment is used to compute the *predicted output distance* of a program, i.e. the maximum variation of the result of the program if closed by any two substitutions that satisfy the Δ distances. The predicted output distance of a program of effect Ξ is computed as $\Delta \cdot \Xi$, where \cdot is the sum of the point-wise multiplication of the gradual sensitivities in Δ and Ξ : $\Delta \cdot \Xi = \sum_{r \in \text{dom}(\Delta) \cup \text{dom}(\Xi)} \Delta(r) * \Xi(r)$. Notice that the predicted output distance is a gradual sensitivity, i.e. a range of possible distances, and it is not a single value.

In order to get an intuitive understanding of metric preservation, let us consider a simple static example. Metric preservation allows us to reason about open programs such as $t = x + x$, which may correspond to the body of the lambda in the program $\Delta r.(\lambda x : \mathbb{R}[r].x + x)$. In this spirit, let

⁴We use the term “termination (in)sensitivity” to concisely deal with both divergence and runtime errors—as discussed by Fennell and Thiemann [2013] in the case of gradual information flow security, a runtime error can be treated as divergence.

$$\begin{aligned}
(v_1, v_2) \in \text{Atom}[G] &\iff v_1 \in \mathbb{T}[G] \wedge v_2 \in \mathbb{T}[G] \\
(v_1, v_2) \in \mathcal{V}_\Delta[\mathbb{R}[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[\mathbb{R}[\Xi]] \wedge \neg(\Delta \cdot (\Xi'_1 \gamma \Xi'_2) \widetilde{<} |u_1 - u_2|) \\
&\quad \text{where } v_i = \langle \mathbb{R}[\emptyset], \mathbb{R}[\Xi'_i] \rangle u_i :: \mathbb{R}[\Xi] \\
(v_1, v_2) \in \mathcal{V}_\Delta[\mathbb{B}[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[\mathbb{B}[\Xi]] \wedge (\Delta \cdot (\Xi'_1 \gamma \Xi'_2) \widetilde{<} \infty \implies u_1 = u_2) \\
&\quad \text{where } v_i = \langle \mathbb{B}[\emptyset], \mathbb{B}[\Xi'_i] \rangle u_i :: \mathbb{B}[\Xi] \\
(v_1, v_2) \in \mathcal{V}_\Delta[(G_1 \rightarrow G_2)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(G_1 \rightarrow G_2)[\Xi]] \wedge \\
&\quad (\forall v'_1, v'_2, (v'_1, v'_2) \in \mathcal{V}_\Delta[G_1]. (v_1 v'_1, v_2 v'_2) \in \mathcal{T}_\Delta[G_2 \text{ (}\vdash\text{)} \Xi]) \\
(v_1, v_2) \in \mathcal{V}_\Delta[(\forall r.G)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(\forall r.G)[\Xi]] \wedge \\
&\quad (\forall \Xi''.(v_1[\Xi''], v_2[\Xi'']) \in \mathcal{T}_\Delta[(\Xi''/r)G \text{ (}\vdash\text{)} \Xi]) \\
(t_1, t_2) \in \mathcal{T}_\Delta[G] &\iff (t_1 \mapsto^* v_1 \wedge t_2 \mapsto^* v_2) \implies (v_1, v_2) \in \mathcal{V}_\Delta[G] \\
(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta[\Gamma] &\iff \text{dom}(\gamma_1) = \text{dom}(\gamma_2) = \text{dom}(\Gamma) \wedge \\
&\quad \forall x \in \text{dom}(\Gamma). (\gamma_1[x], \gamma_2[x]) \in \mathcal{V}_\Delta[\Gamma(x)]
\end{aligned}$$

Fig. 6. Logical relations for gradual sensitivity soundness

us choose $\Gamma = x : r$ and $\Omega = r$ (noticing that they effectively close the term t), then t has effect $2r$. Consequently, if the distance environment is $\Delta = 3r$, i.e., two values of two different executions of effect r are at most at distance 3, then the predicted output distance is $(3r) \cdot (2r) = 3 * 2 = 6$. For instance, consider the following terms:

$$e_1 = (\lambda r. (\lambda x : \mathbb{R}[r]. x + x) (\mathbf{1} :: \mathbb{R}[r])) [\emptyset] \quad e_2 = (\lambda r. (\lambda x : \mathbb{R}[r]. x + x) (\mathbf{4} :: \mathbb{R}[r])) [\emptyset]$$

The expressions e_1 and e_2 will correspond to the use of two different value environments γ_1 and γ_2 , such that $\gamma_1 = \{x \mapsto \mathbf{1} :: \mathbb{R}[r]\}$ and $\gamma_2 = \{x \mapsto \mathbf{4} :: \mathbb{R}[r]\}$. Notice that both values for x have effect r , and are at distance 3, respecting both Γ and Δ . If we reduce both terms, we get $(2 :: \mathbb{R}[2r])$, and $(8 :: \mathbb{R}[2r])$, which are exactly at the predicted output distance (6).

Metric preservation also supports directly reasoning about variables that depend on multiple resources. For instance, let us consider that in the expression $x + x$, x now has effect $1r_1 + 2r_2 + 3r_3$. We can choose Δ as $4r_1 + 5r_2 + 6r_3$, stating that we can only close the expression with two different environments that differ in at most 4, 5 and 6 for the resources r_1 , r_2 and r_3 , respectively. Then, for this setting, the result can vary in at most 64, since $\Delta \cdot \Xi = (4r_1 + 5r_2 + 6r_3) \cdot (2 * (1r_1 + 2r_2 + 3r_3)) = 4 \cdot 2 + 5 \cdot 4 + 6 \cdot 6 = 64$

Metric Preservation in an Imprecise World. We write $(v_1, v_2) \in \mathcal{V}_\Delta[G]$ to denote that values v_1 and v_2 are related at type G and distance environment Δ . The general structure of an interpretation of some type G , is of the form $(v_1, v_2) \in \mathcal{V}_\Delta[G] \iff (v_1, v_2) \in \text{Atom}[G] \wedge \tilde{P}(v_1, v_2, \Delta, G)$, where $(v_1, v_2) \in \text{Atom}[G]$ denotes that both values are of type G ,⁵ and \tilde{P} is a gradual counterpart of the corresponding proposition P in the static interpretation about v_1 and v_2 for Δ and G . For instance, in a static language (such as SAX or FUZZ), two numbers are related if the distance is bounded by the predicted output distance as shown in gray: $(n_1, n_2) \in \text{static}\mathcal{V}_\Delta[\mathbb{R}[\Xi]] \iff (n_1, n_2) \in \text{Atom}[\mathbb{R}[\Xi]] \wedge |n_1 - n_2| \leq \Delta \cdot \Xi$, assuming that Ξ and Δ are fully precise.⁶

A key challenge that this work identifies in gradualizing metric preservation is that the gradual counterpart of this interpretation cannot just lift inequality. One may be tempted to define the

⁵Note that we could have used the typing judgment instead of the interpretation of atoms, but we choose to use the interpretation to later augment the relation with more restrictions in §5.2.

⁶Note that when Ξ and Δ are fully precise, the predicted output distance is a single value, thus the \leq operation can be used directly.

interpretation of numbers using consistent sensitivity subtyping, interpreting $|n_1 - n_2|$ as a one-point gradual sensitivity, as so:

$$(n_1, n_2) \in \text{naive}\mathcal{V}_\Delta[\mathbb{R}[\Xi]] \iff (n_1, n_2) \in \text{Atom}[\mathbb{R}[\Xi]] \wedge |n_1 - n_2| \lesssim \Delta \cdot \Xi$$

Indeed, there are several problems with such a definition. First, since GSMINI_ϵ values are ascribed simple values, we cannot relate numbers directly, therefore we have to relate ascribed numbers (v_1, v_2) and get the actual numeric constants by pattern matching on the ascribed values. Second, metric preservation is a compositional property, and it requires to reason compositionally about value interpretations. In particular, given two related values, their ascriptions should yield (more precise) related values. One novel and important consideration when designing the logical relation for values in the gradual setting is that, when lifting a predicate P to its gradual counterpart, we have to reason about worst-case scenarios for \tilde{P} . Otherwise, if we would take optimistic assumptions, one could always ascribe/refine two related values to a less sensitive type-and-effect (justified by gradual plausibility) breaking compositionality. To illustrate, let $G_1 = \mathbb{R}[[1, 2]r]$, $v_1 = \langle G_1, G_1 \rangle 1 :: G_1$, $v_2 = \langle G_1, G_1 \rangle 3 :: G_1$. Then $(v_1, v_2) \in \text{naive}\mathcal{V}_{[1,1]r}[G_1]$ because $|1 - 3| \lesssim [1, 2]$. Now if we ascribe those values to $G'_1 = \mathbb{R}[1r]$, we have that $\langle G'_1, G'_1 \rangle v_i :: G'_1 \mapsto \langle G'_1, G'_1 \rangle n_i :: G'_1$, where $n_1 = 1, n_2 = 3$. But the resulting values do not belong to the interpretation of G'_1 , because $|1 - 3| \lesssim [1, 1]$ does not hold.

More technically, satisfiability of the predicate \tilde{P} is not preserved after gaining precision. Consider consistent sensitivity subtyping: $2 \lesssim [1, 2]$ is true, but $2 \lesssim [1, 1]$ is not (and $[1, 1] \sqsubseteq [1, 2]$). For this reason, gradual metric preservation must follow a pessimistic approach, and instead *negate the lifting of the negated static proposition*, i.e. $\neg(\neg\tilde{P})$. In particular, as $\neg(s_1 \leq s_2)$ means $s_2 < s_1$, we use the negation of *consistently less than* defined as: $[s_1, s_2] \gtrsim [s_3, s_4] \iff s_1 < s_4$. Notice that proposition $\tilde{P}(x) = \neg(i \gtrsim x)$ preserves satisfiability when i gains precision. And thus, in the previous example, we do not relate v_1 and v_2 at type G_1 , because $\neg([1, 2] \gtrsim 2)$ is false. This design requirement highlights the usefulness of using intervals, and more specifically, the lower bound of gradual sensitivities, as the distance between two values is actually compared against the lower bound of the predicted (gradual) output distance. If the unknown sensitivity were the only form of runtime imprecision, then any time that the runtime information could not be refined to a fully-precise sensitivity, two related values would be forced to be equal (considering that the worst-case sensitivity of $?$ is 0), making the logical relation much stricter than necessary.

Finally, the lifted predicates use the most precise information about types found in evidences—the computationally-relevant bits for runtime tracking. Otherwise, losing precision in the type (e.g. by ascribing two related values to a very imprecise type) could break soundness. This can be observed in the definition of related numbers or related booleans, where we use the sensitivity effect of the evidence to compute the predicted output distance. To illustrate this point, consider $G_2 = \mathbb{R}[[2, 2]r]$, and $v'_1 = \langle G_2, G_2 \rangle 1 :: G_2$, $v'_2 = \langle G_2, G_2 \rangle 3 :: G_2$. The values are related because $\neg([2, 2] \gtrsim 2)$ is true. But, if we ascribe those values to $G'_2 = \mathbb{R}[[0, 2]r]$, then the resulting values would not be related as $\neg([0, 2] \gtrsim 2)$ is false. On the contrary, if we consider only evidence information, we do not have this problem as the underlying evidence of the resulting values maintain their original precision as $G_2 \sqsubseteq G'_2$. Armed with these considerations, we now discuss each interpretation in detail.

(A) *Related numbers and booleans.* As explained before, two numbers are related if the difference between them is bounded by the predicted output distance. The predicted output sensitivity d , is computed using $\Xi'_1 \vee \Xi'_2$ as the (worst-case) predicted sensitivity of the values, i.e. $d = \Delta \cdot (\Xi'_1 \vee \Xi'_2)$. Effects Ξ'_1 and Ξ'_2 are obtained from the evidences, and correspond to the most precise sensitivity information about v_1 and v_2 respectively. We use the join because the precision of the evidences

could be different. As explained before, we want the difference to be less or equal to *all* values within d . In other words, we do *not* want a value within d that is less than the difference to exist, so $\neg(d \lesssim |u_1 - u_2|)$. For the case of booleans, **true** and **false** should only be related at ∞ , i.e. $d < \infty \implies u_1 = u_2$ (statically). Since the inequality is in contravariant position, we can directly use the consistent less than operator to account for the pessimistic analysis.⁷ As a final remark, notice that the left-side sensitivity effect of value evidences is always \emptyset , because the actual effect of the inner simple value is also \emptyset .

(B) *Related functions and resource abstractions*. Two functions or resource abstractions are related if their applications to two related arguments or a sensitivity effect, respectively, yield related computations.

(C) *Related terms*. A pair of terms are related if they both reduce to values, and those two values are related. Observe that this definition is weaker than the one originally proposed by Reed and Pierce [2010]: if either term does not reduce to a value, then the relation vacuously holds; we say that gradual metric preservation is *termination insensitive*. One simple example of a term that behaves differently (in terms of termination) when closed by different substitutions is the term $t = \varepsilon x :: \mathbb{R}[[2, 3]r]$, where $\varepsilon = \langle \mathbb{R}[[2, 3]r], \mathbb{R}[[2, 3]r] \rangle$. If we close it with $\gamma_1 = \{x \mapsto \varepsilon_1 1 :: \mathbb{R}[[1, 2]r]\}$ and $\gamma_2 = \{x \mapsto \varepsilon_2 1 :: \mathbb{R}[[1, 2]r]\}$, where $\varepsilon_1 = \langle \mathbb{R}[\emptyset], \mathbb{R}[1r] \rangle$ and $\varepsilon_2 = \langle \mathbb{R}[\emptyset], \mathbb{R}[[1, 2]r] \rangle$, then $\gamma_1(t)$ will halt with an error, while $\gamma_2(t)$ will not. Notice that the only difference between γ_1 and γ_2 is the precision of evidences, having $\varepsilon_1 \circ^{<} \varepsilon$ being undefined, in contrast to $\varepsilon_2 \circ^{<} \varepsilon$ which evaluates to ε (as no precision is gained). We come back to this aspect in §5.2.

(D) *Related substitutions*. Two substitutions are related at type environment Γ , if each variable in common, x , is bound to related values at type $\Gamma(x)$.

Lastly, to state out main theorem, we need notions of well-formedness of environments so they can be used to close and compute the predicted output distance of a term. These are based on the meta-functions $\text{FV}(\cdot)$ and $\text{FR}(\cdot)$ (defined in the supplementary material), which return the free variables and free resources of a term, respectively.

Definition 4 (Type environment well-formedness). *A type environment Γ is well-formed with respect to an intrinsic term t , denoted $t \vdash \Gamma$ if and only if $\text{FV}(t) \subseteq \text{dom}(t)$ and $\forall x^G \in \text{FV}(t). \Gamma(x) = G$.*

Definition 5 (Distance environment well-formedness). *Well-formedness of a distance environment Δ with respect to a type environment Γ , denoted $\Gamma \vdash \Delta$, is defined as follows:*

$$\frac{}{\cdot \vdash \Delta} \quad \frac{\Gamma \vdash \Delta \quad \text{FR}(G) \subseteq \text{dom}(\Delta)}{\Gamma, x : G \vdash \Delta}$$

Soundness of $\text{GSMINI}_\varepsilon$. If an open intrinsic term typechecks and we have a type environment Γ and a distance environment Δ that closes it (at the type level), then for any two related substitutions γ_1, γ_2 , the computations of the terms closed by γ_1 and γ_2 are related.

THEOREM 5 (FUNDAMENTAL PROPERTY FOR $\text{GSMINI}_\varepsilon$). *Let t be a $\text{GSMINI}_\varepsilon$ term. If $t \in \mathbb{T}[G]$ and $t \vdash \Gamma$, then $\forall \Delta, \gamma_1, \gamma_2$ such that $\Gamma \vdash \Delta$ and $(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta[\Gamma]$, we have $(\gamma_1(t), \gamma_2(t)) \in \mathcal{T}_\Delta[G]$.*

In particular we have the following metric preservation result for base types (where we write $t \Downarrow v$ for $\cdot; \cdot \vdash t : G \rightsquigarrow t' \wedge t' \mapsto^* v$ and $\text{uval}(\varepsilon u :: G) = u$):

COROLLARY 6 (GRADUAL METRIC PRESERVATION FOR GSMINI BASE TYPES). *Let f be a GSMINI function. If f has type $(\mathbb{R}[r] \rightarrow \mathbb{R}[ir])[\emptyset]$, for some gradual sensitivity i , then for any real numbers c_1, c_2, c such that $|c_1 - c_2| \leq c$, if $f c_1 \Downarrow v_1$, and $f c_2 \Downarrow v_2$, then $|\text{uval}(v_1) - \text{uval}(v_2)| \leq c \cdot \pi_2(i)$.*

⁷Technically, when applying the *negating the lifting of the negated static proposition* to the static proposition for booleans, we get $\neg(d \lesssim \infty \wedge u_1 \neq u_2)$, which is equivalent to $d \lesssim \infty \implies u_1 = u_2$.

$$\begin{aligned}
(v_1, v_2) \in \text{Atom}[G] &\iff v_1 \in \mathbb{T}[G] \wedge v_2 \in \mathbb{T}[G] \wedge \text{ev}(v_1) = \text{ev}(v_2) \\
(t_1, t_2) \in \mathcal{T}_\Delta[G] &\iff t_1 \mapsto^* v_1 \implies (\exists v_2. t_2 \mapsto^* v_2 \wedge (v_1, v_2) \in \mathcal{V}_\Delta[G])
\end{aligned}$$

Fig. 7. Termination-sensitive gradual metric preservation

Notice that this result is not as strong as the fundamental property, but it captures the expected static source level guarantee: the observed difference between values is bounded using the upper bound of the specified sensitivity interval. In contrast, the logical relations use the monitored lower bound of the sensitivity effect.

5.2 Termination-Sensitive Gradual Metric Preservation

We have so far established a general gradual metric preservation for GSoul terms that is termination insensitive. In this section, we study a stronger notion of gradual metric preservation that is termination sensitive. Although GSmInI lacks of recursion, and so divergence is not possible, programs can still halt with an error. Later in §6.2, when working with GSoul, which actually features general recursion, we will revisit termination-sensitive gradual metric preservation, considering both divergence and runtime failures. We now explore sufficient conditions for GSoul terms to satisfy a stronger *termination-sensitive* gradual metric preservation.

A key observation is to realize that different termination behavior across runs can be avoided if *sensitivity imprecision is finite*. We draw from D’Antoni et al. [2013] and Toro et al. [2019] to enforce the same behavior of divergence and failures between runs, respectively. Specifically, D’Antoni et al. [2013] restrict the definition of metric preservation to only reason about finite distances, effectively disallowing the different behaviors of reduction due to conditionals and case analysis; and in their work on gradual parametricity, Toro et al. [2019] require related values to have identical evidences (modulo sealing) to avoid different error behavior of reduction due to consistent transitivity errors.

We combine these two ideas to state a termination-sensitive gradual metric preservation. Figure 7 presents the new definition for related atoms, $\text{Atom}[\cdot]$, along with a termination-sensitive characterization of related terms. First, we require related values to have equal evidences, preventing one execution from failing while the other does not due to higher precision in one of the executions. All other cases of the value relation are unchanged from Figure 6, save for the use of the new definition of $\text{Atom}[\cdot]$. Additionally, we add a new restriction to the fundamental property: the output distance cannot plausibly be ∞ , as this would allow both executions to take different branches (recall that inl and inr values are at infinite distance).

THEOREM 7 ((TERMINATION-SENSITIVE) FUNDAMENTAL PROPERTY FOR GSmInI_ε). *Let t be a GSmInI_ε term. If $t \in \mathbb{T}[g[\Xi]]$, $t \vdash \Gamma$ and $\neg(\infty \lesssim \Delta \cdot \Xi)$, then $\forall \Delta, \gamma_1, \gamma_2$ such that $\Gamma \vdash \Delta$ and $(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta[\Gamma]$, we have $(\gamma_1(t), \gamma_2(t)) \in \mathcal{T}_\Delta[g[\Xi]]$.*

Notice that for imprecise terms, this result is only useful due to the availability of sensitivity intervals, as the presence of $?$ in the top-level type trivially makes the condition $\neg(\infty \lesssim \Delta \cdot \Xi)$ false. Finally, the two extra conditions are sufficient to establish termination-sensitive gradual metric preservation, as captured in the following corollary.

COROLLARY 8 ((TERMINATION-SENSITIVE) GRADUAL METRIC PRESERVATION FOR GSmInI BASE TYPES). *Let f be a GSmInI function. If f has type $(\mathbb{R}[r] \rightarrow \mathbb{R}[ir])[\emptyset]$, for some gradual sensitivity i , then for any real numbers c_1, c_2, c such that $|c_1 - c_2| \leq c$ and $c \cdot \pi_2(i) < \infty$, if $f c_1 \Downarrow v_1$ then $f c_2 \Downarrow v_2$ and $|uval(v_1) - uval(v_2)| \leq c \cdot \pi_2(i)$.*

The difference between Corollary 6 and Corollary 8 is that in the latter, the overall distance of the outputs $c \cdot \pi_2(i)$ cannot plausibly be infinite. Consequently, the result is termination sensitive. The fact that this stronger guarantee holds for G_{SMINI} terms whose imprecision is restricted to be finite illustrates how the progressive hardening of imprecision can lead to stronger guarantees. We observe that this should encourage programmers to use bounded imprecision whenever possible, which we see as a strong motivation for supporting sensitivity intervals in gradual source sensitivity languages.

6 FROM G_{SMINI} TO G_{SOUL}

So far we have only presented the core calculus G_{SMINI} (along with its runtime semantics, G_{SMINI_e}), which does not account for the full G_{SOUL} language we have implemented and used in the introduction (§2). We now describe G_{SOUL}, the superset of G_{SMINI} with products, sums and recursive types. In §6.1 we introduce the syntax, type system and runtime semantics of G_{SOUL}. Finally, in §6.2, we discuss the metatheory of G_{SOUL} along with the challenges and necessary changes for proving sensitivity soundness with these new features, especially in the presence of recursive types.

The addition of recursive types requires some care in the design of certain aspects of the language and its metatheory, such as iso-recursive subtyping and the need for a stronger proof technique for metric preservation. In particular, we need to introduce a new auxiliary subtyping judgment and step indexing for proving sensitivity soundness for G_{SOUL_e}. However, it is worth noting that other recursive types definitions such as syntax, typing rules, and reduction rules, can be incorporated without the need to modify existing definitions. Furthermore, product and sum types can be added in a completely modular manner, albeit not without care, specially when considering the gradual interpretation of metric preservation.

6.1 G_{SOUL}: G_{SMINI} with Products, Sums, and Recursive Types

Figure 8 presents the syntax and typing rules for G_{SOUL}, as well as the notions of reduction. The syntax and rules already presented for G_{SMINI} are unchanged and hence omitted. Rules (G_{PAIR}), (G_{INL}) are standard and analogous to (G_{LAM}); constructing a pair or an injection has no sensitivity effect. Similarly to (G_{APP}), in rule (G_{PROJ1}) we use a partial meta-function, *first*, to compute the type-and-effect of the projection. In (G_{CASE}), the type-and-effects of the branches are computed using extended type environments where each newly-bound variable is given a type-and-effect computed using the meta-functions *left* and *right*, which extract the left and right component of the sum type-and-effect, respectively. Then, the resulting type-and-effect is computed by joining the type-and-effect of each branch, and also joining the effect of reducing the sum expression, accounting for the case where branches do not use the bound variables.

To illustrate why we have to join the effect of e_1 in the rule (G_{CASE}), consider the open expression $\text{case } z \text{ of } \{x_1 \Rightarrow 0\} \{x_2 \Rightarrow x_2 + x_2\}$, a resource set $\Omega = r_1, r_2$, and a type environment $\Gamma = z : (\mathbb{R}[\infty r_1] \oplus \mathbb{R}[r_1])[r_2]$, then the type derivation follows as:

$$\begin{array}{c} \Gamma; \Omega \vdash_s z : (\mathbb{R}[\infty r_1] \oplus \mathbb{R}[r_1])[r_2] \\ \text{(G}_{\text{CASE}}\text{)} \frac{\Gamma, x_1 : \mathbb{R}[\infty r_1 + r_2]; \Omega \vdash_s 0 : \mathbb{R}[\emptyset] \quad \Gamma, x_2 : \mathbb{R}[r_1 + r_2]; \Omega \vdash_s x_2 + x_2 : \mathbb{R}[2r_1 + 2r_2]}{\Gamma; \Omega \vdash_s \text{case } z \text{ of } \{x_1 \Rightarrow 0\} \{x_2 \Rightarrow x_2 + x_2\} : \mathbb{R}[\emptyset \vee (2r_1 + 2r_2) \vee r_2]} \end{array}$$

The resulting effect is $(2r_1 + 2r_2) \vee r_2 = 2r_1 + 2r_2$. Also notice that if we now change the expression to be $\text{case } z \text{ of } \{x_1 \Rightarrow 0\} \{x_2 \Rightarrow 1\}$, where the branches are not using their bound variables, then the resulting effect is r_2 , the effect of reducing z to a value.

Rule (G_{FOLD}) allows for subtyping in its body, and rule (G_{UNFOLD}) relies on the meta-function *unf* which performs a substitution for the recursive variable and integrates the latent effect of

$$\begin{aligned}
e &::= \dots \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e \mid \text{fold}^g e \mid \text{unfold } e && (\text{expressions}) \\
&\quad \mid \text{unit} \mid \text{inl}^g e \mid \text{inr}^g e \mid \text{case } e \text{ of } \{x \Rightarrow e\} \{x \Rightarrow e\} \\
g &::= \dots \mid \text{Unit} \mid G \oplus G \mid G \times G \mid \mu\alpha.G && (\text{types}) \\
G &::= g[\Xi] \mid \alpha && (\text{type-and-effects}) \\
u &::= \dots \mid \text{unit} \mid \langle v_1, v_2 \rangle \mid \text{inl}^g v \mid \text{inr}^g v \mid \text{fold}^g v && (\text{simple values})
\end{aligned}$$

Well-typed expressions

$$\boxed{\Gamma; \Omega \vdash_s e : G}$$

$$\begin{aligned}
& \text{(GPAIR)} \quad \frac{\Gamma; \Omega \vdash e_1 : G_1 \quad \Gamma; \Omega \vdash e_2 : G_2}{\Gamma; \Omega \vdash \langle e_1, e_2 \rangle : (G_1 \times G_2)[\emptyset]} & \text{(GPROJ1)} \quad \frac{\Gamma; \Omega \vdash e : G}{\Gamma; \Omega \vdash \text{fst } e : \text{first}(G)} & \text{(GINL)} \quad \frac{\Gamma; \Omega \vdash e : G_1 \quad G_2 = g_2[\emptyset] \quad \Omega \vdash G_2}{\Gamma; \Omega \vdash \text{inl}^{g_2} e : (G_1 \oplus G_2)[\emptyset]} \\
& \text{(GCASE)} \quad \frac{\Gamma; \Omega \vdash e_1 : G_1 \quad \Gamma, x : \text{left}(G_1); \Omega \vdash e_2 : G_2 \quad \Gamma, y : \text{right}(G_1); \Omega \vdash e_3 : G_3}{\Gamma; \Omega \vdash \text{case } e_1 \text{ of } \{x \Rightarrow e_2\} \{y \Rightarrow e_3\} : G_2 \vee G_3 (\vee) \text{ eff}(G_1)} \\
& \text{(GFOLD)} \quad \frac{\Gamma; \Omega \vdash e : G' \quad G' <: \text{unf}((\mu\alpha.G)[\emptyset])}{\Gamma; \Omega \vdash \text{fold}^{\mu\alpha.G} e : (\mu\alpha.G)[\emptyset]} & \text{(GUNFOLD)} \quad \frac{\Gamma; \Omega \vdash e : G}{\Gamma; \Omega \vdash \text{unfold } e : \text{unf}(G)}
\end{aligned}$$

$$\begin{aligned}
\text{first}((G_1 \times G_2)[\Xi]) &= G_1 (\text{+}) \Xi & \text{left}((G_1 \oplus G_2)[\Xi]) &= G_1 (\text{+}) \Xi & \text{right}((G_1 \oplus G_2)[\Xi]) &= G_2 (\text{+}) \Xi \\
\text{unf}((\mu\alpha.G)[\Xi]) &= [(\mu\alpha.G)[\emptyset]/\alpha]G (\text{+}) \Xi & \text{eff}(g[\Xi]) &= \Xi
\end{aligned}$$

Fig. 8. Syntax and type system of GSoul (excerpt)

the inner expression. Recursive type substitution $[\cdot/\cdot]$ replaces a type-and-effect for a recursive variable. Notice that in the definition of unf , the substitution has an empty effect. This ensures that the effect of reducing the expression to a fold value is only accounted for once, and not at each unfolding.

Subtyping and Recursive Types. A last aspect of GSoul that we have to take care of is subtyping in the presence of recursive types. For this, we adopt the *weakly positive subtyping* approach of [Zhou et al. 2022]:⁸

$$\begin{aligned}
& \text{(GSUB-REC)} \quad \frac{G_1 <: G_2 \quad \alpha \in_+ G_1 <: G_2}{\mu\alpha.G_1 <: \mu\alpha.G_2} & \text{(GSUB-REC-REFL)} \quad \frac{G_1 \sim G_2}{\mu\alpha.G_1 <: \mu\alpha.G_2}
\end{aligned}$$

This approach relies on the polarized judgment $\alpha \in_m G_1 \widetilde{<} G_2$, which checks whether α is used positively or negatively—indicated by the m polarity—in type-and-effects G_1 and G_2 ; we write \overline{m} for flipping the polarity m . Rule (GSUB-REC) allows for subtyping in the bodies of the recursive types under the condition that α only appears in positive positions. Recursive variables can still appear in negative positions but only for plausibly equal types, described using \sim , by rule (GSUB-REC-REFL).⁹ To ensure that subtyping remains deterministic, rule (GSUB-REC-REFL) is only used

⁸We could also have used the standard Amber rules [Cardelli 1986], although as argued by Zhou et al. [2022], the metatheory of iso-recursive Amber rules is challenging; their approach is equivalent while enjoying a simpler metatheory.

⁹Plausible equality, written $G_1 \sim G_2$, also known as consistency, is equivalent to $G_1 \widetilde{<} G_2 \wedge G_2 \widetilde{<} G_1$.

Notions of reduction

$$\boxed{t \longrightarrow t}$$

$$\begin{array}{ll}
\text{(IGR-FST)} & \text{fst } (\varepsilon \langle v_1, v_2 \rangle :: G) \longrightarrow \text{ifirst}(\varepsilon) v_1 :: \text{first}(G) \\
\text{(IGR-UNFOLD)} & \text{unfold } (\varepsilon (\text{fold}^{G'} v) :: G) \longrightarrow \text{iunf}(\varepsilon) v :: \text{unf}(G) \\
\text{(IGR-CASE-1)} & \text{case}(\varepsilon_1 \text{inl}^{G_{12}} (\varepsilon_{11} u :: G'_{11}) :: G_1) \longrightarrow \begin{cases} \varepsilon_2 ([\varepsilon'_{11} u :: \text{left}(G_1)/x] t_2) :: G_2 \vee G_3 \langle \vee \rangle \text{eff}(G_1) \\ \text{error} & \text{if not defined} \end{cases} \\
& \text{of } \{ x \Rightarrow t_2^{G_2} \} \{ y \Rightarrow t_3^{G_3} \} \\
\text{[where]} & \varepsilon'_{11} = \varepsilon_{11} \circ^{<} \text{ileft}(\varepsilon_1), \quad \varepsilon_2 = \mathcal{I}_{<}.(G_2, G_2 \vee G_3) \langle \vee \rangle^2 \text{eff}^2(\varepsilon_1)
\end{array}$$

Fig. 9. Reduction rules for GSoul $_{\varepsilon}$ (excerpt)

when $\alpha \in_+ G_1 <: G_2$ is false, otherwise (G_{SUB-REC}) is used. The definition of the weakly positive restriction can be found in the supplementary material.

Dynamic semantics. The reduction rules for products, sums, and recursive types are presented in Figure 9. Evaluation contexts and values are extended with the corresponding constructs, following call-by-value semantics as before. For rules (IGR-FST) and (IGR-SND), we use inversion functions *ifirst* and *isecond* analogous to *icod*. Then, if evidence ε justifies $\varepsilon \triangleright G_1 <: G_2$, then *ifirst*(ε) \triangleright *first*(G_1) $<:$ *second*(G_2), assuming the types involved are products. For rule (IGR-CASE-1), evidence ε'_{11} , for the substituted value, is computed using the inversion function *ileft* (instead of *idom* in (IGR-APP)). Since the body terms have no explicit evidences, in order to produce an evidence ε_2 for the context term we use the interior operator. For ε_2 , to fully reflect the type operations made in the ascription, we must join the information in the sensitivity environment parts of ε_1 . This is done by using the functions eff^2 and $\langle \vee \rangle^2$, the lifted versions of *eff* and $\langle \vee \rangle$ for operating on evidences, respectively. Rule (IGR-CASE-2), corresponding to the *inr* case, is left out as its definition is analogous to (IGR-CASE-1). Finally, the reduction rule (IGR-UNFOLD) is standard, with the subtlety of the use of inversion operator *iunf* for justifying the resulting ascription.

Handling recursive functions. Let us return to the recursive function example of §2. First, we introduce rules for typechecking and reducing fixpoints; as in Fuzz, we can encode fixpoints via the Y-combinator, but in order to simplify the presentation, we here introduce a fixpoint primitive:

$$\begin{array}{ll}
\text{(IGFIX)} & \frac{t \in \mathbb{T}[G]}{\text{fix } (f : G). t \in \mathbb{T}[G]} \quad \text{(IGR-FIX)} \\
& \varepsilon (\text{fix } (f : G). t) :: G' \longrightarrow \varepsilon ([\mathcal{I}_{<}.(G, G) (\text{fix } (f : G). t) :: G/f] t) :: G'
\end{array}$$

Rules (IGFIX) and (IGR-FIX) are standard, modulo the treatment of evidence. In (IGR-FIX), we use the interior operator to justify the ascription of the substituted term, which in this case is trivial: the ascription type-and-effect is the same as the original, thus the interior operator never fails. Finally, since the body of the fixpoint has the same type-and-effect as the fixpoint itself, the original evidence ε can be used to justify the ascription of the result.

We can now define the recursive function *scale* of type-and-effect $\forall r. \text{Nat} \rightarrow \mathbb{R}[r] \rightarrow \mathbb{R}[?r]$:¹⁰

$$\lambda r. \text{fix } \text{scale}. (\lambda (n : \text{Nat}, v : \mathbb{R}[r]). \text{case } (\text{unfold } n) \text{ of } \{ _ \Rightarrow 0 \} \{ m \Rightarrow v + \text{scale } m v \})$$

Let $n = \varepsilon_n 2 :: \text{Nat}[\emptyset]$ and $v = \varepsilon_v 3 :: \mathbb{R}[r_1]$, where $\varepsilon_n = \langle \emptyset, \emptyset \rangle$ and $\varepsilon_v = \langle \emptyset, r_1 \rangle$ (omitting types in evidences to improve readability). Then if we reduce the expression *scale* $[r_1]$ $n v$, after several reduction steps we can observe that the recursive function unfolds as many times as necessary, and so each occurrence of v will simply contribute its sensitivity effect to the final result depending on the remaining operations (in this case, sums):

¹⁰Nat is encoded as $\mu \alpha. ((\text{Unit}[\emptyset] \oplus \alpha)[\emptyset])[\emptyset]$; hereafter, we use specific natural numbers instead of their encoding.

$$\begin{aligned}
(v_1, v_2) \in \mathcal{V}_\Delta^k[\text{Unit}[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[\text{Unit}[\Xi]] \\
&\quad \text{where } v_i = \varepsilon_i \text{unit} :: \text{Unit}[\Xi] \\
(v_1, v_2) \in \mathcal{V}_\Delta^k[(G_1 \rightarrow G_2)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(G_1 \rightarrow G_2)[\Xi]] \wedge \\
&\quad (\forall j \leq k, v'_1, v'_2, (v'_1, v'_2) \in \mathcal{V}_\Delta^{j-1}[G_1]. \\
&\quad (v_1 v'_1, v_2 v'_2) \in \mathcal{T}_\Delta^j[G_2 \text{ (}\vdash\text{)} \Xi]) \\
(v_1, v_2) \in \mathcal{V}_\Delta^k[(\text{Ar}.G)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(\text{Ar}.G)[\Xi]] \wedge (\forall j \leq k, \Xi'' . \\
&\quad (v_1 [\Xi''], v_2 [\Xi'']) \in \mathcal{T}_\Delta^j[(\Xi''/r)G \text{ (}\vdash\text{)} \Xi]) \\
(v_1, v_2) \in \mathcal{V}_\Delta^k[(G_1 \times G_2)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(G_1 \times G_2)[\Xi]] \wedge \\
&\quad (\text{useFst}(v_1), \text{useFst}(v_2)) \in \mathcal{T}_\Delta^k[G_1 \text{ (}\vdash\text{)} \Xi] \wedge \\
&\quad (\text{useSnd}(v_1), \text{useSnd}(v_2)) \in \mathcal{T}_\Delta^k[G_2 \text{ (}\vdash\text{)} \Xi] \\
(v_1, v_2) \in \mathcal{V}_\Delta^k[(G_1 \oplus G_2)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(G_1 \oplus G_2)[\Xi]] \wedge \\
&\quad (\Delta \cdot (\Xi'_1 \vee \Xi'_2) \prec \infty \\
&\quad \implies (\text{useL}(v_1), \text{useL}(v_2)) \in \mathcal{T}_\Delta^k[G_1 \text{ (}\vdash\text{)} \Xi] \vee \\
&\quad (\text{useR}(v_1), \text{useR}(v_2)) \in \mathcal{T}_\Delta^k[G_2 \text{ (}\vdash\text{)} \Xi]) \\
&\quad \text{where } v_i = \langle g_{i1}[\emptyset], g_{i2}[\Xi'_i] \rangle u_i :: (G_1 \oplus G_2)[\Xi] \\
(v_1, v_2) \in \mathcal{V}_\Delta^k[(\mu\alpha.G)[\Xi]] &\iff (v_1, v_2) \in \text{Atom}[(\mu\alpha.G)[\Xi]] \wedge (\forall j \leq k, \\
&\quad (\text{unfold } v_1, \text{unfold } v_2) \in \mathcal{T}_\Delta^j[(\mu\alpha.G)[\emptyset]/\alpha]G \text{ (}\vdash\text{)} \Xi]) \\
(t_1, t_2) \in \mathcal{T}_\Delta^k[G] &\iff \forall j < k. (t_1 \mapsto^j v_1 \wedge t_2 \mapsto^* v_2) \implies (v_1, v_2) \in \mathcal{V}_\Delta^{k-j}[G] \\
\\
\text{useL}(\varepsilon(\text{inl}^{G_{12}}v) :: G) &= \text{ileft}(\varepsilon)v :: \text{left}(G) & \quad \text{useR}(\varepsilon(\text{inr}^{G_{12}}v) :: G) = \text{iright}(\varepsilon)v :: \text{right}(G) \\
\text{useFst}(\varepsilon\langle v_1, v_2 \rangle :: G) &= \text{ifirst}(\varepsilon)v_1 :: \text{first}(G) & \quad \text{useSnd}(\varepsilon\langle v_1, v_2 \rangle :: G) = \text{isecond}(\varepsilon)v_2 :: \text{second}(G)
\end{aligned}$$

Fig. 10. Logical relations for gradual sensitivity soundness of GSoul_ε (excerpt)

$$\text{scale } [r_1] \ n \ v \mapsto^* \varepsilon_2(\varepsilon_v 3 :: r_1 + \varepsilon_1(\varepsilon_v 3 :: r_1 + \varepsilon_0 0 :: ?r_1) :: ?r_1) :: ?r_1 \mapsto^* \langle \emptyset, [2, \infty]r_1 \rangle 6 :: ?r_1$$

where evidences $\varepsilon_2 = \langle [1, \infty]r_1, [1, \infty]r_1 \rangle$, $\varepsilon_1 = \langle [1, \infty]r_1, [1, \infty]r_1 \rangle$ and $\varepsilon_0 = \langle \emptyset, ?r_1 \rangle$ are the results of each unfolding of the fixpoint.

The important insights of this example come from analyzing the final evidence $\langle \emptyset, [2, \infty]r_1 \rangle$. First, during reduction we have learned that resource r_1 was used at least twice. Second, notice how the sensitivity of r_1 matches exactly the value of n , namely 2. More importantly, this generalizes to any value of n : the resulting evidence will always be of the form $\langle \emptyset, [\underline{n}, \infty]r_1 \rangle$, i.e. a result that is known to be at least n -sensitive in resource r_1 . And finally, this resulting evidence means that we can use the result in other contexts, without having to consider the result as ∞ -sensitive in the resources it uses. In particular, for this example, we can successfully pass the result to a function that requires its argument to be at most 2-sensitive in v ; but passing it to a function that requires a 1-sensitive argument would produce a runtime error.

6.2 Metatheory of GSoul

GSoul satisfies all the properties of the core gradual language. More notably, it also satisfies both termination-insensitive and -sensitive gradual metric preservation, as well as their respective corollaries. One important difference between GMINI and GSoul is that the latter supports general recursion. This has a direct impact on the proof of sensitivity soundness for GSoul, because the logical relation presented in §5.1 is not well-founded in presence of recursion. To overcome this, we modify the definition of the logical relations to use *step indexing* [Ahmed 2006; Appel and McAllester 2001]. We now discuss the step-indexed logical relation, presented in Figure 10.

(A) *Related numbers, booleans and unit.* Numbers and booleans are related as defined in Figure 6, independently of the step index. Two values of the unit type are necessarily equal, and hence related for any step index.

(B) *Related functions and resource abstractions.* The definitions remain similar to the ones in §5.1, but now they are indexed by a step index k . Two functions are related at type-and-effect $(G_1 \rightarrow G_2)[\Xi]$ for k steps if, for any $j \leq k$, their applications to two related arguments for $j - 1$ steps yield related computations for j steps. Similarly, two resource abstractions are related for k steps if, for any $j \leq k$, their applications to any sensitivity environment yield related computations for j steps. Notice that for these cases, although the same step index can be used in the recursive definition (when $j = k$), the interpretations are well-founded because in each case a reduction step is always performed, lowering the index by one.

(C) *Related pairs and injections.* Two pairs are related simply if each of their components are related. Any two injections are related for any step index if the predicted (worst-case) output distance $\Delta \cdot (\Xi'_1 \curlywedge \Xi'_2)$ is infinity. If it is possible that the predicted output distance is less than infinity, then the injections must be either both left injections or right injections. Furthermore, the underlying values must be related. Also, and as in the case of booleans, in order to account for the worst case scenario, we need to take the best case on the hypothesis side of the implication as the predicate is in a contravariant position. Therefore, the consistently-less-than operator $\widetilde{\prec}$ is used directly, without the need for double negation. Finally, the partial meta-functions *useL*, *useR*, *useFst*, and *useSnd* are defined to extract the underlying values of the injections and pairs.

(D) *Related folds.* Analogous to functions, two folds are related for k steps if, for any $j \leq k$, their unfolding yield related computations at the expected type-and-effect—as computed by *unf* in (GUNFOLD)—for j steps.

(E) *Related computations.* A pair of closed terms are related for k steps whenever the first term reduces to a value in j steps (where $j < k$) and the second reduces to a value in any number of steps, and those two values are related at that type for $k - j$ steps. Notice that this definition, similar to the first logical relation presented in §5.1, is termination-insensitive. Later in this section, we will revisit the topic of termination-sensitive soundness for GSoul_ε.

We can now state the fundamental property of the logical relation for GSoul_ε, as well as gradual metric preservation for GSoul.

THEOREM 9 (FUNDAMENTAL PROPERTY FOR GSoul_ε). *Let t be a GSoul_ε term. If $t \in \mathbb{T}[G]$ and $t \vdash \Gamma$, then $\forall \Delta, k \geq 0, \gamma_1, \gamma_2$ such that $\Gamma \vdash \Delta$ and $(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta^k[\Gamma]$, we have $(\gamma_1(t), \gamma_2(t)) \in \mathcal{T}_\Delta^k[G]$.*

COROLLARY 10 (GRADUAL METRIC PRESERVATION FOR GSoul BASE TYPES). *Let f be a GSoul function. If f has type $(\mathbb{R}[r] \rightarrow \mathbb{R}[ir])[\emptyset]$, for some gradual sensitivity i , then for any real numbers c_1, c_2, c such that $|c_1 - c_2| \leq c$, if $f \ c_1 \Downarrow v_1$, and $f \ c_2 \Downarrow v_2$, then $|uval(v_1) - uval(v_2)| \leq c \cdot \pi_2(i)$.*

Revisiting termination-sensitive gradual metric preservation. Recursive types not only introduce the need for step indexing in the logical relation, but also make program divergence a possibility. This requires us to revisit what termination-sensitive gradual metric preservation means. The attentive reader may have noticed that in §5.2 we mostly discuss about *different termination behavior across runs* and not specifically about error behavior. Indeed, the changes made in that section are sufficient to also capture divergence as a possible source of different runs not having uniform termination behavior. We thus refer to non-termination as both divergence and failure.

For stating termination-sensitive gradual metric preservation for GSoul we follow the same approach as in §5.2: first, we use a second logical relation that requires evidences of related values to be identical; and second, we disallow possibly-infinite distances in the fundamental property. Again, the definitions for related values remain unchanged with respect to Figure 10, but the logical relations for terms have to be (trivially) updated to use step indexes. We can now prove termination-sensitive gradual metric preservation for GSoul.

THEOREM 11 ((TERMINATION-SENSITIVE) FUNDAMENTAL PROPERTY FOR GSoul_ε). *Let t be a GSoul_ε term. If $t \in \mathbb{T}[g[\Xi]]$, $t \vdash \Gamma$ and $\neg(\infty \lesssim \Delta \cdot \Xi)$, then $\forall \Delta, k \geq 0, \gamma_1, \gamma_2$ such that $\Gamma \vdash \Delta$ and $(\gamma_1, \gamma_2) \in \mathcal{G}_{\Delta}^k[\Gamma]$, we have $(\gamma_1(t), \gamma_2(t)) \in \mathcal{T}_{\Delta}^k[g[\Xi]]$.*

COROLLARY 12 (TERMINATION-SENSITIVE GRADUAL METRIC PRESERVATION FOR BASE TYPES). *Let f be a GSoul function. If f has type $(\mathbb{R}[r] \rightarrow \mathbb{R}[ir])[\emptyset]$, for some gradual sensitivity i , then for any real numbers c_1, c_2, c such that $|c_1 - c_2| \leq c$ and $c \cdot \pi_2(i) < \infty$, if $f c_1 \Downarrow v_1$ then $f c_2 \Downarrow v_2$ and $|uval(v_1) - uval(v_2)| \leq c \cdot \pi_2(i)$.*

All the proofs for GMini and GMini_ε are subsumed by the proofs for GSoul and GSoul_ε, which can be found in the supplementary material.

7 RELATED WORK

Sensitivity and programming languages. The first type system for reasoning about sensitivity is Fuzz [Reed and Pierce 2010], a language for differential privacy using linear types. Several variations have been studied, such as DFuzz [Gabori et al. 2013], FUZZI [Zhang et al. 2019], Adaptive Fuzz [Winograd-Cort et al. 2017]. All of these type systems measure sensitivity and also track and enforce differential privacy. Near et al. [2019] tackle differential privacy in DUET with two mutually-defined languages, one dedicated to sensitivity and one to privacy. JAZZ [Toro et al. 2023] follows the approach of DUET, and includes SAX, a sensitivity language with contextual linear types and delayed sensitivity effects. The starting point of GSoul, the static language SOUL, is very close to SAX. Whereas Fuzz-like languages track the sensitivity of program variables using linear types, Abuah et al. [2022] propose SOLO, which tracks a fixed amount of sensitive resources and avoids linear types. DDuo [Abuah et al. 2021] provides a library for sensitivity tracking in Python i.e. dynamically, being able to tackle expressiveness issues of static sensitivity type systems. However, it does not provide the ability to strengthen the static guarantees of a program, as GSoul does. μ Fuzz [D'Antoni et al. 2013] extends the Fuzz compiler to generate nonlinear constraints, which are then checked by an SMT solver, which results in an automatic type-based sensitivity analysis. There is no prior work integrating static and dynamic sensitivity analysis within one language.

Regarding termination sensitivity, Fuzz [Reed and Pierce 2010] and more recent languages [Abuah et al. 2022; Gabori et al. 2013] focus on the terminating fragment of their languages. Reed and Pierce [2010] discuss the tension between metric preservation and non-termination in their seminal paper, presenting three alternatives: weakening the definition of metric preservation; proving statically that recursive functions terminate, yielding more complex programs; or adding *fuel* to recursive functions, falling back to a default value when running out of fuel. The latter is adopted in

the implementation. μFuzz [D’Antoni et al. 2013] also supports recursive types, and termination-sensitive metric preservation is obtained by reasoning only about finite distances. DUET [Near et al. 2019] avoids divergence via terminating looping primitives. DDuo [Abuah et al. 2021] establishes termination-insensitive metric preservation, although they do not explicitly discuss this aspect.

Before discussing related work in the gradual typing area, let us mention that another line of approaches for sensitivity verification is based on program logics [Barthe et al. 2016, 2012, 2013; Sato et al. 2019]. These approaches are generally very expressive but less automatic than type systems. On the gradualization side, it seems possible to study gradualization of such program logics, for instance by following the gradual verification approach of Bader et al. [2018], further extended to a form of separation logic [Wise et al. 2020]. Extending gradual verification to account for the aforementioned logics for sensitivity would be an interesting venue for future work.

Gradual typing To the best of our knowledge, gradual typing has not been applied to sensitivity typing. It has, however, been applied in languages with one of two particularly interesting properties: languages with type-and-effect disciplines; and languages whose soundness property corresponds to a hyperproperty, such as noninterference [Goguen and Meseguer 1982] or parametricity [Reynolds 1983]. Interestingly, gradual security typing has only been explored for termination- (and error-)insensitive characterizations of noninterference [de Amorim et al. 2020; Fennell and Thiemann 2013; Toro et al. 2018]. Bañados Schwerter et al. [2014, 2016] develop a general approach to gradualize type-and-effects. Toro and Tanter [2015] extends this approach to be able to work in the context of polymorphic effects for the Scala language. However, this line of work is based on the generic type-and-effect system of Marino and Millstein [2009], which cannot directly handle sensitivities as quantities within a range.

Toro et al. [2018] discovered that the addition of mutable references in a security language yields a gradual language that does not satisfy noninterference; ad-hoc changes to address implicit flows recover noninterference, at the expense of the dynamic gradual guarantee. An interesting perspective is to study an extension of GSoul with mutable references, investigating if metric preservation and the dynamic gradual guarantee are both satisfied.

Another interesting remark is the need for sensitivity intervals in order to have a sound evidence representation in GSoul_ε . Bañados Schwerter et al. [2020] study *forward completeness* of evidence representation, which indeed enforces soundness in terms of expected modular type-based semantic invariants. Proving forward completeness for GSoul_ε is an interesting exercise for future work.

Quantitative reasoning. Finally, sensitivity analysis can be seen as one specific case of quantitative reasoning. Several related approaches have been explored such as bounded linear logic [Dal Lago and Gaboardi 2011; Girard et al. 1992], quantitative type theory [Atkey 2018], and graded modal types [Orchard et al. 2019]. These approaches abstract over the specific quantity being analyzed and the underlying accounting mechanism. This work should serve as a useful guide to study if and how to gradualize such general approaches to quantitative program reasoning.

8 CONCLUSION

We present GSoul , a gradual sensitivity calculus with support for recursive types and explicit sensitivity polymorphism, featuring both unbounded and bounded sensitivity imprecision. Gradual sensitivity typing not only allows programmers to seamlessly choose between static and dynamic checking as they see fit, it can also accommodate features, such as recursive functions, that are too conservatively handled by the static discipline.

We have presented the challenges of designing a gradual interpretation of metric preservation, highlighting how a naive lifting of its static counterpart would yield an incorrect specification.

Furthermore, we have explored the termination aspect of metric preservation in the presence of possible runtime errors and divergence.

A challenging venue of future work is the addition of support for mutable references. However, akin to gradual security typing [Toro et al. 2018], adding this feature would likely yield a language that does not satisfy metric preservation by default, and fixing it could endanger the dynamic gradual guarantee. Another possible line of future work involves the extension and application of the principles used in this work to other typing disciplines that rely on function sensitivity, such as differential privacy [Dwork and Roth 2014], as well as other quantitative type-based reasoning techniques [Atkey 2018; Orchard et al. 2019; Petricek et al. 2014].

REFERENCES

- Chiké Abuah, David Darais, and Joseph P. Near. 2022. Solo: a lightweight static analysis for differential privacy. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 150 (oct 2022), 30 pages. <https://doi.org/10.1145/3563313>
- Chike Abuah, Alex Silence, David Darais, and Joseph P. Near. 2021. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–15.
- Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the 15th European Symposium on Programming Languages and Systems (ESOP 2006) (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.), Vol. 3924. Springer-Verlag, Vienna, Austria, 69–83.
- Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and Without Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (Sept. 2017), 39:1–39:28.
- Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Transactions on Programming Languages and Systems* 23, 5 (Sept. 2001), 657–683.
- Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd ACM/IEEE Symposium on Logic in Computer Science (LICS 2018)*. ACM Press, Oxford, UK, 56–65.
- Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A Semantic Account of Metric Preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 545–556. <https://doi.org/10.1145/3009837.3009890>
- Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018) (Lecture Notes in Computer Science)*, Işıl Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer-Verlag, Los Angeles, CA, USA, 25–46.
- Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 2020. Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient. [arXiv:cs.PL/2010.14094](https://arxiv.org/abs/2010.14094)
- Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*. ACM Press, Gothenburg, Sweden, 283–295.
- 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, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. Association for Computing Machinery, New York, NY, USA, 749–758. <https://doi.org/10.1145/2933575.2934554>
- Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. Association for Computing Machinery, New York, NY, USA, 97–110. <https://doi.org/10.1145/2103656.2103670>
- Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 9 (Nov. 2013), 49 pages. <https://doi.org/10.1145/2492061>
- Olivier Bournez, Daniel Graça, and Emmanuel Hainry. 2010. Robust computations with dynamical systems. In *35th international symposium on Mathematical Foundations of Computer Science - MFCS 2010 (Lecture Notes in Computer Science)*, Petr Hliněný and Antonín Kucera (Eds.), Vol. 6281. Springer-Verlag, Brno, Czech Republic, 198–208. https://doi.org/10.1007/978-3-642-15155-2_19
- Luca Cardelli. 1986. Amber. In *Combinators and Functional Programming Languages*, Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–47.
- Swarat Chaudhuri, Sumit Gulwani, Roberto Lubliner, and Sara Navidpour. 2011. Proving Programs Robust. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations*

- of Software Engineering (ESEC/FSE '11). Association for Computing Machinery, New York, NY, USA, 102–112. <https://doi.org/10.1145/2025113.2025131>
- A. Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5 (1940), 56–68.
- Ugo Dal Lago and Marco Gaboardi. 2011. Linear dependent types and relative completeness. IEEE Computer Society Press, 133–142.
- Loris D’Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin Pierce. 2013. Sensitivity Analysis Using Type-Based Constraints. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-Specific Languages (FPCDSL '13)*. Association for Computing Machinery, New York, NY, USA, 43–50. <https://doi.org/10.1145/2505351.2505353>
- Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling Noninterference and Gradual Typing. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. Association for Computing Machinery, New York, NY, USA, 116–129. <https://doi.org/10.1145/3373718.3394778>
- 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 (Aug. 2014), 211–407. <https://doi.org/10.1561/04000000042>
- Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*. 224–239.
- Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 357–370. <https://doi.org/10.1145/2429069.2429113>
- 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>.
- David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM Press, Cambridge, MA, USA, 28–38.
- Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992), 1–66.
- J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. <https://doi.org/10.1109/SP.1982.10014>
- Michael Greenberg. 2015. Space-Efficient Manifest Contracts (POPL '15). Association for Computing Machinery, New York, NY, USA, 181–194. <https://doi.org/10.1145/2676726.2676967>
- Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL 98)*. ACM Press, San Diego, CA, USA, 365–377.
- David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Sympolic Computation* 23, 2 (June 2010), 167–189.
- Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 775–788.
- 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.
- Daniel Marino and Todd Millstein. 2009. A generic type-and-effect system. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation*. 39–50.
- Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism Through Run-Time Sealing, or, Theorems for Low, Low Prices!. In *Proceedings of the 17th European Symposium on Programming Languages and Systems (ESOP 2008) (Lecture Notes in Computer Science)*, Sophia Drossopoulou (Ed.), Vol. 4960. Springer-Verlag, Budapest, Hungary, 16–31.
- Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: An Expressive Higher-Order Language and Linear Type System for Statically Enforcing Differential Privacy. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 172 (Oct. 2019), 30 pages. <https://doi.org/10.1145/3360598>
- Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019), 110:1–110:30.
- Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. *SIGPLAN Not.* 49, 9 (aug 2014), 123–135. <https://doi.org/10.1145/2692915.2628160>
- 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 Conference on Functional Programming (ICFP 2010)*. Association for Computing Machinery, New York, NY, USA, 157–168.

- John C. Reynolds. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason (Ed.). Elsevier, 513–523.
- Tetsuya Sato, Gilles Barthe, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Approximate Span Liftings: Compositional Semantics for Relaxations of Differential Privacy. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 1–14. <https://doi.org/10.1109/LICS.2019.8785668>
- Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012) (Lecture Notes in Computer Science)*, Helmut Seidl (Ed.), Vol. 7211. Springer-Verlag, Tallinn, Estonia, 579–599.
- Jeremy Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*, 81–92.
- Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007) (Lecture Notes in Computer Science)*, Erik Ernst (Ed.). Springer-Verlag, Berlin, Germany, 2–27.
- Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. 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. 2015b. Monotonic References for Efficient Gradual Typing. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015) (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer-Verlag, London, UK, 432–456.
- Matias Toro, David Darais, Chike Abuah, Joe Near, Damián Arquez, Federico Olmedo, and Éric Tanter. 2023. Contextual Linear Types for Differential Privacy. *ACM Transactions on Programming Languages and Systems* (2023). To appear (preprint on arXiv).
- Matias 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.
- Matias Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 17:1–17:30.
- Matias Toro and Éric Tanter. 2015. Customizable Gradual Polymorphic Effects for Scala. In *Proceedings of the 30th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2015)*. ACM Press, Pittsburgh, PA, USA, 935–953.
- Matias Toro and Éric Tanter. 2017. *Gradual Union Types—Complete Definition and Proofs*. Technical Report TR/DCC-2017-1. University of Chile.
- Matias Toro and Éric Tanter. 2020. Abstracting Gradual References. *Science of Computer Programming* 197 (Oct. 2020), 1–65.
- Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A Framework for Adaptive Differential Privacy. *Proc. ACM Program. Lang.* 1, ICFP, Article 10 (Aug. 2017), 29 pages. <https://doi.org/10.1145/3110254>
- Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 228:1–228:28.
- G. Zames. 1996. Input-output feedback stability and robustness, 1959–85. *IEEE Control Systems Magazine* 16, 3 (1996), 61–66. <https://doi.org/10.1109/37.506399>
- Steve Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation. Cornell University.
- Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019. Fuzzi: A Three-Level Logic for Differential Privacy. *Proc. ACM Program. Lang.* 3, ICFP, Article 93 (July 2019), 28 pages. <https://doi.org/10.1145/3341697>
- Yaoda Zhou, Jinxu Zhao, and Bruno C. D. S. Oliveira. 2022. Revisiting Iso-Recursive Subtyping. 44, 4, Article 24 (sep 2022), 54 pages. <https://doi.org/10.1145/3549537>