Gradual Set-Theoretic Types

Victor Lanvin
ENS Paris-Saclay
victor.lanvin@ens-cachan.fr

Problem and motivation. A static type system can be an extremely powerful tool for a programmer, providing early error detection, and offering strong compile-time guarantees on the behavior of a program. However, compared to dynamic typing, static typing often comes at the expense of development speed and flexibility, as statically-typed code might be more difficult to adapt to changing requirements. Gradual typing is a recent and promising approach that tries to get the best of both worlds [16]. The idea behind this approach is to integrate an *unknown* type, usually denoted by "?", which informs the compiler that additional type checks may have to be performed at run time. Therefore, the programmer can *gradually* add type annotations to a program and controls precisely how much checking is done statically versus dynamically. Gradual typing thus allows the programmer to finely tune the distribution of dynamic and static checking over a program. However, gradualization of single expressions has more limited breadth. We argue that adding full-fledged union and intersection types to a gradual type system makes the transition between dynamic typing and static typing smoother and finer grained, giving even more control to the programmer. In particular, we are interested in developing gradual typing for the *semantic subtyping* approach [10], where types are interpreted as sets of values. In this approach union and intersection types are naturally interpreted as the corresponding set-theoretic operations, and the subtyping relation is defined as set-containment, whence the name of *set-theoretic types*. This yields an intuitive and powerful type system in which several important constructions —*eg*, branching, pattern-matching, and overloading— can be typed very precisely. Set-theoretic types, however, exacerbate the shortcomings of static typing. In particular, type reconstruction for intersection type systems is, in general, undecidable. The consequence is that programmers have to add complete type annotations for every variable, which may hinder their development speed; all the more so given that union and intersection type annotations can be syntactically heavy. Adding gradual typing to set-theoretic types may help to alleviate this issue by providing a way to relax the rigidity of certain type annotations via the addition of a touch of dynamic typing, while keeping the full power of static types for critical parts of code.

We said that adding set-theoretic types to a gradual type system makes the transition between dynamic typing and static typing smoother. This is for example the case for function parameters that are to be bound to values of basic types: in the current setting, the only way to gradualize their type is to use "?", while with union and intersection types more precise gradualizations become possible. We illustrate this fact in an ML-like language by progressively refining the following example that we borrow from Siek and Vachharajani [19].

```
let succ : Int -> Int = ...
let not : Bool -> Bool = ...

let f (condition : Bool) (x : ?) : ? =
  if condition then
    succ x
  else
    not x
```

This example cannot be typed using only simple types: the type of `x` as well as the return type of `f` change depending on the value of `condition`. However, this piece of code is perfectly valid in a gradual type system, the compiler will simply add dynamic checks to ensure that the value bound to `x` can be passed as an argument to `succ` or `not` according to the case. Moreover, it will also add checks to ensure that the value returned by `f` is used correctly. Nevertheless, there are some flaws in this piece of code. For example, it is possible to pass a value of any type as the second argument of `f` (the type system ensures that the first argument will always be a Boolean). In particular, if one applies the function `f` to (a Boolean and) a value of type `string`, then the application will always fail, independently from the value of `condition`, and despite the fact that the application is statically considered well-typed. This problem can be avoided by set-theoretic types, in particular by using the union type `Int | Bool` to type the parameter `x` of the function so as to ensure that every second argument of `f` that is neither an integer nor a Boolean will be statically rejected by the type checker. This is obtained by the following code

```
let f (condition : Bool) (x : (Int | Bool))
  : (Int | Bool) =
  if condition then
    succ ((Int) x)
  else
    not ((Bool) x)
```

In order to ensure that the applications of `succ` and `not` are both well typed we added two *type casts* that check at run-time whether the argument has the required type and raise an exception otherwise.

In this second definition of `f` we have ensured, thanks to union types, that every application of `f` has now a chance to succeed. However, this is obtained at the expense of the programmer who has now the burden to insert in the code the type-cases/type-casts necessary to ensure safety (in the sense established by Wright and Felleisen [25]). By using set-theoretic types in conjunction with gradual types, it is possible both to ensure that `f` will only be applied to booleans or integers *and* to delegate the insertion of type casts to the system. This is shown by the following piece of code that our system will compile into the previous one.

```
let f (condition : Bool) (x : (Int | Bool) & ?)
  : (Int | Bool) =
  if condition then
    succ x
  else
    not x
```

In this example, the variable `x` is of type `((Int | Bool) & ?)`, where "`&`" denotes an intersection. This indicates that `x` has both type `(Int | Bool)` *and* type `?`. Intuitively, this means that the function `f` accepts as a second argument a value of any type (which is indicated by `?`), as long as this value is also an integer *or* a Boolean. The effect of having added "`& ?`" in the type of the parameter is that the programmer is no longer required to add the explicit casts in the body of the function: the system will take care of it at compile time. The combination of a union type with

"`& ?`" to type a parameter corresponds to a programming style in which the programmer asks the system to *statically* enforce that the function will be applied only to arguments in the union type and delegates to the system the *dynamic* checking, where/if necessary, for each case in the union; and while adding explicit casts in a five-line example such as the above is quite straightforward, in general (*eg*, in thousand-line modules), it is not always so, whence the interest of having a system that adds all *and only* the casts that are necessary to ensure type safety. Finally, note that the return type of `f` is no longer gradual (as it was in the first definition), since union types allow us to define it without any loss of precision. This allows the system to statically reject all cases in which the value expected from `f` is neither an integer nor a Boolean and which, with the first definition, would be detected only at run-time.

In all the examples above the return type of the function `f` can be easily and automatically deduced and could, therefore, be omitted. But there are cases in which providing the return type of a function allows the system to deduce a better type and, thus, accept more programs. This is particularly true in conjunction with intersection types, since they allow the programmer to specify different return types for different argument configurations. Consider again the function `f` above. Since it always returns either an integer or a Boolean, we used as return type `(Int | Bool)`. By using an intersection type it is possible to give `f` a more precise type in which the return type of `f` depends on the type of `x`:

```
let f : (Bool -> (Int & ?) -> Int)
        &  (Bool -> (Bool & ?) -> Bool) =
    λcondition. λx.
     if condition then
       succ x
     else
       not x
```

This time, in the body of `f`, the variable `x` has type `(Int & ?) | (Bool & ?)`, which is equivalent to `((Int | Bool) & ?)`. Hence, the function can be defined with the same body as before, and it accepts as arguments the same values. However, the return type of `f` now directly depends on the type of `x` (more precisely, on the type of the value bound to `x`): if it is of type `Int`, then the function necessarily returns an integer (that is, if the application does not fail), and the same goes for an argument of type `Bool`.

Having a return type that depends on the type of the input is reminiscent of the typing of overloaded functions (also known as "*ad hoc* polymorphism"). This correspondence is indeed a strong one, since intersections of arrow types can be used to type overloaded functions (*eg*, see Benzaken et al. [1], Castagna et al. [3]; see also Forsythe [15] which uses a limited form of overloading known as *coherent overloading*). As a matter of fact, our function `f` is just a curried function that when applied to a Boolean argument returns an overloaded function. This can better be seen by considering type equivalences: the type we declared above for `f`

```
(Bool -> (Int & ?) -> Int) & (Bool -> (Bool & ?) -> Bool)
```

is equivalent to (*ie*, it is both a subtype and a supertype of) the type

```
Bool -> ((Int & ?) -> Int) & ((Bool & ?) -> Bool))
```

Therefore an equivalent way to define `f` would have been

```
let f (condition : Bool)
    : ((Int & ?) -> Int) & ((Bool & ?) -> Bool)) =
    λx.if condition then
         succ x
       else
         not x
```

which shows in a clear way that the application of `f` to a (necessarily Boolean) argument returns an overloaded function whose result type depends on the type of its argument: the two occurrences of "`?`" in the input types of the overloaded function indicate that, in both cases (*ie*, whatever the type of the argument is), some dynamic cast *may* be needed in the body of the function and, thus, may have to be added at compile-time.

Related work and uniqueness of the approach. Our work combines set-theoretic types with gradual typing. The part on set-theoretic types is based on the *semantic subtyping* framework, as presented by Frisch et al. [10], while for what concerns the addition of gradual typing we followed and adapted the technique based on abstract interpretation by Garcia et al. [12] called "Abstracting Gradual Typing" (AGT). However, the approach proposed by Garcia et al. [12] focuses on consistent subtyping, whereas dealing with set-theoretic types requires more precise properties —most notably for lemmas related to term substitution— hence the need for new operators and for a specific dynamic semantics.

There exist other attempts at integrating gradual typing with union and/or intersection types, but none is as general as the approach we propose, insofar as they just consider either a partial set of type connectives or limited forms thereof. Siek and Tobin-Hochstadt [17] study gradual typing for a language with type-case, union types, and recursive types. Intersection and negation types are not considered and union types are in a restricted version, since it is not possible to form the union of any two types but just types with different top-most constructors: so for instance it is not possible to union two arrows. This limitation is reflected in the type-case expression which can only check the topmost constructor of a value (*eg*, integer, product, arrow, ...) but not its type. The different cases of the type-case construction of Siek and Tobin-Hochstadt [17] are functions that, if selected, are applied to the matched value; this allows a form of *occurrence typing* [23]. Our type-case is more general than the one by Siek and Tobin-Hochstadt [17] since expressions can be checked against any type, and occurrence typing can be encoded. For the sake of simplicity we considered neither product nor recursive types (though all their theory is already developed by Frisch et al. [10]) and disregarded blame, but otherwise our work subsumes the one by Siek and Tobin-Hochstadt [17].

Jafery and Dunfield [13] present a type system that contains both refinement sums and gradual sums. Similarly to our approach, they define a gradually-typed source language and a type-directed translation to a target language that contains casts. However, their approach is very different from ours: their sums are disjoint unions in which elements are explicitly injected by a constructor; as such, their sums do not have the set-theoretic property of unions (*eg*, they are neither idempotent, nor commutative, nor satisfy usual distribution laws). Also, gradual typing is confined to sum types, since the motivation of their work is to allow the programmer to gradually add refinements that make the enforcement of exhaustive pattern matching possible, a problem that does not subsists in our work or, more generally, in languages with set-theoretic types where exhaustiveness of pattern matching is easily verifiable. Finally, Jafery and Dunfield [13] leave intersection types as future work.

Our approach also relates to a recent work by Lehmann and Tanter [14], which presents a way to combine gradual typing with full-fledged refinement types, where formulae can contain unions, intersections, and negations. They encounter problems similar to ours, most notably when trying to find a suitable Galois connection to use AGT. However, our work focuses on set-theoretic types which behave differently from refinement types, in particular when it comes to subtyping, function types, or when evaluating casts.

Results and contributions. The examples given in the first paragraph provide a brief outline of the characteristics of the system we studied. In a nutshell, our work develops a theory for gradual set-theoretic types, that is, types that besides the usual type constructors —*eg*, arrows, products, integers, ...— include a gradual "**?**" basic type and set-theoretic type connectives: union, intersection, *and* negation (in the set-theoretic type approach negations are indissociable from unions and intersections). This amounts to defining and deciding their subtyping relation, using them to type a core functional language, and defining a compilation scheme that inserts all

and only the type casts required to ensure that every non-diverging well-typed expression will either return a value or raise a cast error.

The first part of our work focuses on defining the syntax and semantics of the types we are interested in. In particular, we use abstract interpretation [8] to define the semantics of our gradual types, following a technique introduced by Garcia et al. [12]. Formally, the set STypes of *static types* and the set GTypes of *gradual types* are inductively generated by the following grammars:

$$t \in \mathrm{STypes} ::= b \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$
$$\tau \in \mathrm{GTypes} ::= \mathbf{?} \mid b \mid \tau \to \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

where b ranges over the set of base types (*eg*, `Int`, `Bool`, ...). The set STypes of static types, ranged over by s, t, ..., is formed by basic types, the type *constructor* "$\to$" for function types, type *connectives* for union, intersection and negation types, as well as $\mathbb{0}$ and $\mathbb{1}$ which denote respectively the bottom and the top type. The set GTypes of gradual types, ranged over by σ, τ, ..., is obtained by adding to the static types the unknown type "$\mathbf{?}$", which stands for the absence of type information (not to be confused with $\mathbb{0}$ or $\mathbb{1}$).

Using abstract interpretation, we define two functions γ (concretization function) and α (abstraction function) such that:

$$\gamma : \mathrm{GTypes} \to \mathscr{P}(\mathrm{STypes})$$
$$\alpha : \mathscr{P}(\mathrm{STypes}) \to \mathrm{GTypes}$$

Intuitively, the concretization function γ interprets a gradual type as the set of static types obtained by replacing every occurrence of the unknown type $\mathbf{?}$ by some static type.

The concretization function can then be used to define a conservative extension of static subtyping to gradual types. That is, given the semantic subtyping relation $\leq$ defined by Frisch et al. [10], we can define its conservative extension to gradual types $\tilde{\leq}$ as:

$$\sigma \,\tilde{\leq}\, \tau \iff \exists (s,t) \in \gamma(\sigma) \times \gamma(\tau),\ \ s \leq t$$

Based on this definition of gradual subtyping and of the concretization function, we show two of our main results:

1. For every gradual type τ, the set $(\gamma(\tau), \leq)$ is a closed sublattice of $(\mathrm{STypes}, \leq)$. In particular, every gradual type τ has a maximal and a minimal concretization (denoted respectively by $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$). Moreover, these concretizations can be obtained in linear time from the initial type τ.
2. As a corollary, gradual subtyping can be reduced in linear time to deciding subtyping on set-theoretic types, since we have the following equivalence:

$$\sigma \,\tilde{\leq}\, \tau \iff \sigma^{\Downarrow} \leq \tau^{\Uparrow}$$

Having defined subtyping for gradual set-theoretic types, the second part of our work focuses on typing applications, and in particular, on defining the domain and the result type of an application. To achieve this, we can "lift" the definitions given by Frisch et al. [10] from static types to gradual types, using the same technique as for subtyping. However, this is not immediate, since the operators presented in Frisch et al. [10] are only defined for types in disjunctive normal form, and the concretization function γ does not necessarily produces types in this form. Therefore, to overcome this problem, we defined a so-called *partial applicative concretization* $\gamma_{@}^{+} : \mathrm{GTypes} \to \mathscr{P}_f(\mathscr{P}_f(\mathrm{GTypes}))$, whose goal is to transform a gradual type into a particular disjunctive normal form represented as a finite set (the union) of finite sets (the intersections) of arrow types. Intuitively, this concretization function returns the possible interpretations of a gradual type as a function type, but in disjunctive normal form. This allows us to deduce the domain and result type of an application by lifting the definitions provided in Frisch et al. [10].

The third and last part of our work is threefold. First, we define the syntax and typing rules for a gradually-typed language that makes use of set-theoretic types. This language is a lambda-calculus with explicit function interfaces (which allow the programmer to give functions multiple types, thus providing *overloading*) and dynamic typecases (expressions that branch according to the type of the evaluation of an expression). The typing rules for this language are straightforward and use the previous definitions of gradual subtyping and of the domain and result type of an application. Second, we define the syntax, typing rules and semantics of a *cast language* (or *target language*) which is a language that incorporates dynamic typecasts. The point of these casts is to verify that a gradually-typed expression is always used in a consistent way throughout the execution of a program. In particular, any reduction that would cause the program to be stuck should return a *cast error*. Lastly, we define a compilation procedure that translates a well-typed term of the gradually-typed language to a well-typed term of the cast language. This compilation inserts all the necessary casts to ensure that the execution of the gradually-typed term will never be stuck. This is formalized by our two main results:

1. Compilation is type-preserving. That is, for every typing context Γ, if e is an expression of the gradually-typed language such that $\Gamma \vdash e : \tau$ and e compiles to e' in the cast language, then $\Gamma \vdash e' : \tau$.
2. Every well-typed term of the gradually-typed language compiles to a term that either diverges or reduces to a value or a cast error.

To illustrate the behaviour of our system on an example, consider the term $e = (\lambda^{\{\mathtt{Int}\to\mathtt{Int};(\mathbf{?}\setminus\mathtt{Int})\to\mathtt{Bool}\}} x.\ x)\ true$, which is a valid term in our gradually-typed lambda calculus. The first step is to verify that this term is well-typed. Since it is an application, we must ensure that the type of the argument is "compatible" with the type of the function, and that both sides of the application are well-typed. Determining the type of the left hand side is easy: in our language, the functions are explicitly typed; therefore, the type of the left hand side of the application is simply the intersection of the types that appear in its interface, that is, $(\mathtt{Int} \to \mathtt{Int}) \wedge ((\mathbf{?} \setminus \mathtt{Int}) \to \mathtt{Bool})$. Intuitively, this means that if the function is applied to an integer then it returns an integer, and otherwise returns a boolean. Moreover, the presence of the dynamic type $\mathbf{?}$ informs the compiler that it must add all the checks necessary to (dynamically) ensure that the function effectively returns a boolean.

For the left hand side to be well-typed, the function must verify its interface. That is, the body of the function must have type `Int` under the hypothesis that x has type `Int`, and the body must have type `Bool` under the hypothesis that x has type $\mathbf{?} \setminus \mathtt{Int}$. Both cases hold, since $\mathbf{?} \setminus \mathtt{Int}$ is a subtype of `Bool` (it is, actually, a subtype of every type).

Moreover, if we compute the domain of the function, we obtain $\mathtt{Int} \vee (\mathbf{?} \setminus \mathtt{Int})$, which is equivalent to $\mathbf{?}$. Since $\mathbf{?}$ is a supertype of every type, the function can accept any argument. Thus, we deduce that the application is well-typed.

The second step consists in compiling this application and adding the casts necessary to ensure that the program will not get stuck. There are three subterms we can cast: the left hand side of the application, the argument, or the body of the function. In this case, we already know that the left hand side is a function. Thus there is no need to cast it, but it might be the case that the type of the left hand side is unknown and that we need to ensure dynamically that it is a function. Moreover, we know that the function can accept an argument of type `Bool`. Therefore, there is no need to cast the argument either.

Now, consider the body of the function. Under the hypothesis that x has type `Int`, it is trivially true that the body is well-typed, of type `Int`. However, under the hypothesis that x has type $\mathbf{?} \setminus \mathtt{Int}$, we are not *completely sure* that the body has type `Bool`, since its type is partially unknown. Thus, we need to add a cast to ensure that, in this case, the body has type `Bool`; but this cast should not

be inserted if the argument has type `Int`. The solution is to compile the body of the function by a type-case that distinguishes the two argument configurations, that is:

$$(\lambda^{\{\texttt{Int}\to\texttt{Int}\ ;\ (?\setminus\texttt{Int})\to\texttt{Bool}\}} x.(y = x{\in}\texttt{Int})\ ?\ y : \langle\texttt{Bool}\rangle\ y)\ true$$

Finally, we can apply the semantics of the cast language to evaluate this term. By reducing the application and substituting x by $true$, we obtain:

$$(y = true{\in}\texttt{Int})\ ?\ y : \langle\texttt{Bool}\rangle\ y$$

Since $true$ does not have type `Int`, this type case reduces to

$$\langle\texttt{Bool}\rangle\ true$$

This cast obviously succeeds, and the application returns $true$, which is the expected result.

In conclusion, the main contribution of this work is the definition of the static and the dynamic semantics of a language with gradual types and set-theoretic type connectives and the proof of its soundness. In particular, we show how to lift the definition of domain and result type of an application from set-theoretic types to gradual types and likewise for the subtyping relation. We also show that deciding subtyping for gradual types can be reduced in linear time to deciding subtyping on set-theoretic types and that the same holds true for all subtyping-related decision problems needed for type inference (notably, computing domains and result types). More generally, this work not only enriches gradual type systems with unions and intersections and with the type precision that arises from their use, but also proposes and advocates a new style of programming with gradual types where union and intersection types are used by the programmer to instruct the system to perform fewer dynamic checks.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 51–63. ACM, 2003.

[2] G. Castagna. Covariance and contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers), 2015. Unpublished manuscript, available at the author's web page.

[3] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[4] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages*, POPL '14, pages 5–17. ACM, Jan. 2014.

[5] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages*, POPL '15, pages 289–302. ACM, Jan. 2015.

[6] G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, pages 378–391. ACM, Sept. 2016.

[7] A. Chaudhuri. *Flow: A static type checker for JavaScript*. Facebook, 2014. `https://flowtype.org`.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[9] A. Frisch. Regular tree language recognition with static information. In *IFIP 18th World Computer Congress TC1, 3rd International Conference on Theoretical Computer Science (TCS2004)*, volume 155 of *IFIP*, pages 661–674. Kluwer/Springer, 2004.

[10] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–64, 2008.

[11] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315. ACM, 2015.

[12] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442. ACM, 2016.

[13] K. A. Jafery and J. Dunfield. Sums of uncertainty: Refinements go gradual. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 804–817. ACM, 2017.

[14] N. Lehmann and É. Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 18–20. ACM, 2017.

[15] J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[16] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming Workshop*, pages 81–92. ACM, 2006.

[17] J. G. Siek and S. Tobin-Hochstadt. *The Recursive Union of Some Gradual Types*, pages 388–410. Springer International Publishing, 2016.

[18] J. G. Siek and S. Tobin-Hochstadt. The recursive union of some gradual types. In *A List of Successes That Can Change the World*, pages 388–410. Springer, 2016.

[19] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic languages*, page 7. ACM, 2008.

[20] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 365–376. ACM, 2010.

[21] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[22] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468. ACM, 2016.

[23] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406. ACM, 2008.

[24] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, LNCS, pages 1–16. Springer, 2009.

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