Progressive Types *

Joe Gibbs Politz
Brown University
joe@cs.brown.edu

Hannah Quay-de la Vallee
Brown University
hannahqd@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

As modern type systems grow ever-richer, it can become increasingly onerous for programmers to satisfy them. However, some programs may not require the full power of the type system, while others may wish to obtain these rich guarantees incrementally. In particular, programmers may be willing to exploit the safety checks of the underlying run-time system as a substitute for some static guarantees. Progressive types give programmers this freedom, thus creating a gentler and more flexible environment for using powerful type checkers. In this paper we discuss the idea, motivate it with concrete, real-world scenarios, then show the development of a simple progressive type system and present its (progressive) soundness theorem.

1. Introduction

Recent years have seen a series of strong results on gradual typing [20, 25, 32], which focuses on adding types to otherwise untyped languages (such as most scripting languages). Gradual typing argues that it is difficult for programmers to proceed from untyped to typed programs in a single step; instead, they should be allowed to add types incrementally, on a by-need basis. There are different styles of gradual typing—some enable the free intermingling of typed and untyped code [20], while others confine each kind to separate modules [26]—with correspondingly different guarantees on the behavior of interactions between typed and untyped code. All, however, are designed for the incremental provision of guarantees in programs.

This paper argues that there are actually two, orthogonal, kinds of incrementalism that programmers are likely to want, and explores the one not covered by gradual typing. In gradual typing, the programmer controls the extent of code covered by the type system, but the guarantee that the type system provides is fixed. In contrast, we present *progressive types*, which let programmers choose what guarantees they want from their type system. Static guarantees that programmers desire are checked by the type system; those not chosen are still enforced by the underlying safe run-time system. We argue that many type systems—including those for traditionally typed languages (e.g., a Java that provided the *choice* of allowing NullPointerExceptions)—have progressive elements and can profitably be reformulated to enable progressive typing.

To understand our choice of name, it is helpful to recall the basics of type soundness. Per Milner's formulation [12], a term that types will either not halt or will result in an answer of the right type. The Wright-Felleisen technique for proving it [30] decomposes proofs into two parts: *progress* and *preservation*. Progress says that a well-typed term that is not an answer can take a reduction step. Progress is defined by first defining *faulty expressions*, which are a conservative approximation of terms that will get stuck, and then showing that these faulty expressions are untypable. Since the typable terms must then necessarily be non-faulty, they either diverge or evaluate to values. Preservation says that the term resulting from a step taken by a well-typed term will itself be well-typed. Interleaving progress and preservation steps yields the desired soundness result.

This description hides an important caveat: exceptions, which every realistic sound type system can raise. Progress thus really means that a program is either an answer, can take a well-typed step, or will result in one of that set of exceptions. Progressive types are then a means of tuning how much progress the user wants, without giving up on preservation. Indeed, progressive types grew out of our own experiences as builders and users of various gradual type systems for real-world code.

Describing type soundness as admitting a set of exceptions presents the exceptions as a weakness; in fact, it is a strength in that *only* those exceptions can occur. For a language with a type-based semantics, the set of "exceptions that cannot occur" may not be well-defined (because there is no need to include them in the semantics). In contrast, when types are being "retrofitted" onto an existing safe run-time, it becomes easy to identify precisely which exceptions in the

* The Brown University campus is notoriously full of progressive types.

[Copyright notice will appear here once 'preprint' option is removed.]

run-time system will not be raised (and their corresponding checks can even be removed by an optimizer [22]). Indeed, the classical literature on retrofitted type systems focuses on the exceptions that *cannot* occur:

- "Our [...] aim is to provide for the undefinedness that arises from so-called don't care conditions in language specifications." [13]
- "In typeless languages [...], it is possible to encounter run-time errors such as applying lists to arguments. Can one infer types [...] to catch more errors at compile time?" [24]
- "While Smalltalk is a 'type-safe' language in the sense that [it can produce] a run-time error of the form 'message not understood', it is nevertheless advantageous for the programmer to be informed [...] when the program is being compiled" [2]

In short, every sound retrofitted type system makes a conscious and *fixed* choice about which exceptions it will catch statically and which ones can still occur at run-time. We argue that this choice should instead be delegated to the user of the type system. At one extreme, then, the programmer can choose to have no exceptions prevented statically. This reduces the "type checker" to a mere parser. This is, however, perfectly reasonable: type systems *are* syntactic disciplines (in Reynolds's phrase [18]), and the simplest is a check for adherence to a grammatical term structure. Thus progressive types lay bare the progression from parsing to rich type-checking.

2. Motivating Examples

As motivation, we present two examples where traditional progress guarantees are unnecessary or are downright annoying, and a vision for a tool built with progressive types in mind.

2.1 ADsafety: Safety without Progress

Typed JavaScript [7] is a typed version of JavaScript that handles many idiomatic patterns found in that language. In 2010-2011, two of the authors used Typed JavaScript to verify ADsafe [16], a Web sandboxing library.

The heart of ADsafe is a runtime library, `adsafe.js`. It contains wrappers for built-in browser functionality that block access to sensitive operations on the page (for example, reading the user's cookie or the content of the page outside of an advertisement). The core of our verification strategy was, roughly speaking, to annotate `adsafe.js` and type-check it in Typed JavaScript to verify that it did not leak sensitive data.

The crucial point of relevance here is that ADsafe's sandboxing properties are only dependent upon untrusted code not getting references to certain unsafe values. A runtime error is completely compatible with these goals, as a program that halts before it leaks an unsafe value is a safe program. Thus, it is sufficient for a type-based verification to be concerned only with these values and how they are used, without considering runtime errors.

Further, the runtime library of `adsafe.js`, and the untrusted programs that run against it, can actually produce runtime errors. ADsafe doesn't prevent programs from trying to look up fields on the `null` value, for instance, which signals an exception. To enforce such restrictions in a verification with a traditional type-checker, the code of `adsafe.js` would have required nontrivial refactorings that were irrelevant to the properties we were trying to prove. Instead, we elected to relax the rules in our type-checker to admit runtime errors.

Our implementation of Typed JavaScript for verifying ADsafety reflects this change, which we made by manually editing the appropriate typing rules. There are naturally many problems with this implementation strategy:

- With each new configuration, we would need to re-do our reasoning to ensure that the modified implementation matched our expectations of the allowed errors, and doesn't break soundness in some subtle way.
- The implementation doesn't check a specific set of errors. For verifying ADsafe this was not a problem, since we were permitting all runtime errors. But this does not generalize to a more nuanced setting.
- Changing which errors are allowed requires editing and recompiling the type-checker.

For ADsafe's verification, we cared about specific type-based arguments of safety that were dependent on preservation only, and completely ignored the typical progress lemma. Other projects using Typed JavaScript are similarly exploring weaker variants of the traditional, strong progress result backing Typed JavaScript. Ideally, therefore, Typed JavaScript would be a type-checker parameterized over the specific progress guarantees a user wants.

2.2 Typed Racket

Typed Racket [26] is a type system for the Racket programming language [15], a large and sophisticated descendent of Scheme. Because Typed Racket intends to type all of Racket, it has a very complex type system. Indeed, typing each of several Racket features has led to individual papers [21, 23, 28]. Each of these complex type extensions offers an irreplaceable benefit to the developer who hopes to bless the use of that feature, but can become an enormous burden to the developer who does not care about it.

For instance, consider Typed Racket's numbers [21]. Racket numbers are derived from the complex numeric hierarchy of Scheme [4], including rational and complex numbers, (in)exactitude, fixed- and floating-point, and more. Typed Racket encodes this entire hierarchy, resulting in a limited form of dependent types [21]. Thus, even though Typed Racket's + is not overloaded as in other languages—it

consumes and produces *only* numbers—the listing of its type can be dozens of lines long! In fact, reflecting Racket, there is no universal number type that a programmer can use in all numeric contexts. (There is a built-in type called `Number`, but this is an alias for the union of `Complex` and several other types, and complex numbers are not comparable (e.g., by `<`).)

As a result, we have personally experienced situations where over half the time spent converting an untyped Racket program to Typed Racket is spent on numbers alone. This can be very frustrating in programs where numeric reasoning is largely orthogonal to the point of the program. A programmer who has some means to validate—e.g., through a user interface that lies outside the type system—that "bad" numbers will not arise can painstakingly place casts in every relevant location to ask the type system to circumvent checking. But this is not modular, and anyway hardly inviting to a programmer who (at least for now) *does not care* about (say) numeric errors. In principle, every powerful extension of Typed Racket deserves its own progressive relaxation.

2.3 Progressive IDEs

We envision progressive types as a particularly powerful feature when embedded in a progressive-aware IDE. Figure 1 shows a prototype design for such an interface.

The panel on the right lists errors, and lets the programmer indicate which ones should be allowed. Inline examples give the programmer context about what to expect—if the error is checked statically, the type error is shown, and if the error is allowed at runtime, an example that shows the runtime error is shown. Buttons let programmers toggle checking for specific errors on and off, and aggregate buttons toggle checking for entire groups of errors.

The interface might extend to REPLs, too. When the programmer encounters a type error that corresponds to a progress error (top of the figure), she should be presented with a button that lets her explicitly allow the error. If she allows it, the error panel should update. Conversely, if the program evaluates to an error that isn't statically checked, the IDE presents the programmer with the option of *turning on* checks for that error.

3. Progressive Types: An Example

In this paper, we don't try to prescribe a metatheory for all progressive type systems. Instead, we demonstrate our ideas through a candidate design for a progressive type system. We choose a simple call-by-value λ-calculus with math operations, inspired by Typed Racket, which we dub λ^{τ}_{Ω}. By working through several concrete examples of typing simple programs with errors, we outline what capabilities we expect of progressive systems. Section 3.1 outlines the syntax and semantics of the language we consider, then section 3.2 provides examples, and describes a type system for the language. Section 3.3 discusses the formal guarantees we get from the system, section 3.5 shows how to type the examples, and section 4 discusses some uses of and extensions to the system.

$$
\begin{array}{rcl}
e & := & e(e) \mid c(e) \mid v \mid \mathsf{err}\text{-}\omega \\
v & := & \lambda x : (\tau ; \Omega).e \mid \mathsf{n} \\
\mathsf{n} & := & \mathbb{R} \\
c & := & \div \mid \mathit{add1} \\
\omega & := & \mathsf{app}\text{-}\mathsf{n} \mid \mathsf{app}\text{-}\mathsf{0} \mid \mathsf{div}\text{-}\mathsf{0} \mid \mathsf{div}\text{-}\lambda \mid \mathsf{add1}\text{-}\lambda \\
\Omega & := & \overline{\omega} \\
\tau, \sigma & := & \textit{types, see figure 4}
\end{array}
$$

Figure 2. Syntax of λ^{τ}_{Ω}

3.1 Syntax and Semantics

Figure 2 shows the syntax of λ^{τ}_{Ω}, the core language we consider. The language supports two numeric operators: reciprocal ($\div$) and increment ($\mathit{add1}$). The language is also equipped with first-class functions, which have an explicit type annotation τ. The language of types is presented later in figure 4; we defer their explanation until then.

Errors are expressions in their own right in λ^{τ}_{Ω}, with ω ranging over the possible *error labels*. We use Ω to range over sequences of error labels (indicated by the overline). We use angle brackets to write particular sequences; for example, $\langle \mathsf{app}\text{-}\mathsf{n}, \mathsf{div}\text{-}\mathsf{0} \rangle$ is the sequence containing the two labels app-n and div-0. These sequences have meaning in typing expressions, but no bearing on the semantics of λ^{τ}_{Ω}: The semantics only ever leads to error expressions with a single error label.

Figure 3 shows the semantics of λ^{τ}_{Ω}. Evaluation contexts E enforce left-to-right, call-by-value evaluation of expressions. Evaluation within contexts, applying abstractions via substitution, and deferring primitive operations to a δ function are standard. Type annotations τ are uninterpreted and have no effect on a procedure's evaluation.

Unlike many formal languages, λ^{τ}_{Ω} pays extreme attention to error cases. Misapplication of a number n in the procedure position results in a specific error, err-app-n, rather than getting stuck. Division by 0 or a procedure results in an error, and the errors are distinguished by their labels ω. The only case of reciprocal that succeeds is the expected one: when the numerator is a number and the denominator is a nonzero number. The *add1* operator succeeds on numbers (including zero), and has an error case for procedures.

Both operations' errors for invocation with procedures are distinct and mention the operator: this matches how the runtime of Racket (and many other untyped languages) reports error messages.

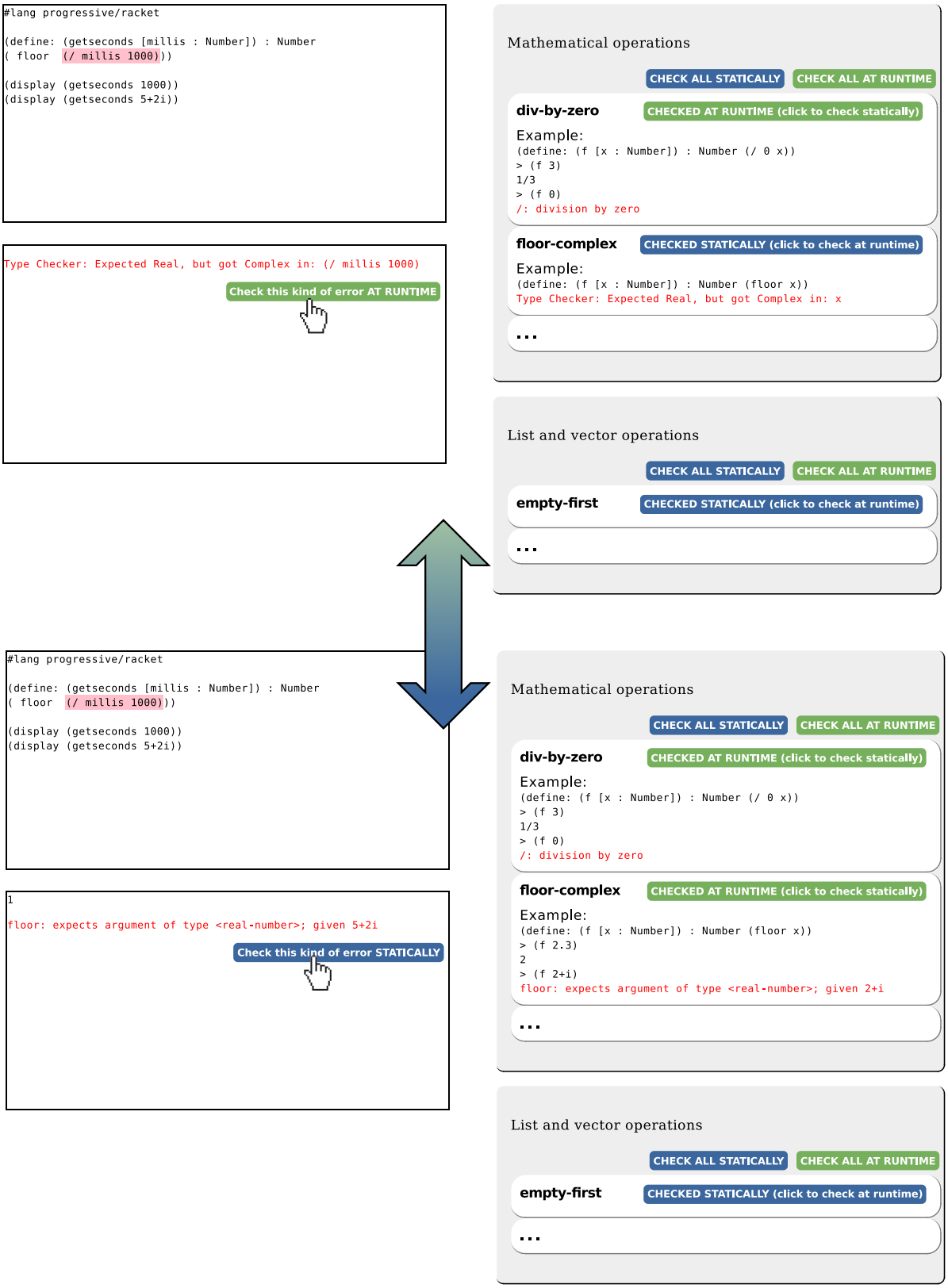

Figure 1. Design for a Progressive Type-Aware IDE. A click on the prompt button leads to the other state in either case.

$E := \bullet \mid E(e) \mid v(E) \mid c(E)$

$\boxed{e \to e}$

$E[e] \to E[e']$ when $e \Rightarrow e'$
$E[\mathsf{err}\text{-}\omega] \to \mathsf{err}\text{-}\omega$

$\boxed{e \Rightarrow e}$

(E-Apply)
$(\lambda x : (\tau; \Omega).e)(v) \Rightarrow e[x/v]$
(E-Apply-0)
$\mathsf{0}(v) \Rightarrow \mathsf{err\text{-}app\text{-}0}$
(E-Apply-Num)
$\mathsf{n}(v) \Rightarrow \mathsf{err\text{-}app\text{-}n} \mid \mathsf{n} \neq 0$
(E-Op)
$c(v) \Rightarrow \delta(c, v)$

$\boxed{c \times v \to v + \mathsf{err}\text{-}\omega}$

$$\begin{array}{lcll} \delta(\div, \mathsf{n}) & = & 1/n & \mid \mathsf{n} \neq 0 \\ \delta(\div, 0) & = & \mathsf{err\text{-}div\text{-}0} & \\ \delta(\div, \lambda x : (\tau; \Omega).e) & = & \mathsf{err\text{-}div\text{-}}\lambda & \\ & & & \\ \delta(\mathit{add1}, \mathsf{n}) & = & \mathsf{n} + 1 & \\ \delta(\mathit{add1}, \lambda x.e) & = & \mathsf{err\text{-}add1\text{-}}\lambda & \end{array}$$

Figure 3. Semantics of λ^{τ}_{Ω}

These errors cover all evaluation cases.[1] That is, λ^{τ}_{Ω} enjoys *untyped progress*: For every expression e, either:

1. e is a value v;
2. $e = \mathsf{err}\text{-}\omega$; or,
3. $e \neq \mathsf{err}\text{-}\omega$, and there exists some e' with $e \to e'$.

The proof is a straightforward case analysis of the syntax and reduction relation, and we don't present it here. Rather, we mechanize the semantics with PLT Redex [5], and use its testing tools [10] to check this.

Untyped progress is a useful model of what Racket's existing safe runtime enforces for untyped programs. Rather than having behavior that is completely undefined for the misuse of operators, it returns well-defined errors; this says quite a bit more than just "getting stuck." As we present progressive types, we will see how to reason about these error states, and what it takes to guarantee that they will not occur in well-typed programs.

[1] We require that expressions be closed—a different formulation might allow unbound identifiers to evaluate to a particular kind of error.

3.2 Typing λ^{τ}_{Ω}

In typing λ^{τ}_{Ω}, we should be able to address the entire spectrum of programs, from the fully untyped, which may evaluate to any error, to the fully typed, which can provably only evaluate to a programmer-selected subset of errors. To do this, we need our type system to:

1. ascribe meaningful types to programs that evaluate to errors, and
2. be parameterized over a selectable subset of the errors in the language.

The first criterion means typing programs such as these (where the type annotations are intentionally left unspecified for now, with the τs as placeholders):

$\div(0)$
$(\lambda x : \tau. \div (x))(0)$
$(\lambda x : \tau. \div (\mathit{add1}(x)))(-1)$
$(\lambda f : \tau_1.f(2)(4))(\lambda x : \tau_2.\mathit{add1}(x))$

The first program results in an immediate error: it divides by 0 directly, resulting in err-div-0. In the second program, 0 is substituted for x in the procedure body, and the program results in err-div-0 on the next step. The third program is similar, but has an addition that succeeds before the division by 0. In the fourth program, the procedure that increments its argument is correctly called and yields 3, but then 3 is applied to 4, resulting in err-app-n.

What types should these programs have, and what do those types mean? In the first case, we can easily prove that it will evaluate to an error. The second example is less clear. Typical compositional type-checking would have us type-check the function in isolation, using information from its annotation τ to make conclusions about its body. Separately we would have to ensure that only τ-typed values are substituted for the argument. If we choose a type that allows 0 to flow into the body, we would need to allow for err-div-0 errors. If we choose a type that excludes 0, then the application to 0 won't type-check.

The third example is more subtle. If we want to statically prove the lack of err-div-0, we'd need to reason about the behavior of $\mathit{add1}$, and the possible arguments to it that might yield 0. Most type systems don't accept this burden, and programmers live with the possibility of division-by-zero errors, guarding them with dynamic checks as best they can.

In contrast, most type systems would not give a type to the fourth program, which could (and does) result in err-app-n. It's difficult to say what it even means to give a type to this program; we'll see one possible typing in section 3.5.

Choosing Types There are myriad type systems we could imagine for type-checking λ^{τ}_{Ω}, given these examples and many others. However, our examples do highlight an important feature of the dynamic semantics of λ^{τ}_{Ω} that can inform our design of a type language. The δ-function and reduction

$$\begin{array}{rcl} \tau, \sigma & := & \mathbf{0} \mid \mathbf{N} \mid \bot \mid \tau \cup \tau \\ & \mid & \tau \xrightarrow{\Omega} \tau \\ \Gamma & := & \bullet \mid \Gamma[x : \tau] \end{array}$$

Figure 4. Types for λ^{τ}_{Ω}

relation considers values of three distinct kinds: procedures, the value 0, and nonzero numbers. It does not distinguish them further, and makes decisions based on this partitioning of values alone. We take this distinction as a starting point for designing types: our type language should be able to represent procedures, zero, nonzero reals, and any combination of them.

Figure 4 shows the complete type language for λ^{τ}_{Ω}. We use τ to range over types. The type $\mathbf{0}$ represents the number 0 and $\mathbf{N}$ represents all other numbers. A union of two types, $\tau \cup \tau$, represents the type of all values of either type. We employ a bottom type, $\bot$, to represent computations that can only terminate in an error. Type environments Γ are standard.

Arrow types, $\tau \xrightarrow{\Omega} \tau$, include an *error set*. We read a type $\tau \xrightarrow{\Omega} \tau'$ as a function from values of type τ to values of type τ', *or any of the errors in* Ω. That is, a function type includes the kinds of errors that may occur during the evaluation of values that inhabit the type.

Typing Programs Now we consider typing expressions that evaluate to different kinds of errors, and parameterizing the type system over different sets of errors. We do this in a straightforward way. We define not only type environments, Γ, as assumptions of each typing derivation, but also a set of *allowed errors*, Ω, which ranges over all the subsets of error labels in ω. The type system proves judgments of the form:

$$\Omega; \Gamma \vdash e : \tau$$

which means that e will evaluate to a value of type τ, diverge, or *terminate with an error in* Ω. The rest of this section walks through the typing judgments for λ^{τ}_{Ω}, focusing on where Ω is used to make the type system progressive.

Typing Simple Expressions Typing numeric values can never result in an error, so their typing holds for any Ω and Γ. Typing variables is standard lookup in the type environment Γ. These straightforward rules are in figure 5.

To type-check an error, we check that the error's label is in Ω. If they are typable, error expressions always have type $\bot$ (figure 6).

Typing δ When we get to less inert expressions, typing becomes interesting. In order to support progressive typing, we need to provide types for unconventional expression forms, like $\div(\lambda x(\tau; \Omega).e)$, where a procedure is used as the argument to a primitive operation. This program should be typable conditioned on whether err-div-λ is an allowed error

$$\text{T-Zero}\ \frac{}{\Omega; \Gamma \vdash 0 : \mathbf{0}} \qquad \text{T-Num}\ \frac{\mathsf{n} \neq 0}{\Omega; \Gamma \vdash \mathsf{n} : \mathbf{N}}$$

$$\text{T-Var}\ \frac{}{\Omega; \Gamma \vdash x : \Gamma(x)}$$

Figure 5. Typing Constants and Identifiers

$$\text{T-Err}\ \frac{\omega \in \Omega}{\Omega; \Gamma \vdash \mathsf{err}\text{-}\omega : \bot}$$

Figure 6. Typing Errors

$$\boxed{\delta_\tau : c \times \tau \times \Omega \to \tau}$$

$$\begin{array}{lcl} \delta_\tau(c, \bot, \Omega) & = & \bot \\ \delta_\tau(c, \tau_1 \cup \tau_2, \Omega) & = & \delta_\tau(c, \tau_1, \Omega) \cup \delta_\tau(c, \tau_2, \Omega) \\ \\ \delta_\tau(\div, \mathbf{N}, \Omega) & = & \mathbf{N} \\ \delta_\tau(\div, \mathbf{0}, \Omega) & = & \bot \text{ when } \mathsf{div\text{-}0} \in \Omega \\ \delta_\tau(\div, \tau_1 \xrightarrow{\Omega_1} \tau_2, \Omega_2) & = & \bot \text{ when } \mathsf{div}\text{-}\lambda \in \Omega_2 \\ \\ \delta_\tau(\mathit{add1}, \mathbf{N}, \Omega) & = & \mathbf{N} \cup \mathbf{0} \\ \delta_\tau(\mathit{add1}, \mathbf{0}, \Omega) & = & \mathbf{N} \\ \delta_\tau(\mathit{add1}, \tau_1 \xrightarrow{\Omega_1} \tau_2, \Omega_2) & = & \bot \text{ when } \mathsf{add}\text{-}\lambda \in \Omega_2 \end{array}$$

$$\text{T-Op}\ \frac{\Omega; \Gamma \vdash e : \tau}{\Omega; \Gamma \vdash c(e) : \delta_\tau(c, \tau, \Omega)}$$

Figure 7. Typing δ

or not, since

$$\div(\lambda x(\tau; \Omega).e) \to \mathsf{err}\text{-}\mathsf{div}\text{-}\lambda$$

Generalizing fully, we want to be able to type an application of a primitive operation to *any* value, as long as the appropriate error labels are in Ω.

This full generalization is captured in δ_τ in figure 7. To ascribe a type to a primitive operation expression, $c(e)$, we must have a type for the subexpression e, assuming the same bindings and allowed errors. Then, δ_τ takes this type and distributes it over unions, yielding $\bot$ when the type guarantees that the expression will evaluate to an error ω that is in the set of allowed errors Ω. This lets us give a type to our the first example if we choose Ω correctly:

$$\Omega = \langle \mathsf{div\text{-}0} \rangle$$

$$\delta_\tau(\div, \mathbf{0}, \langle \mathsf{div\text{-}0} \rangle) = \bot \qquad \mathsf{div\text{-}0} \in \langle \mathsf{div\text{-}0} \rangle$$

$$\frac{\overline{\langle \mathsf{div\text{-}0} \rangle ; \bullet \vdash 0 : \mathbf{0}}}{\langle \mathsf{div\text{-}0} \rangle ; \bullet \vdash \div(0) : \delta_\tau(\div, \mathbf{0}, \langle \mathsf{div\text{-}0} \rangle)}$$

If we had chosen an Ω that didn't include div-0, δ_τ would have been undefined for $\mathbf{0}$, and no typing would have been possible for this expression (using $\uparrow$ to indicate that δ_τ is undefined on its inputs):

$$\Omega = \langle \mathsf{app\text{-}0} \rangle$$

$$\delta_\tau(\div, \mathbf{0}, \langle \mathsf{app\text{-}0} \rangle) \uparrow \qquad \mathsf{div\text{-}0} \notin \langle \mathsf{app\text{-}0} \rangle$$

$$\frac{\overline{\langle \mathsf{app\text{-}0} \rangle ; \bullet \vdash 0 : \mathbf{0}}}{\langle \mathsf{app\text{-}0} \rangle ; \bullet \vdash \div(0) : \text{undefined}}$$

Procedures and Application We now need to tackle procedures and their application. The rule for procedures is in figure 8:

$$\text{T-Fun}\frac{\Omega_2; \Gamma[x : \tau_1] \vdash e : \tau_2}{\Omega_1; \Gamma \vdash \lambda x : (\tau_1; \Omega_2).e : \tau_1 \xrightarrow{\Omega_2} \tau_2}$$

Figure 8. Typing Procedures

To type-check procedures, we use the typical strategy of extending the environment, but we *replace* the Ω_1, the current set of acceptable errors, with the procedure's annotation Ω_2 to type- check its body. The annotated error set is then included in the type for the function.[2]

Type-checking application is the most subtle operation. A number of factors come into play:

1. We need to ensure that a function, when applied in a particular environment of allowed errors Ω, cannot cause errors outside of Ω.
2. We need to give types to programs where a number appears in procedure position, but only if the appropriate errors are allowed.
3. To ensure the usual correctness guarantees for fully-typed programs, we require the usual guarantee of type preservation; namely, that the type of the parameter matches the expectation annotated on the formal argument.

As with primitives, we employ a metafunction to type-check applications. The *apply* metafunction is shown in figure 9. It propagates $\bot$ as in δ_τ. If number- or zero-typed values appear in the function position, it allows them if the appropriate error is in Ω, resulting in $\bot$.

For arrow types, *apply* checks: (1) that the argument expression will evaluate to a value of the same type as the annotation, and (2) that the set of errors Ω_1 that the application can result in is contained within the errors in Ω_2 that the context expects.

Finally, *apply* distributes over unions in the procedure position. For example, take this application of *apply* to the union of a number and an arrow type:

$$\begin{array}{ll} & \mathit{apply}(\mathbf{N} \cup \mathbf{0} \xrightarrow{\langle\rangle} \mathbf{0}, \mathbf{0}, \langle \mathsf{app\text{-}n} \rangle) \\ = & \mathit{apply}(\mathbf{N}, \mathbf{0}, \langle \mathsf{app\text{-}n} \rangle) \cup \mathit{apply}(\mathbf{0} \xrightarrow{\langle\rangle} \mathbf{0}, \mathbf{0}, \langle \mathsf{app\text{-}n} \rangle) \\ = & \bot \cup \mathbf{0} \end{array}$$

Note that the function type, $\mathbf{0} \xrightarrow{\langle\rangle} \mathbf{0}$, doesn't have to have the error label app-n: only the context in which the application is being performed does. It is sufficient that $\langle\rangle \in \langle \mathsf{app\text{-}n} \rangle$: the function's application causes no errors, which is allowed in a context that accepts app-n errors.

Subtyping We introduced union types, but no introduction mechanism for them. Further, we cannot type some programs that should intuitively type-check. For example, we cannot handle

$$(\lambda x : ((\mathbf{0} \xrightarrow{\langle\rangle} \mathbf{0}) \cup \mathbf{N}; \langle\rangle).x)(5)$$

because we lack a mechanism for expressing that 5 is substitutable for not just $\mathbf{N}$, but for any union of types that includes $\mathbf{N}$. To remedy this, we introduce a standard subsumption

$$\boxed{\mathit{apply} : \tau \times \tau \times \Omega \to \tau}$$

$$\begin{array}{lcll} \mathit{apply}(\bot, \tau_2, \Omega) & = & \bot & \\ \mathit{apply}(\tau_1, \bot, \Omega) & = & \bot & \\ \\ \mathit{apply}(\mathbf{0}, \tau', \Omega) & = & \bot & \text{when } \mathsf{app\text{-}0} \in \Omega \\ \mathit{apply}(\mathbf{N}, \tau', \Omega) & = & \bot & \text{when } \mathsf{app\text{-}n} \in \Omega \\ \\ \mathit{apply}(\tau_1 \xrightarrow{\Omega_1} \tau_2, \tau', \Omega_2) & = & \tau_2 & \text{when } \tau' = \tau_1 \\ & & & \text{and } \Omega_1 \subseteq \Omega_2 \\ \\ \mathit{apply}(\tau_1 \cup \tau_2, \tau', \Omega) & = & \mathit{apply}(\tau_1, \tau', \Omega) & \\ & \cup & \mathit{apply}(\tau_2, \tau', \Omega) & \end{array}$$

$$\text{T-App}\frac{\Omega; \Gamma \vdash e_1 : \tau_1 \qquad \Omega; \Gamma \vdash e_2 : \tau_2}{\Omega; \Gamma \vdash e_1(e_2) : \mathit{apply}(\tau_1, \tau_2, \Omega)}$$

Figure 9. Typing Application

[2] If all procedures were not already annotated with error sets, we could imagine nested procedures closing over the ambient error set. This would allow one procedure, perhaps acting as a module, to dictate the allowed errors for all the procedures defined within it.

$$\boxed{\tau \leqslant \tau}$$

$$\text{S-Bottom}\ \frac{}{\bot \leqslant \tau} \qquad \text{S-Refl}\ \frac{}{\tau \leqslant \tau}$$

$$\text{S-Union-L}\ \frac{\tau \leqslant \tau_1}{\tau \leqslant \tau_1 \cup \tau_2} \qquad \text{S-Union-R}\ \frac{\tau \leqslant \tau_2}{\tau \leqslant \tau_1 \cup \tau_2}$$

$$\text{S-Union-Join}\ \frac{\tau_1 \leqslant \tau \qquad \tau_2 \leqslant \tau}{\tau_1 \cup \tau_2 \leqslant \tau}$$

$$\text{S-Arrow}\ \frac{\tau_3 \leqslant \tau_1 \qquad \Omega_1 \subseteq \Omega_2 \qquad \tau_2 \leqslant \tau_4}{\tau_1 \xrightarrow{\Omega_1} \tau_2 \leqslant \tau_3 \xrightarrow{\Omega_2} \tau_4}$$

$$\text{T-Sub}\ \frac{\Omega; \Gamma \vdash e : \tau_1 \qquad \tau_1 \leqslant \tau_2}{\Omega; \Gamma \vdash e : \tau_2}$$

Figure 10. Subtyping λ_{math}

rule, and subtyping rules for determining when a value of one type is substitutable for another (figure 10). The notable addition to the subtyping relation is the *covariance* of the error position in S-Arrow. That is, a procedure that produces *fewer* errors is substitutable for one that produces more.

3.3 Soundness

Our soundness theorem differs from the traditional one by taking progressivity into account:

Progress: If $\Omega; \Gamma \vdash e : \tau$ then

1. $e \in v$, or
2. $e = \mathsf{err}\text{-}\omega$ where $\omega \in \Omega$, or
3. $e \neq \mathsf{err}\text{-}\omega$, and there exists e' such that $e \to e'$.

This proof has a similar structure to untyped soundness from section 3.1, but it ensures that the error is one of the allowed errors.

We also prove a more classical preservation theorem:

Preservation: If $\Omega; \Gamma \vdash e : \tau$ and $e \to e'$ then $\Omega; \Gamma \vdash e' : \tau$.

We include proofs of these lemmas in the supplemental materials at `http://www.cs.brown.edu/research/plt/dl/progressive-types/`.

3.4 Discussion

Some of the features of λ^{τ}_{Ω} and its associated theory warrant highlighting.

Type "Errors" Informally, there are exactly two reasons why a λ^{τ}_{Ω} program might not have a type. In an untypable program, it is always the case that the type system can't prove one of:

1. All errors the program may evaluate to are in the Ω declared for type-checking.
2. In all applications, the type of the argument is compatible with the left-hand side of each arrow type (if any) in the type of the procedure.

The first is a *progress* error: the program might fail to make progress in a way the programmer didn't intend. It arises in *apply*, δ_{τ}, and T-Err when the appropriate error isn't present for a case that goes to $\bot$, and also in *apply* when the error set of the procedure's annotation is checked against the current context's error set. The second is a *preservation* error: the program might compute a result that isn't what the programmer intended. This occurs only in *apply*, in the case of matching an arrow type with its argument's type. Our progressive type system for λ^{τ}_{Ω} allows us to relax any and all progress errors, with the assumption that preservation remains intact.

The goal of progressive types is to provide control over different kinds of terminating errors. It cannot relax preservation—which, if violated, results in a program that computes nonsense from the type system's point of view. Thus, in order to provide these guarantees, the progressive type-checker may still report a *type error* when configured to allow all runtime errors.

Delayed Evaluation of Errors T-Fun (figure 8) allows the annotation on a procedure to express the intended errors that should be allowed to occur when the function is applied. The type checker is not concerned about the relationship between the outer allowed errors and the errors allowed in the body of the function at the time of the procedure's definition. There are two points of interest here:

1. The static declaration of runtime errors is similar to the `throws` clause in Java, which enumerates the exceptions a method might throw. Progressive types make this manifest for *all* the errors in the system. This is in contrast to Java, which has errors that avoid the need for annotation, like `NullPointerException`.
2. A procedure that might cause a particular error when applied can be successfully type-checked in a context that does not allow that error. This makes sense for functions that may be exported across module boundaries into more lenient contexts.

3.5 Type-checking the Examples

We return to the examples from the beginning of section 3.2. We have already seen how we can type-check the simple division by zero, so we start with the slightly more complicated second example, which needs an annotation:

$$(\lambda x : (\tau?; \Omega?). \div (x))(0)$$

To type-check the function, it needs to allow div-0. The smallest type that we can use for τ is just $\mathbf{0}$. With those types:

$$\frac{\langle \mathsf{div\text{-}0} \rangle \,;\bullet \vdash 0 : \mathbf{0} \qquad \langle \mathsf{div\text{-}0} \rangle \,;\bullet \vdash (\lambda x : (\mathbf{0}; \langle \mathsf{div\text{-}0} \rangle).\div(x)) : \mathbf{0} \xrightarrow{\langle \mathsf{div\text{-}0} \rangle} \bot \qquad apply(\mathbf{0} \xrightarrow{\langle \mathsf{div\text{-}0} \rangle} \bot, \mathbf{0}, \langle \mathsf{div\text{-}0} \rangle) = \bot}{\langle \mathsf{div\text{-}0} \rangle \,;\bullet \vdash (\lambda x : (\mathbf{0}; \langle \mathsf{div\text{-}0} \rangle).\div(x))(0) : \bot}$$

The third example also needs annotations:

$$(\lambda x : (\tau?; \Omega?).\div(add1(x)))(-1)$$

The number type **N** is required for τ, and div-0 is required to type-check the body of the function. To recur into the body, the proof proceeds as in the last example, but type-checking the body requires two uses of δ_τ:

$$\frac{\dfrac{[x : \mathbf{N}](x) = \mathbf{N} \qquad \delta_\tau(add1, \mathbf{N}, \langle \mathsf{div\text{-}0} \rangle) = \mathbf{N} \cup \mathbf{0}}{\langle \mathsf{div\text{-}0} \rangle \,; [x : \mathbf{N}] \vdash add1(x) : \mathbf{N} \cup \mathbf{0}} \qquad \delta_\tau(\div, \mathbf{N} \cup \mathbf{0}, \langle \mathsf{div\text{-}0} \rangle) = \mathbf{N} \cup \bot}{\langle \mathsf{div\text{-}0} \rangle \,; [x : \mathbf{N}] \vdash \div(add1(x)) : \mathbf{N} \cup \bot}$$

The use of *add1* results in a union $\mathbf{N} \cup \mathbf{0}$, since the type system can't prove anything more specific. The second use of δ_τ distributes over this union, yielding **N** for the left branch, and $\bot$ for the right branch, thanks to the presence of div-0 in the error set passed to δ_τ. If we use subsumption, we can get a final type of **N**, and the typing judgment can be read as a single statement: This program will evaluate to a number, not terminate, or result in a div-0 error.

Next, we consider the final example:

$$(\lambda f : (\tau_1; \Omega_1).f(2)(4))(\lambda x : (\tau_2; \Omega_2).\div(add1(x)))$$

One assignment that makes this program typecheck is:

$$\begin{array}{rcl} \tau_1 & \equiv & \mathbf{N} \xrightarrow{\langle \mathsf{div\text{-}0} \rangle} \mathbf{N} \\ \Omega_1 & \equiv & \langle \mathsf{app\text{-}n}, \mathsf{div\text{-}0} \rangle \\ \tau_2 & \equiv & \mathbf{N} \\ \Omega_2 & \equiv & \langle \mathsf{div\text{-}0} \rangle \end{array}$$

This example shows that the function with parameter f needs to allow the div-0 error, because it will invoke the other function in its dynamic extent. It needs to include the app-n error because it applies the result of the application of $f(2)$, which is a number.

These examples show typing a program that immediately evaluates to an error, a program that abstracts procedures over errors, a program that abstracts the typing of the δ function over a union of error-causing and acceptable values, and a program that misapplies a number as a function. They demonstrate the flexibility that progressive typing provides, and the strategy used to maintain the guarantees of staying within a declared set of errors.

4. Uses and Extensions

Having worked through the development of a progressively-typed language, we now discuss the relationship of progressive typing to programming and other typing strategies.

4.1 Unityping

Scott pointed out that languages like Scheme can be implicitly typed in terms of a single type that encompasses all their expressions [19]. The "unityped" argument implies that every step of execution goes from a well-(uni-)typed state to another. All the stuck states (errors) are simply more instances of unityped expressions. This strategy has been used before in mixing typed and untyped code by Matthews and Findler, who mix a Scheme-like language and an ML-like language [9]. They do so by using unityping rules to ascribe **TST** (the Scheme type) to all of the Scheme expressions in the computation.

To illustrate how this is useful, we outline some of the properties of a slightly richer, almost-unitype for λ^τ_Ω. If we add equirecursive types to λ^τ_Ω, we can express a useful, broad type:

$$U = \mu x.\mathbf{0} \cup \mathbf{N} \cup (x \xrightarrow{\Omega^*} x)$$

where Ω^* is the set of all possible errors. Note that U is not quite the type of *every* value in λ^τ_Ω. For example, the value

$$\lambda x : \mathbf{0}.x$$

does not type to U: the contravariance of arrow subtyping (S-Arrow) precludes using subsumption to go from the calculated type $\mathbf{0} \to \mathbf{0}$ to U. The unityping property of U is a bit more subtle:

Unityping Completeness: Let Ω^* be the set of all errors. For all expressions e, let e' be e with all instances of $(\lambda x : (\tau; \Omega).e'')$ replaced with $(\lambda x : (U; \Omega^*).e'')$. Then $\Omega^*; \cdot \vdash e' : U$.

That is, every program doesn't necessarily type to U, but if we replace all the type annotations with U, the program will type-check to U. If all we want is the statement of unityping, this replacement of type annotations with the unitype is a simple inference strategy indeed!

We obtain confidence in the above claim via specification and automation in PLT Redex [5]. Our Redex model extends λ^τ_Ω with equirecursive types, and defines a procedure that replaces all type annotations with U and all error annotations with Ω^*.[3] Then, for one million randomly generated expressions, we test that replacing their annotations in this way yields an expression that type-checks, and satisfies the single-step progress and preservation lemmas above. This experiment required that we produce an algorithm for subtyping, which we did by applying a subtyping check at

[3] In our implementation, the unitype-annotation procedure is affectionately named `harperize`.

procedure applications. We do not know if our implementation is complete with respect to the declarative presentation; providing such an algorithm is valuable future work.

A similar "inference," with analogous restrictions for preservation, was extremely useful in our verification of ADsafe. First, we showed via testing that untrusted widgets are effectively U-typed. Then, we annotated ADsafe's runtime library, where security-sensitive checks are implemented, with much more precise types. The runtime of an ADsafe computation is an untrusted widget composed with calls into the runtime library, which involves passing U-typed values into ADsafe's reference monitor.

One subtle point of the verification is that untrusted widgets can provide ADsafe with callbacks for event handlers, and ADsafe will pass wrapped values into the callbacks. In order to securely use the untrusted callbacks, ADsafe needs to avoid passing non-U typed values into these callback functions. While the type-checker will allow nearly any operation on U it will prevent the application of a U-typed value to a value that is not a subtype of U. This guarantee helped us establish ADsafety, and it was merely a statement of preservation, not one of traditional progress.

Our progressive presentation of λ^{τ}_{Ω} lets us clearly understand what tradeoff the unityping strategy is making. It is trading away all guarantees about progress in exchange for the ability to say something meaningful about types being preserved, even if the types are quite coarse.

4.2 If-Splitting

Type systems for dynamic languages now feature some form of *if-splitting* [7, 26, 27], which was proposed by Reynolds [17], and has been implemented in several systems [6, 31]. This allows the type system to refine types in the branches of a conditional, based on static knowledge of the runtime type test information. It allows programmers in Typed Racket and Typed JavaScript to split apart unions of types with tag tests, and use the pieces of the union in different branches. For example, Typed Racket can statically type the following type-safe program:

```
(define: (get-seconds [millis : Number]) : Number
  (if (not (real? millis))
      (error "get-seconds: millis was complex")
      (floor (/ millis 1000))))
```

This works because Number is really a union of several numeric types, including Complex. Typed Racket's if-splitting allows the type-checker to refine the type of millis in the else branch, instead of yielding a type error.

In this example, the conditional inside get-seconds enables its author to provide a useful error message. A seemingly better solution, however, would be for get-seconds to take only non-Complex types in the first place. However, this would potentially require changing the callers of get-seconds, possibly by having *them* include if-splits, until the input source of the value that reaches get-millis is reached. Somewhere, the program needs to either allow the error of taking the floor of a complex number; if-split, and provide alternate functionality; or else prove (or assert) that the input cannot yield complex numbers.

If-splitting is a useful tool in this chain of refactoring and reasoning, allowing the programmer to choose where they provide alternate functionality. However, they ultimately force these decisions, which may be a distraction, onto the programmer if they wish for the type-checker to succeed. In contrast, progressive types enable the programmer to eschew this style by explicitly making a conscious choice to allow particular errors rather than check for them. Since Typed Racket is retrofitted onto a system that performs the runtime safety checks already, the programmer is not sacrificing any (dynamic) safety. The provision of choice therefore gives programmers a wide range of options for bringing code into the typed world.

4.3 Intersection and Dependent Types

The operators $\div$ and $add1$ in λ^{τ}_{Ω} are first-order: they can't be passed as values and used in any context other than primitive operation expressions. In other presentations, such as for Typed Racket [27], primitives are first-class values with rich arrow types, and Racket's numeric primitive have a weak form of dependent types through intersection types [21]. The primitives of λ^{τ}_{Ω} could be expressed with similar rich types, leveraging intersections. For example, $\div$, which has different behavior for all three different kinds of type, could be expressed as (again with Ω^* as the set of all errors):

$$\begin{array}{rcl} \div & : & \mathbf{N} \xrightarrow{\langle\rangle} \mathbf{N} \\ & \cap & \mathbf{0} \xrightarrow{\langle \mathsf{div\text{-}0} \rangle} \bot \\ & \cap & \left(\bot \xrightarrow{\Omega^*} \top\right) \xrightarrow{\langle \mathsf{div}\text{-}\lambda \rangle} \bot \end{array}$$

That is, for a **N**-typed argument, it yields a number with no errors; for a **0**-typed argument, it doesn't return and causes a div-0 error; and for function arguments, it doesn't return and causes a div-λ error. This formulation is reminiscient of the latent predicates of Hochstadt [27], and the cases that only lead to errors could even potentially be used to if-split on computations that are guaranteed to terminate in an error.

This formulation is useful because it allows a function that causes an error only on particular inputs to be used safely in other contexts. Without some form of dependent types, the only type in our current system for $\div$ as a value would require adding a $\top$ type and defining:

$$\div : \left(\mathbf{N} \cup \mathbf{0} \cup \bot \xrightarrow{\Omega^*} \top\right) \xrightarrow{\langle \mathsf{div\text{-}0}, \mathsf{div}\text{-}\lambda \rangle} \mathbf{N}$$

This is too broad to type-check safe uses like $\div(5)$ in an error environment that doesn't expect div-0 and div-λ. We are therefore interested in pursuing the applications of intersection types more fully.

4.4 Generalizing Progressive Types

We have presented progressive types strictly with respect to runtime failures. Our extensions to traditional systems are the addition of error contexts Ω, and their use as "switches" within metafunctions used by the typing rules. Because our calculus has only a single reduction rule for each error, this actually means that adding or removing an error corresponds to allowing or disallowing a single case of the reduction relation. For example, we can state:

If

- app-0 $\notin \Omega$,
- $\Omega; \cdot \vdash e : \tau$, and
- $e \twoheadrightarrow e'$,

then (E-Apply-0) was not used in $e \twoheadrightarrow e'$. This is a simple corollary of preservation and an inspection of the reduction relation.

In Typed JavaScript, we statically prevent certain operations that are not runtime errors. For example, subtracting strings in JavaScript doesn't lead to an error, but rather to IEEE NaN:

`"subtract" - "a string"` $\rightarrow$ NaN

We disallow string subtraction because we believe it is unlikely that the programmer intends it. This runs contrary to the progressive philosophy: Simply because it is unlikely doesn't mean that it should be disallowed by fiat. This is quite a specific step in the reduction relation, and we have just seen how to disable specific steps that correspond to errors. It would be useful to have a similar toggle for matters of judgment such as this.

We could make the toggle switch be the string subtraction case of the δ function. This would suggest that each reduction rule (and case of the δ function) get its own switch, and place in the extended "type environment." We could also, in the style of declaring allowed errors, declare allowed *values*, and declare that no expression should evaluate to NaN. In JavaScript, this would cross-cut a number of rules, but might actually be a useful summary of what the programmer would like! (For instance, the production of the value Undefined is often indicative of the program taking an undefined step.)

5. Related Work

Modern gradual type systems prevent the program from executing until the portion being typed has passed all checks. In contrast, the work on *soft typing* [31] was built on the principle that the type system may not be rich enough to type a particular program. As a result, when a program fails to type, a soft type checker inserts checks corresponding to the unproven primitive applications. The user is then free to execute the program, secure in the knowledge that any primitive not proven safe statically will be checked dynamically (but others need not be [8]). Progressive typing shares with soft typing the philosophy that programmers should be free to explore program behavior not only with types but also through dynamic execution.

Progressive typing differs from soft typing in at least two ways. First, only certain *kinds* of run-time checks are permitted to remain in the progressively-typed program, so the developer has a guarantee that all other kinds of checks have been checked statically and can be eliminated. Second, while soft typing is based on the idea that the type system may be too *weak* (and hence does not offer means to weaken it further), progressive typing is based on the idea that the type system may be too *strong* (and hence too onerous).

Like types [32] focus on the values a program computes with, rather than the eventual result of the program. When a programmer provides a like-annotation of type τ for an identifier, she is allowing that values that are not of type τ may be bound to that identifier. If the program misuses the like-typed value, a runtime error results. Thus, a like-typed program delays the error for as long as possible (until the program would be producing nonsensical results). This is a relaxation of both preservation and progress, in that more runtime errors are possible, and some identifiers may be substituted for incorrect values. The additional errors can only occur on like-typed variables, and in exchange, more programs are like-typable. In contrast, progressive types only relax progress, and give guarantees about the result of the computation rather than which values might be used incorrectly along the way.

Pluggable type systems [3] lay out a set of desiderata for optional type systems. It argues, among other things, that "types should be optional", and that there should be "multiple type system [sic] for different needs." Progressive types are not completely optional; our formalism requires an annotation burden to obtain the guarantees of preservation once any errors are to be avoided. However, a progressive type system is designed explicitly to allow the instantiation of a type system for many different needs in static error detection.

Pluggable type systems have been studied in detail for Java in work on the Checker Framework [14], and JavaCOP [11]. These systems allow type system designers to mix and match various static checking features on top of the same underlying checker. The Checker Framework, in fact, builds up a non-null type system for Java that offers the choice we suggested in section 1. Each additional feature in a pluggable system may either admit more programs, or reject more. Progressive types are geared towards making the type system more lenient one error at a time; however, it isn't always clear which type system is the right one to use to prevent a particular error. For example, there are many choices of type systems that prove the lack of divide-by-zero errors at runtime, with varying levels of utility (the type checker that rejects programs containing $\div$ outright is of dubious merit in many contexts). Thus, different type systems may

be appropriate for different choices of error sets. Progressive types could guide the choice of which type system features need to be "plugged in" in these cases.

The Glasgow Haskell compiler has recently seen the addition of type errors that can be deferred until runtime [29]. DuctileJ does much the same for Java [1]. Instead of the compiler signalling an error at type-checking time, these systems defer the proof failure that the type checker observes until runtime, using casts that do runtime type-checking to yield a runtime failure. This shares the progressive philosophy that sometimes the type system may be "too strong," and it would be best to weaken the type checker rather than have it be a barrier to running code. Progressive types takes a dual approach to the design of the weakening. The related systems start with a static typing discipline in mind; our sample progressive system draws its design from the underlying untyped language. This focus on the underlying failure cases in the language makes progressive typing suitable for retrofitting, and suggests a design for new type systems that need this flexibility.

Our semantics of errors is based heavily on the presentation in *Semantics Engineering with PLT Redex* [5, Chapter 7]. That work doesn't present a type system to accompany the errors, but clearly lays out error labels' interaction with evaluation contexts.

Acknowledgments

We thank Arjun Guha and Ben Lerner for collaborating on various progressive variants of Typed JavaScript, and for numerous useful discussions about this work. We are also grateful for exchanges with Ravi Chugh. Asumu Takikawa, Sam Tobin-Hochstadt, and Matthias Felleisen pointed out important deficiencies in our presentation of related work. Caitlin Santone provided her graphic design skills to help create figure 1. This work is partially supported by the US NSF and by Google; standard disclaimers apply.

References

[1] M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 521–530, Waikiki, Hawaii, USA, May 25–27, 2011.

[2] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for smalltalk. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1982.

[3] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[4] S. Egner, R. A. Kelsey, and M. Sperber. Cleaning up the Tower: Numbers in Scheme. 2004.

[5] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[6] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Programming Languages Design and Implementation (PLDI)*, 1996.

[7] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, 2011.

[8] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *Functional programming languages and computer architecture*, 1995.

[9] Jacob Matthews and Robert Bruce Findler. Operational Semantics for Multi-Language Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[10] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[11] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. JavaCOP: Declarative Pluggable Types for Java. *ACM Transactions on Programming Languages and Systems*, 32, 2010.

[12] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[14] M. M. Papi, M. Ali, T. L. C. Jr., J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *International Symposium on Software Testing and Analysis*, 2008.

[15] PLT. The Racket Programming Lanugage. `http://racket-lang.org`.

[16] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[17] J. C. Reynolds. Automatic computation of data-set definitions. In *IFIP Congress*, 1968.

[18] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Information Processing*, 83:513–523, 1983.

[19] D. Scott. Lambda calculus: Some models, some philosophy. In *The Kleene Symposium*, pages 223–265, 1980.

[20] J. G. Siek and W. Taha. Gradual typing for functional languages. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2006.

[21] V. St-Amour and S. Tobin-Hochstadt. Typing the Numeric Tower. In *Practical Aspects of Declarative Languages*, 2011.

[22] P. Steenkiste. The Implementation of Tags and Run-Time Type Checking. *Topics in Advanced Language Implementation*, pages 3–24.

[23] T. S. Strickland, S. Tobin-Hochstadt, and M. Felleisen. Practical Variable-Arity Polymorphism. In *European Symposium on Programming*, 2009.

[24] N. Suzuki. Inferring Types in Smalltalk. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1981.

[25] S. Tobin-Hochstadt and M. Felleisen. Interlanguage Migration: From Scripts to Programs. In *ACM SIGPLAN Dynamic Languages Symposium*, 2006.

[26] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 395–406, 2008.

[27] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

[28] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Programming Languages Design and Implementation*, 2011.

[29] D. Vytiniotis, S. P. Jones, and J. P. M. aes. Equality proofs and deferred type errors, A compiler pearl. In *International Conference on Functional Programming (ICFP)*, 2012.

[30] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.

[31] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[32] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integration of Typed and Untyped Code in a Scripting Language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2010.