Really Natural Linear Indexed Type Checking

Arthur Azevedo de Amorim
University of Pennsylvania

Marco Gaboardi
University of Dundee

Emilio Jesús Gallego Arias
University of Pennsylvania

Justin Hsu
University of Pennsylvania

arXiv:1503.04522v1 [cs.LO] 16 Mar 2015

Abstract

Recent works have shown the power of *linear indexed type systems* for enforcing complex program properties. These systems combine *linear types* with a language of *type-level indices*, allowing more fine-grained analyses. Such systems have been fruitfully applied in diverse domains, including implicit complexity and differential privacy.

A natural way to enhance the expressiveness of this approach is by allowing the indices to depend on runtime information, in the spirit of dependent types. This approach is used in *DFuzz*, a language for differential privacy. The *DFuzz* type system relies on an index language supporting real and natural number arithmetic over constants and variables. Moreover, *DFuzz* uses a subtyping mechanism to make types more flexible. By themselves, linearity, dependency, and subtyping each require delicate handling when performing type checking or type inference; their combination increases this challenge substantially, as the features can interact in non-trivial ways.

In this paper, we study the type-checking problem for *DFuzz*. We show how we can reduce type checking for (a simple extension of) *DFuzz* to constraint solving over a first-order theory of naturals and real numbers which, although undecidable, can often be handled in practice by standard numeric solvers.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Type structure

Keywords type checking, type inference, linear types, subtyping, sensitivity analysis

1. Introduction

Linear indexed type systems have been used to ensure safety properties of programs with respect to different kinds of resources; examples include usage analysis [24, 25], implicit complexity [4, 5, 14], sensitivity analysis [10, 23], automatic timing analysis [12, 13], and more. Linear indexed types use a type-level *index language* to describe resources and *linear types* to reason about the program's resource usage in a compositional way.

One limitation of such systems is that resource usage is inferred independently of the control flow of a program—e.g. the typing rule for branching usually approximates resources by taking the maximal usage of one of the branches, and recursion imposes even greater restrictions. To improve this scenario, some authors have proposed extending such systems with dependent types, using type indices to capture both resource usage and the *size information* of a program's input. This significantly enriches the resulting analysis by allowing resource usage to depend on runtime information. Linear dependently typed systems have been used in several domains, including implicit complexity [4, 16] and sensitivity analysis [10].

Of course, there is a price to be paid for the increase in expressiveness: type checking and type inference become inevitably more complex. In linear indexed type systems, these tasks are often done in two stages: a standard Hindley-Milner-like pass, followed by a constraint-solving procedure. In some cases, the generated constraints can be solved automatically by using custom algorithms [17] or off-the-shelf SMT solvers [7, 13]. However, the constraints are specific to the index language, and richer index languages often lead to more complex constraints.

Type-checking *DFuzz*

In this paper we will focus on the type-checking problem for a particular programming language with linear dependent types: *DFuzz* [10], a dependently-typed extension of the *Fuzz* programming language [23].

Fuzz uses linear indexed types to reason about programs in the context of differential privacy. Its indices are real numbers that provide upper bounds on the *sensitivity* of a program, a quantity that measures the distance between outputs on nearby inputs. In this setting, type checking and inference result in a simple but effective static analysis for function sensitivity. Indeed, as shown by D'Antoni et al. [7], both of these can be performed efficiently by using an SMT solver to discharge the numeric proof obligations arising from the type system.

While *Fuzz* works well on a variety of simple programs, it has a fundamental limitation: sensitivity information cannot depend on runtime information, such as the size of a data structure. This is what *DFuzz* is designed to handle. *DFuzz* indices combine information about the *size* of data structures with information about the *sensitivity* of functions. Technically, this is achieved by considering an index language with variables ranging over integers (to refer to runtime sizes) and reals (to refer to runtime sensitivities). This richer index language, combined with dependent pattern-

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

matching and subtyping, achieves increased expressiveness in the analysis, providing sensitivity bounds beyond *Fuzz*'s capabilities.

However, adding variables to the index language has a significant impact on the difficulty of type checking. Concretely, since the index language also supports addition and multiplication, index terms are now *polynomials* over the index variables. Instead of constraints between real constants like in *Fuzz*, type checking constraints in *DFuzz* may involve general polynomials.

A natural first approach is to try to extend the algorithm proposed by D'Antoni et al. [7] to work with the new index language by simply generating additional constraints when dealing with the new language constructs. This would be similar in spirit to the work of Dal Lago et al. [6] for type inference for dℓPCF, a linear dependently typed system for complexity analysis. A crucial difference between that setting and *DFuzz* is that the index language of dℓPCF can be extended by arbitrary (computable) functions. This makes the approach to type inference for dℓPCF proposed by Dal Lago and Petit the most natural, since such functions can be used as direct solutions to some of the introduced constraints.

However, such an approach does not work as well for *DFuzz*, which opts for a much smaller index language. While it may be possible to extend *DFuzz*'s index language with general functions, we opt to keep the index language simple. Instead, since the type system of *DFuzz* also supports subtyping, we consider a different approach inspired by techniques from the literature on subtyping [21] and on constraint-based type-inference approaches [15, 19, 22].

The main idea is to type-check a program by inferring some set of sensitivities for it, and then testing whether the resulting type is a subtype of the desired type. To obtain completeness (relative to checking the subtype), one must ensure that the inferred sensitivities are the "best" possible for that term. Unfortunately, the *DFuzz* index language is not rich enough for expressing such sensitivities. For instance, some cases require taking the maximum of two sensitivity expressions, something that cannot be done in the language of polynomials. We solve this problem by extending the index language with three syntactic constructs, resulting in a new type system that we name *EDFuzz*. This new system has meta-theoretic properties that are similar to those of *DFuzz*, but also simplifies the search for minimal sensitivities. Using these new constructs, we design a sensitivity-inference algorithm for *EDFuzz* which we show sound and complete, modulo constraint resolution.

We now face the problem of solving the constraints generated by our algorithm. First, we show how to compile the constraints generated by the algorithmic systems to constraints in the first-order theory over mixed integers and reals. This way, we can still use a numeric solver without resorting to custom symbolic resolution. Unfortunately, the presence of natural numbers in the constraints has important consequences: we show that *DFuzz* type-checking is undecidable by reducing from Hilbert's tenth problem, a standard undecidable problem.

While this result shows that we can't have a terminating type-checker that is both sound and complete, not everything is lost. We first show that by approximating the constraints, we obtain a sound and computable method to type-check *EDFuzz* programs. We show that this procedure can successfully type-check a fragment of *EDFuzz* which we call *UDFuzz*; almost all of the examples proposed by Gaboardi et al. [10] belong to this class. Of course, *UDFuzz* is a strict subset of *EDFuzz*, and it is not hard to come up with well-typed programs in *EDFuzz* that are invalid under *UDFuzz*.

Finally, we present a constraint simplification procedure that can significantly reduce the complexity of our translated constraints (measured by the number of alternating quantifiers), even when checking full *EDFuzz*.

Contributions

We briefly overview the *DFuzz* programming language in Section 2, to move to an informal exposition of the main challenges involved in Section 3. Then, we present the main contributions of the paper:

- *EDFuzz*: an extension of *DFuzz* with a more expressive sensitivity language that gives programs more precise types (Section 4);
- a sound and complete algorithm that reduces type checking and sensitivity inference in *EDFuzz* to constraint solving over the first-order theory of $\mathbb{N}$ and $\mathbb{R}$ (Section 5 and Section 6);
- a proof of undecidability of type checking in *DFuzz* (and *EDFuzz*) (Section 7);
- a sound translation from the previous type-checking constraints to the first-order theory of the real numbers, a decidable theory (Section 8.1); and
- a simplification procedure to make the constraints more amenable to automatic solving (Section 8.2).

Additionally, we have developed a prototype implementation of the above, which we discuss in Section 9.

2. The *DFuzz* System

DFuzz [10] is a language for writing and verifying differentially private programs. At its core lies a type system for tracking function *sensitivity*:

Definition 1. *Given two metric spaces X, Y, the sensitivity (or Lipschitz constant) of a function $f : X \to Y$ is a number k such that $d_Y(f(x), f(x')) \leq k d_X(x, x')$ for all $x, x' \in X$. In this case, we say that f is k-sensitive (or k-Lipschitz continuous).*

The precise relationship between differential privacy and function sensitivity is beyond the scope of this paper; we refer the reader to previous work [10, 23] for more information. What is important for present purposes is that *DFuzz* uses a linear dependently-typed system for analyzing function sensitivity. Let us begin with a brief presentation of *DFuzz* before discussing the type-checking challenges.

2.1 Syntax and Types

DFuzz is an extension of PCF with dependent indexed linear types. Indices consist of numeric constants; index-level variables, which range over *sizes* (natural numbers) or *sensitivities* (positive reals extended with ∞, denoted $\mathbb{S}$); and addition and multiplication of indices. The syntax of *DFuzz*, including types, terms, and the index language, is shown in Figure 1, which we briefly overview. Here, we omit some features of the original system to keep our presentation simple.

- Abstraction and application for index variables are captured by the $\Lambda i : \kappa.e$ and $e[R]$ terms, with κ representing the kind for i. We refer to variables of kind n as *size variables*, while variables of kind r are *sensitivity variables*.
- Singleton types $\mathbb{N}[S]$ and $\mathbb{R}[R]$ are used to related type-level sizes and sensitivities with term-level sizes and sensitivities.
- Dependent pattern matching over $\mathbb{N}[S]$ types is captured by the **case** construction.
- Linear functions indexed by R are written $!_R\sigma \multimap \tau$.
- Variable environments Γ carry an additional annotation for assignments $x :_{[R]} \sigma$, representing the current sensitivity R for the variable x.
- Index variable environments ϕ specify the kinding of index variables.

$$\begin{array}{lcll}
\kappa & ::= & \mathrm{r} \mid \mathrm{n} & \text{(kinds)}\\
\mathbb{S} & ::= & \mathbb{R}^{\geq 0}\cup\{\infty\} & \text{(extended positive reals)}\\
S & ::= & i \mid 0 \mid S+1 & \text{(sizes)}\\
R & ::= & \mathbb{S} \mid i \mid S \mid R+R \mid R\cdot R & \text{(sensitivities)}\\
\sigma,\tau & ::= & \mathbb{R} \mid \mathbb{R}[R] \mid \mathbb{N}[S] \mid !_R\sigma \multimap \tau & \text{(types)}\\
 & \mid & \forall i:\kappa.\ \sigma \mid \sigma\otimes\tau \mid \sigma \,\&\, \tau & \\
e & ::= & x \mid \mathbb{N} \mid \mathbf{s}\ e \mid \mathbb{R}^{\geq 0} \mid \mathbf{fix}\ (x:\sigma).e & \text{(expressions)}\\
 & \mid & \lambda x :_{[R]} \sigma.e \mid e_1\ e_2 & \\
 & \mid & \Lambda i:\kappa.\ e \mid e[R] & \\
 & \mid & \langle e_1,e_2\rangle \mid \pi_i\ e & \\
 & \mid & (e_1,e_2) \mid \mathbf{let}\ (x,y)=e\ \mathbf{in}\ e' & \\
 & \mid & \mathbf{case}\, e\, \mathbf{of}\ 0 \Rightarrow e_0 \mid n_{[i]}+1 \Rightarrow e_s & \\
\Gamma,\Delta & ::= & \emptyset \mid \Gamma, x:_{[R]}\sigma & \text{(environments)}\\
\phi,\psi & ::= & \emptyset \mid \phi, i:\kappa & \text{(sens. environments)}\\
\Phi,\Psi & ::= & \top \mid \Phi, S=0 \mid \Phi, S=i+1 & \text{(constraints)}
\end{array}$$

Figure 1. *DFuzz* Types and Expressions

- Constraint environments Φ store assumptions introduced under dependent pattern matching. Often, we will think of a constraint environment as the conjunction of its constraints.

2.2 Environment Operations

As in many similar systems, *DFuzz* defines operations on variable environments. Specifically, we can add two environments Γ, Δ, and scale a single environment Γ can by a sensitivity expression R. We define environment multiplication $R \cdot \Gamma$ as the operation taking every element $x_i :_{[r_i]} \sigma_i$ of Γ to $x_i :_{[R\cdot r_i]} \sigma_i$. Environment addition is defined iff all the common assignments of Γ, Δ map to the same type, that is to say, forall x_i in $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$, $(x_i :_{[R_i]} \sigma_i) \in \Gamma \iff (x_i :_{[S_i]} \sigma_i) \in \Delta$, where we write $\mathrm{dom}(\Gamma)$ for the domain of an environment. In this case:

$$\begin{array}{rcll}
\Gamma+\Delta & = & \{x_i :_{[R_i+S_i]} \sigma & \mid x_i \in \mathrm{dom}(\Gamma)\cap\mathrm{dom}(\Delta)\}\\
 & \cup & \{x_j :_{[R_j]} \sigma_j & \mid x_j \in \mathrm{dom}(\Gamma)-\mathrm{dom}(\Delta)\}\\
 & \cup & \{x_k :_{[R_k]} \sigma_k & \mid x_k \in \mathrm{dom}(\Delta)-\mathrm{dom}(\Gamma)\}
\end{array}$$

2.3 Subtyping

DFuzz has a notion of subtyping, which intuitively corresponds to a standard property of function sensitivity: a k-sensitive function is also k'-sensitive for all $k' \geq k$. Furthermore, subtyping in *DFuzz* is the mechanism that allows types to use information from the constraint environment; in this use, subtyping allows a form of type coercion. We consider here a slightly simpler definition of subtyping than the one used in Gaboardi et al. [10]. In the environments we require subtyping to preserve the internal type. This slight modification will allow us to simplify some rules of the type-checking algorithm.

The semantics of the subtying relation is defined by interpreting sensitivity expressions as functions that produce sensitivity values. Formally, let R be a sensitivity expression, well-typed under environment ϕ, and ρ a suitable variable *valuation* (i.e., a function that maps each variable $i : \kappa$ in ϕ to an element of $[\![\kappa]\!]$, with $[\![\mathrm{n}]\!] = \mathbb{N}$ and $[\![\mathrm{r}]\!] = \mathbb{S}$). We then define $[\![R]\!]_\rho$ as follows:

$$\begin{array}{rcll}
[\![0]\!]_\rho & := & 0 & \\
[\![S+1]\!]_\rho & := & [\![S]\!]_\rho + 1 & \\
[\![i]\!]_\rho & := & \rho(i) & i \text{ a variable}\\
[\![r]\!]_\rho & := & r & r \text{ a constant}\\
[\![R_1+R_2]\!]_\rho & := & [\![R_1]\!]_\rho + [\![R_2]\!]_\rho & \\
[\![R_1\cdot R_2]\!]_\rho & := & [\![R_1]\!]_\rho \cdot [\![R_2]\!]_\rho &
\end{array}$$

Then, the standard ordering $\geq$ on $\mathbb{S}$ induces an ordering on index terms, which we can then extend to a subtype relation $\sqsubseteq$ on types and environments; the rules can be found in Figure 2. Note that checking happens under the current constraint environment Φ, so subtyping may use information recovered from a dependent match.

$$\frac{}{\phi;\Phi \models \sigma \sqsubseteq \sigma}\ \sqsubseteq\text{-Refl}$$

$$\frac{\phi;\Phi \models \sigma' \sqsubseteq \sigma \qquad \phi;\Phi \models \tau \sqsubseteq \tau'}{\phi;\Phi \models \sigma \,\&\, \tau \sqsubseteq \sigma' \,\&\, \tau'}\ (\sqsubseteq . \&)$$

$$\frac{\phi;\Phi \models \sigma \sqsubseteq \sigma' \qquad \phi;\Phi \models \tau \sqsubseteq \tau'}{\phi;\Phi \models \sigma \otimes \tau \sqsubseteq \sigma' \otimes \tau'}\ (\sqsubseteq . \otimes)$$

$$\frac{\begin{array}{c}\phi;\Phi \models R \leq R'\\ \phi;\Phi \models \sigma' \sqsubseteq \sigma \qquad \phi;\Phi \models \tau \sqsubseteq \tau'\end{array}}{\phi;\Phi \models\ !_R\sigma \multimap \tau \sqsubseteq\ !_{R'}\sigma' \multimap \tau'}\ (\sqsubseteq . \multimap)$$

$$\frac{\phi, i:\kappa;\Phi \models \sigma \sqsubseteq \tau \qquad i \text{ fresh in } \phi}{\phi;\Phi \models \forall i:\kappa.\ \sigma \sqsubseteq \forall i:\kappa.\ \tau}\ (\sqsubseteq . \forall)$$

$$\frac{\forall (x :_{[R']} \sigma) \in \Delta, \exists R, (x :_{[R]} \sigma) \in \Gamma \wedge \big(\phi;\Phi \models R_i \geq R'_i\big)}{\phi;\Phi \models \Gamma \sqsubseteq \Delta}\ \sqsubseteq\text{-Env}$$

Figure 2. *DFuzz* Subtyping Relation

The leaves of the subtype derivation are assertions $\phi;\Phi \models R_1 \geq R_2$. These are defined logically as

$$\forall \rho \in \mathsf{val}(\phi).[\![\Phi]\!]_\rho \Rightarrow [\![R_1]\!]_\rho \geq [\![R_2]\!]_\rho,$$

where $\mathsf{val}(\phi)$ is the set of all valid valuations for environment ϕ, and $[\![\Phi]\!]_\rho$ is the conjunction of the denotations of each formula in Φ, defined the usual way.

2.4 Typing

Typing judgments for *DFuzz* are of the form

$$\phi;\Phi \mid \Gamma \vdash e : \sigma$$

meaning that term e has type σ under environments ϕ and Γ and constraints Φ; full rules are shown in Figure 3.

We highlight here just the most complex rule, the dependent pattern-matching rule $(\mathbb{N}\ E)$, which allows each branch to be typed under different assumptions on the type $\mathbb{N}[S]$ of the scrutinee (e). The left branch e_0 is typed under the assumption $S = 0$, while the right branch e_s is typed under the assumption $S = i + 1$ for some i. Combined with the rule for fixpoints (Fix), this allows us to express programs whose sensitivity depends on the number of iterations or number of input elements. These rules also require implicitly that all sensitivity (and size) expressions be well-typed under the appropriate environments, which we note $\phi \vdash R$. Readers interested in more details can consult Gaboardi et al. [10]; we follow their presentation closely except for a few points, which we detail in the Appendix.

2.5 Examples

We close the overview of *DFuzz* with some examples, to give an idea of the increase in expressiveness brought by dependent types. We take the liberty of including some features that were not introduced before to make the examples more interesting.

We begin by considering multiplication of a real number by a natural number. Without dependent types, the best type we can assign to multiplication is $!_\infty\mathbb{N} \multimap\ !_\infty\mathbb{R} \multimap \mathbb{R}$, which is not very informative. However, thanks to dependent types we can introduce

$$\frac{\phi;\Phi \mid \Delta \vdash e : \sigma \qquad \phi;\Phi \models \Gamma \sqsubseteq \Delta}{\phi;\Phi \mid \Gamma \vdash e : \sigma}\ (\sqsubseteq.\mathrm{L}) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \sigma \qquad \phi;\Phi \models \sigma \sqsubseteq \tau}{\phi;\Phi \mid \Gamma \vdash e : \tau}\ (\sqsubseteq.\mathrm{R})$$

$$\frac{r \in \mathbb{R}}{\phi;\Phi \mid \Gamma \vdash r : \mathbb{R}}\ (\mathrm{Const}_{\mathbb{R}}) \qquad \frac{n = [\![S]\!]}{\phi;\Phi \mid \Gamma \vdash n : \mathbb{N}[S]}\ (\mathrm{Const}_{\mathbb{N}})$$

$$\frac{}{\phi;\Phi \mid \Gamma, x :_{[1]} \sigma \vdash x : \sigma}\ (\mathrm{Var}) \qquad \frac{\phi;\Phi \mid \Gamma, x :_{[\infty]} \sigma \vdash e : \sigma}{\phi;\Phi \mid \infty \cdot \Gamma \vdash \mathbf{fix}\ (x : \sigma).e : \sigma}\ (\mathrm{Fix})$$

$$\frac{\phi;\Phi \mid \Gamma, x :_{[R]} \sigma \vdash e : \tau}{\phi;\Phi \mid \Gamma \vdash \lambda x :_{[R]} \sigma.e : !_R\sigma \multimap \tau}\ (\multimap I) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e_1 : !_R\sigma \multimap \tau \qquad \phi;\Phi \mid \Delta \vdash e_2 : \sigma}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau}\ (\multimap E)$$

$$\frac{\phi, i : \kappa;\Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi, \Gamma}{\phi;\Phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma}\ (\forall I) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi;\Phi \mid \Gamma \vdash e[S] : \sigma[S/i]}\ (\forall E)$$

$$\frac{\phi;\Phi \mid \Gamma_1 \vdash e_1 : \sigma \qquad \phi;\Phi \mid \Gamma_2 \vdash e_2 : \tau}{\phi;\Phi \mid \Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \sigma \otimes \tau}\ (\otimes I) \qquad \frac{\phi;\Phi \mid \Delta \vdash e : \sigma \otimes \tau \qquad \phi;\Phi \mid \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \vdash e' : \mu}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e' : \mu}\ (\otimes E)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi;\Phi \mid \Gamma \vdash e_2 : \tau}{\phi;\Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \mathbin{\&} \tau}\ (\&\ I) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \sigma_1 \mathbin{\&} \sigma_2}{\phi;\Phi \mid \Gamma \vdash \pi_i\ e : \sigma_i}\ (\&\ E)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e : \mathbb{N}[S]}{\phi;\Phi \mid \Gamma \vdash \mathbf{s}\, e : \mathbb{N}[S+1]}\ (\mathrm{S}\ I) \qquad \frac{\begin{array}{c}\phi;\Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi;\Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n};\Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \text{ fresh in } \phi\end{array}}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\ 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma}\ (\mathbb{N}\ E)$$

Figure 3. *DFuzz* Typing Rules

a scaling primitive with the following type:

$$\times : \forall i : \mathrm{n}.\ !_\infty\mathbb{N}[i] \multimap\ !_i\mathbb{R} \multimap \mathbb{R}$$

By partially applying this operator, we obtain a scaling function with the appropriate sensitivity, e.g.

$$(3 \times -) : !_3\mathbb{R} \multimap \mathbb{R}.$$

DFuzz uses probability distributions for differential privacy. The type system includes a primitive for adding noise drawn from the Laplace distribution to its input, with the following type:

$$\mathsf{add\_noise} : \forall \epsilon : \mathrm{r}.!_\epsilon\mathbb{R} \multimap \bigcirc\mathbb{R}$$

where $\bigcirc\mathbb{R}$ is the type of probability distributions over $\mathbb{R}$. Here, ϵ is a parameter for controlling the amount of added noise. This noise determines how "far apart" the resulting distributions will be; as it turns out, given the distance function used for probability distributions in *DFuzz*, this results in an ϵ-sensitive function.

Finally (and more interestingly), the standard map function on lists is given the following type in *DFuzz*:

$$\mathsf{map} : \forall i\, R\, \sigma\, \tau.!_i(!_R\sigma \multimap \tau) \multimap\ !_R\mathsf{list}(\sigma)[i] \multimap \mathsf{list}(\tau)[i]$$

Here, $\mathsf{list}(\sigma)[i]$ is the type of lists of elements of some type σ with length equal to i. Because we have length-indexed lists, we can correctly track the sensitivity of map on its function argument, which is precisely the length of its list argument. *Fuzz*, in contrast, would require us to replace i by ∞.

3. The Challenge of Type-checking Linear Dependent Types

Type-checking a language with linear indexed types presents several challenges, which are only compounded when dependent types and subtyping are added to the mix. In this section, we take a closer look at these challenges.

3.1 To Split, or not to Split?

The first problem we face is due to linearity. Given a term and an environment, we need a way to "split" the environment into appropriate subenvironments that can be used in the recursive calls to type-check subterms.

Automatically inferring the right environments in our setting is difficult, due to the index language for *DFuzz*. Indeed, index terms are polynomials over index variables, which may range over the reals or the naturals. For instance, we may know that a particular variable x has sensitivity $i^2 \cdot j^2 + 3$ in our environment. However, it is not clear how to split such sensitivity information between two environments that share the variable x. In fact, as we will show below, in general it is not always possible to find a split. One might hope to simplify the type-checking task by requiring the programmer to provide a few type annotations, like in non-linear type systems. Unfortunately, this approach is impractical for the splitting problem because the annotations must describe the split for every variable binding in the environment!

To better understand this obstacle, let us consider two general approaches to type-checking linear type systems, which we call the *top-down* and *bottom-up* strategies.

The Downfall of Top-Down

For the type-checking problem, suppose we are given the environment Γ, a term e, and a purported type σ. The goal is to decide if $\Gamma \vdash e : \sigma$ is derivable. The *top-down* strategy takes an environment and a term, and attempts to partition the environment and recursively type the subterms of e.

The main difficulty of this approach centers around splitting the environment, a problem that is most clear in the application rule. Here is a simplified version:

$$\frac{\Gamma \vdash f : !_R\sigma \multimap \tau \qquad \Delta \vdash e : \sigma}{\Gamma + R \cdot \Delta \vdash f\ e : \tau}$$

So given a type-checking problem $\Sigma \vdash f\ e : \sigma'$ our first difficulty is to pick R, Γ, and Δ such that $\Sigma = \Gamma + R \cdot \Delta$. We could try to guess R, but unfortunately it may depend on the choice of Γ. Since our index language contains the real numbers, the number of possible splittings isn't even finite.

A natural idea is to delay the choice of this split. For instance, we may create a placeholder variable R and placeholder environments Γ', Δ', asserting $\Sigma = \Gamma' + R \cdot \Delta'$ and recursively type-checking f and e. After reaching the leaves of the derivation, we would have a set of constraints whose satisfiability would imply that the program type-checks.

Unfortunately, the constraints seem difficult to solve due to the syntactical nature of our indices. In other words, the "placeholder variables" are really *meta-variables* that range over index terms, which could potentially depend on bound index variables. In order to prove soundness of such a system with respect to the formal typing system, the solver must return success *only* if there is a solution where all the meta-variables can be instantiated to an index term—a syntactic object. This is at odds with the way most solvers work—*semantically*—finding arbitrary solutions over their domain. It is not clear how to solve these existential constraints automatically for the specific index language of *DFuzz*.

The Rise of Bottom-Up?

A different approach is a *bottom-up* strategy: suppose we are again given an environment Γ, a term e, and a type σ, and we want to check if $\Gamma \vdash e : \sigma$ is derivable. The main idea is to avoid splitting environments by calculating the minimal sensitivities needed for typing each subexpression. For each typing rule, these minimal sensitivities can be combined to find the resulting minimal sensitivities for e. Once this is done, we just need to check whether these optimal sensitivities are compatible with Γ and σ via subtyping.

Let's consider how this works in more detail by analyzing a few important cases. At the base case, we type-check variables in a minimal environment (that is, empty but for the variable) by assigning it the minimal sensitivity required:

$$\frac{}{x :_{[1]} \sigma \vdash x : \sigma}$$

Recall that we have weakening on the left so can add non-occurring variables to the environment later.

Now, the key benefit of the bottom-up approach becomes evident in the application rule: we can completely avoid the splitting problem. When faced with a type-checking instance $\Sigma \vdash f\ e : \sigma$, we recursively find optimal Γ, R, and Δ for checking f and e; then, checking that $\Sigma \sqsubseteq \Gamma + R \cdot \Delta$ suffices.

Unfortunately, things don't look so easy in the additive rules. Let's examine the introduction rule for $\&$:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \ \&\ \sigma_2}$$

This rule forces both environments to have the same sensitivities, but the bottom-up idea may infer different environments for each expression:

$$\frac{\Gamma_1 \vdash e_1 : \sigma_1 \qquad \Gamma_2 \vdash e_2 : \sigma_2}{\Sigma? \vdash \langle e_1, e_2 \rangle : \sigma_1 \ \&\ \sigma_2}$$

Now we need to guess a best environment $\Sigma?$, but the *DFuzz* sensitivity language is too weak to express this value. For instance, if we consider sensitivity expressions r^2 and r depending on a sensitivity variable r, we can show that there is no minimal polynomial upper bound for them under the point-wise order on polynomials[1].

To maintain the minimality invariant, we can extend the sensitivity language with a new syntactic construct $\mathbf{max}(R_1, R_2)$ for sensitivity-inference purposes only, which should denote the maximum of two sensitivity values. We could then safely set $\Sigma? := \mathbf{max}(\Gamma_1, \Gamma_2)$, where the expression combines sensitivities for the bindings on both environments as expected.

However, there is a problem with this approach: the resulting algorithm is not sound with respect to the original type system, because it allows more terms to be typed even when sensitivities in the final type do not mention the new construct! To see this, assume that our algorithm produces a derivation $\Gamma' \vdash e : \sigma'$ using extended sensitivities. Now, soundness amounts to showing that for all Γ, σ mentioning only standard sensitivities such that $\Gamma \sqsubseteq \Gamma'$ and $\sigma' \sqsubseteq \sigma$, there exists a typing derivation $\Gamma \vdash e : \sigma$ that uses only the original sensitivity language. Let's try to sketch how this proof would work by restricting our attention to a particular instance of the application rule:

$$\frac{\phi; \emptyset \mid \emptyset \vdash f : !_{R_f}\sigma \multimap \tau \qquad \phi; \emptyset \mid x :_{[\hat{R}_x]} \mu \vdash e : \sigma}{\phi; \emptyset \mid x :_{[R_f \cdot \hat{R}_x]} \mu \vdash f\ e : \tau}$$

where $\hat{R}_x$ is an extended sensitivity expression. By induction, we know that for all standard sensitivity expressions R_x such that $R_x \geq \hat{R}_x$, we can obtain a standard derivation $x :_{[R_x]} \mu \vdash e : \sigma$. We also have standard R_{xf} such that $R_{xf} \geq R_f \cdot \hat{R}_x$. Thus, all we need to do is to calculate from R_f, R_{xf} standard sensitivities R'_f, R'_x to be able to apply both induction hypotheses. The following result shows that this is not always possible.

Lemma 2. *Given standard sensitivities expressions R_{xf}, R_f and an extended sensitivity expression $\hat{R}_x$ such that $R_{xf} \geq R_f \cdot \hat{R}_x$, it is not the case that one can always find standard R'_f, R'_x such that $R_{xf} \geq R'_f \cdot R'_x \wedge R'_f \geq R_f \wedge R'_x \geq \hat{R}_x$.*

Proof. Take $R_{xf} = r^2 + 1$, $R_f = r$ and $\hat{R}_x = \mathbf{max}(2, r)$. As we can see, we have $r^2 + 1 \geq r \cdot \mathbf{max}(2, r)$, with equality iff $r = 1$. Suppose there exist standard sensitivity expressions R'_f, R'_x that satisfy the statement. Because $R'_f \geq r$ and $R'_x \geq \mathbf{max}(2, r)$, we know by asymptotic analysis that the degree of R'_f and R'_x must be at least 1. Furthermore, because $r^2+1 \geq R'_f \cdot R'_x$, their degree must be exactly 1, with leading coefficient equal to 1. Write $R'_f = r + a$ and $R'_x = r + b$, where a, b are positive constants. The lower bound on R'_x implies $b \geq 2$. For $r = 1$, we have $R'_f \cdot R'_x \geq 3a + 3 \geq 3$. However, the lower and upper bounds for $R'_f \cdot R'_x$ coincide at that point, forcing $R'_f \cdot R'_x = 2$; contradiction. Thus, no such R'_f, R'_x can exist. □

It is not hard to adapt the above into a counterexample for the soundness of the algorithm with respect to the original system. However, we can recover soundness by extending the sensitivity language for the basic typing rules as well.

3.2 Avoiding the Avoidance Problem

After the addition of least upper bounds for sensitivities, the bottom-up approach is in a good working state for the basic system. However, other constructs in the language introduce further challenges. In particular, let's examine a simple version of the abstraction rule for sensitivity variables:

$$\frac{\phi, i : \kappa \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Gamma}{\phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma}$$

When this rule is interpreted in a top-down approach, usually no problem arises; we would just introduce the new sensitivity variable and proceed with type checking.

[1] Indeed, it can be seen that *DFuzz* does not possess minimal types. Refer to the Appendix for a more detailed proof.

However, when the type-checking direction is reversed, we hit a version of the avoidance problem [8, 11, 18]. The avoidance problem usually appears in slightly different scenarios related to existential types, and could be informally stated as finding a best type free of a particular variable. In our case, we must find the "best" Γ free of i. It may not be obvious how i could have been propagated to Γ, but indeed, a function f in e could have a type such as $!_i\sigma \multimap \tau$, and applying f will introduce i into the environment in the bottom-up approach.

Fortunately, in our setting, we can easily solve the avoidance problem by further extending the sensitivity language. The "best" way of freeing a sensitivity expression R of a variable i is to take the supremum of R over all possible values of i, which we denote by $\mathbf{sup}(i, R)$[2]. Then, the minimal environment is $\mathbf{sup}(i, \Gamma)$, where the supremum is extended to each binding in the environment.

3.3 Undependable Dependencies

The last case to consider in our informal overview is **case**, also referred as dependent pattern matching.

The dependent pattern matching can be considered as a special case of the two previous difficulties. Like the least upper bound, we must compute a least upper bound of the resources used in two branches. However, now the information coming from the successor branch may also contain sensitivities depending on the newly introduced refinement variable, which cannot occur in the upper bound; similar to the avoidance problem we just discussed. On top of that, information coming from both sides is conditional on the particular refinements induced by the match, so any new sensitivity information that we propagate cannot really depend on the refinements.

We now face a choice: we can introduce refinement types over sensitivity and size variables of the form $\{\sigma \mid P(\vec{i})\}$, which would allow us to express the sensitivity inference for **case** in term of the least upper bound and supremum operations. However, we take a simpler path and add a conditional operator on natural number expressions S, $\mathbf{case}(S, R_0, i, R_s)$, interpreted as R_0 if S is 0 or $R_s[i \mapsto S - 1]$ if $S \geq 1$.

In the next sections we proceed to formally introduce the extended sensitivities and its semantics; we discuss the type-checking algorithm, which depends on solving inequality constraints over the extended sensitivities; and we study several approaches to constraint solving and discuss decidability issues.

4. Extended *DFuzz*: *EDFuzz*

We define a conservative extension to *DFuzz*'s type system, *EDFuzz*, which is basically *DFuzz* with an extended sensitivity language for the indices. We summarize the new sensitivity terms, ranged over by meta-variable $\hat{R}$:

- $\mathbf{max}(\hat{R}_1, \hat{R}_2)$ is the pointwise least upper bound of sensitivity terms $\hat{R}_1, \hat{R}_2$.
- $\mathbf{sup}(i, \hat{R})$ is the pointwise least upper bound of $\hat{R}$ over all i.
- $\mathbf{case}(S, \hat{R}_0, i, \hat{R}_s)$ is the conditional function on the size expression S that is valued $\hat{R}_0$ when $S = 0$, and $\hat{R}_s[i \mapsto S - 1]$ when S is a strictly positive integer.

The semantics of extended terms is defined as follows.

Definition 3 (Extended sensitivity semantics)**.** *We extend the semantics of sensitivities to the new constructs in the following way (the old cases stay the same):*

$$[\![\mathbf{sup}(i : \kappa, \hat{R})]\!]_\rho := \sup_{r \in [\![\kappa]\!]} \{[\![\hat{R}]\!]_{\rho \cup [i=r]}\}$$

$$[\![\mathbf{max}(\hat{R}_1, \hat{R}_2)]\!]_\rho := \max([\![\hat{R}_1]\!]_\rho, [\![\hat{R}_2]\!]_\rho)$$

$$[\![\mathbf{case}(S, \hat{R}_0, i, \hat{R}_s)]\!]_\rho := \begin{cases} [\![\hat{R}_0]\!]_\rho & \textit{if} \quad [\![S]\!]_\rho = 0 \\ [\![\hat{R}_s]\!]_{\rho \cup [i=n-1]} & \textit{if} \quad [\![S]\!]_\rho = n \geq 1. \end{cases}$$

We define analogous operations on environments in the obvious way. For instance, if $x :_{[R_1]} \sigma \in \Gamma_1$ and $x :_{[R_2]} \sigma \in \Gamma_2$, then $x :_{[\mathbf{max}(R_1,R_2)]} \sigma \in \mathbf{max}(\Gamma_1, \Gamma_2)$. As previously, two-argument operations on environments are only defined when every variable that is bound on both environments is assigned the same type by them.

It is not hard to show that any derivation valid in *DFuzz* remains valid in *EDFuzz*. Furthermore, *DFuzz*'s metatheory only relies on sensitivity terms having an interpretation as total function from free variables to a real number, rather than on any specific property about the interpretation itself. The extended interpretation is total, and hence the metatheory of *DFuzz* extends to *EDFuzz*.

5. Type Checking and Inference

We present a sound and complete type-checking and sensitivity-inference algorithm for *EDFuzz*. The algorithm assumes an oracle for deciding the subtyping relation; in this sense, our algorithm is relatively complete. We defer discussion about solving subtyping constraints to the next section.

The type-checking problem for *EDFuzz* is the usual one: given a full context, term, and type, the goal is to check if there is a derivation deriving the type from the context.

Definition 4 (Type Checking)**.** *Given an environment Γ, a term e, a type σ, the* type-checking problem *for* EDFuzz *is to determine whether a derivation $\emptyset; \emptyset \mid \Gamma \vdash e : \sigma$ exists.*

Before we move to sensitivity inference, we introduce some notation for working with contexts. It will be convenient to work with contexts with no top-level annotations, i.e., contexts with bindings $(x : \sigma)$, where σ is a proper *EDFuzz* type. We will call such contexts *context skeletons*. For notation, $\overline{\Gamma}$ will mean the context Γ with all top-level annotations removed, while $\Gamma^\bullet$ will represent an arbitrary context skeleton.

In our context, sensitivity inference means inferring the sensitivity annotations in both an environment and a type. The input is an annotated term[3] and a context with without top-level annotations. The goal is to reconstruct a type for the term, a full proper *EDFuzz* context (e.g., with all top-level annotations) along with a derivation, if possible.

Definition 5 (Sensitivity Inference)**.** *Given an environment skeleton $\Gamma^\bullet$ and a term e, the* sensitivity-inference problem *is to compute an environment Γ and a type σ with a derivation of $\emptyset; \emptyset \mid \Gamma \vdash e : \sigma$, such that $\overline{\Gamma} = \Gamma^\bullet$.*

5.1 The Algorithm

We can fulfill both goals using an algorithm that takes as inputs a term e, an environment free of sensitivity annotations $\Gamma^\bullet$ and a refinement constraint Φ. The algorithm will output an annotated environment Δ and a type σ. We write a call to the sensitivity inference algorithm as:

$$\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \sigma.$$

[2] Contrary to $\mathbf{max}(-,-)$, it would have been possible to define this construct as a function over sensitivity expressions, without the need to extend their syntax. This would still be true even after introducing index-level case sensitivity expression for analyzing dependent pattern matching. As the translation is somewhat intricate and leads to more complex constraints, we chose to add it directly to the syntax of sensitivity expressions.

[3] We discuss annotations in Section 5.2

$$\frac{}{\phi; \Phi; \Gamma^\bullet; r \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet); \mathbb{R}}\ (\text{Const})$$

$$\frac{n = \llbracket S \rrbracket}{\phi; \Phi; \Gamma^\bullet; n \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet); \mathbb{N}[S]}\ (\text{Const}_{\mathbb{N}})$$

$$\frac{}{\phi; \Phi; \Gamma^\bullet, x : \sigma; x \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma; \sigma}\ (\text{Var})$$

$$\frac{\begin{array}{l}\phi; \Phi; \Gamma^\bullet, x : \sigma; e \Longrightarrow \Gamma, x :_{[R']} \sigma; \tau \\ \phi; \Phi \models R \geq R'_{\square\uparrow}\end{array}}{\phi; \Phi; \Gamma^\bullet; \lambda(x :_{[R]} \sigma).\ e \Longrightarrow \Gamma; !_R\sigma \multimap \tau}\ (\multimap I)$$

$$\frac{\begin{array}{l}\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma; !_R\sigma \multimap \tau \\ \phi; \Phi; \Delta^\bullet; e_2 \Longrightarrow \Delta; \sigma' \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi; \Gamma^\bullet; e_1\ e_2 \Longrightarrow \Gamma + R \cdot \Delta; \tau}\ (\multimap E)$$

$$\frac{\begin{array}{l}\phi; \Phi; \Gamma^\bullet, x : \sigma; e \Longrightarrow \Gamma, x :_{[R]} \sigma; \sigma' \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi; \Gamma^\bullet; \mathbf{fix}\, x : \sigma.\ e : \sigma \Longrightarrow \infty \cdot \Gamma; \sigma}\ (\text{Fix})$$

$$\frac{\phi, i : \kappa; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma}{\phi; \Phi; \Gamma^\bullet; \Lambda i : \kappa.\ e \Longrightarrow \mathbf{sup}(i, \Gamma); \forall i : \kappa.\ \sigma}\ (\forall I)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi; \Gamma^\bullet; e[S] \Longrightarrow \Gamma; \sigma[S/i]}\ (\forall E)$$

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma_1; \sigma_1 \\ \phi; \Phi; \Gamma^\bullet; e_2 \Longrightarrow \Gamma_2; \sigma_2\end{array}}{\phi; \Phi; \Gamma^\bullet; \langle e_1, e_2 \rangle \Longrightarrow \Gamma_1 + \Gamma_2; \sigma_1 \otimes \sigma_2}\ (\otimes I)$$

$$\frac{\begin{array}{l}\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \sigma \otimes \tau \\ \phi; \Phi; \Gamma^\bullet, x : \sigma, y : \tau; e' \Longrightarrow \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau; \mu\end{array}}{\phi; \Phi; \Gamma^\bullet; \mathbf{let}(x, y) = e\ \mathbf{in}\ e' \Longrightarrow \Gamma + \mathbf{max}(R_{1\,\square\uparrow}, R_{2\,\square\uparrow}) \cdot \Delta; \mu}\ (\otimes E)$$

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma_1; \sigma_1 \\ \phi; \Phi; \Gamma^\bullet; e_2 \Longrightarrow \Gamma_2; \sigma_2\end{array}}{\phi; \Phi; \Gamma^\bullet; \langle e_1, e_2 \rangle \Longrightarrow \mathbf{max}(\Gamma_1, \Gamma_2); \sigma_1 \mathbin{\&} \sigma_2}\ (\&\ I)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma_1 \mathbin{\&} \sigma_2}{\phi; \Phi; \Gamma^\bullet; \pi_i e \Longrightarrow \Gamma; \sigma_i}\ (\&\ E)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \mathbb{N}[S]}{\phi; \Phi; \Gamma^\bullet; \mathbf{s}\ e \Longrightarrow \Gamma; \mathbb{N}[S + 1]}\ (\text{S}\ I)$$

$$\frac{\begin{array}{l}\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \mathbb{N}[S] \qquad \phi; \Phi, S = 0; \Gamma^\bullet; e_0 \Longrightarrow \Gamma_0; \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1; \Gamma^\bullet, x : \mathbb{N}[i]; e_s \Longrightarrow \Gamma_s, x :_{[R']} \mathbb{N}[i]; \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\begin{array}{l}\phi; \Phi; \Gamma^\bullet; \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\ 0 \mapsto e_0 \mid x_{[i]} + 1 \mapsto e_s \\ \Longrightarrow \mathbf{case}(S, \Gamma_0, i, \Gamma_s) + \mathbf{case}(S, 0, i, R'_{\square\uparrow}) \cdot \Delta; \sigma\end{array}}\ (\mathbb{N}\ E)$$

Figure 4. Algorithmic Rules for *EDFuzz*

Figure 4 presents the full algorithm in a judgmental style. The algorithm is based on a syntax-directed version of DFuzz that enjoys several nice properties; full technical details and notation definitions can be found in the Appendix. Here, we just sketch how the transformation works in the proofs of soundness and completeness.

Theorem 6 (Algorithmic Soundness)**.** *Suppose* $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$*. Then, there is a derivation of* $\phi; \Phi \mid \Gamma \vdash e : \sigma$*.*

Proof. We define two intermediate systems: The first one internalizing certain properties of weakening and a second, syntax-directed. The algorithm is a direct transcription of the syntax-directed system and soundness can be proved by induction on the number of steps. We prove soundness of the syntax-directed system by induction on the syntax-directed derivation. □

Theorem 7 (Algorithmic Completeness)**.** *If* $\phi; \Phi \mid \Gamma \vdash e : \sigma$ *is derivable, then* $\phi; \Phi; \overline{\Gamma}; e \Longrightarrow \Gamma'; \sigma'$ *and* $\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \wedge \sigma' \sqsubseteq \sigma$*.*

Proof. We show that a "best" syntax-directed derivation can be build from any standard derivation by induction on the original derivation plus monotonicity and commutativity properties of the subtype relation. Completeness for the algorithm follows. □

5.2 Removing Sensitivity Annotations

We briefly discuss the role annotations play in our algorithm. *DFuzz* programs have three different annotations: the type of the argument for lambda terms (including the sensitivity), the return type for case, and the type for fixpoints.

The sensitivity annotations ensure that inferred types are free of terms with extended sensitivities. This is useful for some optimizations on subtype checking (introduced later in the paper). However, the general encoding of subtyping checks works with full extended types, thus the sensitivity annotations can be safely omitted and the system will infer types containing extended sensitivities.

Due to technical difficulties in inferring the minimal sensitivity in the presence of higher-order functions, the argument type in functions (σ in $\lambda(x : \sigma)$) must be annotated, and we require the type of fixpoints to be annotated.

6. Constraint Solving

The type-checking algorithm introduced in the previous section produces inequality constraints over the extended sensitivity language. While these extended sensitivity terms may appear complicated, we can translate them into equivalent formulas over the first-order theory of arithmetic over $\mathbb{R}$ and $\mathbb{N}$. While we show in the next section that the formulas we generate are usually undecidable, they can still be handled by standard solvers. Moreover, in Section 8.1 we will present a sound (although not complete) computable procedure to check the constraints.

To define our translation, it suffices to convert formulas with extended sensitivities into equivalent ones that use only standard sensitivities, for we can replace quantification over $\mathbb{S}$ by equivalent formulas that only quantify over $\mathbb{R}$ and $\mathbb{N}$. For instance, a formula of the form $\forall i : \mathbb{S}.P$, where P has only quantifiers over $\mathbb{R}$ or $\mathbb{N}$, can be translated into $(\forall i : \mathbb{R}.i \geq 0 \Rightarrow P) \wedge P'$, where P' is the result of substituting ∞ for i in P and performing all possible simplifications.

The idea behind our translation is simple: we use a first-order formula to uniquely specify each extended sensitivity term. Specifically, we define a predicate $T(R)$ for each extended sensitivity term R, such that $\llbracket T(R)(r) \rrbracket_\rho$ holds exactly when r is equal to the interpretation of R under the valuation ρ. For instance, consider the translation for $R_1 + R_2$:

$$T(R_1{+}R_2)(r) := \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r = r_1{+}r_2.$$

For ρ a valuation for R_1, R_2, we have $r_1 = \llbracket R_1 \rrbracket_\rho$ and $r_2 = \llbracket R_2 \rrbracket_\rho$. Then the only r that satisfies this predicate is

$$r = r_1 + r_2 = \llbracket R_1 \rrbracket_\rho + \llbracket R_2 \rrbracket_\rho = \llbracket R_1 + R_2 \rrbracket_\rho,$$

as desired.

For a more involved example, consider the translation of $\mathbf{max}(R_1, R_2)$:

$$\begin{aligned} &T(\mathbf{max}(R_1, R_2))(r) \\ &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge \\ &(r_1 \geq r_2 \wedge r = r_1 \vee r_2 \geq r_1 \wedge r = r_2). \end{aligned}$$

Again, for any valuation ρ of R_1, R_2, we have $r_1 = \llbracket R_1 \rrbracket_\rho$ and $r_2 = \llbracket R_2 \rrbracket_\rho$. The final conjunction states that r must be the largest among r_1 and r_2, which is precisely the semantics we have given $\llbracket \mathbf{max}(R_1, R_2) \rrbracket_\rho$. The full translation is in Figure 5.

We formalize our intuitive explanation of the translation with the following lemma.

Lemma 8. *For every sensitivity expression R and $r \in \mathbb{S}$, and for every valuation ρ whose domain contains the free variables of R, $\llbracket T(R)(r) \rrbracket_\rho \iff r = \llbracket R \rrbracket_\rho$*

Proof. By induction on R. We have already considered the $R_1 + R_2$ and $\mathbf{max}(R_1, R_2)$ cases above. □

Using the translation of terms, we can translate sensitivity constraints generated by our typing algorithm. We map each constraint of the form

$$\phi; \Phi \models R_1 \geq R_2$$

to

$$\forall \phi, \Phi \Rightarrow \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r_1 \geq r_2$$

Thanks to Lemma 8, this translation is equivalent to the semantics of sensitivity constraints given in Section 2.

7. Undecidability of Type-checking

As we have seen in the previous section, constraints over our extended sensitivity language can be translated to simple first-order formulas. Taken by itself, this is not entirely satisfactory, as the first-order theory of $\mathbb{N}$ is already undecidable. A nice illustration of this is Hilbert's tenth problem, which asks if a polynomial equation of the form $P(\vec{x}) = 0$ over several variables has any solutions over the natural numbers. After several years of investigation, this property was finally shown to be undecidable.

In this section, we will show that this result makes *DFuzz* type-checking undecidable. We begin with an auxiliary lemma.

Lemma 9. *Given polynomials P, Q over n variables with coefficients in $\mathbb{N}$, checking $\forall \vec{i} \in \mathbb{N}^n, P(\vec{i}) \geq Q(\vec{i})$ is undecidable.*

Proof. We will use a solution to our problem to solve Hilbert's tenth problem. Suppose we are given a polynomial P with integer coefficients, and we want to decide whether $\exists \vec{i} \in \mathbb{N}^n, P(\vec{i}) = 0$. This is equivalent to deciding $\neg \forall \vec{i} \in \mathbb{N}^n, P(\vec{i})^2 \geq 1$. Write $P(\vec{i})^2 = P^+(\vec{i}) - P^-(\vec{i})$, where P^+ and P^- have only positive coefficients. Then our condition is equivalent to $\neg \forall \vec{i} \in \mathbb{N}^n, P^+(\vec{i}) \geq P^-(\vec{i}) + 1$. Thus, we can solve Hilbert's tenth problem by using P^+ and $P^- + 1$ as inputs to our problem, which shows that it is undecidable. □

We can then show the following

Theorem 10. DFuzz *type checking is undecidable.*

Proof. Suppose we are given P and Q as previously. Consider the types $\sigma = \forall \vec{i}, !_0 \mathbb{N}^n[\vec{i}] \multimap !_{Q(\vec{i})} \mathbb{R} \multimap \mathbb{R}$ and $\tau = \forall \vec{i}, !_0 \mathbb{N}^n[\vec{i}] \multimap !_{P(\vec{i})} \mathbb{R} \multimap \mathbb{R}$. Then $\sigma \sqsubseteq \tau$ is equivalent to $\forall \vec{i}, P(\vec{i}) \geq Q(\vec{i})$. On the other hand, using recursion and dependent pattern matching, it is possible to write a function that multiplies a real number by a polynomial $Q(\vec{v})$ with variables ranging over $\mathbb{N}$. Its minimal type will clearly be σ. Therefore, type-checking it against τ is equivalent to deciding $\sigma \sqsubseteq \tau$; since P and Q are arbitrary, this is undecidable by Lemma 9. □

8. Approaches to Constraint Solving

Given that type-checking *DFuzz* (and hence also *EDFuzz*) is undecidable, is there anything more we can do besides feeding the constraints to a solver and hoping for the best? In this section, we discuss two possible directions to tackle these constraints. For both of these approaches, we require that *all annotations in the term be standard sensitivities*, rather than extended. Then, we have the following lemma. (We defer the proof to the Appendix.)

Lemma 11 (Standard Annotations). *Assume annotations in a term e range over standard sensitivities and $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$. Then:*

- *σ has no extended sensitivities; and*
- *all constraints required for the algorithm are of the form $\phi; \Phi \models R \geq R'$ where R is a standard sensitivity term.*

8.1 Modifying the subtype relation

The first approach is to restrict *EDFuzz* to a fragment that enjoys decidable type checking, which we call *UDFuzz*. The main difference between both languages is the interpretation of subtyping constraints: in *UDFuzz*, constraint variables are interpreted uniformly, ranging over *all* possible sensitivity values, regardless of their kind. As noted in Section 6, we can translate such formulas into the first-order theory of real arithmetic; since this theory is decidable, so is *UDFuzz* type checking.

Of course, this only makes sense if we can show that *UDFuzz* is *sound* with respect to *EDFuzz*. As it turns out, it suffices to restrict *UDFuzz* annotations to standard sensitivities—as we'll see, this forces the subtyping relation of *UDFuzz* to be a subrelation of the one of *EDFuzz*. This restriction rules out some programs that are typeable under *EDFuzz*, but is expressive enough to cover interesting ones, including most of the original examples [10].

Formally, besides the restriction on annotations, *UDFuzz* is the system obtained from *EDFuzz* by replacing all constraints of the form $\phi; \Phi \models R \geq R'$ with *uniform* constraints $\phi; \Phi \models^U R \geq R'$, which have the following interpretation:

$$\forall \rho \in \mathsf{val}_U(\phi). \llbracket \Phi \rrbracket^U_\rho \Rightarrow \llbracket R \rrbracket^U_\rho \geq \llbracket R' \rrbracket^U_\rho$$

Here, $\mathsf{val}_U(\phi)$ is the set of all *uniform* valuations, that map variables in $\mathrm{dom}(\phi)$ to values in $\mathbb{S}$. The denotation $\llbracket \cdot \rrbracket^U_\rho$ of formulas and sensitivity and size terms is the same as before, except for two cases:

$$\llbracket \mathbf{sup}(i : \kappa, \hat{R}) \rrbracket^U_\rho := \sup_{r \in \mathbb{S}} \{ \llbracket \hat{R} \rrbracket^U_{\rho \cup [i=r]} \}$$

$$\llbracket \mathbf{case}(S, \hat{R}_0, i, \hat{R}_s) \rrbracket^U_\rho := \begin{cases} \llbracket \hat{R}_1 \rrbracket^U_\rho & \text{if} \quad \llbracket S \rrbracket^U_\rho = 0 \\ 0 & \text{if} \quad \llbracket S \rrbracket^U_\rho \in (0,1) \\ \llbracket \hat{R}_2 \rrbracket^U_{\rho \cup [i=r-1]} & \text{if} \quad \llbracket S \rrbracket^U_\rho = r \geq 1. \end{cases}$$

$$
\begin{aligned}
\kappa &:= \mathbb{N} \mid \mathbb{S} \\
T(i)(r) &:= i = r \\
T(R_1 + R_2)(r) &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r = r_1 + r_2 \\
T(R_1 \cdot R_2)(r) &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r = r_1 \cdot r_2 \\
T(\mathbf{max}(R_1, R_2))(r) &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge (r_1 \geq r_2 \wedge r = r_1 \vee r_2 \geq r_1 \wedge r = r_2) \\
T(\mathbf{case}(S, R_0, i, R_s))(r) &:= \exists r_s : \mathbb{N}, T(S)(r_s) \wedge (r_s = 0 \wedge T(R_0)(r) \vee \exists i : \mathbb{N}, r_s = i + 1 \wedge T(R_s)(r)) \\
T(\mathbf{sup}(i : \kappa, R))(r) &:= \mathbf{bound}(i : \kappa, R, r) \wedge \forall r'.\mathbf{bound}(i : \kappa, R, r') \Rightarrow r' \geq r \\
\mathbf{bound}(i : \kappa, R, r) &:= \forall i : \kappa . \exists r' : \mathbb{S}. T(R)(r') \wedge r' \leq r
\end{aligned}
$$

Figure 5. Constraint Translation

We first show that this uniform semantics is an extension of the standard semantics.

Lemma 12. *Suppose R is a standard sensitivity term, typed under environment ϕ. Then, for any standard valuation $\rho \in \mathsf{val}(\phi)$, we have*

$$[\![R]\!]^U_\rho = [\![R]\!]_\rho .$$

Proof. Immediate from the definition of the interpretation. □

We are now ready to prove that the uniform interpretation of constraints is sound with respect to the original interpretation.

Theorem 13. *Suppose R, R' are well-typed in environment ϕ, with R standard. Suppose that $\phi; \Phi \models^U R \geq R'$ is valid. Then $\phi; \Phi \models R \geq R'$ is also valid.*

Proof. It is clear that for any standard valuation $\rho \in \mathsf{val}(\phi)$, we have $[\![R']\!]^U_\rho \geq [\![R']\!]_\rho$. Assuming this, the hypothesis of the theorem yields $[\![R]\!]^U_\rho \geq [\![R']\!]^U_\rho \geq [\![R']\!]_\rho$ for every standard valuation $\rho \in \mathsf{val}(\phi)$. But R is a standard sensitivity, so $[\![R]\!]^U_\rho = [\![R]\!]_\rho$ by Lemma 12, and we are done. □

Thanks to Lemma 11, all *UDFuzz* constraints are of this form, which shows that the subtype relation of *UDFuzz* is a subrelation of the subtype relation in *EDFuzz*. By reasoning analgous to Lemma 8, we can show that relaxing the first order translation of constraints captures this uniform interpretation. More formally:

Lemma 14. *For every sensitivity term R, let $T^U(R)$ be a unary predicate defined exactly as in Figure 5, but replacing quantification over $\mathbb{N}$ with quantificiation over $\mathbb{S}$ and with the modified* **case** *translation:*

$$
\begin{aligned}
&T^U(\mathbf{case}(S, R_0, i, R_s))(r) := \\
&\quad \exists r_s : \mathbb{S}, \quad T^U(S)(r_s) \wedge (r_s = 0 \wedge T^U(R_0)(r)) \\
&\qquad\qquad \vee \quad (0 < r_s < 1 \wedge r = 0) \\
&\qquad\qquad \vee \quad (\exists i : \mathbb{S}, i \geq 0 \wedge r_s = i + 1 \wedge T^U(R_s)(r))
\end{aligned}
$$

Then, $r \in \mathbb{S}$, and for every uniform valuation ρ whose domain contains the free variables of R, $[\![T^U(R)(r)]\!]^U_\rho \iff r = [\![R]\!]^U_\rho$.

By this lemma, we can give a sound, complete and decidable type-checking algorithm for *UDFuzz*.

Theorem 15. *Suppose we use our algorithmic system, with the constraints*

$$\phi; \Phi \models^U R_1 \geq R_2$$

handled by translation to the first order formula

$$\forall \phi, \Phi \Rightarrow \exists r : \mathbb{S}, T^U(R_2)(r) \wedge R_1 \geq r,$$

where all quantifiers are over $\mathbb{S}$. Since the theory of $\mathbb{S}$ is decidable, this gives an effective type-checking procedure for UDFuzz.

Proof. Note that R_1 is a standard sensitivity term, so the translated formula is indeed a first order formula over the theory of $\mathbb{S}$. By Lemma 14, the translated formula is logically equivalent to $[\![\Phi]\!]^U_\rho \Rightarrow [\![R_1]\!]^U_\rho \geq [\![R_2]\!]^U_\rho$ for all uniform valuations $\rho \in \mathsf{val}_U(\phi)$, which in turn implies $\phi; \Phi \models R_1 \geq R_2$ by Theorem 13. This shows that the algorithmic system is sound and complete with respect to *UDFuzz*. □

Remark 16. UDFuzz *is a strict subset of* EDFuzz*; informally, it contains* EDFuzz *programs with typing derivations that do not use facts true over $\mathbb{N}$ but not over $\mathbb{R}$. One key way that subtyping is used in* EDFuzz *is for equational manipulations of the indices; for instance, subtyping may be needed to change the index expression $3(i + 1)$ to $3i + 3$. This reasoning is available in* UDFuzz *as well; indeed, most of the example programs in* DFuzz *are typeable under* UDFuzz *as well. (The only exception is k-medians, which extends the index language with a division function that we have not investigated.)*

However, there are many programs that lie in EDFuzz *but not in* UDFuzz*—constraints as simple as $\forall i.\ i^2 \geq i$ are true when quantifing over the naturals but not when quantifying over the reals. Valid* EDFuzz *programs that use these facts in their typing derivation will not lie in* UDFuzz.

8.2 Constraint Simplification

The second approach is to simplify the constraints generated by the translation of Section 6, so that they can be better handled by solvers. Since alternating quantifiers are a source of complexity in formulas, we devised a rewriting procedure for producing constraints with no alternating quantifiers. Here, we continue to require that all source annotations must be standard sensitivity terms.

To begin, we generalize our three extended constructs with a new *constrained least upper bound* (**club**) operation, with form $\mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\}$. Here, ϕ is a size and sensitivity variable environment, Φ is a constraint environment, and R is a sensitivity term, extended or standard. The judgment for a well-formed **club** is

$$\phi \vdash \mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\},$$

where each R_j has kind r under $\phi, \phi_j; \Phi_j$, and $\phi, \{\phi_j\}_j$ have disjoint domain. Intuitively, **club** is a maximum over a set of sensitivities, restricting to sensitivities where the associated constraint is satisfied. Sensitivities where the constraints are not satisfied are ignored. Formally, let ϕ contain the free variables of **club**, and let $\rho \in \mathsf{val}(\phi)$ be any standard valuation. We can give the following interpretation of **club**:

$$
\begin{aligned}
&(\!|\mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\}|\!)_\rho := \\
&\max_{j \in [n]} \max\{[\![R_j]\!]_{\rho \cup \rho_j} \mid \rho_j \in \mathsf{val}(\phi_j) \text{ and } [\![\Phi_j]\!]_{\rho \cup \rho_j}\}.
\end{aligned}
$$

We define the maximum over an empty set to be 0.

Now, we can encode the extended sensitivity terms using only **club**, through the following translation function:

$$\begin{aligned}
C(\mathbf{max}(\hat{R}_1, \hat{R}_2)) &:= \mathbf{club}\{(\emptyset; \emptyset; C(\hat{R}_1)), (\emptyset; \emptyset; C(\hat{R}_2))\} \\
C(\mathbf{sup}(i, \hat{R})) &:= \mathbf{club}\{(i; \emptyset; C(\hat{R}))\} \\
C(\mathbf{case}(S, i, \hat{R}_0, \hat{R}_s)) &:= \mathbf{club}\{(\emptyset; S = 0; C(\hat{R}_0)), \\
&\qquad (i; S = i + 1; C(\hat{R}_s))\} \\
C(\hat{R}_1 + \hat{R}_2) &:= C(\hat{R}_1) + C(\hat{R}_2) \\
C(\hat{R}_1 \cdot \hat{R}_2) &:= C(\hat{R}_1) \cdot C(\hat{R}_2) \\
C(R) &:= R \qquad \text{otherwise.}
\end{aligned}$$

While we may now have nested **club**, we extend the interpretation in the natural way. We can show that the translation faithfully preserves the semantics of the extended terms, with the following lemma.

Lemma 17. *Suppose $\phi \vdash R$ and $\rho \in \mathsf{val}(\phi)$ is a standard valuation. Then, $\llparenthesis C(R) \rrparenthesis_\rho = [\![R]\!]_\rho$.*

Proof. By induction on R. □

Now, we can simplify the compiled constraints. First, we can push all standard sensitivity terms to the leaves of the expression. More formally, we have the following lemma.

Lemma 18. *Suppose $\phi \vdash R \cdot \mathbf{club}\{(\phi_i; \Phi_i; C_i)\}_i + R'$, where R, R' are standard sensitivity terms, and C_i is an arbitrary sensitivity term possibly involving* **club**. *Then, for any standard valuation $\rho \in \mathsf{val}(\phi)$,*

$$\llparenthesis R{\cdot}\mathbf{club}\{(\phi_i; \Phi_i; C_i)\}_i{+}R' \rrparenthesis_\rho = \llparenthesis \mathbf{club}\{(\phi_i; \Phi_i; R \cdot C_i + R')\}_i \rrparenthesis_\rho.$$

Proof. By the definition of the interpretations, and the mathematical fact $a \cdot \max_i\{b_i\} + c = \max_i\{a \cdot b_i + c\}$ for $a, b, c \geq 0$. □

Thus, without loss of generality we may reduce the compiled sensitivity constraint to an expression of the form Q, with grammar

$$Q ::= \emptyset \mid Q_1{+}Q_2 \mid Q_1{\cdot}Q_2 \mid \mathbf{club}\{(\phi_i; \Phi_i; Q_i)\} \mid \mathbf{club}\{(\phi_i; \Phi_i; R_i)\},$$

where R_i are standard sensitivity terms. We will use the metavariable V to denote an arbitrary (possibly empty) collection of triples $(\phi_i; \Phi_i; R_i)_i$, and the metavariable W to denote an arbitrary (possibly empty) collection of triples $(\phi_i; \Phi_i; Q_i)_i$. Throughout, we will implicitly work up to permutation of the arguments to **club**: for instance, $\mathbf{club}\{(X), (Y)\}$ will be considered the same as $\mathbf{club}\{(Y), (X)\}$. We will also work up to commutativity of addition and multiplication: $Q_1 + Q_2$ will be considered the same as $Q_2 {+} Q_1$, and likewise with multiplication. We present the constraint simplification rules as a rewrite relation $\mapsto$. As typical, we will write $\mapsto^*$ for the reflexive, transitive closure of $\mapsto$. The full rules are in Figure 6.

We can prove correctness of our constraint simplification with the following lemma.

Lemma 19. *Suppose $Q \mapsto Q'$, and suppose $\phi \vdash Q$ and $\phi \vdash Q'$. Then, for any standard valuation $\rho \in \mathsf{val}(\phi)$, we have $\llparenthesis Q \rrparenthesis_\rho = \llparenthesis Q' \rrparenthesis_\rho$.*

Proof. By induction on the derivation of $Q \mapsto Q'$. The cases Plus, Mult and Red are immediate by induction. The other cases all follow by the semantics of **club**; details are in the Appendix. □

The simplification relation terminates in the following particular simple form.

Lemma 20. *Let Q be a sensitivity term involving* **club**. *Along any reduction path, Q reduces in finitely many steps to a term of the form $\mathbf{club}\{V\} = \mathbf{club}\{(\phi_1; \Phi_1; R_1), \dots, (\phi_n; \Phi_n; R_n)\}$.*

Proof. First, note that any reduction of Q must terminate in finitely many steps: by induction on the derivation of the reduction, it's clear that each reduction removes one **club** subterm, and no reductions introduce **club** subterms. So, suppose that Q is a term with no possible reductions.

By induction on the structure of Q, we claim that Q is of the desired form. Say if $Q = Q_1 + Q_2$, if either Q_1, Q_2 can reduce, then Plus applies. If not, then by induction, CPlus applies. The same reasoning follows for $Q = Q_1 \cdot Q_2$: either Mult applies, or CMult does. Finally, if Q is a single **club** term, if Red and Flat both don't apply, then Q is of the desired form. □

Finally, checking a constraint $\forall \phi.\, \Phi \Rightarrow R \geq \mathbf{club}\{V\}$ is simple.

Lemma 21. *Let R be a standard sensitivity term, and let V be*

$$V = (\phi_1; \Phi_1; R_1), \dots, (\phi_n; \Phi_n; R_n)$$

where each R_j is a standard sensitivity term without **club**. *Then, $\phi; \Phi \models R \geq \mathbf{club}\{V\}$ is logically equivalent to*

$$\forall \phi.\ \bigwedge_{j \in [n]} \forall \phi_j.\ \Phi \wedge \Phi_j \Rightarrow R \geq R_k.$$

Proof. Immediate by the semantics of $\mathbf{club}\{V\}$. □

Putting together all the pieces, given a constraint $\phi; \Phi \models R \geq R'$, with R standard, we can transform $C(R')$ to a term of the form Q by pushing all standard sensitivity terms to the leaves. Then, we normalize $Q \mapsto^* \mathbf{club}\{V\}$ by Lemma 20 arbitrarily. By Lemma 19, the interpretation of Q and $\mathbf{club}\{V\}$ are the same, so we can reduce the constraint $\phi; \Phi \models R \geq \mathbf{club}\{V\}$ to a first order formula over mixed naturals and $\mathbb{S}$, with no alternating quantifiers, by Lemma 21.

9. Implementation and Usability

We have implemented our algorithm for *EDFuzz*, including the constraint simplification described in the previous section, in a prototype type-checker. The tool is written in OCaml, and uses the Why3 framework to check the generated numeric inequalities with SMT solvers. We have successfully type-checked a range of examples, including all but one of the examples from the original *DFuzz* paper. The remaining example involves a "safe division" operation on the sensitivity language; we believe this operation can also be handled with our techniques. The solvers had no problem solving the mixed natural/real constraints on our examples, even though the problem is undecidable.

In our experience, the type-checker was quite usable. To give an idea of the annotation burden in a typical example, consider the raw, annotated program below.

```
function cdf
 forall (i:size) (b:list(num)[i]) (db:[i]num bag)
 : list(num)[i] {
   listcase b of list(num)[i] {
     []           ⇒  nil @ [num]
   | x :: xs [m]  ⇒
     let (lt, gt)  = bagsplit@[num]
       (fun (n:num) : bool {n < x}) db;
     let count  = (bagsize lt);
     let bigger = cdf[e][m] xs gt;
     cons @ [num][m] count bigger } }
```

This is a modified version of an original *DFuzz* example. It uses a few extensions to the system we have described, including additional primitive types (bag) and lists with a basic form of polymorphism.

$$\frac{}{\mathbf{club}\{(\phi;\Phi;\mathbf{club}\{(\phi_i;\Phi_i;R_i)\}_i),V\} \mapsto \mathbf{club}\{(\phi\cup\phi_i;\Phi\wedge\Phi_i;R_i),V\}_i}\ \text{Flat}$$

$$\frac{}{\mathbf{club}\{(\phi_i;\Phi_i;R_i)\}_i + \mathbf{club}\{(\phi'_j;\Phi'_j;R'_j)\}_j \mapsto \mathbf{club}\{(\phi_i\cup\phi'_j;\Phi_i\wedge\Phi'_j;R_i+R'_j)\}_{ij}}\ \text{CPlus}$$

$$\frac{}{\mathbf{club}\{(\phi_i;\Phi_i;R_i)\}_i \cdot \mathbf{club}\{(\phi'_j;\Phi'_j;R'_j)\}_j \mapsto \mathbf{club}\{(\phi_i\cup\phi'_j;\Phi_i\wedge\Phi'_j;R_i\cdot R'_j)\}_{ij}}\ \text{CMult}$$

$$\frac{Q_1 \mapsto Q'_1}{Q_1 + Q_2 \mapsto Q'_1 + Q'_2}\ \text{Plus} \qquad \frac{Q_1 \mapsto Q'_1}{Q_1 \cdot Q_2 \mapsto Q'_1 \cdot Q_2}\ \text{Mult} \qquad \frac{Q \mapsto Q'}{\mathbf{club}\{(\phi;\Phi;Q),W\} \mapsto \mathbf{club}\{(\phi;\Phi;Q'),W\}}\ \text{Red}$$

Figure 6. **club** Reduction

Our experience with error reporting was generally good. The tool points out the location of the failed check, which was usually not far from the actual error. The error messages leave a bit to be desired—usually, a polynomial inequality that can't be proved—we leave improving this aspect to future work.

The implementation and examples are available online.[4]

10. Related work

There is a vast literature on type checking for various combinations of indexed types, linear types, dependent types and subtyping. A distinctive feature of our approach is that our index language represents natural and real number expressions. As we have shown in the previous sections, this makes type checking non-trivial.

The work most closely related to ours is Dal Lago et al. [6], who studied the type-inference problem for dℓPCF, a relatively-complete type system for complexity analysis introduced in Dal Lago and Gaboardi [4]. dℓPCF uses ideas similar to *DFuzz* but brings the idea of linear dependent types to the limit. Indeed, dℓPCF index language contains function symbols that are given meaning by an equational program. The equational program then plays the role of an oracle for the type system—dℓPCF is in fact a family of type systems parametrized over the equational program. The main contribution of Dal Lago et al. [6] is an algorithm that, given a PCF program, generates a type and the set of constraints that must be satisfied in order to assign the return type to the input term.

In our terminology, their work is similar to the top-down approach we detailed in Section 3. As we discussed there, the complication of this approach is that it requires solving constraints over expressions—with possible function symbols—of the index-level language. As shown by Dal Lago and Petit, a clear advantage of the dℓPCF formulation is that instead of introducing an existential variable over expressions, one can introduce a new function symbol that will then be given meaning by the equational program generated by the constraints—i.e., the constraints give a description of the semantics of the program, which can be turned in an equational program, that in turn gives meaning to the function symbols of the index language appearing in the type. Clearly, this approach cannot be reduced to numeric resolution and need instead a combination of numeric and symbolic solving technology. The authors show that these constraints can be anyway handled by using the Why3 framework. Some constraints are discharged automatically by some of the solvers available in Why3 while others requires an interactive resolution using Coq.

As explained in Section 3, the situation with *DFuzz* is different. Indeed, *DFuzz* can be seen as a simplified version of dℓPCF—simplifying in particular the typing for the fixpoint and without variable bindings in !-types—extended however to deal with indices representing real numbers and using quantifications over index variables. A key distinction of *DFuzz* is that the set of constructors for the language of sensitivity is *fixed*—one cannot add arbitrary functions. Moreover, the extension to real numbers gives a different behavior from how natural numbers are used in dℓPCF—e.g., our example for the lack of minimal type would make no sense in dℓPCF. These distinctions make the type checking problem very different.

For another approach that is closely related to our work, recall that *DFuzz* is an extension of *Fuzz*. The sensitivity-inference and sensitivity-checking problems for *Fuzz* have been studied in D'Antoni et al. [7]. These problems are simpler than the one studied here since in *Fuzz* there is no dependency, no quantification and no subtyping. Indeed, the constraints generated are much simpler and can be solved quickly by an SMT solver.

Similarly, Eigner and Maffei [9] have studied an extension of *Fuzz* for modeling protocols. In their work they also give an algorithmic version of their type system. Their type system presents challenges similar to *Fuzz*, which they handle with algebraic manipulations. More precisely, their algorithmic version uses a technique similar to the one developed in Cervesato et al. [2] for the splitting of resources: when a rule with multiple premises is encountered the algorithmic system, first allocate all the resources to the first branch and then allocate the remaining resources to the second branch. Unfortunately, this approach cannot be easily applied to *DFuzz* due to the presence of index variables and dependent pattern matching.

From a different direction, recent works [1, 13] have shown how linear indexed type systems can be made more abstract and useful to analyze abstract resources. In particular, this kind of analyses is connected to comonadic notions of computations [20]. The type-inference algorithm described in Ghica and Smith [13] is parametric on an abstract notion of resource. This resource can be instantiated on a language for sensitivities similar to the one in *Fuzz*. So, this abstract type-inference procedure could be also used for sensitivity analysis.

DFuzz is one of several languages combining linear and dependent types. For example, ATS [3] is designed around a dependent type system enriched with a notion of resources that is a type-level representation of memory locations; these resources are managed using a linear discipline. ATS uses these features to verify the correctness of memory and pointer management.

Even if the use of linear types in ATS is very different from the one presented here, our type checking algorithm shares some similarities with ATS's one. The main difference is that ATS uses interactive theorem proving to discharge proof obligations while, thanks to the restricted scope of our analysis, our constraints can be handled by numeric solvers. In contrast, DML [26]—a predecessor of ATS which did not use linear types—uses an approach similar to ours by solving proof obligations using automatic numeric

[4] https://github.com/ejgallego/dfuzz

resolution. This required limitations on the operations available in the index language, similar to *DFuzz*.

Another work considering lightweight dependent types is the one by Zhu and Jagannathan [27]. In particular they propose a technique based on dependent types to reduce the verification of higher order programs to the verification of a first order language. While the goal of their work is similar in spirit to ours, their technique has only superficial similarities with the one presented here.

Finally, our work has been informed by the wide literature on type-checking, far too large to summarize here. For instance, the problem of dealing with subtyping rules by using syntax-directed systems has been studied by Pierce and Steffen [21], and others.

11. Conclusions and Future Work

We have presented a type-checking and sensitivity-inference algorithm for *EDFuzz*—a simple extension of *DFuzz*—featuring a linear indexed dependently type system. While we have shown that *DFuzz* type checking is undecidable in the general case, our approach generates constraints over the first-order theory over the reals and naturals, for which there are standard (though necessarily incomplete) solvers.

Overall, our design was guided by two principles: to stay as close to *DFuzz* as possible, and to provide a practical type checking procedure. While we do require extensions to *DFuzz*, there is a clear motivation for the introduction of each new construct. The idea of making a limited enrichment of the index language in order to simplify type-checking may be applicable to other linear indexed type systems. Furthermore, designers of such systems would do well to keep implementability in mind: seemingly unimportant decisions that simplify the metatheory may have a serious impact on type-checking.

References

[1] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *European Symposium on Programming (ESOP), Grenoble, France*. Springer, 2014.

[2] I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232 (1—2):133–163, 2000.

[3] C. Chen and H. Xi. Combining programming with theorem proving. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Tallinn, Estonia*, pages 66–77, 2005. ISBN 1-59593-064-7.

[4] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *IEEE Symposium on Logic in Computer Science (LICS), Toronto, Ontario*, pages 133–142. IEEE, 2011.

[5] U. Dal Lago and U. Schöpp. Functional programming in sublinear space. In *ACM Transactions on Programming Languages and Systems*, pages 205–225. Springer, 2010.

[6] U. Dal Lago, B. Petit, et al. The geometry of types. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 167–178, 2013.

[7] L. D'Antoni, M. Gaboardi, E. J. Gallego Arias, A. Haeberlen, and B. C. Pierce. Sensitivity analysis using type-based constraints. In *Workshop on Functional Programming Concepts in Domain-specific Languages (FPCDSL)*, FPCDSL '13, pages 43–50, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2380-2.

[8] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, POPL '03, pages 236–249, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5.

[9] F. Eigner and M. Maffei. Differential privacy by typing in security protocols. In *IEEE Computer Security Foundations Symposium, New Orleans, Louisiana*, pages 272–286, 2013.

[10] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, POPL '13, pages 357–370, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7.

[11] G. Ghelli and B. Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1–2):75 – 96, 1998.

[12] D. R. Ghica and A. Smith. Geometry of synthesis III: Resource management through type inference. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, volume 46, pages 345–356. ACM, 2011.

[13] D. R. Ghica and A. Smith. Bounded linear types in a resource semiring. In *European Symposium on Programming (ESOP), Grenoble, France*. Springer, 2014.

[14] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[15] B. Heeren, B. Heeren, J. Hage, J. Hage, D. Swierstra, and D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical report, 2002.

[16] U. D. Lago and B. Petit. Linear dependent types in a call-by-value scenario. In D. D. Schreye, G. Janssens, and A. King, editors, *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP), Leuven, Belgium*, pages 115–126. ACM, 2012. ISBN 978-1-4503-1522-7.

[17] U. D. Lago and U. Schöpp. Type inference for sublinear space functional programming. In K. Ueda, editor, *Asian Symposium on Programming Languages and Systems (APLAS), Shanghai, China*, volume 6461 of *Lecture Notes in Computer Science*, pages 376–391. Springer, 2010. ISBN 978-3-642-17163-5.

[18] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems. PhD thesis*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1996.

[19] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[20] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: Unified static analysis of context-dependence. In *International Colloquium on Automata, Languages and Programming (ICALP), Riga, Latvia*, pages 385–397. Springer, 2013.

[21] B. C. Pierce and M. Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, pages 511–530, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[22] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[23] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, ICFP '10, pages 157–168, New York, NY, USA, 2010. ISBN 978-1-60558-794-3.

[24] P. Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), New Haven, Connecticut*, volume 26, pages 255–273. ACM, 1991.

[25] D. A. Wright and C. A. Baker-Finch. Usage analysis with natural reduction types. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Workshop on Static Analysis (WSA) , Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 1993. ISBN 3-540-57264-3.

[26] H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 214–227. ACM, 1999.

[27] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Rome, Italy*, pages 295–314. Springer, 2013.

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \qquad \phi; \Phi \models \Gamma \sqsubseteq \Delta}{\phi; \Phi \mid \Gamma \vdash e : \sigma}\ (\sqsubseteq .\mathrm{L}) \qquad \frac{\phi; \Phi \mid \Gamma \vdash e : \sigma \qquad \phi; \Phi \models \sigma \sqsubseteq \tau}{\phi; \Phi \mid \Gamma \vdash e : \tau}\ (\sqsubseteq .\mathrm{R}) \qquad \frac{}{\phi; \Phi \mid \Gamma \vdash \mathrm{r} : \mathbb{R}}\ (\mathrm{Const}_{\mathbb{R}})$$

$$\frac{n = [\![S]\!]}{\phi; \Phi \mid \Gamma \vdash n : \mathbb{N}[S]}\ (\mathrm{Const}_{\mathbb{N}}) \qquad \frac{}{\phi; \Phi \mid \Gamma, x :_{[1]} \sigma \vdash x : \sigma}\ (\mathrm{Var}) \qquad \frac{\phi; \Phi \mid \Gamma_1 \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \sigma \otimes \tau}\ (\otimes I)$$

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \vdash e' : \mu \qquad R \neq \square}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{let}(x, y) = e \;\mathbf{in}\; e' : \mu}\ (\otimes E) \qquad \frac{\phi; \Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \mathbin{\&} \tau}\ (\&\ I)$$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma_1 \mathbin{\&} \sigma_2}{\phi; \Phi \mid \Gamma \vdash \pi_i\ e : \sigma_i}\ (\&\ E) \qquad \frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash e : \tau \qquad R \neq \square}{\phi; \Phi \mid \Gamma \vdash \lambda(x :_{[R]} \sigma).e : !_R\sigma \multimap \tau}\ (\multimap I)$$

$$\frac{\phi; \Phi \mid \Gamma \vdash e_1 : !_R\sigma \multimap \tau \qquad \phi; \Phi \mid \Delta \vdash e_2 : \sigma}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau}\ (\multimap E) \qquad \frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi, \Gamma}{\phi; \Phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma}\ (\forall I)$$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash e[S] : \sigma[S/i]}\ (\forall E) \qquad \frac{\phi; \Phi \mid \Gamma, x :_{[\infty]} \sigma \vdash e : \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash \mathbf{fix}\, x : \sigma.\ e : \sigma}\ (\mathrm{Fix}) \qquad \frac{\phi; \Phi \mid \Gamma \vdash e : \mathbb{N}[S]}{\phi; \Phi \mid \Gamma \vdash e + 1 : \mathbb{N}[S + 1]}\ (\mathrm{S}\ I)$$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \# R \qquad R \neq \square\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e \,\mathbf{return}\, \sigma \,\mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma}\ (\mathbb{N}\ E)$$

Figure 7. *DFuzz* $_\square$ Type Judgment

A. Differences Compared to Gaboardi et al. [10]

While we hew closely to the presentation of *DFuzz* in Gaboardi et al. [10], we make a few technical changes.

- The environment weakening operation $\Gamma \sqsubseteq \Gamma'$ in *DFuzz* allows the types to change. That is, a binding $x :_{[R]} \sigma \in \Gamma$ can be weakened to $x :_{[R']} \sigma'$ for $\sigma \sqsubseteq \sigma'$ two syntactically different types. We take a more restricted weakening rule, where the types must be syntactically the same; we are unaware of any programs that need the more general rule.
- We take the interpretation of $\infty \cdot 0$ to be ∞, rather than 0.
- We assume some additional type annotations in the source language, as discussed in Section 5

B. The *DFuzz* $_\square$ system

The first system has the goal to enjoy environment "uniformity", in the sense that sensitivity information in the environments may be missing. We denote such an assignment $x :_\square \sigma$. This is a subtle technical point for crucial to enable syntax-directed typability.

We modify subtyping for environments such that $\Gamma \sqsubseteq \Delta$ requires Γ, Δ to have the same domain. The new rule is:

$$\frac{\begin{array}{c}\forall(x_i :_{[R_i]} \sigma_i, x_i :_{[R'_i]} \sigma_i) \in (\Gamma, \Delta) \\ \mathrm{dom}(\Delta) = \mathrm{dom}(\Gamma) \qquad \phi; \Phi \models R_i \geq R'_i \vee R'_i = \square\end{array}}{\phi; \Phi \models \Gamma \sqsubseteq \Delta}\ \sqsubseteq\text{-Env}$$

This subsumes regular variable weakening. Environment operations must be aware of $\square$, with $\square + i = i$, $i \cdot \square = \square$ for the annotations.

Definition 22 (Box erasure)**.** *For any environment* Γ*, we define the* $\square$*-erasure operation* $|\Gamma| = \{x :_{[R]} \sigma \mid x :_{[R]} \sigma \in \Gamma \wedge R \neq \square\}$.

We introduce the $\square$ system in Figure 7.

We prove that derivations in a system with $\square$ are in direct correspondence with derivation in a system without it.

Lemma 23. *Assume* $\phi; \Phi \mid \Gamma \vdash e : \sigma$ *in the* $\square$ *system, then* $\phi; \Phi \mid |\Gamma| \vdash e : \sigma$ *in the system without it.*

Proof. By induction on the typing derivation. The base cases and cases where the environment is not modified are immediate. Subtyping on the left is proven by weakening.

The rest of cases are split in two:

- All cases featuring variables in the top rule, also have the condition $R \neq \square$, this is enough.
- For the cases involving environment operations, the proofs is completed by following properties:

$$|R \cdot \Gamma| = R \cdot |\Gamma| \qquad |\Gamma + \Delta| = |\Gamma| + |\Delta|$$

□

Lemma 24. *Assume* $\phi; \Phi \mid \Gamma \vdash e : \sigma$ *in the system without* $\square$*, then* $\phi; \Phi \mid \Gamma \vdash e : \sigma$ *in the system with it.*

Proof. The proof is mostly routine by induction on the derivation, but relies in the following fact of the $\Box$ system: $\phi; \Phi \mid \Gamma \vdash e : \sigma$ implies $\phi; \Phi \mid \Gamma, x :_{\Box} \tau \vdash e : \sigma$. Then, using this lemma we can adjust the environments so that subtyping goes through in the system with $\Box$. □

A $\Box$-elimination operation $R_{\Box\uparrow}$, which sends environment annotations to sensitivities will prove useful in the the syntax directed system. It is defined as $\Box_{\Box\uparrow} = 0$, $R_{\Box\uparrow} = R$ otherwise. Remember that $\Box$ doesn't belong to the sensitivity language, so any annotation that is used in places where a sensitivity is expected must be wrapped with $-_{\Box\uparrow}$.

Definition 25 (Extension to environments operations). *Operations on extended sensitivites that were extended to environments in a pointwise fashion, now must take into account the presence of* $\Box$.

- $\mathbf{max}(R_1, R_2)$ *operates now as* $\mathbf{max}(\Box, \Box) = \Box$, $\mathbf{max}(\Box, R) = R$, $\mathbf{max}(R, \Box) = R$, *the original term otherwise.*
- $\mathbf{sup}(i, R)$ *is extended in the natural way* $\mathbf{sup}(i, \Box) = \Box$, *the original term otherwise.*
- $\mathbf{case}(S, i, R_0, R_s)$ *operates now* $\mathbf{case}(S, i, \Box, \Box) = \Box$, $\mathbf{case}(S, i, R_0, R_s) = \mathbf{case}(S, i, R_{0\Box\uparrow}, R_{s\Box\uparrow})$ *otherwise.*

C. Subtyping Proofs

From now on we can consider only environments of similar length. We prove a few necessary facts about subtyping.

Lemma 26 (Environment manipulation). *Environment subtyping is preserved by addition and scalar multiplication. More formally:*

- *If* $\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \wedge \Delta \sqsubseteq \Delta'$, *then* $\phi; \Phi \models \Gamma + \Delta \sqsubseteq \Gamma' + \Delta'$; *and*
- *if* $\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \wedge R \geq R'$, *then* $\phi; \Phi \models R \cdot \Gamma \sqsubseteq R' \cdot \Gamma'$.

Proof. These follow from the interpretation of subtyping assertions. Note that the subtyping relation preserves the skeleton of the environments, thus making sure that the operations are always defined. □

Lemma 27 (Properties of extended sensitivities). *Extended sensitivities satisfy the following properties:*

- $\phi; \Phi \models R \geq \mathbf{max}(R_1, R_2)$ *if and only if* $\phi; \Phi \models R \geq R_1 \wedge R \geq R_2$;
- $\phi; \Phi \models R \geq \mathbf{sup}(i, R')$ *with* $i\#\phi$ *if and only if* $\phi, i; \Phi \models R \geq R'$; *and*
- $\phi; \Phi \models R \geq \mathbf{case}(S, i, R_0, R_s)$ *with* $i\#\phi$ *if and only if*

$$\phi; \Phi, S = 0 \models R \geq R_0 \quad \textit{and} \quad \phi, i; \Phi, S = i + 1 \models R \geq R_s.$$

As an immediate corollary, setting R *to be* $\mathbf{max}(R_1, R_2), \mathbf{sup}(i, R'), \mathbf{case}(S, i, R_0, R_s)$ *yields*

- $\phi; \Phi \models \mathbf{max}(R_1, R_2) \geq R_1 \wedge R \geq R_2$;
- $\phi, i; \Phi \models \mathbf{sup}(i, R') \geq R'$; *and*
- $\phi; \Phi, S = 0 \models \mathbf{case}(S, i, R_0, R_s) \geq R_0$ *and* $\phi, i; \Phi, S = i + 1 \models \mathbf{case}(S, i, R_0, R_s) \geq R_s$.

Proof. These follow from the interpretation of extended sensitivities. □

Lemma 28. *Suppose* $\phi, i : \kappa; \Phi \models \sigma \sqsubseteq \tau$ *and* $i\#\Phi$. *Then for any* $\phi \models S : \kappa$, *we have*

$$\phi; \Phi \models \sigma[S/i] \sqsubseteq \tau[S/i].$$

Proof. By induction on the subtype derivation. For the base cases, we know

$$\forall \phi, i : \kappa.\ (\Phi \Rightarrow R \geq R'),$$

and we need to prove

$$\forall \phi.\ (\Phi \Rightarrow R[S/i] \geq R'[S/i]),$$

but this is clear from the interpretation of R, R'. □

D. The Syntax-Directed system

The syntax-directed system is presented in Figure 8. It works over a uniform environment, using $\Box$ annotations to "mark", variables not occurring in the original *DFuzz* derivation.

We first prove the system sound with respect the non syntax-directed one.

Lemma 29 (Syntax-directed soundness). *If* $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$ *has a derivation, then* $\phi; \Phi \mid \Gamma \vdash e : \sigma$.

Proof. By induction on the derivation proving $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$.

Case: (Var)

$$\frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^{\bullet}), x :_{[1]} \sigma \vdash_{\mathcal{S}} x : \sigma} \ (\mathrm{Var})$$

Immediate, the same rule applies.

Case: $(\otimes I)$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash_{\mathcal{S}} (e_1, e_2) : \sigma \otimes \tau} \ (\otimes I)$$

Immediate by induction; the same rule applies.

$$\frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^\bullet) \vdash_{\mathcal{S}} \mathrm{r} : \mathbb{R}} \quad (\mathrm{Const}_{\mathbb{R}}) \qquad \frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma \vdash_{\mathcal{S}} x : \sigma} \quad (\mathrm{Var})$$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash_{\mathcal{S}} (e_1, e_2) : \sigma \otimes \tau} \quad (\otimes I) \qquad \frac{\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau \vdash_{\mathcal{S}} e' : \mu}{\phi; \Phi \mid \Gamma + \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \mathbf{\ in\ } e' : \mu} \quad (\otimes E)$$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma \mathbin{\&} \tau} \quad (\& \ I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma_1 \mathbin{\&} \sigma_2}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \pi_i e : \sigma_i} \quad (\& \ E)$$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \phi; \Phi \models R \geq R^\bullet{}_{\Box\uparrow}}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda(x :_{[R]} \sigma).\, e : !_R \sigma \multimap \tau} \quad (\multimap I) \qquad \frac{\begin{array}{c}\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1\ e_2 : \tau} \quad (\multimap E)$$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma \qquad i \text{ fresh in } \Phi}{\phi; \Phi \mid \mathbf{sup}(i, \Gamma) \vdash_{\mathcal{S}} \Lambda i : \kappa.\, e : \forall i : \kappa.\, \sigma} \quad (\forall I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \forall i : \kappa.\, \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e[S] : \sigma[S/i]} \quad (\forall E)$$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash_{\mathcal{S}} \mathbf{fix}\, x : \sigma.\, e : \sigma} \quad (\mathrm{Fix})$$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma_0 \vdash_{\mathcal{S}} e_0 : \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n :_{[R]} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) + \mathbf{case}(S, i, 0, R_{\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\ 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\ E)$$

$$\mathrm{Ectx}(\Gamma^\bullet) := \Delta \quad \text{with} \quad \left\{ \begin{array}{lll} \mathrm{dom}(\Gamma^\bullet) & = \mathrm{dom}(\Delta) & \\ \Delta(b) & \equiv \_ :_{\Box} \_ & \text{for all } b \in \mathrm{dom}(\Gamma^\bullet) \end{array} \right.$$

Figure 8. *DFuzz* Type Judgment, Syntax-directed Version

Case: $(\otimes E)$

$$\frac{\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau \vdash_{\mathcal{S}} e' : \mu}{\phi; \Phi \mid \Gamma + \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \mathbf{\ in\ } e' : \mu} \quad (\otimes E)$$

By induction, we have

$$\phi; \Phi \mid \Delta \vdash e : \sigma \otimes \tau \quad \text{and} \quad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \sigma \vdash e' : \mu$$

By Lemma 27, $\phi; \Phi \models \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \geq R_{i\Box\uparrow}$ for $i = 1, 2$. Abbreviating $R^\bullet := \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow})$ and applying weakening we have:

$$\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma, y :_{[R^\bullet]} \tau \vdash e' : \mu$$

with $R^\bullet \neq \Box$ so we have exactly what we need to apply $(\otimes E)$.

Case: $(\& \ I)$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma \mathbin{\&} \tau} \quad (\& \ I)$$

By induction, we have

$$\phi; \Phi \mid \Gamma_1 \vdash e_1 : \sigma \quad \text{and} \quad \phi; \Phi \mid \Gamma_2 \vdash e_2 : \tau.$$

By Lemma 27, we have

$$\phi; \Phi \models \mathbf{max}(\Gamma_1, \Gamma_2) \sqsubseteq \Gamma_1 \quad \text{and} \quad \phi; \Phi \models \mathbf{max}(\Gamma_1, \Gamma_2) \sqsubseteq \Gamma_2.$$

By weakening, we can derive

$$\phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash e_1 : \sigma \quad \text{and} \quad \phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash e_2 : \tau,$$

when we can conclude by $(\& \ I)$.

Case: $(\& \ E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma_1 \mathbin{\&} \sigma_2}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \pi_i e : \sigma_i} \quad (\& \ E)$$

Immediate; the same rule applies.

Case: $(\multimap I)$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \phi; \Phi \models R \geq R^\bullet{}_{\Box\uparrow}}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda(x :_{[R]} \sigma).\, e : !_R \sigma \multimap \tau} \quad (\multimap I)$$

By induction, we have

$$\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash e : \tau$$

and we know $R \neq \Box$ and:

$$\phi; \Phi \models R \geq R^\bullet.$$

By weakening, we have

$$\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \tau,$$

and we can conclude by $(\multimap I)$.

Case: $(\multimap E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1\ e_2 : \tau} \ (\multimap E)$$

By induction, we have

$$\phi; \Phi \mid \Gamma \vdash e_1 : !_R \sigma \multimap \tau \quad \text{and} \quad \phi; \Phi \mid \Delta \vdash e_2 : \sigma'$$

and we also know

$$\phi; \Phi \models \sigma' \sqsubseteq \sigma.$$

By subtyping on the right, we can derive

$$\phi; \Phi \mid \Delta \vdash e_2 : \sigma,$$

and we can conclude with $(\multimap E)$.

Case: $(\forall I)$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma \qquad i \text{ fresh in } \Phi}{\phi; \Phi \mid \mathbf{sup}(i, \Gamma) \vdash_{\mathcal{S}} \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma} \ (\forall I)$$

By induction, we have

$$\phi; i : \kappa; \Phi \mid \Gamma \vdash e : \sigma$$

and i fresh in Φ. By Lemma 27, we have

$$\phi; \Phi \models \mathbf{sup}(i, \Gamma) \sqsubseteq \Gamma,$$

and so by weakening, we have

$$\phi, i : \kappa; \Phi \mid \mathbf{sup}(i, \Gamma) \vdash e : \sigma.$$

Now, we can conclude with $(\forall I)$.

Case: $(\forall E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e[S] : \sigma[S/i]} \ (\forall E)$$

Immediate; the same rule applies.

Case: (Fix)

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash_{\mathcal{S}} \mathbf{fix}\, x : \sigma.\ e : \sigma} \ (\text{Fix})$$

By induction; we have

$$\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \sigma'.$$

But we also have $\phi; \Phi \models \sigma' \sqsubseteq \sigma$. By subtyping, we get

$$\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \sigma$$

and we can conclude with (Fix).

Case: $(\mathbb{N}\ E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma_0 \vdash_{\mathcal{S}} e_0 : \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n :_{[R]} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) + \mathbf{case}(S, i, 0, R_{\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\ 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \ (\mathbb{N}\ E)$$

By induction, we have

$$\begin{aligned}&\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \\ &\phi; \Phi, S = 0 \mid \Gamma_0 \vdash e_0 : \sigma_0 \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n : !_R \mathbb{N}[i] \vdash e_s : \sigma_s.\end{aligned}$$

By Lemma 27, we have

$$\begin{aligned}&\phi; \Phi, S = 0 \models \mathbf{case}(S, i, \Gamma_0, \Gamma_s) \sqsubseteq \Gamma_0 \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \mathbf{case}(S, i, \Gamma_0, \Gamma_s) \sqsubseteq \Gamma_s \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \mathbf{case}(S, i, 0, R_{\Box\uparrow}) \geq R_{\Box\uparrow}\end{aligned}$$

with $R_{\Box\uparrow} \neq \Box$, and we also know

$$\begin{aligned} &\phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma. \end{aligned}$$

By subtyping on the left and right, we have

$$\begin{aligned} &\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \\ &\phi; \Phi, S = 0 \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) \vdash e_0 : \sigma \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s), n : !_{R^\bullet} \mathbb{N}[i] \vdash e_s : \sigma, \end{aligned}$$

where $R^\bullet = \mathbf{case}(S, i, 0, R_{\Box\uparrow})$. We can then conclude by $(\mathbb{N}\ E)$.

$$\frac{\begin{array}{c} \phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \# R \qquad R \neq \Box \end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\; 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \; (\mathbb{N}\ E)$$

□

We now prove completeness, that is to say, for every derivation in the original system, the syntax-directed one will have a derivation, possibly even a better from a subtype point of view.

We first need a few auxiliary lemmas:

Lemma 30. *Suppose that $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$ is derivable. Then, for any logically equivalent Ψ such that $\phi \models \Phi \Leftrightarrow \Psi$, there is a derivation of $\phi; \Psi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$ with the same height.*

Proof. By induction on the derivation. The only place the constraint environment is used is when checking constraints of the form

$$\phi; \Phi \models R \geq R'.$$

But since Ψ and Φ are logically equivalent, we evidently have

$$\phi; \Psi \models R \geq R'$$

as well. □

Lemma 31 (Inner Weakening for the Syntax-directed system)**.** *Assume a derivation $\Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \tau$, a type σ' such that $\sigma' \sqsubseteq \sigma$. Then, there exists a type τ' and a derivation $\Gamma, x :_{[R]} \sigma' \vdash_{\mathcal{S}} e : \tau'$ such that $\tau' \sqsubseteq \tau$.*

Proof. By induction over the typing derivation. The base cases are immediate. In the induction hypothesis we get to pick the appropriate type and we get a better type in all the cases. □

Lemma 32 (Syntax-directed completeness)**.** *If $\phi; \Phi \mid \Gamma \vdash e : \sigma$ has a derivation, then there exists Γ', σ' such that $\phi; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e : \sigma'$ has a derivation, $\phi; \Phi \models \Gamma \sqsubseteq \Gamma'$, $\phi; \Phi \models \sigma' \sqsubseteq \sigma$.*

Proof. By induction on the derivation proving $\phi; \Phi \mid \Gamma \vdash e : \sigma$.

Case: $(\sqsubseteq .\mathrm{L})$

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \qquad \phi; \Phi \models \Gamma \sqsubseteq \Delta}{\phi; \Phi \mid \Gamma \vdash e : \sigma} \; (\sqsubseteq .\mathrm{L})$$

Immediate, by induction; the desired environment is Δ.

Case: $(\sqsubseteq .\mathrm{R})$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma \qquad \phi; \Phi \models \sigma \sqsubseteq \tau}{\phi; \Phi \mid \Gamma \vdash e : \tau} \; (\sqsubseteq .\mathrm{R})$$

Immediate, by induction; the desired subtype is σ.

Case: (Var)

$$\frac{}{\phi; \Phi \mid \Gamma, x :_{[1]} \sigma \vdash x : \sigma} \; (\mathrm{Var})$$

Immediate; the same rule applies.

Case: $(\otimes I)$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \sigma \otimes \tau} \; (\otimes I)$$

By induction, we have $\Gamma'_1, \Gamma'_2, \sigma', \tau'$ such that

$$\phi; \Phi \models \Gamma_1 \sqsubseteq \Gamma'_1 \wedge \Gamma_2 \sqsubseteq \Gamma'_2 \quad \text{and} \quad \phi; \Phi \models \sigma' \sqsubseteq \sigma \wedge \tau' \sqsubseteq \tau$$

and derivations

$$\phi; \Phi \mid \Gamma'_1 \vdash_{\mathcal{S}} e_1 : \sigma' \quad \text{and} \quad \phi; \Phi \mid \Gamma'_2 \vdash_{\mathcal{S}} e_2 : \tau'.$$

Then we can conclude by $(\otimes I)$, since Lemma 26 shows

$$\phi; \Phi \models \Gamma_1 + \Gamma_2 \sqsubseteq \Gamma'_1 + \Gamma'_2 \quad \text{and} \quad \phi; \Phi \models \sigma' \otimes \tau' \sqsubseteq \sigma \otimes \tau.$$

Case: $(\otimes E)$

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \vdash e' : \mu \qquad R \neq \square}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{let}(x, y) = e \mathbf{\ in\ } e' : \mu} \quad (\otimes E)$$

By induction and inversion on the subtype relation, we have $\Delta', \Gamma', \sigma', \sigma'', \tau', \tau'', \mu', R_1, R_2$ such that

$$\begin{aligned} &\phi; \Phi \models \Delta \sqsubseteq \Delta' \\ &\phi; \Phi \models \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \sqsubseteq \Gamma', x :_{[R_1]} \sigma'', y :_{[R_2]} \tau'' \\ &\phi; \Phi \models \sigma' \sqsubseteq \sigma \wedge \tau' \sqsubseteq \tau \end{aligned}$$

this implies $\sigma' \sqsubseteq \sigma''$, $\tau' \sqsubseteq \tau''$, $R \geq R_{1\square\uparrow}$, and $R \geq R_{2\square\uparrow}$. We have derivations:

$$\phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e : \sigma' \otimes \tau' \quad \text{and} \quad \phi; \Phi \mid \Gamma', x :_{[R_1]} \sigma'', y :_{[R_2]} \tau'' \vdash_{\mathcal{S}} e' : \mu'$$

By Lemma 31, we have a derivation:

$$\phi; \Phi \mid \Gamma', x :_{[R_1]} \sigma', y :_{[R_2]} \tau' \vdash_{\mathcal{S}} e' : \mu''$$

with $\mu'' \sqsubseteq \mu'$. Hence, we can produce a syntax-directed derivation now:

$$\phi; \Phi \mid \Gamma' + \mathbf{max}(R'_{1\square\uparrow}, R'_{2\square\uparrow}) \cdot \Delta' \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \mathbf{\ in\ } e' : \mu''.$$

By Lemma 27, we have that $\phi; \Phi \models R \geq \mathbf{max}(R'_{1\square\uparrow}, R_{2\square\uparrow})$ and by Lemma 26,

$$\phi; \Phi \models \Gamma + R \cdot \Delta \sqsubseteq \Gamma' + \mathbf{max}(R'_{1\square\uparrow}, R_{2\square\uparrow}) \cdot \Delta',$$

so we are done: the environment $\Gamma' + \mathbf{max}(R'_{1\square\uparrow}, R'_{2\square\uparrow}) \cdot \Delta'$ and subtype τ'' suffice.

Case: $(\& I)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \mathbin{\&} \tau} \quad (\& I)$$

By induction, there exists

$$\begin{aligned} \phi; \Phi \models \Gamma \sqsubseteq \Gamma'_1 \quad &\text{and} \quad \phi; \Phi \models \Gamma \sqsubseteq \Gamma'_2 \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma \quad &\text{and} \quad \phi; \Phi \models \tau' \sqsubseteq \tau \end{aligned}$$

such that

$$\phi; \Phi \mid \Gamma'_1 \vdash_{\mathcal{S}} e_1 : \sigma' \quad \text{and} \quad \phi; \Phi \mid \Gamma'_2 \vdash_{\mathcal{S}} e_2 : \tau'.$$

By $(\& I)$, we have

$$\phi; \Phi \mid \mathbf{max}(\Gamma'_1, \Gamma'_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma' \mathbin{\&} \tau'.$$

We are done, since by Lemmas 26 and 27,

$$\phi; \Phi \models \sigma' \mathbin{\&} \tau' \sqsubseteq \sigma \mathbin{\&} \tau \quad \text{and} \quad \phi; \Phi \models \Gamma \sqsubseteq \mathbf{max}(\Gamma'_1, \Gamma'_2) \sqsubseteq \Gamma'_i.$$

So, the desired environment is $\mathbf{max}(\Gamma'_1, \Gamma'_2)$, and the desired subtype is $\sigma' \mathbin{\&} \tau'$.

Case: $(\& E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma_1 \mathbin{\&} \sigma_2}{\phi; \Phi \mid \Gamma \vdash \pi_i \; e : \sigma_i} \quad (\& E)$$

Immediate, by induction.

Case: $(\multimap I)$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash e : \tau \qquad R \neq \square}{\phi; \Phi \mid \Gamma \vdash \lambda(x :_{[R]} \sigma).e : !_R \sigma \multimap \tau} \quad (\multimap I)$$

By induction, there exists

$$\phi; \Phi \models \Gamma, x :_{[R]} \sigma \sqsubseteq \Gamma', x : !_{R'} \sigma \quad \text{and} \quad \phi; \Phi \models \tau' \sqsubseteq \tau$$

such that

$$\phi; \Phi \mid \Gamma', x :_{[R']} \sigma \vdash_{\mathcal{S}} e : \tau'.$$

By inversion on the subtype relation, we have

$$\phi; \Phi \models R \geq R'_{\square\uparrow} \wedge \tau' \sqsubseteq \tau.$$

and we are done, since

$$\phi; \Phi \models !_{R'_{\square\uparrow}} \sigma \multimap \tau' \sqsubseteq !_R \sigma \multimap \tau \quad \text{and} \quad \phi; \Phi \models \Gamma \sqsubseteq \Gamma'.$$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \phi; \Phi \models R \geq R^{\bullet}{}_{\square\uparrow}}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda(x :_{[R]} \sigma).\, e : !_R \sigma \multimap \tau} \quad (\multimap I)$$

Case: $(\multimap E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e_1 : !_R \sigma \multimap \tau \qquad \phi; \Phi \mid \Delta \vdash e_2 : \sigma}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau} \quad (\multimap E)$$

By induction, there exists $\Gamma', \Delta', R', \sigma', \tau', \sigma''$ such that

$$\begin{aligned} &\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \\ &\phi; \Phi \models \Delta \sqsubseteq \Delta' \\ &\phi; \Phi \models !_{R'} \sigma' \multimap \tau' \sqsubseteq !_R \sigma \multimap \tau \\ &\phi; \Phi \models \sigma'' \sqsubseteq \sigma, \end{aligned}$$

and derivations

$$\phi; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e_1 : !_{R'} \sigma' \multimap \tau' \quad \text{and} \quad \phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e_2 : \sigma''.$$

By inversion on the subtype relation, we have

$$\phi; \Phi \models R \geq R' \quad \text{and} \quad \phi; \Phi \models \sigma'' \sqsubseteq \sigma \sqsubseteq \sigma' \quad \text{and} \quad \phi; \Phi \models \tau' \sqsubseteq \tau.$$

By Lemma 27, the environment $\Gamma' + R' \cdot \Delta'$ and subtype τ' suffice.

Case: $(\forall I)$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi, \Gamma}{\phi; \Phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma} \quad (\forall I)$$

By induction, there exist

$$\phi, i : \kappa; \Phi \models \sigma' \sqsubseteq \sigma \quad \text{and} \quad \phi, i : \kappa; \Phi \models \Gamma \sqsubseteq \Gamma'$$

such that

$$\phi, i : \kappa; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e : \sigma'.$$

Thus, we have the derivation

$$\phi; \Phi \mid \mathbf{sup}(i, \Gamma') \vdash_{\mathcal{S}} \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma'$$

and

$$\phi; \Phi \models \forall i : \kappa.\ \sigma' \sqsubseteq \forall i : \kappa.\ \sigma.$$

By Lemma 27, we actually have

$$\phi; \Phi \models \Gamma \sqsubseteq \mathbf{sup}(i, \Gamma') \sqsubseteq \Gamma',$$

so the environment $\mathbf{sup}(i, \Gamma')$ and subtype $\forall i : \kappa.\ \sigma'$ suffices.

Case: $(\forall E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash e[S] : \sigma[S/i]} \quad (\forall E)$$

By induction, there exists

$$\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \quad \text{and} \quad \phi; \Phi \models \forall i : \kappa.\ \sigma' \sqsubseteq \forall i : \kappa.\ \sigma$$

such that

$$\phi; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e : \forall i : \kappa.\ \sigma'.$$

So, we have a derivation

$$\phi; \Phi \mid \Gamma'' \vdash_{\mathcal{S}} e[S/i] : \sigma'[S/i].$$

By Lemma 28,

$$\phi; \Phi \models \sigma'[S/i] \sqsubseteq \sigma[S/i],$$

so the environment Γ' and subtype $\sigma'[S/i]$ suffice.

Case: (Fix)

$$\frac{\phi; \Phi \mid \Gamma, x :_{[\infty]} \sigma \vdash e : \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash \mathbf{fix}\, x : \sigma.\ e : \sigma} \quad (\text{Fix})$$

By induction, we have

$$\phi; \Phi \models \Gamma, x : !_\infty \sigma \sqsubseteq \Gamma', x : !_R \sigma \quad \text{and} \quad \phi; \Phi \models \sigma' \sqsubseteq \sigma$$

such that

$$\phi; \Phi \mid \Gamma', x : !_R \sigma \vdash_{\mathcal{S}} e : \sigma'.$$

We can then conclude by (Fix): the desired environment is $\infty \cdot \Gamma'$ and the desired type is σ.

Case: $(\mathbb{N}\ E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \# R \qquad R \neq \Box\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\ 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\ E)$$

By induction, there exists

$$\phi; \Phi \models \Delta \sqsubseteq \Delta' \quad \text{and} \quad \phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e : \mathbb{N}[S'] \quad \text{and} \quad \phi; \Phi \models \mathbb{N}[S'] \sqsubseteq \mathbb{N}[S].$$

By inversion, $\phi; \Phi \models S = S'$. Also by induction,

$$\begin{aligned} &\phi; \Phi, S = 0 \models \Gamma \sqsubseteq \Gamma'_0 \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \Gamma, n : !_R \mathbb{N}[i] \sqsubseteq \Gamma'_s, n : !_{R'} \mathbb{N}[i] \\ &\phi; \Phi, S = 0 \models \sigma'_0 \sqsubseteq \sigma \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma'_s \sqsubseteq \sigma \end{aligned}$$

such that

$$\begin{aligned} &\phi; \Phi, S = 0 \mid \Gamma'_0 \vdash_{\mathcal{S}} e_0 : \sigma'_0 \\ &\phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma'_s, n : !_{R'} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma'_s. \end{aligned}$$

By Lemma 30, we also have derivations

$$\begin{aligned} &\phi; \Phi, S' = 0 \mid \Gamma'_0 \vdash_{\mathcal{S}} e_0 : \sigma'_0 \\ &\phi, i : \mathrm{n}; \Phi, S' = i + 1 \mid \Gamma'_s, n : !_{R'} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma'_s \end{aligned}$$

since $\phi; \Phi \models S = S'$.
Hence, we have a derivation

$$\begin{aligned} &\phi; \Phi \mid \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) + R^\bullet \cdot \Delta' \\ &\vdash_{\mathcal{S}} \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma, \end{aligned}$$

where $R^\bullet$ is $\mathbf{case}(S', i, 0, {R'}_{\square\uparrow})$. We have

$$\begin{aligned} &\phi; \Phi, S' = 0 \models \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) \sqsubseteq \Gamma'_0 \\ &\phi, i : \mathrm{n}; \Phi, S' = i + 1 \models \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) \sqsubseteq \Gamma'_s \end{aligned}$$

so by Lemma 27

$$\phi; \Phi \models \Gamma \sqsubseteq \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s),$$

and

$$\phi, i : \mathrm{n}; \Phi, S' = i + 1 \models R \geq R^\bullet \geq {R'}_{\square\uparrow} \quad \text{and} \quad \phi, \Phi \models R \geq R^\bullet$$

thanks to $R \neq \square$.
By weakening, we have

$$\begin{aligned} &\phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e : \mathbb{N}[S'] \\ &\phi; \Phi, S = 0 \mid \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) \vdash_{\mathcal{S}} e_0 : \sigma \\ &\phi, i : \mathrm{n}; \Phi, S' = i + 1 \mid \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s), n : !_{R^\bullet} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma, \end{aligned}$$

so we can conlude with $(\mathbb{N}\ E)$. The environment $\mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) + R^\bullet \cdot \Delta'$ and type σ suffice (recall that $\phi; \Phi \models R \geq R^\bullet$, and $\phi; \Phi \models R \cdot \Delta \sqsubseteq R^\bullet \cdot \Delta'$ by Lemma 26).

□

D.1 Algorithm Proofs

Theorem 33 (Algorithmic Soundness)**.** *Suppose $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$. Then, there is a derivation of $\phi; \Phi; \Gamma \vdash_{\mathcal{S}} e : \sigma$.*

Proof. By induction on the algorithmic derivations we see that every algorithmic step has an exact correspondence with a syntax-directed derivation. We do a few representative cases:

Case (Var)

$$\frac{}{\phi; \Phi; \Gamma^\bullet, x : \sigma; x \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma; \sigma} \ (\text{Var})$$

$$\frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma \vdash_{\mathcal{S}} x : \sigma} \ (\text{Var})$$

Case $(\multimap E)$

$$\frac{\begin{array}{c} \phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma; !_R \sigma \multimap \tau \\ \phi; \Phi; \Delta^\bullet; e_2 \Longrightarrow \Delta; \sigma' \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma \end{array}}{\phi; \Phi; \Gamma^\bullet; e_1\ e_2 \Longrightarrow \Gamma + R \cdot \Delta; \tau} \ (\multimap E)$$

$$\frac{\begin{array}{c} \phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma \end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1\ e_2 : \tau} \ (\multimap E)$$

Case $(\otimes E)$

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Delta; \sigma \otimes \tau \\ \phi; \Phi; \Gamma^{\bullet}, x : \sigma, y : \tau; e' \Longrightarrow \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau; \mu\end{array}}{\phi; \Phi; \Gamma^{\bullet}; \mathbf{let}(x, y) = e \;\mathbf{in}\; e' \Longrightarrow \Gamma + \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \cdot \Delta; \mu} \quad (\otimes E)$$

$$\frac{\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau \vdash_{\mathcal{S}} e' : \mu}{\phi; \Phi \mid \Gamma + \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \;\mathbf{in}\; e' : \mu} \quad (\otimes E)$$

□

Theorem 34 (Algorithmic Completeness)**.** *Suppose* $\phi; \Phi; \Gamma \vdash_{\mathcal{S}} e : \sigma$ *is derivable. Then* $\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Gamma; \sigma$*.*

Proof. By induction on the syntax-directed derivation. The proof is mostly direct, we show a few representative cases.

Case $(\multimap E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1 \; e_2 : \tau} \quad (\multimap E)$$

By induction, we have derivations

$$\phi; \Phi; \Gamma^{\bullet}; e_1 \Longrightarrow \Gamma; !_R \sigma \multimap \tau \quad \text{and} \quad \phi; \Phi; \Delta^{\bullet}; e_2 \Longrightarrow \Delta; \sigma'.$$

Note that $\Gamma^{\bullet} = \Delta^{\bullet}$ for the syntax-directed derivation to be defined, so we can apply the algorithmic rule $(\multimap E)$:

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^{\bullet}; e_1 \Longrightarrow \Gamma; !_R \sigma \multimap \tau \\ \phi; \Phi; \Delta^{\bullet}; e_2 \Longrightarrow \Delta; \sigma' \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi; \Gamma^{\bullet}; e_1 \; e_2 \Longrightarrow \Gamma + R \cdot \Delta; \tau} \quad (\multimap E)$$

Case (Fix)

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash_{\mathcal{S}} \mathbf{fix}\, x : \sigma.\; e : \sigma} \quad (\text{Fix})$$

By induction, we have

$$\phi; \Phi; \Gamma^{\bullet}, x : \sigma; e \Longrightarrow \Gamma, x :_{[R]} \sigma; \sigma'$$

and we can apply the algorithm rule (Fix):

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^{\bullet}, x : \sigma; e \Longrightarrow \Gamma, x :_{[R]} \sigma; \sigma' \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi; \Gamma^{\bullet}; \mathbf{fix}\, x : \sigma.\; e : \sigma \Longrightarrow \infty \cdot \Gamma; \sigma} \quad (\text{Fix})$$

Case $(\otimes E)$

$$\frac{\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau \vdash_{\mathcal{S}} e' : \mu}{\phi; \Phi \mid \Gamma + \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \;\mathbf{in}\; e' : \mu} \quad (\otimes E)$$

We know that $\Gamma^{\bullet} = \Delta^{\bullet}$. By induction, we know that:

$$\begin{array}{l}\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Delta; \sigma'_1 \otimes \sigma'_2 \\ \phi; \Phi; \Gamma^{\bullet}, x_1 : \sigma_1, x_2 : \sigma_2; e' \Longrightarrow \Gamma, x :_{[R_1]} \sigma_1, y :_{[R_2]} \sigma_2; \tau\end{array}$$

and we know $\phi; \Phi \models \sigma'_1 \sqsubseteq \sigma_1 \wedge \sigma'_2 \sqsubseteq \sigma_2$, so we apply the algorithmic case $(\otimes E)$:

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Delta; \sigma \otimes \tau \\ \phi; \Phi; \Gamma^{\bullet}, x : \sigma, y : \tau; e' \Longrightarrow \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau; \mu\end{array}}{\phi; \Phi; \Gamma^{\bullet}; \mathbf{let}(x, y) = e \;\mathbf{in}\; e' \Longrightarrow \Gamma + \mathbf{max}(R_{1\Box\uparrow}, R_{2\Box\uparrow}) \cdot \Delta; \mu} \quad (\otimes E)$$

Case $(\mathbb{N}\; E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma_0 \vdash_{\mathcal{S}} e_0 : \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n :_{[R]} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) + \mathbf{case}(S, i, 0, R_{\Box\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{case}\, e \,\mathbf{return}\, \sigma \,\mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\; E)$$

We know that $\Gamma^{\bullet} = \Delta^{\bullet}$. By induction, we know that:

$$\begin{array}{l}\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Delta; \mathbb{N}[S] \\ \phi; \Phi, S = 0; \Gamma^{\bullet}; e_0 \Longrightarrow \Gamma_0; \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1; \Gamma^{\bullet}, x : \mathbb{N}[i]; e_s \Longrightarrow \Gamma_s, x :_{[R']} \mathbb{N}[i]; \sigma_s\end{array}$$

and we know

$$\phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \quad \text{and} \quad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma.$$

We can conclude with the algorithmic rule ($\mathbb{N}\ E$):

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \mathbb{N}[S] \qquad \phi; \Phi, S = 0; \Gamma^\bullet; e_0 \Longrightarrow \Gamma_0; \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1; \Gamma^\bullet, x : \mathbb{N}[i]; e_s \Longrightarrow \Gamma_s, x :_{[R']} \mathbb{N}[i]; \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\ 0 \mapsto e_0 \mid x_{[i]} + 1 \mapsto e_s \\ \Longrightarrow \mathbf{case}(S, \Gamma_0, i, \Gamma_s) + \mathbf{case}(S, 0, i, R'_{\square\uparrow}) \cdot \Delta; \sigma\end{array}} \ (\mathbb{N}\ E)$$

□

E. Minimal Types

Lemma 35. DFuzz *does not have minimal types.*

Proof. Using dependent recursion, we can define a function $\mathbf{use} : \forall i : \mathrm{n}.\ !_0\mathbb{N}[i] \multimap\ !_i\mathbb{R} \multimap \mathbb{R}$ that multiplies a real number by a natural number. Consider the following term e:

$$\Lambda i : \mathrm{n}.\ \lambda e : \mathbb{N}[i], x : \mathbb{R}.\ \langle x, \mathbf{use}[i]\ e\ x + \mathbf{use}[i]\ e\ x \rangle.$$

Evidently, the minimal type should have the form

$$\emptyset; \emptyset \mid \emptyset \vdash e : \forall i : \mathrm{n}.\ !_0\mathbb{N}[i] \multimap\ !_q\mathbb{R} \multimap \mathbb{R} \mathbin{\&} \mathbb{R}$$

for some sensitivity expression q. What should q be? Note that $q(i)$ can be *a priori* a polynomial in i with positive, real coefficients. By inspecting the typing rules, we find that

$$i : \mathrm{n}; \emptyset \models q(i) \geq 1 \wedge q(i) \geq 2i.$$

Furthermore, the subtyping judgments show that

$$\forall i : \mathrm{n}.\ !_0\mathbb{N}[i] \multimap\ !_a\mathbb{R} \multimap \mathbb{R} \mathbin{\&} \mathbb{R} \sqsubseteq \forall i : \mathrm{n}.\ !_0\mathbb{N}[i] \multimap\ !_b\mathbb{R} \multimap \mathbb{R} \mathbin{\&} \mathbb{R}$$

is equivalent to $i : \mathrm{n}; \emptyset \models a \leq b$. Suppose that $q(i)$ is the minimal such polynomial for the sensitivity in the type of e. If the degree of q is strictly greater than 1, then the polynomial $2i + 1$ satisfies $2i + 1 \geq 1 \wedge 2i + 1 \geq 2i$, and is eventually smaller than q for large i (since q has higher degree and has non-negative coefficients).

On the other hand, q can't have degree 0 since it must be larger than $2i$ for all i. If q has degree 1, then its leading coefficient must be at least 2. Now, the polynomial $i^2 + 1$ satisfies $i^2 + 1 \geq 1 \wedge i^2 + 1 \geq 2i$. Finally, note

$$q \geq 2i + 1 \geq i^2 + 1$$

for $i \in \{0, 1\}$. Hence, there is no minimal sensitivity q, and hence no minimal type for e. □

F. Auxiliary Lemmas

Lemma 36 (Standard Annotations)**.** *Assume annotations in a term e range over regular sensitivities and $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$. Then:*

- *σ has no extended sensitivities; and*
- *all the constraints are of the form $\phi; \Phi \models R \geq R'$ where R is a standard sensitivity term.*

This directly implies Lemma 11.

Proof. The first point is clear by inspecting the rules in Figure 8: by induction, the type of any expression has only regular sensitivities. The second point is also clear: in all subtype checks in Figure 8, both types have no extended sensitivities by the first point. The only place where we check against an extended sensitivity is in rule ($\multimap I$), with constraint

$$\phi; \Phi \models R \geq R'.$$

Here, the R is a standard sensitivity term since it is an annotation, but the R' may be an extended sensitivity. □

Lemma 37 (Constraint Simplification)**.** *Suppose $Q \mapsto Q'$, and suppose $\phi \vdash Q$ and $\phi \vdash Q'$. Then, for any standard valuation $\rho \in \mathsf{val}(\phi)$, we have $(\!|Q|\!)_\rho = (\!|Q'|\!)_\rho$.*

Proof. By induction on the derivation of $Q \mapsto Q'$. The cases Plus, Mult and Red are immediate by induction. The other cases all follow by the semantics of $\mathbf{club}$.

Case Flat: The semantics of Q under valuation ρ is equivalent to the larger of

$$\max_{\rho'}\{\max_{i,\rho''}\{[\![R_i]\!]_{\rho \cup \rho' \cup \rho''} \mid \phi_i; \Phi_i \models \rho'\} \mid \phi; \Phi \models \rho'\}$$

and $N = (\!|\mathbf{club}\{V\}|\!)_\rho$. The first expression can be seen to be

$$M = \max_{i,\rho',\rho''}\{[\![R_i]\!]_{\rho \cup \rho' \cup \rho''} \mid \phi, \phi_i; \Phi \wedge \Phi_i \models \rho', \rho''\},$$

and the semantics of Q' under the valuation can be seen to be $\max(M, N)$, as desired.

Case CPlus: The interpretation of Q under valuation ρ is

$$\max_i \max\{[\![R_i]\!]_{\rho\cup\rho_i} \mid \phi_i;\Phi_i \models \rho_i\} + \max_j \max\{[\![R'_j]\!]_{\rho\cup\rho'_j} \mid \phi'_j;\Phi'_j \models \rho'_j\}$$

The first maximum is achieved at some i^*, and the second maximum is achieved at j^*. Then,

$$\max\{[\![R_{i^*}]\!]_{\rho\cup\rho_i} \mid \phi_{i^*};\Phi_{i^*} \models \rho_i\} + \max\{[\![R'_{j^*}]\!]_{\rho\cup\rho'_j} \mid \phi'_{j^*};\Phi'_{j^*} \models \rho'_j\}$$

is at most

$$\max\{[\![R_{i^*} + R'_{j^*}]\!]_{\rho\cup\rho_i\cup\rho'_j} \mid \phi_{i^*},\phi'_{j^*};\Phi_{i^*} \wedge \Phi'_{j^*} \models \rho_i,\rho'_j\} \leq (\!|\mathbf{club}\{(\phi_i \cup \phi'_j;\Phi_i \wedge \Phi'_j;R_i + R'_j)\}_{ij}|\!)_\rho$$

since ϕ_{i^*}, ϕ_{j^*} are assumed to be disjoint. For the reverse direction, consider the semantics of Q':

$$\max_{ij} \max\{[\![R_i + R'_j]\!]_{\rho\cup\rho_i\cup\rho'_j} \mid \phi_i,\phi'_j;\Phi_i \wedge \Phi'_j \models \rho_i,\rho'_j\}$$

If there are no valuations such that $\phi_i \cup \phi'_j;\Phi_i \wedge \Phi'_j \models \rho_i,\rho'_j$, then we are done (we've defined the max of an empty set to be 0). If the maximum is achieved at some ρ,ρ' at i^*, j^*, then we know $\phi_{i^*};\Phi_{i^*} \models \rho$ and $\phi_{j^*};\Phi_{j^*} \models \rho'$, when the maximum is at most $(\!|Q|\!)_\rho$.

Case CMult: This case follows like the previous case.

□