Problem Statement

Our goal is to implement a language for differential privacy where researchers/analysts can build new differential privacy techniques into their programs.

To solve the above problem, the PLAID lab at UVM created the programming language: Duet. This is a general purpose programming language with the rules or differential privacy built into the type system. It is therefore very important that the type system correctly follows the fundamental properties and promises of differential privacy. We would like to prove the correctness of Duet's type system. However, writing a proof in english/standard math can be error prone, so we plan to use the proof assistant Agda to model and write our proof of correctness for Duet's type system.

The general structure of the proof of the fundamental property of metric preservation in contextual Duet is as follows:

- formalization of the core language: expressions (terms), types, values and environments.
- formalization of the language rules: ground truth dynamic semantics, typing judgements and probabilistic semantics.
- formalization of the logical relations of the language.
- necessary lemmas and the main proof body.

So far the progress made has been making a proof for the correctness of Duet 1 and a proof for the correctness of Duet 2 following the above process.

The proof for Duet 1 found an issue with the semantics for the treatment of case splitting expressions as well as generally finding correctness for most of the Duet type system. The proof for Duet 2 has to consider more complexity in the type system, as Duet 2 now has the added features of contextual types and delayed sensitivity and privacy environments.

Key Concept: Formalizing core language

Formalizing the core language is the process of modeling Duet's AST and semantics in Agda. For example the data type `Term` is the AST of the sensitivity language. In `Term` we have constructors for every piece of AST such as `` `𝔹_ : ∀ {N} → 𝔹 → Term N `` which is the boolean constructor. This takes an implicit argument `N` as the number of free variables in scope, and a boolean value, and then returns a `Term` with `N` free variables in scope. Another example is the sensitivity lambda:

```
sƛ_∣_⇒_ : ∀ {N} → (τ1 : τ N) → (s : Sens) → (e : Term (S N)) → Term N
```

which takes

- N: the number of free variables in scope
- τ_1: the type of the input parameter
- s: the sensitivity bound on the input parameter
- e: the body of the lambda, which is a `Term` with `N + 1` free variables.

and is a `Term` with `N` free variables.

This was the process for every piece of syntax we wanted to formalize in the proof. The rest of the `Term` data type can be found in the source code.

Key Concept: Quantities and Algebra with Quantities

Sensitivity and privacy values in Differential Privacy can be unbounded, this is a property that we also model in Duet with a data type called `qty` pronounced “quantity". Below is the definition of the `qty` data structure. The two cases allow for an embedded value of the type `A`, or the unbounded value `` `∞ ``.

```
data qty {ℓ} (A : Set ℓ) : Set ℓ where
  ⟨_⟩ : A → qty A
  `∞ : qty A
```

We then define algebraic operations for the `qty` type that deal with the unbounded value and then when both values are embedded, pass through to the definitions for the operation on the embedded type `A`. For example, addition is defined very simply as follows:

```
_+[qty]_ : qty A → qty A → qty A
  _ +[qty] `∞ = `∞
  `∞ +[qty] _ = `∞
  ⟨ x ⟩ +[qty] ⟨ y ⟩ = ⟨ x + y ⟩
```

The key here is that <anything> plus infinity is infinity and an embedded value plus and an embedded value is an embedding of the addition. However this could not be done for all operations.

Key Concept: Truncation

An important operator we use in the privacy language a lot is the truncation operation. It’s definition is:

```
]_⌈'_'⌋ : ∀ {ℓ₁ ℓ₂} {A : Set ℓ₁} {B : Set ℓ₂}
  {{_ : has[+] A}} {{_ : has[≡?] A}} {{_ : has[+] B}}
  → qty A → qty B → qty B
] x ⌈' y '⌋ with x ≡? ⟨ zero ⟩
… | [≠] = y
… | [≡] = ⟨ zero ⟩
```

Truncation is a binary operation. It checks if x is zero, if not then return y else return zero. This is useful to the privacy language because we don't scale ϵ privacy costs by any value we just care if an ϵ value is used or not. We also have truncation for vectors `[vec]⌋ xs ⌈' q '` which will truncate all values in xs to q.

Findings: Issue in Duet 1

During the first proof of Duet 1, we found a bug in the sensitivity of variables case splitting on a value. The bug happened when there was a program like so:

```
if x == 0
then 1
else 100000
```

This should have a sensitivity of 100000 because the value can change by at most 100000 depending on the value of x. However, Duet 1 would only get a sensitivity of 1 because x is used once.

Let the above expression be called e. When we evaluate e we would get noise added to the answer of $[[e]] + Lap(\frac{s}{\epsilon})$ which is a very small amount of noise when $s = 1$ so if x were zero then it would be $100000 + Lap(\frac{1}{\epsilon})$ rather than $100000 + Lap(\frac{100000}{\epsilon})$ and the signal would heavily outweigh the noise, removing ambiguity from the answer given by Duet.

Key concept: De Bruijn Indices

In our formalization of Duet in Agda, we replace named variables with a unique index into an *N*-length vector for each variable, where *N* is the number of free variables in the program. The major advantage of using De Bruijn Indices is not having to deal with uniqueness under scoping and alpha renaming issues.

A drawback of using De Bruijn Indices is that sometimes they can be conceptually tricky and hard to decipher. Particularly it can be difficult to define substitution using De Bruijn Indices. This problem is somewhat exacerbated in the formalization of contextual Duet, which has delayed "contexts" or environments embedded in types. These contexts are involved in several specialized substitution operations particular to contextual Duet as discussed below:

We define *pred* (predecessor) as a index specific decrement of an index type. This is useful for specifying the result type of closing over an environment by one variable, as we will see later on.

```
pred : ∀ (N : ℕ) → idx N → ℕ
pred (s N) z = N
pred (s N) (s ι) = s (pred N ι)
```

We define the *pluck* operation to remove a specific variable from a De Bruijn vector representation, given its particular index. We usually pluck a free variable when we are about to replace it with another term and close over it.

```
pluck : ∀ {ℓ} {A : Set ℓ} {N} (ι : idx N) → ⟮ A ⟯[ N ] → ⟮ A ⟯[ pred N ι ]
```

In contextual Duet, we formalize sensitivity and privacy environments as Σ and Σ_p respectively using the De Bruijn vector representation. We then define operations *substΣ/Σ* and *substΣ/*Σ_p respectively to represent the substitution of the appropriate environment (sensitivity or privacy) for a single variable in the corresponding environment. In both cases this operation involves an indexing to find the variable in question, some form of scaling or truncation on the incoming environment based on the variable's current reference, a pluck on that variable, and then elementwise addition of both environments. This takes us, in both cases, from an N-length vector, to a pred N-length vector (i.e. N-1 at a specific index).

```
substΣ/Σ : ∀ {N} → (ι : idx N) → Σ[ pred N ι ] → Σ[ N ] → Σ[ pred N ι ]
substΣ/Σₚ : ∀ {N} → (ι : idx N) → Σ[ pred N ι ] → Σₚ[ N ] → Σₚ[ pred N ι ]
```

We define several weakening operations to represent the addition/inclusion of a new free variable into an environment, typically used when a new variable comes into scope such as in a lambda or let expression. We define weakening on sensitivity/privacy environments (which are essentially equivalent). Weakening on types, which contain these environments, calls out to environment weakening. Weakening in some cases must be defined as index specific, such as in $\Uparrow^t\langle\_\rangle$ while in other cases we assume/know that the new variable will be at the 0th position as in $\Uparrow^t$.

```
wkΣ : ∀ {N} → (ι : idx N) → Σ[ pred N ι ] → Σ[ N ]
⇧ˢ′<_> : ∀ {N} → idx N → Σ[ N ] → Σ[ s N ]
⇧ˢ<_> : ∀ {N} → idx (s N) → Σ[ N ] → Σ[ s N ]
⇧ᵗ<_> : ∀ {N} → idx (s N) → τ N → τ (s N)
⇧ᵗ : ∀ {N} → τ N → τ (s N)
⇧ˢ : ∀ {N} → Σ[ N ] → Σ[ (s N) ]
```

Just as we have substitution for De Bruijn vectors into other De Bruijn vectors, another level up we define substitution for De Bruijn vectors into types that contain De Bruijn vectors. This is essentially a call out to the former substitution operation when appropriate. *substΣ/τ* accepts a specific index, while *cut* assumes again that the index is 0.

```
substΣ/τ : ∀ {N} → (ι : idx N) → Σ[ pred N ι ] → τ N → τ (pred N ι)
cut : ∀ {N} → Σ[ N ] → τ (s N) → τ N
```

We define instantiation to represent closing over a set of *N* free variables in a De Bruijn vector or types containing these, with an incoming vector of sensitivity values. Instantiation usually boils down to the dot product of two equal-length vectors, or equal length subsets of these vectors. In privacy instantiation we usually truncate the incoming vector to 1 elementwise before taking the dot product.

```
instantiateΣ/Σ : ∀ {N N′} → Σ[ N ] → Σ[ N′ + N ] → qty ℕ ∧ Σ[ N′ ]
instantiateΣ/τ : ∀ {N N′} → Σ[ N ] → τ (N′ + N) → τ N′
_⟨⟨_⟩⟩ : ∀ {N} → Σ[ N ] → τ N → τ Z
```

substSx/τ<_> is defined to represent the substitution of a single variable for a sensitivity value in a type. In practice this is usually the first (most recently bound) variable in the De Bruijn vector, and we assume this in the *substSx/τ* operation. This usually comes into play in function application.

```
substSx/τ<_> : ∀ {N} (ι : idx N) → Sens → τ N → τ (pred N ι)
substSx/τ : ∀ {N} → Sens → τ (s N) → τ N
```

Key concept: Probabilistic Semantics

To formalize non-determinism in the semantics of the privacy language we introduce the probability distribution monad $\mathcal{D}$. This allows us to reason about probability with monad laws bind and return, appealing to the differential privacy literature for known facts about probabilty distributions when necessary.

```
𝒟 : ∀ {ℓ} → Set ℓ → Set ℓ
-- this represents the probability of a specific sample coming from
-- a distribution of the corresponding type
Pr[_≒_]≡[_] : ∀ {ℓ} {A : Set ℓ} → 𝒟 A → A → ℝ → Set
instance
  has[»=][𝒟] : ∀ {ℓ} → has[»=] {ℓ} 𝒟
gauss : 𝒟 ℝ
laplace : 𝒟 ℝ
_ : (do x ← return $ 𝕣 1
        y ← laplace
        return $ x + y)
  ≡ (do y ← laplace
        return $ 𝕣 1 + y)
_ = lunit[»=]
```

This allows us to formalize the probabilistic semantics as a set of monadic inference rules.

A straightforward example is the *return* term in the privacy language, which makes use of the *return* monad law to talk about distributions of known well-typed values.

```
-- RETURN
⊢`return: ∀ {N} {γ : γ[N]} {e : Term N} {𝓋₁ : 𝓋} {τ} {⊢τ : ⊢ 𝓋₁ ⦂ τ }
→ γ ⊢ e ⇓ 𝓋₁
-----------------------------------------------------------
→ γ ⊢ (`return e) ⇓ₚ return (∃ 𝓋₁ , ⊢τ )
```

Privacy function application assumes the distribution of output values directly.

```
-- APP
⊢`papp: ∀ {N} {γ : γ[N]} {e′ : PTerm (s N)} {e₁ e₂ : Term N} {v₁ : 𝒱} {τ} {v₂
: 𝒟 (∃ v ∶ 𝒱 ST ⊢ v ∶ τ )}
  → γ ⊢ e₁ ⇓ (pλ∶ e′ ∥ γ )
  → γ ⊢ e₂ ⇓ v₁
  → v₁ :: γ ⊢ e′ ⇓ₚ v₂
  ------------------------------------------------------------
  → γ ⊢ (e₁ `papp e₂) ⇓ₚ v₂
```

For *bind* we rely on the probability distribution sample existential to draw a sample from e_1's output, which is then bound in e_2. The output of *bind* can then be defined as the first projection of the *E* existential dependent pair premise.

```
-- BIND
⊢`bind: ∀ {N} {γ : γ[N]} {e₂ : PTerm (s N)} {e₁ : PTerm N} {v₁ : 𝒱} {τ} {⊢τ :
⊢ v₁ ∶ τ }
  → γ ⊢ e₁ ⇓ₚ return ⟨∃ v₁ , ⊢τ ⟩
  → (E : ∃ v₂ ST v₁ :: γ ⊢ e₂ ⇓ₚ v₂)
  → let v₃ = do ⟨∃ v₁ , ⊢v₁ ⟩ ← (return ⟨∃ v₁ , ⊢τ ⟩) ; dπ₁ E in
  ------------------------------------------------------------
  γ ⊢ (`bind e₁ ∥ e₂) ⇓ₚ v₃
```

Key concept: Logical Relations

The proof of the fundamental property of metric preservation in contextual Duet requires that we state hypotheses and prove facts about the relationship between two members of the same set or category. For example, we may wish to prove something about the relationship between two values of the same "type", or two expressions. In particular, we usually want to say something about their type (that they have the same type) and the sensitivity or privacy "distance" between them.

In comparison with the paper/English version of this proof, the mechanization of the logical relations requires extra machinery to push through. Specifically, because many of the relations involve talking about values in the Duet language, we need to formalize "value types", value type judgements and value type environments. Also, in cases involving reduction of expressions to values, or typing of expressions, we also assume well-typedness of corresponding values, which is sound under assumption/proof of type preservation and progress in contextual Duet.

Each logical relation is briefly discussed below:

The sensitivity expression relation relates two different expressions evaluated under two different value environments by deferring to the value relation of the values produced after evaluation by the ground truth dynamic semantics. As mentioned earlier, since we relate expressions via the value relation, expressions are related by a value type at some sensitivity. Note that we must also assume well-typedness of the relevant values.

```
-- sensitivity expression relation
⟨_⊢_,_⊢_⟩∈ℰ⟦_:_⟧: ∀ {N} γ[N] → Term N → γ[N] → Term N → Sens → τ Z → Set
⟨γ₁⊢e₁,γ₂⊢e₂⟩∈ℰ⟦s:τ⟧ = ∀ v₁ v₂ → (ϵ₁: ⊢ v₁ ∶ τ) → (ϵ₂: ⊢ v₂ ∶ τ) → (γ₁ ⊢ e₁ ⇓
v₁) ∧ (γ₂ ⊢ e₂ ⇓ v₂) → ⟨ v₁ , v₂ ⟩∈𝒱′⟦ τ : ϵ₁ , ϵ₂ : s ⟧
```

The privacy expression relation is less straightforward due to non-determinism in the privacy language. This makes use of the probability monad and the sample probability operator to formalize the distance between two privacy expressions via the standard differential privacy inequality.

```
-- privacy expression relation
⟨_⊢_,_⊢_⟩∈ℰₚ⟦_:_⟧: ∀ {N} γ[N] → PTerm N → γ[N] → PTerm N → Priv → τZ → Set
  ⟨ γ₁ ⊢ e₁ , γ₂ ⊢ e₂ ⟩∈ℰₚ⟦ p : τ ⟧ = ∀ v₁ v₂ r₁ r₂ →
    (ϵ₁ : ⊢ v₁ ∶ τ) →
    (ϵ₂ : ⊢ v₂ ∶ τ) →
    (γ₁ ⊢ e₁ ⇓ₚ return ⟨∃ v₁ , ϵ₁ ⟩) ∧ (γ₂ ⊢ e₂ ⇓ₚ return ⟨∃ v₂ , ϵ₂ ⟩) →
    Pr[ return v₁ ≟ v₁ ]≡[ r₁ ] →
    Pr[ return v₂ ≟ v₂ ]≡[ r₂ ] → r₁ ≤ᵣ ((ℯ^R (p2r p)) × r₂)
```

The value environment relation is assumed in the proof of the fundamental property, however, in certain cases in the mechanization it becomes necessary to manually extend the relation to include new values in the value environments. For this reason, we formalize the value environment relation as the *null* and *cons* constructor functions where the constructor case accepts an instance of the value relation to extend the value environment relation.

```
-- value environment relation
⟨_,_⟩∈𝒢⟦_:_⟧: ∀ {N} → γ[ N ] → γ[ N ] → Σ[ N ] → 𝕋[ N ] → Set
⟨ [] , [] ⟩∈𝒢⟦ [] : [] ⟧ = 𝟙
⟨ v₁ :: γ₁ , v₂ :: γ₂ ⟩∈𝒢⟦ s :: Σ : τ :: Γ ⟧ = ∃ δ₁ ⦂ (⊢ v₁ ⦂ τ) ST ∃ δ₂ ⦂ (⊢ v₂ ⦂
τ) ST ⟨ v₁ , v₂ ⟩∈𝒱′⟦ τ : δ₁ , δ₂ : s ⟧ ∧ ⟨ γ₁ , γ₂ ⟩∈𝒢⟦ Σ : Γ ⟧
```

The value relation is straightforward, assuming all-typedness of values as discussed earlier.

```
-- value relation
⟨_,_⟩∈𝒱′⟦_:_,_:_⟧: ∀ (v₁ v₂ : 𝓋) (t : τ z) → ⊢ v₁ ⦂ t → ⊢ v₂ ⦂ t → Sens →  Set
```

Proof of fundamental property of metric preservation

$$
\begin{aligned}
fp : \forall\, \{N\}\, \{\Gamma : \Gamma[N]\} \{ &\llbracket\, e\, \tau\, \Sigma\, \gamma_1\, \gamma_2\, \Sigma'\, \Sigma_0 \rrbracket \\
&\to \llbracket \vdash \gamma_1 \to \llbracket \vdash \gamma_2 \to \Gamma, \Sigma_0 \vdash e \mathrel{⦂} \tau, \Sigma \\
&\to \langle \gamma_1, \gamma_2 \rangle \in \mathcal{G}\llbracket \Sigma\prime : \mathbb{\Gamma} \rrbracket \\
&\to \langle \gamma_1 \vdash e, \gamma_2 \vdash e \rangle \in \mathcal{E}\llbracket\, \Sigma \dot{\times} \Sigma' : \Sigma' \langle\langle \tau \rangle\rangle \,\rrbracket \\
fp_2 : \forall\, \{N\}\, \{\Gamma : \Gamma[N]\} \{ &\llbracket\, e\, \tau\, \Sigma\, \gamma_1\, \gamma_2\, \Sigma'\, \Sigma_0 \rrbracket \\
&\to \llbracket \vdash \gamma_1 \to \llbracket \vdash \gamma_2 \to \Gamma, \Sigma_0 \vdash_p e \mathrel{⦂} \tau, \Sigma \\
&\to \langle \gamma_1, \gamma_2 \rangle \in \mathcal{G}\llbracket \Sigma\prime : \mathbb{\Gamma} \rrbracket \\
&\to \langle \gamma_1 \vdash e, \gamma_2 \vdash e \rangle \in \mathcal{E}_p\llbracket [vec] \rceil \Sigma\prime \lceil {}^{\mathsf{r}} one^{\mathsf{r}} \dot{\times} \Sigma : (\Sigma\prime \langle\langle \tau \rangle\rangle) \rrbracket
\end{aligned}
$$

The fundamental property proof is by induction on the terms of the language. In the mechanization we need to explicitly assume well-typednedness of the value environments, which is implicit in the value environment relation. Above we have two fundamental properties, the first is for the sensitivity language and the second is for the privacy language.