1 LET

LET, as outlined in the Background section, is a conventional high-level language, so I will talk about a few design decisions made during its implementation to take advantage of dependent types.

1.1 Variables, environments and de Bruijn indices

Variable names are pointers to variable bindings in the environment. A de Bruijn index is an alternate way of referring to a binding using its level, with the innermost level being 0. So we replace variable names with natural numbers and avoid additional code complexity:

let x = ⟨⟩ in let y = (⟨⟩,⟨⟩) in (y,x) **becomes** let ⟨⟩ in let (⟨⟩,⟨⟩) in (0,1)

However, we cannot allow just any natural number to be used in place of a variable name. This can lead to errors during evaluation. In the example above, if 0 is replaced with 5, we will get an error because there is no binding at level 5! In other words, this variable is not in scope. The data structures used to represent indices and environments help us prevent this problem.

Consider a LET value environment ρ and its corresponding type environment Γ. ρ is a vector of values, Γ is a vector of value types, and ∀x ∈ [0 ,..., len(ρ) - 1].ρ[x] is of type Γ[x]. In our implementation of LET, we represent ρ as a value of type Γ env, where Γ is a length-indexed vector that holds values of type 𝕓, i.e., Γ is of type Vec 𝕓 n and n = len(Γ). In this way, we 'connect' ρ to its type environment Γ.

```
data _env : ∀{n : ℕ} → (Vec 𝕓 n) → Set where
  ε : [] env
  _+ₑ_ : ∀{n : ℕ}{Γ : Vec 𝕓 n}{b : 𝕓}
           → Γ env
           → (x : val b)
         ----------
           → (b :: Γ) env
```

(a) Data type for environments

```
varₑ : ∀{n : ℕ}{Γ : Vec 𝕓 n}
     → (x : Fin n)
     ----------
     → Γ ⊢exp: (lookup Γ x)
```

(b) Constructor for variables

Figure 1: Variables and environments

The type environment Γ for any LET expression is known at compile time. Each binding in the current environment contributes one element to Γ. The Agda term var_e x represents a LET variable where x corresponds to a de Bruijn index, and the typechecker ensures that x is of type Fin n where n = len(Γ). Therefore, x must be in the range [0 ,.., len(Γ) - 1] and var_e x refers to a valid binding. By using Fin n instead of ℕ for x, we prevent references to variables that are out of scope.

1.2 The type of an expression is its typing judgement.

Instead of implementing LET typing rules separately, which might lead to expressions that are untypeable, we incorporate the typing derivation into the construction of an expression. This is especially useful when the preconditions are linked in some way, for instance in **case** statements, where the left and right branches must have the same type under the extended environment. So for expressions, we use a data type indexed by value types 𝕓 and Γ i.e., Vec 𝕓 n. A LET expression **e** is represented in Agda as a value of type Γ ⊢exp: b where $\mathbf{\Gamma} \vdash \mathbf{e} :\ b$ according to LET typing rules. Below we present the typing rules for **fst** and **case**.

$$\frac{\Gamma \vdash e : b_1 \times b_2}{\Gamma \vdash \text{fst } e : b_1}\ \text{fst}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : b_1 + b_2 \\ \Gamma,\, x : b_1 \vdash e_1 : b_3 \\ \Gamma,\, y : b_2 \vdash e_2 : b_3\end{array}}{\Gamma \vdash \text{case } e \text{ in L}x \to e_1 \text{ , R}y \to e_2\text{: } b_3}\ \text{case}$$

And the corresponding Agda constructors:

```
fstₑ : ∀{n : ℕ}{b₁ b₂ : 𝕓}{Γ : Vec 𝕓 n}
        → Γ ⊢exp: (b₁ × b₂)
        ----------
        → Γ ⊢exp: b₁
```

```
ₑcase_ₑL_ₑR_ : ∀{n : ℕ}{b₁ b₂ b₃ : 𝕓}{Γ : Vec 𝕓 n}
  → Γ ⊢exp: (b₁ + b₂)
  → (b₁ :: Γ) ⊢exp: b₃
  → (b₂ :: Γ) ⊢exp: b₃
  ----------
  → Γ ⊢exp: b₃
```

Figure 2: Construction of **fst** and **case** expressions using typing rules

When we use an expression somewhere, say in an interpreter, Γ is known. To form a typeable **case** expression which is used for branching, we need a LET expression e and a proof that $\mathbf{\Gamma} \vdash \mathbf{e} :\ b_1 + b_2$ for some b_1, b_2. In Agda, this proposition is implicit in the type of e, which is its proof. Similarly, we receive e_1 and e_2 which are proofs that they have the same type b_3 in environments extended with b_1 and b_2 respectively. The typechecker concludes that the **case** expression we have built will have the type b_3 in Γ.

This representation has another big advantage:

1.3 A type-safety proof in the declaration of the interpreter

```
evalₑ : ∀{n : ℕ}{Γ : Vec 𝕓 n}{b : 𝕓}
          → Γ env → Γ ⊢exp: b → val b
```

LET is a total language so all its constructs will terminate. This fact is verified by Agda's termination checker which ensures that the interpreter evalₑ terminates. In addition, we know from the declaration that it terminates with a value of the correct type **b** given a type environment Γ and an expression e which has type **b** in Γ. So we have shown that LET is type-safe simply by defining evalₑ! The definition of evalₑ is conventional. We conclude our discussion of LET with two examples:

```
evalₑ ρ (leftₑ e) = left (evalₑ ρ e)

evalₑ ρ (ₑlet e₁ ₑin e₂) = evalₑ (ρ +ₑ (evalₑ ρ e₁)) e₂
```