Do a Parametricity Proof with Agda

Hey, you!

Have you seen this mystery function?

$$f \ : \ \forall\alpha.\, \alpha \to \alpha$$

You ever suspect it's actually this?

$$\Lambda\alpha.\, \lambda x.\, x$$

Use Agda and embark on a convoluted journey to prove yourself right!

First, encode some of System F

You'll need function and polymorphic types for sure

```
-- Function type
_⇒_ : Type → Type → Type
-- Polymorphic type
all[_]⇒_ : Id → Type → Type
```

Function, polymorphic, and application expressions are also a must

```
ƛ[_]⇒_ : Id → Expr → Expr
-- Polymorphic
Λ[_]⇒_ : Id → Expr → Expr
-- Poly-app
_[_] : Expr → Type → Expr
-- Application
_·_ : Expr → Expr → Expr
```

Finally, step rules and substitution rules for both types and expressions are needed!

Step rules are data types.
Substitution rules are just functions.

In Another File, Encode Equivalence Rules

These are data types that tell Agda how to step through equivalence statements.

```
data _~⟦_⟧_ : Expr → Type → Expr → Set where
```

Example for polymorphic type equivalence.

```
tylam : ∀ { v₁ v₂ τ R α α α' }
  → (v₁ [ α ]) ~⟦ [ R := α ]t τ ⟧ (v₂ [ α' ])
  ------------------------------
  → v₁ ~⟦ all[ α ]⇒ τ ⟧ v₂
```

These equivalence statements are iff statements, so we want to tell Agda how to go backwards, too.

```
tylam-inv : ∀ { v₁ v₂ τ R α α α' }
  → v₁ ~⟦ all[ α ]⇒ τ ⟧ v₂
  ------------------------------
  → (v₁ [ α ]) ~⟦ [ R := α ]t τ ⟧ (v₂ [ α' ])
```

Make some postulates

With postulates, we can give ourselves assumptions to work with.

Give yourself parametricity

```
postulate
parametricity :
∀ { τ M } → (∅ ⊢ M ⦂ τ) → M ~⟦ τ ⟧ M
```

And the mystery function

```
f : Expr
f-type : ∀ { α } →
∅ ⊢ f ⦂ (all[ α ]⇒ ((` α) ⇒ (` α)))
```

And an "inversion" of one of our equivalence rules.

```
expr/inv-1 : ∀ { e₁ e₂ τ v₁ }
  → e₁ ~⟦ τ ⟧ e₂
  ------------------------------
  → (e₁ —→* v₁)
```

Now for the hard part!

Prove some stuff

Given f from the last page and this id function

```
id = Λ[ "α" ]⇒ (ƛ[ "x" ]⇒ ` "x")
```

We wanna show this

f-is-id

```
∀ { α } →
f ~⟦ all[ α ]⇒ ((` α) ⇒ (` α)) ⟧ id
```

To do that, we need to show this

Hint: use parametricity! lemma-1

```
where
∀ { v α } → ((f [ α ]) · v) —→* v
```

And this

Hint: use step rules idv->v

```
∀ { v α } → ((id [ α ]) · v) —→* v
```

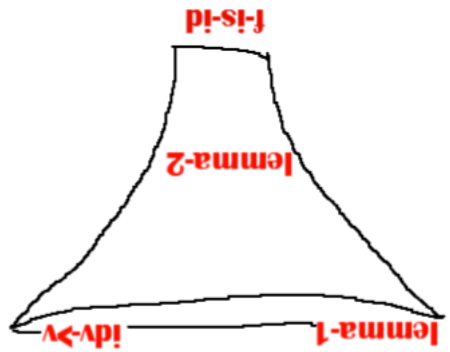
And finally this

lemma-2

```
∀ { α α' τ α } →
(f [ α ]) ~⟦ τ ⇒ τ ⟧ (id [ α' ])
```

Put it all together!

I think of proving things in Agda like putting things down a funnel.
I can't put all the code in here, so here's a drawing of what it should "feel" like:



Chris Martens's Notes

Frank Pfenning's Notes

The Lambda Calculus chapter of Programming Language Foundations in Agda

The Question Mark

Electric Zine Maker

Thank you for existing.