Formal Proof of Type Preservation of the Dictionary Passing Transform for System F

Marius Weidner

Chair of Programming Languages, University of Freiburg

February 25, 2023

Typeclasses

Overloading Equality in Haskell

```
class Eq α where
  eq :: α → α → Bool

instance Eq Nat where
  eq x y = x ≐ y
instance Eq α ⇒ Eq [α] where
  eq []       []       = True
  eq (x : xs) (y : ys) = eq x y && eq xs ys

.. eq 42 0 .. eq [42, 0] [42, 0] ..
```

Desugaring Typeclasses

Overloading Equality in System F_O

```
decl eq in

inst eq : Nat → Nat → Bool
  = λx. λy. .. in
inst eq : ∀α. [eq : α → α → Bool] ⇒ [α] → [α] → Bool
  = Λα. ƛ(eq : α → α → Bool). λxs. λys. .. in

.. eq 42 0 .. eq Nat [42, 0] [42, 0] ..
```

Dictionary Passing Transform

Overloading Equality in System F_O

```
decl eq in
inst eq : Nat → Nat → Bool
  = λx. λy. .. in
inst eq : ∀α. [eq : α → α → Bool] ⇒ [α] → [α] → Bool
  = Λα. ƛ(eq : α → α → Bool). λxs. λys. .. in
.. eq 42 0 .. eq Nat [42, 0] [42, 0] ..
```

System F_O Transformed to System F

```
let eq₁ : Nat → Nat → Bool
  = λx. λy. .. in
let eq₂ : ∀α. (α → α → Bool) → [α] → [α] → Bool
  = Λα. λeq₁. λxs. λys. .. in

.. eq₁ 42 0 .. eq₂ Nat eq₁ [42, 0] [42, 0] ..
```

Elegant Syntax Representations in Agda

System F_O Syntax

```
data Term : Sorts → Sort r → Set where
  `_              : s ∈ S → Term S s
  decl`o`in_      : Term (S ▷ oₛ) eₛ → Term S eₛ
  inst`_`=_`in_   : Term S oₛ → Term S eₛ → Term S eₛ → Term S eₛ
  _:_             : Term S oₛ → Term S τₛ → Term S cₛ
  ƛ_⇒_            : Term S cₛ → Term S eₛ → Term S eₛ
  [_]⇒_           : Term S cₛ → Term S τₛ → Term S τₛ
  - ...
```

Substitution Defined on Terms

```
_[_] : Term (S ▷ s') s → Term S s' → Term S s
t [ t' ] = sub (singleₛ idₛ t') t
```

Agda Formalization of System F_O

Context

```
data Ctx : Sorts → Set where
  ∅ : Ctx []
  _▶_ : Ctx S → Term S (item-of s) → Ctx (S ▷ s)
  _▸_ : Ctx S → Cstr S → Ctx S
```

Constraint Solving

```
data [_]∈_ : Cstr S → Ctx S → Set where
  here : [ (‘ o : τ) ]∈ (Γ ▸ (‘ o : τ))
  under-bind : {I : Term S (item-of s′)} →
    [ (‘ o : τ) ]∈ Γ → [ (‘ there o : wk τ) ]∈ (Γ ▶ I)
  under-inst : [ c ]∈ Γ → [ c ]∈ (Γ ▸ c′)
```

Extrinsic Typing Rules

System F_{O} Typing

```
data _⊢_:_ : Ctx S → Term S s → Term S (kind-of s) → Set where
  ⊢inst :
    Γ ⊢ e₂ : τ →
    Γ ▶ (‘ o : τ) ⊢ e₁ : τ’ →
    Γ ⊢ inst‘ ‘ o ‘= e₂ ‘in e₁ : τ’
  ⊢‘o :
    [ ‘ o : τ ]∈ Γ →
    Γ ⊢ ‘ o : τ
  ⊢ƛ :
    Γ ▶ c ⊢ e : τ →
    Γ ⊢ ƛ c ⇒ e : [ c ]⇒ τ
  ⊢⊘ :
    Γ ⊢ e : [ ‘ o : τ ]⇒ τ’ →
    [ ‘ o : τ ]∈ Γ →
    Γ ⊢ e : τ’
```

Fun Lemmas on Our Way to Type Preservation

Type Transform Preserves Weakening

F.ren (⊢ρ⇝ρ $\vdash\rho$) (τ⇝τ τ) ≡ τ⇝τ (F^O.ren ρ τ)

Type Transform Preserves Substitution

F.sub (⊢σ⇝σ $\vdash\sigma$) (τ⇝τ τ) ≡ τ⇝τ (F^O.sub σ τ)

Instance Resolution Transforms to Correct Variable

```
o:τ∈Γ⇝Γx≡τ : ∀ {Γ : Fᴼ.Ctx Fᴼ.S} →
  (o:τ∈Γ : [ ‘ Fᴼ.o : Fᴼ.τ ]∈ Γ) →
  F.lookup (Γ⇝Γ Γ) (o:τ∈Γ⇝x o:τ∈Γ) ≡ (τ⇝τ Fᴼ.τ)
```

Type Preservation of the Dictionary Passing Transform

Typed System $\mathrm{F_O}$ tranforms to typed System F

```
⊢t⇝⊢t : ∀ {Γ : Fᴼ.Ctx Fᴼ.S} {t : Fᴼ.Term Fᴼ.S Fᴼ.s}
            {T : Fᴼ.Term Fᴼ.S (Fᴼ.kind-of Fᴼ.s)} →
  (⊢t : Γ Fᴼ.⊢ t : T) →
  (Γ⇝Γ Γ) F.⊢ (⊢t⇝t ⊢t) : (T⇝T Γ T)
⊢t⇝⊢t (⊢'o o:τ∈Γ) = ⊢'x (o:τ∈Γ⇝Γx≡τ o:τ∈Γ)
⊢t⇝⊢t (⊢ƛ {c = (' o : τ)} ⊢e) = ⊢λ (subst (_ F.⊢ ⊢t⇝t ⊢e :_)
  τ⇝wk-inst·τ≡wk-inst·τ⇝τ (⊢t⇝⊢t ⊢e))
⊢t⇝⊢t (⊢⊘ ⊢e o:τ∈Γ) = ⊢· (⊢t⇝⊢t ⊢e) (⊢'x (o:τ∈Γ⇝Γx≡τ o:τ∈Γ))
- ...
```

Future Work: Hindley Milner & Semantic Preservation

Overloading in Hindley Milner

- Constraint abstractions cannot require poly types
- All instances must differ in the type of there first argument
  1. Deterministic instance resolution
  2. Preserve Algorithm W

Proving Semantic Preservation

- System F_O would require typed semantics
  1. Prove that
- Hindley Milner based System could support untyped Semantics
  1. Prove that