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

Marius Weidner

Chair of Programming Languages, University of Freiburg

March 5, 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 & Renaming

Renaming

$$\mathsf{Ren} : \mathsf{Sorts} \to \mathsf{Sorts} \to \mathsf{Set}$$
$$\mathsf{Ren}\ S_1\ S_2 = \forall\ \{s\} \to \mathsf{Var}\ S_1\ s \to \mathsf{Var}\ S_2\ s$$

$$\mathsf{wk} : \mathsf{Term}\ S\ s \to \mathsf{Term}\ (S \triangleright s')\ s$$
$$\mathsf{wk} = \mathsf{ren}\ \mathsf{there}$$

Substitution

$$\mathsf{Sub} : \mathsf{Sorts} \to \mathsf{Sorts} \to \mathsf{Set}$$
$$\mathsf{Sub}\ S_1\ S_2 = \forall\ \{s\} \to \mathsf{Var}\ S_1\ s \to \mathsf{Term}\ S_2\ s$$

$$\_[\_] : \mathsf{Term}\ (S \triangleright s')\ s \to \mathsf{Term}\ S\ s' \to \mathsf{Term}\ S\ s$$
$$t\ [\ t'\ ] = \mathsf{sub}\ (\mathsf{single}_s\ \mathsf{id}_s\ t')\ t$$

Overloading Formalized

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 Renaming

$\mathsf{F.ren}\ (\vdash\!\rho\!\rightsquigarrow\!\rho\ \vdash\!\rho)\ (\tau\!\rightsquigarrow\!\tau\ \tau) \equiv \tau\!\rightsquigarrow\!\tau\ (\mathsf{F}^{O}.\mathsf{ren}\ \rho\ \tau)$

Type Transform Preserves Substitution

$\mathsf{F.sub}\ (\vdash\!\sigma\!\rightsquigarrow\!\sigma\ \vdash\!\sigma)\ (\tau\!\rightsquigarrow\!\tau\ \tau) \equiv \tau\!\rightsquigarrow\!\tau\ (\mathsf{F}^{O}.\mathsf{sub}\ \sigma\ \tau)$

Instance Resolution Transforms to Unqiue Variable

$$
\begin{aligned}
&\mathrm{o{:}\tau{\in}\Gamma{\rightsquigarrow}\Gamma x{\equiv}\tau} : \forall\ \{\Gamma : \mathsf{F}^{O}.\mathsf{Ctx}\ F^{O}.S\} \to (o{:}\tau{\in}\Gamma : [\ `\ F^{O}.o : F^{O}.\tau\ ]{\in}\ \Gamma) \to \\
&\quad \mathsf{F.lookup}\ (\Gamma\!\rightsquigarrow\!\Gamma\ \Gamma)\ (\mathrm{o{:}\tau{\in}\Gamma{\rightsquigarrow}x}\ o{:}\tau{\in}\Gamma) \equiv (\tau\!\rightsquigarrow\!\tau\ F^{O}.\tau)
\end{aligned}
$$

Type Preservation of the Dictionary Passing Transform

Typed System 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 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))
- ...
```

Further Work: Hindley Milner & Semantic Preservation

Overloading in Hindley Milner

- Constraint abstractions cannot require poly types
  1. Introduce sorts m_s and p_s in favour of single sort τ_s
- All instances must differ in the type of their first argument for each overloaded variable
  1. Preserves Algorithm W

Proving Semantic Preservation

- Overloaded languages require typed semantics
- Prove that, if $\vdash e \hookrightarrow \vdash e'$ then
  $\exists\ [\ e''\ ]\ (\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e'\ \vdash e \hookrightarrow^* e'') \times (\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e'\ \vdash e' \hookrightarrow^* e'')$