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

Marius Weidner

Chair of Programming Languages, University of Freiburg

March 7, 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. λ(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. λ(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 ▷ o_s) e_s → Term S e_s
  inst`_`=_`in_     : Term S o_s → Term S e_s → Term S e_s → Term S e_s
  _:_               : Term S o_s → Term S τ_s → Term S c_s
  ƛ_⇒_              : Term S c_s → Term S e_s → Term S e_s
  [_]⇒_             : Term S c_s → Term S τ_s → Term S τ_s
```

Substitution & Renaming

Renaming

```
Ren : Sorts → Sorts → Set
Ren S₁ S₂ = ∀ {s} → Var S₁ s → Var S₂ s

wk : Term S s → Term (S ▷ s') s
wk = ren there
```

Substitution

```
Sub : Sorts → Sorts → Set
Sub S₁ S₂ = ∀ {s} → Var S₁ s → Term S₂ s

_[_] : Term (S ▷ s') s → Term S s' → Term S s
t [ t' ] = sub (single_s 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

F.ren ($\vdash \rho \rightsquigarrow \rho \vdash \rho$) ($\tau \rightsquigarrow \tau\ \tau$) $\equiv$ $\tau \rightsquigarrow \tau$ (F^O.ren $\rho\ \tau$)

Type Transform Preserves Substitution

F.sub ($\vdash \sigma \rightsquigarrow \sigma \vdash \sigma$) ($\tau \rightsquigarrow \tau\ \tau$) $\equiv$ $\tau \rightsquigarrow \tau$ (F^O.sub $\sigma\ \tau$)

Instance Resolution Transforms to Unqiue Variable

$o{:}\tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau$: $\forall\ \{\Gamma : F^O.\mathrm{Ctx}\ F^O.S\} \to (o{:}\tau \in \Gamma : [\ `\ F^O.o : F^O.\tau\ ] \in \Gamma) \to$
F.lookup ($\Gamma \rightsquigarrow \Gamma\ \Gamma$) ($o{:}\tau \in \Gamma \rightsquigarrow x\ o{:}\tau \in \Gamma$) $\equiv$ ($\tau \rightsquigarrow \tau\ F^O.\tau$)

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
  $\exists\ [\ e''\ ]\ (\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e' \vdash e \hookrightarrow^* e'') \times (\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e' \vdash e' \hookrightarrow^* e'')$