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

Marius Weidner

Chair of Programming Languages, University of Freiburg
weidner@cs.uni-freiburg.de

Bachelor Thesis

Examiner: Prof. Dr. Peter Thiemann
Advisor: Hannes Saffrich

Abstract. Most popular strongly typed programming languages support function overloading. In combination with polymorphism this leads to essential language constructs, like type classes in Haskell or traits in Rust. We introduce System F_O, a minimal language extension to System F, with support for overloading. We show that the Dictionary Passing Transform from System F_O to System F is type preserving.

1 Introduction

1.1 Motivation

A common use cases for function overloading is operator overloading. Without overloading, code becomes less readable, since we would need to define a unique name for each operator on each type. Haskell, for example, solves this problem using type classes. Essentially, type classes allow to declare function names with multiple meanings. We can give one or more meanings to a type class by instantiating the type class on different types. When we invoke an overloaded function name, we determine the correct instance based on the types of the supplied arguments.

Example: Overloading Equality in Haskell

Our goal is to overload the function `eq : α → α → Bool` with different meanings for different types substituted for `α`. We want to be able to call `eq` on both `Nat` and `[Nat]` respectively. In Haskell we would solve the problem as follows:

```
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] ..
```

First, type class `Eq` is declared and instantiated for `Nat`. Next, `Eq` is instantiated for `[α]`, given that an instance `Eq` exists for type `α`. Finally, we can call `eq` on elements of type `[Nat]`, since the constraint `Eq α ⇒ ..` in the second instance resolves to the first instance.

1.2 Introducing System F_O

In our minimal language extension to System F we give up high level language constructs like Haskell's type classes. Instead, System F_O desugars type class functionality to just overloaded variables. Using the `decl o in e'` expression we can introduce an new overloaded variable `o`. If declared as overloaded, `o` can be instantiated for type `τ` of expression `e` using the `inst o = e in e'` expression. In contrast to Haskell, it is allowed to overload `o` with arbitrary types. Shadowing other instances of the same type is allowed. Constraints can be introduced using the constraint abstraction `ƛ(o : τ). e'` resulting in a expression of constraint type `[o : τ] ⇒ τ'`. Constraints are eliminated implicitly by the typing rules.

Example: Overloading Equality in System F_O

Recall the Haskell example from above. The same functionality can be expressed in System F_O as follows:

```
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] ..
```

First, we declare `eq` to be an overloaded identifier and instantiate `eq` for `Nat`. Next, we instantiate `eq` for `[α]`, given the constraint introduced by the constraint abstraction `ƛ` is satisfied. For convenience type annotations for instances are given. The actual implementations of the instances are omitted. Because System F_O is based on System F, we are required to bind type variables using type abstractions `Λ` and eliminate type variables using type application.
A little caveat: the second instance needs to recursively call instance `eq` for sublists but System F_O's formalization does not actually support recursive instances or recursive let bindings. Extending System F_O with recursive instances and let bindings should be straight forward but is subject to further work.

1.3 Translating between System F_O and System F

The Dictionary Passing Transform translates well typed System F_O expressions to well typed System F expressions. We drop `decl o in` expressions and replace `inst o = e` `in e'` expressions with `let` o_τ `= e in e'` expressions, where o_τ is a unique name with respect to type τ of `e`.

Example: Dicitionary Passing Transform

Recall the System F_O example from above. We use indices to ensure unique names. Applying the Dictionary Passing Transform results in the following well typed System F expression:

```
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] ..
```

1.4 Related Work

- SystemO - SystemFc - ..?

2 Preliminary

2.1 Dependently Typed Programming in Agda

Agda [CITE] is a dependently typed programming language developed at Gothenburg University.

2.2 Design Decisions for the Agda Formalization

Sorts pure type systems

HÄ

3 System F

3.1 Specification

Sorts

We define two sorts for our Syntax.

```
data Sort : Set where
  eₛ : Sort
  τₛ : Sort

Sorts : Set
Sorts = List Sort
```

Syntax

```
data Term : Sorts → Sort → Set where
  `_         : s ∈ S → Term S s
  tt         : Term S eₛ
  λ`x→_      : Term (S ▷ eₛ) eₛ → Term S eₛ
  Λ`α→_      : Term (S ▷ τₛ) eₛ → Term S eₛ
  _·_        : Term S eₛ → Term S eₛ → Term S eₛ
  _•_        : Term S eₛ → Term S τₛ → Term S eₛ
  let`x=_`in_ : Term S eₛ → Term (S ▷ eₛ) eₛ → Term S eₛ
  `⊤         : Term S τₛ
  _⇒_        : Term S τₛ → Term S τₛ → Term S τₛ
  ∀`α_       : Term (S ▷ τₛ) τₛ → Term S τₛ
```

We will use shorthands Var S s = $s \in S$, Expr S = Term S e_s and Type S = Term S τ_s.

Renaming

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

idᵣ : Ren S S
idᵣ = id

wkᵣ : Ren S (S ▷ s)
wkᵣ = there

extᵣ : Ren S₁ S₂ → Ren (S₁ ▷ s) (S₂ ▷ s)
extᵣ ρ (here refl) = here refl
extᵣ ρ (there x) = there (ρ x)

dropᵣ : Ren S₁ S₂ → Ren S₁ (S₂ ▷ s)
dropᵣ ρ x = there (ρ x)

ren : Ren S₁ S₂ → (Term S₁ s → Term S₂ s)
ren ρ (` x) = ` (ρ x)
ren ρ tt = tt
ren ρ (λ`x→ e) = λ`x→ (ren (extᵣ ρ) e)
ren ρ (Λ`α→ e) = Λ`α→ (ren (extᵣ ρ) e)
ren ρ (e₁ · e₂) = (ren ρ e₁) · (ren ρ e₂)
ren ρ (e • τ) = (ren ρ e) • (ren ρ τ)
ren ρ (let`x= e₂ `in e₁) = let`x= (ren ρ e₂) `in ren (extᵣ ρ) e₁
ren ρ `⊤ = `⊤
ren ρ (τ₁ ⇒ τ₂) = ren ρ τ₁ ⇒ ren ρ τ₂
ren ρ (∀`α τ) = ∀`α (ren (extᵣ ρ) τ)

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

idₛ : Sub S S
idₛ = `_

extₛ : Sub S₁ S₂ → Sub (S₁ ▷ s) (S₂ ▷ s)
extₛ σ (here refl) = ` here refl
extₛ σ (there x) = ren wkᵣ (σ x)

dropₛ : Sub S₁ S₂ → Sub S₁ (S₂ ▷ s)
dropₛ σ x = wk (σ x)

singleₛ : Sub S₁ S₂ → Term S₂ s → Sub (S₁ ▷ s) S₂
singleₛ σ t (here refl) = t
singleₛ σ t (there x) = σ x

sub : Sub S₁ S₂ → (Term S₁ s → Term S₂ s)
sub σ (` x) = (σ x)
sub σ tt = tt
sub σ (λ`x→ e) = λ`x→ (sub (extₛ σ) e)
sub σ (Λ`α→ e) = Λ`α→ (sub (extₛ σ) e)
sub σ (e₁ · e₂) = sub σ e₁ · sub σ e₂
sub σ (e • τ) = sub σ e • sub σ τ
sub σ (let`x= e₂ `in e₁) = let`x= sub σ e₂ `in (sub (extₛ σ) e₁)
sub σ `⊤ = `⊤
sub σ (τ₁ ⇒ τ₂) = sub σ τ₁ ⇒ sub σ τ₂
sub σ (∀`α τ) = ∀`α (sub (extₛ σ) τ)

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

3.2 Soundness

4 System F_O

4.1 Specification

5 Dictionary Passing Transform

6 Conclusion and Further Work

6.1 Hindley Milner with Overloading

6.2 Semantic Preservation of System $\mathbf{F_O}$

6.3 Conclusion

References

Declaration.

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Place, Date

Signature