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, for example 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 Overloading in Haskell

Without overloaded function names code becomes less readable, since we would need to define a unique name for every function, for example equality, on each type. Haskell, solves this problem using type classes. Essentially, type classes allow to declare overloaded function names with generic type signatures. We can give one of many specific meanings to a type class, by instantiating the type class for concrete types. When we invoke the overloaded function name, we determine the correct instance based on the concrete types of the applied arguments. Furthermore, Haskell allows to constrain bound type variables α via type constraints `Tc α ⇒ ..` to only be substituted by a concrete type τ, if there exists an instance of `Tc` for τ.

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 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` with a single generic function `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 language extension to System F we give up high level language constructs. Instead, System F_O desugars type class functionality to 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 expressions 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] ..
```

For convenience type annotations for instances are given. 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. 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 `eq` for sublists but System F_O's formalization does not actually support recursive let bindings. Extending System F and System F_O with recursive let bindings and thus recursive instances is known to be sound.

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. The overall goal will be to formally show that the

Dictionary Passing Transform is in fact correct. The translation drops `decl o in` expressions and replaces `inst o = e in e'` expressions with `let` $\texttt{o}_\tau$ `= e in e'` expressions, where $\texttt{o}_\tau$ is an unique name with respect to type τ of `e`. Constraint abstractions `ƛ (o : τ). e'` translate to normal lambda bindings λ$\texttt{o}_\tau$`. e'`. Similarly constraint types `[o : τ] ⇒ τ'` are translated to function types `τ → τ'`. Invocations of overloaded function names are translated to the correct variable name bound by the former instance, now let binding. Implicitly resolved constraints in System $\mathrm{F_O}$ must be explicitly passed as arguments in System F.

Example: Dicitionary Passing Transform

Recall the System $\mathrm{F_O}$ 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] ..
```

First we drop the `decl` expression and replace `inst` definitions with `let` bindings. Inside the second instance the constraint abstraction is translated into a normal function. Invocations of `eq` are translated to the correct unique names $\texttt{eq}_i$. When invoking $\texttt{eq}_2$ the correct instance to resolve the former constraint must be eliminated explicitly by passing $\texttt{eq}_1$ as argument.

1.4 Related Work

There exist other Systems to formalize overloading.

Bla, Bla & Bla introduced System O [CITE], a language extension to the Hindley Milner System, preserving full type inference. Aside from using Hindley Milner as base system, System O differs from System $\mathrm{F_O}$ by embedding constraints into ∀-types. Constraints can not be introduced on the expression level, instead constraints are introduced via explicit type annotations of instances. ... ?

2 Preliminary

2.1 Dependently Typed Programming in Agda

Agda is a dependently typed programming language and proof assistant. [CITE] Agdas type system is based on Martin Löf's intuitionistic type theory [CITE] and allows to construct proofs based on the Curry Howard correspondence. The Curry Howard correspondence is an isomorphic relationship between programs written in dependently typed languages and mathematical proofs written in first order logic. Because of the Curry Howard correspondence, programs in Agda correspond to proofs and formulae correspond to types. Hence, if a type checked Agda program implies that our proofs are sound, given we do not use unsafe Agda features and assuming Agda is implemented correctly. Agda is appealing to programmers, because proving in Agda is similar to functional programming using common concepts, for example pattern matching, currying and inductive data types. Further, Agda has a couple useful support features, for example proving with interactive holes and automatic proof search.

2.2 Design Decisions for the Agda Formalization

To formalize System F and System F_O in Agda we will use a single data type Term indexed by sorts s to represent the syntax. Sorts distinguish between different kind of terms, for example sort e_s for expressions and τ_s for types. Using only a single data type to formalize the syntax yields more elegant proofs involving contexts, substitutions and renamings. In consequence we must use extrinsic typing, because intrinsically typed terms Term $e_s \vdash$ Term τ_s would need to be indexed by themselves.

3 System F

3.1 Specification

We will first look at System F, our target language of the Dictionary Passing Transform. The specification includes Syntax, Typing and Semantic.

Sorts

System F only requires two sorts, e_s for expressions and τ_s for types.

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

Sorts : Set
Sorts = List Sort
```

Going forward, we use s as variable name for sorts and S for a list of sorts.

Syntax

System F's syntax is represented in a single data type Term indexed by a list of sorts S and a sort s. The length of S represents the amount of bound variables and the elements s_i of the list provide the sort of the variable bound at that position. The second index s represents the sort of the term itself.

```
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
```

Variables ‘ x are represented as references $s \in S$ to an element in S. Memberships of type $s \in S$ are defined similar to natural numbers and can either be here refl where refl is prove we found our element or there x where x is another membership. In consequence we can only reference already bound variables, in a similar fashion to debruijn indices. The unit element tt and unit type ‘⊤ represent base types. Lambda abstractions λ‘x→ e’ result in function types $\tau_1 \Rightarrow \tau_2$ and type abstractions Λ‘α→ e’ result in forall types ∀‘α τ’. To eliminate abstractions we use application $e_1 \cdot e_2$ for lambda abstractions and type application $e \bullet \tau$ for type abstractions. Let bindings let‘x= e_2 ‘in e_1 combine abstraction and application. We will use shorthands Var S s = $s \in S$, Expr S = Term S e_s and Type S = Term S τ_s and variable names x, e and τ respectively as well as t for arbitrary Term S s.

Renaming

Renamings ρ of type Ren S_1 S_2 are defined as total functions mapping variables Var S_1 s to variables Var S_2 s preserving the sort s of the variable.

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

Applying a renaming Ren S_1 S_2 to a term Term S_1 s yield a new term Term S_2 s where variables are represented as references $s \in S_2$ to elements in S_2.

```
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)
```

When going under a binder, the renaming is extended using ext_r : Ren S_1 S_2 → Ren ($S_1 \triangleright s$) ($S_2 \triangleright s$). The weakening of a term can be defined as shifting all variables by one.

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

Since variables are represented as references to a list, we shift them by wrapping a given reference in the there constructor.

Substitution

Substitutions σ of type Sub S_1 S_2 are similar to renamings but rather than mapping variables to variables, substitutions map variables to terms.

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

Applying a substitution to a term sub : Sub S_1 S_2 $\rightarrow$ (Term S_1 s $\rightarrow$ Term S_2 s) is analogous to the applying a renaming. Single substitution is constructed by composing single_s : Sub S_1 S_2 $\rightarrow$ Term S_2 s $\rightarrow$ Sub (S_1 ▷ s) S_2 with identity substitution id_s = ‘_ of type Sub S S.

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

Context

The typing context Ctx S is indexed by sorts S similar to terms.

```
data Ctx : Sorts → Set where
  ∅ : Ctx []
  _▶_ : Ctx S → Types S s → Ctx (S ▷ s)
```

A context can either be empty ∅ or cons Γ ▶ T where T is of type Types S s. Type Types S s is defined as

```
Types : Sorts → Sort → Set
Types S eₛ = Type S
Types S τₛ = ⊤
```

and has two overlapping meanings. First, Types S s represents the type of what is stored in the context for a variable of sort s. For expressions e we expect the context to store the corresponding type τ. For types τ we store the corresponding kind of unit type ⊤, since System F only has one kind for types. Additionally Types S s represents the outcome of the typing relation $\Gamma \vdash t : T$ for terms.

Typing

The typing relation $\Gamma \vdash t : T$ relates terms t to their typing of type Types S s in context Γ.

```
data _⊢_:_ : Ctx S → Term S s → Types S s → Set where
  ⊢‘x :
    lookup Γ x ≡ τ →
    --------
    Γ ⊢ ‘ x : τ
  ⊢⊤ :
    -------
    Γ ⊢ tt : ‘⊤
```

```
⊢λ :
  Γ ▶ τ ⊢ e : wk τ' →
  --------------
  Γ ⊢ λ'x→ e : τ ⇒ τ'
⊢Λ :
  Γ ▶ tt ⊢ e : τ →
  --------------
  Γ ⊢ Λ'α→ e : ∀'α τ
⊢· :
  Γ ⊢ e₁ : τ₁ ⇒ τ₂ →
  Γ ⊢ e₂ : τ₁ →
  ------------
  Γ ⊢ e₁ · e₂ : τ₂
⊢• :
  Γ ⊢ e : ∀'α τ' →
  ------------
  Γ ⊢ e • τ : τ' [ τ ]
⊢let :
  Γ ⊢ e₂ : τ →
  Γ ▶ τ ⊢ e₁ : wk τ' →
  --------------
  Γ ⊢ let'x= e₂ 'in e₁ : τ'
⊢τ :
  -----
  Γ ⊢ τ : tt
```

Semantics

3.2 Soundness

Progress

Subject Reduction

4 System F_O

4.1 Specification

Sorts

Syntax

Renaming

Substitution

Context

Constraint Solving

Typing

5 Dictionary Passing Transform

5.1 Translation

Sorts

Terms

Renaming

Substitution

Context

5.2 Type Preservation

Renaming

Substitution

Variables

Terms

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