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 General

Overloading function names is a practical technique to overcome verbosity in real world programming languages. In every language there exist commonly used function names, especially in the form of infix operators, for example equality and arithmetics, that are defined for a variety of type combinations. Overloading the meaning of common function names and operators for multiple types eliminates the necessity for a unique name for each operator, on each type. For example, Python uses so called magic methods, that allow to overload commonly used operators used on user defined classes and Java utilizes method overloading. Both Python and Java implement rather restricted forms of overloading. Haskell supports overloading in a less restricted fashion. Haskell uses typeclasses, to solve the overloading problem.

1.2 Overloading in Haskell using Typeclasses

Essentially, typeclasses 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, the type checker determines the correct instance based on the types of the applied arguments. Furthermore, Haskell allows to constrain bound type variables α via type constraints `Tc α ⇒ τ'` to only be substituted by a concrete type τ, if there exists an instance `Tc 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 be able to call `eq` on both `Nat` and `[α]`, where `α` is a type that `eq` is already defined on. 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 polymorphic function `eq` is declared. Next, we instantiate `Eq` for `Nat`. After that, `Eq` is instantiated for `[α]`, given that an instance `Eq` exists for type `α`. Finally, we can call `eq` on elements of both `Nat` and `[Nat]`, where in the latter case, the type constraint `Eq α ⇒ ..` in the second instance resolves to the first instance.

1.3 Introducing System $\mathrm{F_O}$

In our language extension to System F [CITE] we give up high level language constructs. System $\mathrm{F_O}$ 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 $\mathrm{F_O}$

Recall the Haskell example from above. The same functionality can be expressed in System $\mathrm{F_O}$ 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 λ 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 straight forward.

1.4 Transforming System F_O to System F

The Dictionary Passing Transform translates well typed System F_O expressions to well typed System F expressions. The translation drops `decl o in` expressions and replaces `inst o = e in e'` expressions with `let` o_τ `= e in e'` expressions, where o_τ is an unique name with respect to type `τ` of `e`. Constraint abstractions λ `(o : τ). e'` translate to lambda bindings `λ`o_τ`. e'`. Similarly constraint types `[o : τ]` $\Rightarrow$ `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 F_O must be explicitly passed as arguments in System F.

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] ..
```

First we drop the `decl` expression and transform `inst` definitions to `let` bindings with unique names. Inside the second instance the constraint abstraction is translated into a lambda abstraction. 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.

1.5 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 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 [CITE]. 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, type checked Agda programs imply that 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 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 $\mathrm{F_O}$ 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 e, τ_s for types τ and κ_s for kind $\star$. 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 $\mathsf{Term}\ \mathsf{e}_s \vdash \mathsf{Term}\ \tau_s$ would need to be indexed by themselves. In the actual implementation Term has another index S, a list of sorts representing the sort of bound variables, similar to Debruijn Indices [CITE].

2.3 Verbal Formulation of the Type Preservation Proof

Our goal will be to prove that the Dictionary Passing Transform is type preserving. Let $\vdash_{F_O} t$ be any well formed System $\mathrm{F_O}$ term $\Gamma \vdash_{F_O} t : T$ where t is $\mathsf{Term}_{F_O}\ s$ and T is $\mathsf{Term}_{F_O}\ s'$ and s' is the sort of the typing result for terms of sort s. There exist two cases for typings: $\Gamma \vdash e : \tau$ and $\Gamma \vdash \tau : \star$. Let $\rightsquigarrow : (\Gamma \vdash_{F_O} t : T) \to \mathsf{Term}_F\ s$ be the Dictionary Passing Transform, translating well typed System $\mathrm{F_O}$ terms to untyped System F terms. Further let $\rightsquigarrow_\Gamma : \mathsf{Ctx}_{F_O} \to \mathsf{Ctx}_F$ be the transform of untyped contexts and $\rightsquigarrow_T : \mathsf{Term}_{F_O}\ s' \to \mathsf{Term}_F\ s'$ the transform of untyped types and kinds. We show that for all well typed System $\mathrm{F_O}$ terms $\vdash_{F_O} t$ the Dictionary Passing Transform results in well typed System F programs, that is $(\rightsquigarrow_\Gamma\ \Gamma) \vdash_F (\rightsquigarrow \vdash_{F_O} t) : (\rightsquigarrow_T\ T)$.

3 System F

3.1 Specification

Sorts

The formalization of System F requires three sorts: e_s for expressions, τ_s for types and κ_s for kinds.

```
data Sort : Ctxable → Set where
  eₛ : Sort ⊤ᶜ
  τₛ : Sort ⊤ᶜ
  κₛ : Sort ⊥ᶜ

Sorts : Set
Sorts = List (Sort ⊤ᶜ)
```

Sorts are indexed by boolean data type Ctxable indicating if terms of the sort can appear in contexts. Going forward, we use s as variable name for sorts and S for lists of sorts.

Syntax

The syntax of System F is represented as a single data type Term indexed by a list of sorts S and sort s. The length of S represents the amount of bound variables and the elements s_i of the list represent the sort of the variable bound at that position. The index S is inspired by Debruijn indices where we reference variables using numbers that count the amount many binders we need to go back to where the variable was bound. The list S extends this idea by allowing to reference variables of different sorts. The second index s represents the sort of the term itself.

```
data Term : Sorts → Sort r → 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
  ⋆           : Term S κₛ
```

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 using memberships in S. 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. All types τ have kind $\star$. 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 yields a new term Term S_2 s where variables are now 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)
ren ρ ⋆ = ⋆
```

When we encounter 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, shifting variables is simply wrapping them 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 → (Term S_1 s → Term S_2 s) is analogous to the applying a renaming. Function t [t’] substitutes the last bound variable in t with t’.

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

A single substitution single_s : Sub S_1 S_2 → Term S_2 s → Sub ($S_1 \triangleright s$) S_2 introduces t’ to an existing substitution σ. In the case of _[_] we let σ be the identity substitution id_s : Sub S S.

Context

Similar to terms typing contexts Γ of type Ctx S are indexed by sorts S. In consequence only types/kinds for bound expression/type variables can be stored in Γ.

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

A context can either be empty ∅ or cons Γ ▶ T where T is a term of the kind of sort s. The function kind-of maps sorts that can appear in contexts to the sorts of their kind.

```
kind-of e_s = τ_s
kind-of τ_s = κ_s
```

Expressions have kind τ_s, while types have kind κ_s. We will use T as shorthand for the term with sort kind-of s.

Typing

The typing relation $\Gamma \vdash t : T$ relates terms t to their typing kind T in context Γ.

```
data _⊢_:_ : Ctx S → Term S s → Term S (kind-of 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 ▶ ⋆ ⊢ 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 : ⋆
```

Rule ⊢`x says that variables ` x have type τ if x has type τ in Γ. Next, ⊢⊤ states that unit expressions tt has type `⊤. The rule for abstractions ⊢λ introduces a variable of type τ to body e. Because type τ' cannot use the introduced expression variable, we let τ' have one variable bound less and weaken it to be compatible with context Γ ▶ τ. Hence τ' is compatible in the list of bound variables S with τ to form the resulting type $\tau \Rightarrow \tau$'. Analogously type abstraction rule ⊢Λ introduces a type of kind ⋆ to body e. Application rules ⊢· and ⊢• handle application of expressions and types to expressions of type $\tau \Rightarrow \tau$' and ∀'α τ' respectively. Rule ⊢*let* combines the abstraction and application rule. Finally, rule ⊢τ indicates that all types τ are well formed and have kind ⋆. Type variables are correctly typed per definition and type constructors ∀'α and ⇒ accept arbitrary types as their arguments.

Typing Renaming & Substitution

Because of extrinsic typing both renamings and substitutions need to have typed forms. In contrast to typed substituted we do not allow arbitrary renamings. We formalized typed renamings as order preserving embeddings. The order is preserved in a sense that, if variable x_1 of type $s_1 \in S_1$ references an element with index smaller than some other variable x_2 then x_1 must reference an element with smaller index after the renaming in S_2. This restriction is necessary because variables might depend on the variables before them.

```
data _:_⇒ᵣ_ : Ren S₁ S₂ → Ctx S₁ → Ctx S₂ → Set where
  ⊢idᵣ : ∀ {Γ} → _:_⇒ᵣ_ {S₁ = S} {S₂ = S} idᵣ Γ Γ
  ⊢extᵣ : ∀ {ρ : Ren S₁ S₂} {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {T' : Term S₁ (kind-of s)} →
    ρ : Γ₁ ⇒ᵣ Γ₂ →
    (extᵣ ρ) : (Γ₁ ▶ T') ⇒ᵣ (Γ₂ ▶ ren ρ T')
  ⊢dropᵣ : ∀ {ρ : Ren S₁ S₂} {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {T' : Term S₂ (kind-of s)} →
    ρ : Γ₁ ⇒ᵣ Γ₂ →
    (dropᵣ ρ) : Γ₁ ⇒ᵣ (Γ₂ ▶ T')
```

The identity renaming ⊢id$_r$ is typed per definition. The extension of a renaming ⊢ext$_r$ allows to extend both Γ_1 and Γ_2 by T' and renamed T' respectively. Constructor ⊢ext$_r$ corresponds to the typed version of function ext$_r$ used when encountering a binder. Further, constructor ⊢drop$_r$ allows us to introduce T' only in Γ_2. Hence ⊢drop$_r$ ⊢id$_r$ corresponds to the typed weakening of a term. Typed Substitutions are defined as total function, similar to untyped substitutions.

```
_:_⇒ₛ_ : Sub S₁ S₂ → Ctx S₁ → Ctx S₂ → Set
_:_⇒ₛ_ {S₁ = S₁} σ Γ₁ Γ₂ = ∀ {s} (x : Var S₁ s) → Γ₂ ⊢ σ x : (sub σ (lookup Γ₁ x))
```

Typed substitutions ⊢σ map variables x to the corresponding typed term σ x, that is $\Gamma \vdash \sigma\ x$: (sub σ (lookup Γ_1 x)).

Semantics

The semantics are formalized call-by-value, that is, there is no reduction under binders. Values are indexed by there irreducible expression.

```
data Val : Expr S → Set where
  v-λ : Val (λ'x→ e)
  v-Λ : Val (Λ'α→ e)
  v-tt : ∀ {S} → Val (tt {S = S})
```

System F has three values. The two closure values v-λ and v-Λ for abstractions waiting for their argument and unit value v-tt. We formalize semantics as small step semantics, where each constructor represents a single reduction step e ↪ e'. We distinguish between β and ξ rules. Meaningful computation in the form of substitution is done by β rules while ξ rules reduce sub expressions.

```
data _↪_ : Expr S → Expr S → Set where
  β-λ :
    Val e₂ →
    (λ‘x→ e₁) · e₂ ↪ (e₁ [ e₂ ])
  β-Λ :
    (Λ‘α→ e) • τ ↪ e [ τ ]
  β-let :
    Val e₂ →
    let‘x= e₂ ‘in e₁ ↪ (e₁ [ e₂ ])
  ξ-·₁ :
    e₁ ↪ e →
    --------
    e₁ · e₂ ↪ e · e₂
  ξ-·₂ :
    e₂ ↪ e →
    Val e₁ →
    e₁ · e₂ ↪ e₁ · e
  ξ-• :
    e ↪ e' →
    --------
    e • τ ↪ e' • τ
  ξ-let :
    e₂ ↪ e →
    let‘x= e₂ ‘in e₁ ↪ let‘x= e ‘in e₁
```

Rules β-λ and β-Λ give meaning to application and type application in the form of substituting the applied term into the abstraction. Further, β-let is equivalent to application rule β-λ. Rules ξ-·$_i$ and ξ-• evaluate sub expressions of application until e_1 and e_2, or e respectively, are values. Finally, ξ-let reduces the bound expression e_2 until e_2 is a value and β-let can be applied.

3.2 Soundness

Progress

We prove progress, that is, a typed expression $\Gamma \vdash e : \tau$ can either be further reduced to some e' or e is a value, by induction over the typing rules.

```
progress :
  ∅ ⊢ e : τ →
  (∃[ e' ] (e ↪ e')) ⊎ Val e
progress ⊢⊤ = inj₂ v-tt
progress (⊢λ _) = inj₂ v-λ
progress (⊢Λ _) = inj₂ v-Λ
progress (⊢· {e₁ = e₁} {e₂ = e₂} ⊢e₁ ⊢e₂) with progress ⊢e₁ | progress ⊢e₂
... | inj₁ (e₁' , e₁↪e₁') | _ = inj₁ (e₁' · e₂ , ξ-·₁ e₁↪e₁')
... | inj₂ v | inj₁ (e₂' , e₂↪e₂') = inj₁ (e₁ · e₂' , ξ-·₂ e₂↪e₂' v)
... | inj₂ (v-λ {e = e₁}) | inj₂ v = inj₁ (e₁ [ e₂ ] , β-λ v)
progress (⊢• {τ = τ} ⊢e) with progress ⊢e
... | inj₁ (e' , e↪e') = inj₁ (e' • τ , ξ-• e↪e')
... | inj₂ (v-Λ {e = e}) = inj₁ (e [ τ ] , β-Λ)
progress (⊢let {e₂ = e₂} {e₁ = e₁} ⊢e₂ ⊢e₁) with progress ⊢e₂
... | inj₁ (e₂' , e₂↪e₂') = inj₁ ((let`x= e₂' `in e₁) , ξ-let e₂↪e₂')
... | inj₂ v = inj₁ (e₁ [ e₂ ] , β-let v)
```

Cases ⊢⊤, ⊢λ and ⊢Λ result in values. Application cases ⊢·, ⊢• and ⊢let follow directly from the induction hypothesis.

Subject Reduction

We prove subject reduction, that is, reductions preserve typing. More specifically, an expression e with type τ still has type τ after being reduced to e'. We prove subject reduction by induction over the reduction rules.

```
subject-reduction : ∀ {Γ : Ctx S} →
  Γ ⊢ e : τ →
  e ↪ e' →
  Γ ⊢ e' : τ
subject-reduction (⊢· (⊢λ ⊢e₁) ⊢e₂) (β-λ v₂) = e[e]-preserves ⊢e₁ ⊢e₂
subject-reduction (⊢· ⊢e₁ ⊢e₂) (ξ-·₁ e₁↪e) = ⊢· (subject-reduction ⊢e₁ e₁↪e) ⊢e₂
subject-reduction (⊢· ⊢e₁ ⊢e₂) (ξ-·₂ e₂↪e x) = ⊢· ⊢e₁ (subject-reduction ⊢e₂ e₂↪e)
subject-reduction (⊢• (⊢Λ ⊢e)) β-Λ = e[τ]-preserves ⊢e ⊢τ
subject-reduction (⊢• ⊢e) (ξ-• e↪e') = ⊢• (subject-reduction ⊢e e↪e')
subject-reduction (⊢let ⊢e₂ ⊢e₁) (β-let v₂) = e[e]-preserves ⊢e₁ ⊢e₂
subject-reduction (⊢let ⊢e₂ ⊢e₁) (ξ-let e₂↪e') = ⊢let (subject-reduction ⊢e₂ e₂↪e') ⊢e₁
```

Cases ξ-·$_1$, ξ-·$_2$, ξ-• and ξ-let follow directly from the induction hypothesis. For beta reduction cases β-λ, β–Λ and β-let we need to prove that substitution preserves typing for both substitutions e [e] and e [τ]. B Both lemmas follow from a more general lemma ⊢σ-preserves.

```
⊢σ-preserves : ∀ {σ : Sub S₁ S₂} {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂}
                 {t : Term S₁ s} {T : Term S₁ (kind-of s)} →
  σ : Γ₁ ⇒ₛ Γ₂ →
  Γ₁ ⊢ t : T →
  Γ₂ ⊢ (sub σ t) : (sub σ T)
```

Lemma ⊢σ-preserves follows by induction over typing rules and some other lemmas about the interaction between renamings and substitutions. Soundness follows as a consequence of progress and subject reduction.

4 System F_O

4.1 Specification

Sorts

```
data Sort : Ctxable → Set where
  oₛ : Sort ⊤ᶜ
  cₛ : Sort ⊥ᶜ
  - ...
```

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
```

... Cstr S = Term S c_s

Renaming & Substitution

```
_[_] : Type (S ▷ τₛ) → Type S → Type S
τ [ τ' ] = sub (single-typeₛ idₛ τ') τ
```

Context

```
item-of eₛ = τₛ
item-of τₛ = κₛ
item-of oₛ = κₛ
```

..

```
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’)
```

Typing

```
kind-of e_s = τ_s
kind-of τ_s = κ_s
kind-of o_s = τ_s
```

```
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’
```

Typing Renaming & Substitution

```
data _:_⇒ᵣ_ : Ren S₁ S₂ → Ctx S₁ → Ctx S₂ -> Set where
  ⊢ext-instᵣ : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {τ} {o} →
    ρ : Γ₁ ⇒ᵣ Γ₂ →
    --------------------
    ρ : (Γ₁ ▸ (o : τ)) ⇒ᵣ (Γ₂ ▸ (ren ρ o : ren ρ τ))
  ⊢drop-instᵣ : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {τ} {o} →
    ρ : Γ₁ ⇒ᵣ Γ₂ →
    -------
    ρ : Γ₁ ⇒ᵣ (Γ₂ ▸ (o : τ))
  - ...
```

```
data _:_⇒s_ : Sub S₁ S₂ → Ctx S₁ → Ctx S₂ -> Set where
  ⊢ids : ∀ {Γ} → _:_⇒s_ {S₁ = S} {S₂ = S} ids Γ Γ
  ⊢keeps : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {I : Term S₁ (item-of s)} →
    σ : Γ₁ ⇒s Γ₂ →
    ------------------
    exts σ : Γ₁ ▶ I ⇒s Γ₂ ▶ sub σ I
  ⊢drops : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {I : Term S₂ (item-of s)} →
    σ : Γ₁ ⇒s Γ₂ →
    -------------
    drops σ : Γ₁ ⇒s (Γ₂ ▶ I)
  ⊢types : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {τ : Type S₂} →
    σ : Γ₁ ⇒s Γ₂ →
    -------
    single-types σ τ : Γ₁ ▶ ⋆ ⇒s Γ₂
  ⊢keep-insts : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {τ} {o} →
    σ : Γ₁ ⇒s Γ₂ →
    --------------------
    σ : (Γ₁ ▸ (o : τ)) ⇒s (Γ₂ ▸ (sub σ o : sub σ τ))
  ⊢drop-insts : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {τ} {o} →
    σ : Γ₁ ⇒s Γ₂ →
    -------
    σ : Γ₁ ⇒s (Γ₂ ▸ (o : τ))
```

5 Dictionary Passing Transform

5.1 Translation

Sorts

```
s⇝s : Fᴼ.Sort ⊤ᶜ → F.Sort ⊤ᶜ
s⇝s es = es
s⇝s os = es
s⇝s τs = τs

Γ⇝S : Fᴼ.Ctx Fᴼ.S → F.Sorts
Γ⇝S ∅ = []
Γ⇝S (Γ ▸ c) = Γ⇝S Γ ▷ F.es
Γ⇝S {S ▷ s} (Γ ▶ x) = Γ⇝S Γ ▷ s⇝s s
```

Terms

```
τ⇝τ : ∀ {Γ : Fᴼ.Ctx Fᴼ.S} →
  Fᴼ.Type Fᴼ.S →
  F.Type (Γ⇝S Γ)
τ⇝τ (‘ x) = ‘ x⇝x x
τ⇝τ ‘⊤ = ‘⊤
τ⇝τ (τ₁ ⇒ τ₂) = τ⇝τ τ₁ ⇒ τ⇝τ τ₂
τ⇝τ {Γ = Γ} (F^O.∀‘α τ) = F.∀‘α τ⇝τ {Γ = Γ ▶ ⋆} τ
τ⇝τ ([ o : τ ]⇒ τ’) = τ⇝τ τ ⇒ τ⇝τ τ’

T⇝T : ∀ {Γ : F^O.Ctx F^O.S} →
  F^O.Term F^O.S (F^O.kind-of F^O.s) →
  F.Term (Γ⇝S Γ) (F.kind-of (s⇝s F^O.s))
T⇝T {s = e_s} τ = τ⇝τ τ
T⇝T {s = o_s} τ = τ⇝τ τ
T⇝T {s = τ_s} _ = ⋆

⊢t⇝t : ∀ {Γ : F^O.Ctx F^O.S} {t : F^O.Term F^O.S F^O.s} {T : F^O.Term F^O.S (F^O.kind-of F^O.s)} →
  Γ F^O.⊢ t : T →
  F.Term (Γ⇝S Γ) (s⇝s F^O.s)
⊢t⇝t (⊢‘x {x = x} Γx≡τ) = ‘ x⇝x x
⊢t⇝t (⊢‘o o:τ∈Γ) = ‘ o:τ∈Γ⇝x o:τ∈Γ
⊢t⇝t ⊢⊤ = tt
⊢t⇝t (⊢λ ⊢e) = λ‘x→ (⊢t⇝t ⊢e)
⊢t⇝t (⊢Λ ⊢e) = Λ‘α→ (⊢t⇝t ⊢e)
⊢t⇝t (⊢ƛ ⊢e) = λ‘x→ (⊢t⇝t ⊢e)
⊢t⇝t (⊢· ⊢e₁ ⊢e₂) = ⊢t⇝t ⊢e₁ · ⊢t⇝t ⊢e₂
⊢t⇝t (⊢• {τ = τ} ⊢e) = ⊢t⇝t ⊢e • (τ⇝τ τ)
⊢t⇝t (⊢⊘ ⊢e o:τ∈Γ) = ⊢t⇝t ⊢e · ‘ o:τ∈Γ⇝x o:τ∈Γ
⊢t⇝t (⊢let ⊢e₂ ⊢e₁) = let‘x= ⊢t⇝t ⊢e₂ ‘in ⊢t⇝t ⊢e₁
⊢t⇝t (⊢decl ⊢e) = let‘x= tt ‘in ⊢t⇝t ⊢e
⊢t⇝t (⊢inst ⊢e₂ ⊢e₁) = let‘x= ⊢t⇝t ⊢e₂ ‘in ⊢t⇝t ⊢e₁
```

Renaming

```
⊢ρ⇝ρ : ∀ {ρ : F^O.Ren F^O.S₁ F^O.S₂} {Γ₁ : F^O.Ctx F^O.S₁} {Γ₂ : F^O.Ctx F^O.S₂} →
  ρ F^O.: Γ₁ ⇒ᵣ Γ₂ →
  F.Ren (Γ⇝S Γ₁) (Γ⇝S Γ₂)
⊢ρ⇝ρ ⊢idᵣ = id
⊢ρ⇝ρ (⊢extᵣ ⊢ρ) = F.extᵣ (⊢ρ⇝ρ ⊢ρ)
⊢ρ⇝ρ (⊢dropᵣ ⊢ρ) = F.dropᵣ (⊢ρ⇝ρ ⊢ρ)
⊢ρ⇝ρ (⊢ext-instᵣ ⊢ρ) = F.extᵣ (⊢ρ⇝ρ ⊢ρ)
⊢ρ⇝ρ (⊢drop-instᵣ ⊢ρ) = F.dropᵣ (⊢ρ⇝ρ ⊢ρ)
```

Substitution

```
⊢σ⇝σ : ∀ {σ : Fᴼ.Sub Fᴼ.S₁ Fᴼ.S₂} {Γ₁ : Fᴼ.Ctx Fᴼ.S₁} {Γ₂ : Fᴼ.Ctx Fᴼ.S₂} →
  σ Fᴼ.: Γ₁ ⇒ₛ Γ₂ →
  F.Sub (Γ⇝S Γ₁) (Γ⇝S Γ₂)
⊢σ⇝σ ⊢idₛ = F.‘_
⊢σ⇝σ (⊢keepₛ ⊢σ) = F.extₛ (⊢σ⇝σ ⊢σ)
⊢σ⇝σ (⊢dropₛ ⊢σ) = F.dropₛ (⊢σ⇝σ ⊢σ)
⊢σ⇝σ (⊢typeₛ {τ = τ} ⊢σ) = F.singleₛ (⊢σ⇝σ ⊢σ) (τ⇝τ τ)
⊢σ⇝σ (⊢keep-instₛ ⊢σ) = F.extₛ (⊢σ⇝σ ⊢σ)
⊢σ⇝σ (⊢drop-instₛ ⊢σ) = F.dropₛ (⊢σ⇝σ ⊢σ)
```

Context

```
Γ⇝Γ : (Γ : Fᴼ.Ctx Fᴼ.S) → F.Ctx (Γ⇝S Γ)
Γ⇝Γ ∅ = ∅
Γ⇝Γ (Γ ▶ I) = (Γ⇝Γ Γ) ▶ I⇝T I
Γ⇝Γ (Γ ▸ (‘ o : τ)) = (Γ⇝Γ Γ) ▶ τ⇝τ τ
```

5.2 Type Preservation

Terms

```
⊢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 (⊢λ {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))
- ...
```

Variables

```
Γx≡τ⇝Γx≡τ : ∀ {Γ : Fᴼ.Ctx Fᴼ.S} {τ : Fᴼ.Type Fᴼ.S} (x : Fᴼ.Var Fᴼ.S eₛ) →
  Fᴼ.lookup Γ x ≡ τ →
  F.lookup (Γ⇝Γ Γ) (x⇝x x) ≡ (τ⇝τ τ)
Γx≡τ⇝Γx≡τ {Γ = Γ ▶ τ} (here refl) refl = ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ Fᴼ.⊢wkᵣ τ
Γx≡τ⇝Γx≡τ {Γ = Γ ▶ _} {τ’} (there x) refl = trans
  (cong F.wk (Γx≡τ⇝Γx≡τ x refl))
  (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ Fᴼ.⊢wkᵣ (Fᴼ.lookup Γ x))
Γx≡τ⇝Γx≡τ {Γ = Γ ▸ c@(‘ o : τ’)} {τ} x refl = (
  begin
    F.wk (F.lookup (Γ⇝Γ Γ) (x⇝x x))
  ≡⟨ cong F.wk (Γx≡τ⇝Γx≡τ x refl) ⟩
    F.wk (τ⇝τ τ)
  ≡⟨ ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢wk-inst_r τ ⟩
    τ⇝τ (F^O.ren F^O.id_r τ)
  ≡⟨ cong τ⇝τ (id_rτ≡τ τ) ⟩
    τ⇝τ τ
  ∎)


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

Renaming

```
  (⊢ρ⇝ρ ⊢ρ) (x⇝x x) ≡ x⇝x (ρ x)

F.ren (⊢ρ⇝ρ ⊢ρ) (τ⇝τ τ) ≡ τ⇝τ (F^O.ren ρ τ)τ⇝τ {Γ = Γ ▶ I} (F^O.wk τ’) ≡ F.wk
(τ⇝τ τ’)τ⇝τ {Γ = Γ ▸ (‘ o : τ’)} τ ≡ F.wk (τ⇝τ τ)
```

Substitution

```
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x : {σ : F^O.Sub F^O.S_1 F^O.S_2} {Γ_1 : F^O.Ctx F^O.S_1} {Γ_2 : F^O.Ctx F^O.S_2} →
    (⊢σ : σ F^O.: Γ_1 ⇒_s Γ_2) →
    (x : F^O.Var F^O.S_1 τ_s) →
    F.sub (⊢σ⇝σ ⊢σ) (‘ x⇝x x) ≡ τ⇝τ (F^O.sub σ (‘ x))
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢id_s x = refl
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢keep_s ⊢σ) (here refl) = refl
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢keep_s {σ = σ} ⊢σ) (there x) = trans
    (cong F.wk (⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢σ x)) (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ F^O.⊢wk_r (σ x))
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢drop_s {σ = σ} ⊢σ) x = trans
    (cong F.wk (⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢σ x)) (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ F^O.⊢wk_r (σ x))
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢type_s ⊢σ) (here refl) = refl
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢type_s ⊢σ) (there x) = ⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢σ x
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢keep-inst_s {σ = σ} ⊢σ) x = trans (cong F.wk (⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢σ x)) (
    begin
      F.wk (τ⇝τ (σ x))
    ≡⟨ (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢wk-inst_r (σ x)) ⟩
      τ⇝τ (F^O.ren F^O.id_r (σ x))
    ≡⟨ cong τ⇝τ (id_rτ≡τ (σ x)) ⟩
      τ⇝τ (σ x)
    ∎)
  ⊢σ⇝σ·x⇝x≡τ⇝σ·x(⊢drop-inst_s {σ = σ} ⊢σ) x = trans (cong F.wk (⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢σ x)) (
    begin
      F.wk (τ⇝τ (σ x))
    ≡⟨ ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢wk-inst_r (σ x) ⟩
      τ⇝τ (F^O.ren F^O.id_r (σ x))
    ≡⟨ cong τ⇝τ (id_rτ≡τ (σ x)) ⟩
      τ⇝τ (σ x)
    ∎)


  ⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ : ∀ {σ : F^O.Sub F^O.S_1 F^O.S_2} {Γ_1 : F^O.Ctx F^O.S_1} {Γ_2 : F^O.Ctx F^O.S_2} →
    (⊢σ : σ F^O.: Γ_1 ⇒_s Γ_2) →
    (τ : F^O.Type F^O.S_1) →
  F.sub (⊢σ⇝σ ⊢σ) (τ⇝τ τ) ≡ τ⇝τ (F^O.sub σ τ)
  ⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ ⊢σ τ = ?
```

```
  τ'⇝τ'[τ⇝τ]≡τ⇝τ'[τ] τ τ' = ⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ ⊢single-type_s τ'
```

6 Further Work and Conclusion

6.1 Hindley Milner with Overloading

In this scenario our source language for the Dictionary Passing Transform would be an extended Hindley Milner HM_{O} and our target language HM. HM is a restricted form of System F. HM has two sorts m_s for mono types and p_s for poly types in favour of arbitrary types τ_s. Poly types can include forall quantifiers, while mono types consist only of primitive types and type variables. Usually all language constructs are restricted to mono types, except let bound variables. Hence polymorphism in HM is also called let polymorphism [CITE]. In consequence, constraint abstractions would only allow to introduce constraints for overloaded variables with mono types. Further, we need to restrict all instances for some overloaded variable o to differ in the type of their first argument. With these restrictions type inference, using an extended version of Algorithm W, is preserved [CITE]. Formalizing the changes and restrictions mentions above should be a fairly straight forward adjustment to the formalization given in this thesis.

6.2 Semantic Preservation of System F_{O}

For now System F_{O} does not have semantics formalized. Semantics for System F_{O} would need to be typed semantics, because applications ' o · e_1 .. · e_n need type information to reduce properly. We need to resolve the correct instance for o based on the type arguments e_1 .. e_n. Let ⊢e ↪ ⊢e' be such a typed small step semantic for System F_{O}. We would need to prove something similar to: If ⊢e ↪ ⊢e' then ∃[e”] (⊢e↪e'⇝e↪e' ⊢e ↪* e”) × (⊢e↪e'⇝e↪e' ⊢e' ↪* e”), where ⊢e↪e'⇝e↪e' translates typed System F_{O} reductions to a untyped System F reductions. Instead of translating reduction steps directly, we prove that both translated ⊢e and ⊢e' reduce to e” in System F using finite many reduction steps. We need this more general formulation, because there might be more reduction steps in the translated System F program than in the System F_{O} program. For example, an implicitly resolved constraint in System F_{O} needs to be explicitly passed using a additional application step in System F. We believe semantic preservation can be shown via induction over the semantic rules.

6.3 Conclusion

We have formalized both System F and System F_{O} in Agda. System F_{O} acts as core calculus, capturing the essence of higher level type features, for example typeclasses in Haskell and traits in Rust. Using Agda we formalized the Dictionary Passing Transform between System F and System F_{O} and proved it to be type preserving.

References

Declaration