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 typeclasses in Haskell or traits in Rust. We introduce System $\mathrm{F_O}$, a minimal language extension to System F, with support for overloading. We show that the Dictionary Passing Transform from System $\mathrm{F_O}$ 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 and operators that are defined for a variety of type combinations. Overloading the meaning of function names for different type combinations solves the unique name problem and helps overcome verbosity. Python uses magic methods to overload commonly used operators on user defined classes and Java utilizes method overloading. Both Python and Java implement rather restricted forms of overloading. Haskell solves the overloading problem with a more general concept called typeclasses.

1.2 Overloading in Haskell using Typeclasses

Essentially, typeclasses allow to declare function names with generic type signatures. We can give one of possibly many meanings to a typeclass by instantiating the typeclass for some concrete type. Instantiating a typeclass gives a concrete implementation to all the functions defined by the typeclass. When we invoke an overloaded function name defined by a typeclass, we expect the compiler to determine the correct instance based on the types of the arguments applied. Furthermore, Haskell allows to constrain bound type variables `α` via type constraints `Tc α ⇒ τ'`, to only be substituted by concrete types `τ`, if there exists an instance `Tc τ`.

Example: Overloading Equality in Haskell

In this example we want to overload the function `eq : α → α → Bool` with different meanings for different substitutions $\{\alpha \mapsto \tau\}$. We want to be able to call `eq` on both {`α` ↦ `Nat`} and {`α` ↦ `[β]`}, where `β` is a concrete type and there exists an instance `eq : β → β → Bool`. The intuition here is that we want to be able to compare natural numbers `Nat` and lists `[β]`, if the elements of type `β` are known to be comparable.

```
class Eq α where
  eq :: α → α → Bool

instance Eq Nat where
  eq x y = x ≐ y
instance Eq β ⇒ Eq [β] where
  eq []       []       = True
  eq (x : xs) (y : ys) = eq x y && eq xs ys

.. eq 42 0 .. eq [42, 0] [42, 0] ..
```

First, typeclass `Eq`, with a single generic function signature `eq :: α → α → Bool`, is declared. Next, we instantiate `Eq` for {`α` ↦ `Nat`}. After that, `Eq` is instantiated for {`α` ↦ `[β]`}, given that an instance `Eq β` can be found. Finally, we can call `eq` on elements of both `Nat` and `[Nat]`. In the latter case, the type constraint `Eq β ⇒ ..` in the second instance resolves to the first instance.

1.3 Introducing System F_O

In our language extension to System F [CITE] we give up high level language constructs. System F_O desugars typeclass 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, we allow to overload `o` with arbitrary types. Locally shadowing other instances of the same type is allowed. Constraints can be introduced using the constraint abstraction `ƛ (o : τ). e'`. Constraint abstractions result 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. For convenience type annotations for instances are given.

```
decl eq in

inst eq : Nat → Nat → Bool
  = λx. λy. .. in
inst eq : ∀β. [eq : β → β → Bool] ⇒ [β] → [β] → Bool
  = Λβ. ƛ(eq : β → β → 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 equality on `Nat`. Next, we instantiate `eq` for equality on lists `[β]`, given the constraint `eq : β → β → Bool` introduced by the constraint abstraction `ƛ` is satisfied. 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 recursion. 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 removes all `decl o in e` expressions. Instance expressions `inst o = e in e'` are replaced with `let` $\texttt{o}_\tau$ `= e in e'` expressions, where $\texttt{o}_\tau$ is an unique name with respect to type `τ` of expression `e`. Constraint abstractions `ƛ (o : τ). e'` translate to normal abstractions `λ`$\texttt{o}_\tau$`. e'`. Hence, constraint types `[o : τ] ⇒ τ'` are translated to function types `τ → τ'`. Invocations of overloaded function names `o` translate to the correct unique variable name $\texttt{o}_\tau$ bound by the translated instance. 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 represent unique names. Applying the Dictionary Passing Transform to well typed System F_O results in well typed System F.

```
let eq₁ : Nat → Nat → Bool
  = λx. λy. .. in
let eq₂ : ∀β. (β → β → Bool) → [β] → [β] → Bool
  = Λβ. 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 normal lambda abstraction. Invocations of `eq` are translated to the correct unique variables $\texttt{eq}_i$. When invoking $\texttt{eq}_2$ the correct instance to resolve the former constraint, now higher order function, must be applied explicitly by passing instance $\texttt{eq}_1$ 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 $\forall$-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. Thus, type checked Agda programs imply the correctness of the corresponding proofs, given we do not use unsafe Agda features and assuming Agda is implemented correctly.

2.2 Design Decisions for the Agda Formalization

To formalize System F and System F_O in Agda we use a single data type Term indexed by sorts s to represent the syntax. Sorts distinguish between different kinds of terms. For example, sort e_s categorizes expressions e, τ_s categorizes τ and κ_s is used to categorizes the only existing kind $\star$. Using 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 and Agda does not allow that. In the actual implementation Term has another index S, that we will ignore for now.

2.3 Verbal Formulation of the Type Preservation Proof

Our goal will be to prove that the Dictionary Passing Transform is type preserving. Let $\vdash t$ be any well formed System $\mathrm{F_O}$ term $\Gamma \vdash_{F_O} t : T$, where t is a $\mathsf{Term}_{F_O}\ s$, T is a $\mathsf{Term}_{F_O}\ s'$ 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 that translates 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 contexts and $\rightsquigarrow_T : \mathsf{Term}_{F_O}\ s' \to \mathsf{Term}_F\ s'$ be the transform of untyped types and kinds. We show that for all well typed System $\mathrm{F_O}$ terms $\vdash t$ the Dictionary Passing Transform results in a well typed System F term $(\rightsquigarrow_\Gamma \Gamma) \vdash_F (\rightsquigarrow \vdash t) : (\rightsquigarrow_T T)$.

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 are indexed by boolean data type Ctxable. Index $\top^C$ indicates that variables for terms of some sort s can be bound. In contrast, $\bot^C$ says that variables for terms of some s cannot be bound. Hence, System F support abstractions over expressions and types, but not over kinds. Going forward we use shorthand $\mathsf{Sorts} = \mathsf{List}\ (\mathsf{Sort}\ \top^C)$, variable s for sorts and variable S for lists of contextable sorts.

Syntax

The syntax of System F is represented in a single data type Term, indexed by sorts S and sort s. The index s represents the sort of the term itself. The second index S is inspired by Debruijn indices. Debruijn indices reference variables using a number that counts the amount of binders that are in scope between the binding of the variable and the position it is used. In Agda terms are often indexed by the amount of bound variables. The variable constructor then only accepts Debruijn indices that are smaller or equal than the current amount of bound variables. Thus, unbound variables can not be referenced by definition. But indexing our term with a number is not sufficient, since System F has both expression and type variables. To solve this problem, we need to extend the idea of Debruijn indices and distinguish variables of different sorts. We let S be a list of sorts instead of a number. 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 debruijn index.

```
data Term : Sorts → Sort r → Set where
  ‘_          : s ∈ S → Term S s
```

$$
\begin{array}{ll}
\mathsf{tt} & : \mathsf{Term}\ S\ \mathsf{e}_s \\
\lambda\text{`x}\to\_ & : \mathsf{Term}\ (S \rhd \mathsf{e}_s)\ \mathsf{e}_s \to \mathsf{Term}\ S\ \mathsf{e}_s \\
\Lambda\text{`}\alpha\to\_ & : \mathsf{Term}\ (S \rhd \tau_s)\ \mathsf{e}_s \to \mathsf{Term}\ S\ \mathsf{e}_s \\
\_\cdot\_ & : \mathsf{Term}\ S\ \mathsf{e}_s \to \mathsf{Term}\ S\ \mathsf{e}_s \to \mathsf{Term}\ S\ \mathsf{e}_s \\
\_\bullet\_ & : \mathsf{Term}\ S\ \mathsf{e}_s \to \mathsf{Term}\ S\ \tau_s \to \mathsf{Term}\ S\ \mathsf{e}_s \\
\mathsf{let`x}{=}\_\text{`in}\_ & : \mathsf{Term}\ S\ \mathsf{e}_s \to \mathsf{Term}\ (S \rhd \mathsf{e}_s)\ \mathsf{e}_s \to \mathsf{Term}\ S\ \mathsf{e}_s \\
\text{`}\top & : \mathsf{Term}\ S\ \tau_s \\
\_\Rightarrow\_ & : \mathsf{Term}\ S\ \tau_s \to \mathsf{Term}\ S\ \tau_s \to \mathsf{Term}\ S\ \tau_s \\
\forall\text{`}\alpha\_ & : \mathsf{Term}\ (S \rhd \tau_s)\ \tau_s \to \mathsf{Term}\ S\ \tau_s \\
\star & : \mathsf{Term}\ S\ \kappa_s
\end{array}
$$

Variables $\text{`}\ x$ are represented as references $s \in S$ to an element in S. Memberships of type $s \in S$ are defined similar to natural numbers. Memberships 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. The unit element tt and unit type $\text{`}\top$ represent base types. Lambda abstractions $\lambda\text{`x}\to\ e'$ result in function types $\tau_1 \Rightarrow \tau_2$ and type abstractions $\Lambda\text{`}\alpha\to\ e'$ result in forall types $\forall\text{`}\alpha\ \tau'$. To eliminate abstractions we use application $e_1 \cdot e_2$ and type application $e \bullet \tau$ to eliminate type abstractions. Let bindings $\mathsf{let`x}{=}\ e_2\ \text{`in}\ e_1$ combine abstraction and application. All types τ have kind $\star$. We use shorthands $\mathsf{Var}\ S\ s = s \in S$, $\mathsf{Expr}\ S = \mathsf{Term}\ S\ \mathsf{e}_s$, $\mathsf{Type}\ S = \mathsf{Term}\ S\ \tau_s$ and variable names x, e and τ respectively, as well as t for arbitrary $\mathsf{Term}\ S\ s$.

Renaming

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

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

Applying a renaming $\mathsf{Ren}\ S_1\ S_2$ to a term $\mathsf{Term}\ S_1\ s$ yields a new term $\mathsf{Term}\ S_2\ s$ where variables are now represented as references to elements in S_2.

$$
\begin{array}{l}
\mathsf{ren} : \mathsf{Ren}\ S_1\ S_2 \to (\mathsf{Term}\ S_1\ s \to \mathsf{Term}\ S_2\ s) \\
\mathsf{ren}\ \rho\ (\text{`}\ x) = \text{`}\ (\rho\ x) \\
\mathsf{ren}\ \rho\ \mathsf{tt} = \mathsf{tt} \\
\mathsf{ren}\ \rho\ (\lambda\text{`x}\to\ e) = \lambda\text{`x}\to\ (\mathsf{ren}\ (\mathsf{ext}_r\ \rho)\ e) \\
\mathsf{ren}\ \rho\ (\Lambda\text{`}\alpha\to\ e) = \Lambda\text{`}\alpha\to\ (\mathsf{ren}\ (\mathsf{ext}_r\ \rho)\ e) \\
\mathsf{ren}\ \rho\ (e_1 \cdot e_2) = (\mathsf{ren}\ \rho\ e_1) \cdot (\mathsf{ren}\ \rho\ e_2) \\
\mathsf{ren}\ \rho\ (e \bullet \tau) = (\mathsf{ren}\ \rho\ e) \bullet (\mathsf{ren}\ \rho\ \tau) \\
\mathsf{ren}\ \rho\ (\mathsf{let`x}{=}\ e_2\ \text{`in}\ e_1) = \mathsf{let`x}{=}\ (\mathsf{ren}\ \rho\ e_2)\ \text{`in}\ \mathsf{ren}\ (\mathsf{ext}_r\ \rho)\ e_1 \\
\mathsf{ren}\ \rho\ \text{`}\top = \text{`}\top \\
\mathsf{ren}\ \rho\ (\tau_1 \Rightarrow \tau_2) = \mathsf{ren}\ \rho\ \tau_1 \Rightarrow \mathsf{ren}\ \rho\ \tau_2 \\
\mathsf{ren}\ \rho\ (\forall\text{`}\alpha\ \tau) = \forall\text{`}\alpha\ (\mathsf{ren}\ (\mathsf{ext}_r\ \rho)\ \tau) \\
\mathsf{ren}\ \rho\ \star = \star
\end{array}
$$

When we encounter a binder for a term of sort s, the renaming is extended using $\mathsf{ext}_r : \mathsf{Ren}\ S_1\ S_2 \to \mathsf{Ren}\ (S_1 \rhd s)\ (S_2 \rhd s)$. 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, using the sub function, is analogous to applying a renaming using ren. Substitution operator 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 singleₛ : Sub S_1 S_2 → Term S_2 s → Sub (S_1 ▷ s) S_2 introduces t' to an existing substitution σ'. In the case of _[_] we let σ' be the identity substitution idₛ : Sub S S.

Context

Similar to terms, typing contexts Γ of type Ctx S are also indexed by S. In consequence only types and kinds for already bound 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 sort kind-of s. The function kind-of maps contextable sorts s to sorts s'. For variables of sort s a term of sort s' is stored in Γ.

```
kind-of eₛ = τₛ
kind-of τₛ = κₛ
```

Expressions variables require Γ to store the corresponding type and types need their kind stored in Γ. 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 result 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 a variable ‘ x has type τ if x has type τ in Γ. Next, ⊢⊤ states that unit expression tt has type ‘⊤. The rule for abstractions ⊢λ introduces a variable of type τ to body e. Because body 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 with τ to form the resulting function type $\tau \Rightarrow \tau$’. Type abstraction rule ⊢Λ introduces a type of kind ⋆ to body e and results in forall type ∀‘α τ, where τ is the type of body e. Application is handled by rule ⊢· and says that, if e_1 is a function from τ_1 to τ_2 and e_2 has type τ_1, then e_1 · e_2 has type τ_2. Similarly, type application rule ⊢• states that, if e has type ∀‘α τ’ a can be substituted with another type τ in τ’. 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 counterparts. We formalize typed renamings $\vdash\rho$ as order preserving embeddings. Thus, if variable x_1 of type $s_1 \in S_1$ references an element with an index smaller than some other variable x_2 in S_1, then renamed x_1 must still reference an element with a smaller index than renamed x_2 in S_2. Arbitrary renaming would allow swapping types in the context and thus potentially violate the telescoping. Telescoping allows types in the context to depend on type variables bound before them.

```
  data _:_⇒ᵣ_ : Ren S₁ S₂ → Ctx S₁ → Ctx S₂ → Set where
    ⊢idᵣ : ∀ {Γ} → _:_⇒ᵣ_ {S₁ = S} {S₂ = S} idᵣ Γ Γ
```

```
⊢ext_r : ∀ {ρ : Ren S₁ S₂} {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {T' : Term S₁ (kind-of s)} →
  ρ : Γ₁ ⇒_r Γ₂ →
  (ext_r ρ) : (Γ₁ ▶ T') ⇒_r (Γ₂ ▶ ren ρ T')
⊢drop_r : ∀ {ρ : Ren S₁ S₂} {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {T' : Term S₂ (kind-of s)} →
  ρ : Γ₁ ⇒_r Γ₂ →
  (drop_r ρ) : Γ₁ ⇒_r (Γ₂ ▶ T')
```

The identity renaming $\vdash$id$_r$ is typed per definition. The extension of a renaming $\vdash$ext$_r$ allows to extend both Γ_1 and Γ_2 by T' and renamed T' respectively. Constructor $\vdash$ext$_r$ corresponds to the typed version of function ext$_r$, that is used when a binder is encountered. Further, constructor $\vdash$drop$_r$ allows us to introduce T' only in Γ_2. Hence, $\vdash$drop$_r$ $\vdash$id$_r$ corresponds to the typed weakening of a term.
Typed Substitutions are defined as a 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 $\vdash\sigma$ map variables $x \in S_1$ to the corresponding typing of $\sigma\ x$ in Γ_2. The typing result of $\sigma\ x$ is the original type of x in Γ_1 applied to σ.

Semantics

The semantics are formalized call-by-value. That is, there is no reduction under binders. Values are indexed by their corresponding 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 and unit value v-tt. We formalize small step semantics where each constructor represents a single reduction step $e \hookrightarrow e'$. We distinguish between β and ξ rules. Meaningful computation in the form of substitution is done by β rules while ξ rules only 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 by substituting the applied expression or term into the abstraction body. Reduction β-let is equivalent to application. Rules ξ-·$_i$ and ξ-• evaluate sub expressions of applications 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 e can either be further reduced to some e' or e is a value. The proof follows 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, reduction preserves 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 the substitutions preserve typing for both e [e] and e [τ]. Both e[e]-preserves and e[τ]-preserves 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 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

In addition to the sorts of System F, System F_O introduces two new sorts: o_s for overloaded variables and c_s for constraints.

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

Terms of sort o_s can only be constructed using the variable constructor ‘_. Variables for constraints do not exist and thus c_s is indexed by $\bot^C$.

Syntax

We only discuss additions to the syntax of System F.

```
data Term : Sorts → Sort r → Set where
  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
  - ...
```

Declarations decl‘o‘in e introduce a new overloaded variable o. Hence, S is extended by sort o_s inside the body e. Expression inst‘ ‘ $o = e_2$ ‘in e_1 gives overloaded variable o an additional meaning e in e’. Constraints c can be constructed using constructor ‘ o : τ. Constraints are part of both constraint abstractions λ $c \Rightarrow e$ and constraint types [c]⇒ τ, used to introduce constraints to the expression and type level respectively. Going forward, we will use shorthand Cstr S = Term S c_s.

Renaming & Substitution

Renamings and substitutions in System F_O are formalized identically to renamings and substitutions in System F. The only difference is that we define the substitution operator only on types.

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

Because we do not formalize semantics for System F_O only substitutions of types in types are necessary. Type in type substitution appears in the typing rule for type application.

Context

In addition to the normal context items, we also store constraints inside the context.

```
data Ctx : Sorts → Set where
  _▸_ : Ctx S → Cstr S → Ctx S
  - ...
```

We write Γ ▸ c to pick up constraint c. Constraints give an additional meaning to a overloaded variable that is already bound. Hence index S is not modified.

Constraint Solving

The search for constraints in a context is formalized analogously to membership proofs $s \in S$. The subtle difference is, that we do reference constraints in Γ and not S.

```
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-cstr : [ c ]∈ Γ → [ c ]∈ (Γ ▸ c')
```

The here constructor is analogous to the here constructor of memberships and can be used when the last item in Γ is the constraint c, the constraint that we searched for. If the last item in the context is not the constraint c, c can be further inside the context, either behind a item stored in Γ (under-bind) or constraint (under-cstr).

Typing

Again, we only discuss typing rules not already discussed in the System F specification.

```
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’
  ⊢decl :
    Γ ▶ ⋆ ⊢ e : wk τ →
    Γ ⊢ decl‘o‘in e : τ
  - ...
```

Rule ⊢‘o for overloaded variables says that, if we can resolve the constraint o : τ in Γ, then o has type τ. The rule for constraint abstraction ⊢ƛ appends constraint c to Γ and thus assumes c to be valid in body e. Expressions e with constraint type [c]⇒ τ’ have the constraint implicitly eliminated using the ⊢⊘ rule, given constraint c can be resolve in Γ. Finally, the rule ⊢decl introduces a new overloaded variable to body e. Similar to the abstraction rule, type τ is weakened to be compatible in S with Γ not extended by o to act as the resulting type of the typing.

Typing Renaming & Substitution

Typed renamings are identical to the typed renamings in System F, except there are two new cases for the constraints that can appear inside contexts.

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

Constructor ⊢ext-cstr$_r$ allows to introduce a constraint c to Γ_1 and renamed c to Γ_2, similar to ⊢ext$_r$. Extending by a constraint needs to be used when encountering constraint abstractions or instance expression. Further, constraint o : τ can only be introduced to Γ_2 using constructor ⊢drop-cstr$_r$. The latter corresponds to a typed

weakening, similar to $\vdash\mathsf{drop}_r$, but instead of introducing an unused variable we introduce an unused constraint. Because picking up constraints does not modify the index S of contexts, ρ does not need to wrapped in ext_r or drop_r respectively.
Other than in System F, arbitrary substitutions will not be allowed in System $\mathrm{F_O}$. Similar to the substitution operator we restrict typed substitutions in System $\mathrm{F_O}$ to substitutions of types in types. This restriction simplifies proofs for the type preservation of the Dictionary Passing Transform.

```
data _:_⇒s_ : Sub S₁ S₂ → Ctx S₁ → Ctx S₂ → Set where
  ⊢types : ∀ {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {τ : Type S₂} →
    σ : Γ₁ ⇒s Γ₂ →
    single-types σ τ : Γ₁ ▶ ⋆ ⇒s Γ₂
  - ...
```

Constructors $\vdash\mathsf{ext}_s$, $\vdash\mathsf{drop}_s$, $\vdash\mathsf{ext\text{-}cstr}_s$ and $\vdash\mathsf{drop\text{-}cstr}_s$ are not shown. All of them function the same way as their counterparts in typed renamings. Constructor $\vdash\mathsf{type}_s$ allows to introduce a new type τ and complements the $\mathsf{single\text{-}type}_s$ function. Hence Γ_1 is extended by kind $\star$ and Γ_2 remains unchanged. The intuition here is that, if we would allow all terms to be introduced using a $\vdash\mathsf{term}_s$ constructor, typed substitutions in System $\mathrm{F_O}$ would be arbitrary again.

5 Dictionary Passing Transform

5.1 Translation

Sorts

The translation of System $\mathrm{F_O}$ sorts to System F sorts only considers sorts that are contextable. The two missing non-contextable sorts c_s and κ_s do not need to be translated for our purpose. Intuitively there does not even exist a sensible translation for c_s.

```
s⇝s : F^O.Sort ⊤^C → F.Sort ⊤^C
s⇝s e_s = e_s
s⇝s o_s = e_s
s⇝s τ_s = τ_s
```

Sort e_s and τ_s translate to their corresponding counterparts in System F. Overloaded variables in System $\mathrm{F_O}$ are translate to normal variables in System F. Thus sort o_s translates to e_s.
Translating lists S directly is not possible, because there might appear additional sorts inside the list after the translation. New sorts must be added for variable bindings introduced by the translation. For example, a inst' ' $o = e_2$ 'in e_1 expression does not bind a new variable in e_1, but translates to a let'x= e_2 'in e_1 binding. Hence S must have a new entry e_s at the corresponding position to further function as valid index for the translated e_1. To solve this problem the System $\mathrm{F_O}$ context Γ is used to build the translated S. The context stores the relevant information about introduced constraints and thus where new bindings will occur, that were not present in System $\mathrm{F_O}$.

```
Γ⇝S : F^O.Ctx F^O.S → F.Sorts
Γ⇝S ∅ = []
```

$$\mathsf{\Gamma{\leadsto}S}\ (\Gamma \blacktriangleright c) = \mathsf{\Gamma{\leadsto}S}\ \Gamma \triangleright \mathsf{F.e}_s$$
$$\mathsf{\Gamma{\leadsto}S}\ \{S \triangleright s\}\ (\Gamma \blacktriangleright x) = \mathsf{\Gamma{\leadsto}S}\ \Gamma \triangleright \mathsf{s{\leadsto}s}\ s$$

The empty context $\emptyset$ corresponds to the empty list $[]$. For each constraint in Γ an additional sort e_s is appended to S, representing the new binder introduced by the translation. If we find that a normal item is stored in the context, s is directly translated to $\mathsf{s{\leadsto}s}\ s$.

5.2 Variables

Similar to lists S, the translation for variables x needs context information.

$$\mathsf{x{\leadsto}x} : \forall\ \{\Gamma : \mathsf{F}^{O}.\mathsf{Ctx}\ F^{O}.S\} \rightarrow$$
$$\quad \mathsf{F}^{O}.\mathsf{Var}\ F^{O}.S\ F^{O}.s \rightarrow \mathsf{F.Var}\ (\mathsf{\Gamma{\leadsto}S}\ \Gamma)\ (\mathsf{s{\leadsto}s}\ F^{O}.s)$$
$$\mathsf{x{\leadsto}x}\ \{\Gamma = \Gamma \blacktriangleright \tau\}\ (\mathsf{here\ refl}) = \mathsf{here\ refl}$$
$$\mathsf{x{\leadsto}x}\ \{\Gamma = \Gamma \blacktriangleright \tau\}\ (\mathsf{there}\ x) = \mathsf{there}\ (\mathsf{x{\leadsto}x}\ x)$$
$$\mathsf{x{\leadsto}x}\ \{\Gamma = \Gamma \blacktriangleright c\}\ x = \mathsf{there}\ (\mathsf{x{\leadsto}x}\ x)$$

If an item is stored in the context we can translate the variable directly. Whenever a constraint is encountered, x is wrapped in an additional there. This is because, the expression that introduced the constraint will translate to an expression with an additional new binding, that needs to be respected in System F.
Furthermore, resolved constraints translate to the correct unique expression variable.

$$\mathsf{o{:}\tau{\in}\Gamma{\leadsto}x} : \forall\ \{\Gamma : \mathsf{F}^{O}.\mathsf{Ctx}\ F^{O}.S\} \rightarrow$$
$$\quad [\ `\ F^{O}.o\ :\ F^{O}.\tau\ ]{\in}\ \Gamma \rightarrow \mathsf{F.Var}\ (\mathsf{\Gamma{\leadsto}S}\ \Gamma)\ \mathsf{F.e}_s$$
$$\mathsf{o{:}\tau{\in}\Gamma{\leadsto}x}\ \mathsf{here} = \mathsf{here\ refl}$$
$$\mathsf{o{:}\tau{\in}\Gamma{\leadsto}x}\ (\mathsf{under\text{-}bind}\ o{:}\tau{\in}\Gamma) = \mathsf{there}\ (\mathsf{o{:}\tau{\in}\Gamma{\leadsto}x}\ o{:}\tau{\in}\Gamma)$$
$$\mathsf{o{:}\tau{\in}\Gamma{\leadsto}x}\ (\mathsf{under\text{-}cstr}\ o{:}\tau{\in}\Gamma) = \mathsf{there}\ (\mathsf{o{:}\tau{\in}\Gamma{\leadsto}x}\ o{:}\tau{\in}\Gamma)$$

The idea is the same as before, we wrap the variable in an additional there, for each constraint in the context.

Context

The translation of contexts is mostly a direct translation. We only look at the translation of constraints stored in the context.

$$\mathsf{\Gamma{\leadsto}\Gamma} : (\Gamma : \mathsf{F}^{O}.\mathsf{Ctx}\ F^{O}.S) \rightarrow \mathsf{F.Ctx}\ (\mathsf{\Gamma{\leadsto}S}\ \Gamma)$$
$$\mathsf{\Gamma{\leadsto}\Gamma}\ (\Gamma \blacktriangleright (`\ o\ :\ \tau)) = (\mathsf{\Gamma{\leadsto}\Gamma}\ \Gamma) \blacktriangleright \mathsf{\tau{\leadsto}\tau}\ \tau$$
$$\text{-}\ \ldots$$

Following the idea from above, constraints $o : \tau$ stored inside Γ translate to normal items in the translated Γ. The item introduced is the translated type $\mathsf{\tau{\leadsto}\tau}\ \tau$ required by the constraint. Again, whenever we pick up a constraint in System $\mathrm{F_O}$ there will be a new expression binding in System F, that accepts the constraint as higher order function. Thus, the corresponding type for that binding is expected in Γ at that position.

Renaming & Substitution

Typed renamings in System $\mathrm{F_O}$ get translated to untyped renamings in System F.

```
⊢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 (⊢ext-cstrᵣ ⊢ρ) = F.extᵣ (⊢ρ⇝ρ ⊢ρ)
⊢ρ⇝ρ (⊢drop-cstrᵣ ⊢ρ) = F.dropᵣ (⊢ρ⇝ρ ⊢ρ)
- ...
```

Typed renamings $\vdash$id$_r$, $\vdash$ext$_r$ and $\vdash$drop$_r$ translate to their untyped counterparts. Because constraints in contexts translate to actual bindings, both $\vdash$ext-cstr$_r$ and $\vdash$drop-cstr$_r$ translate to normal $\vdash$ext$_r$ and $\vdash$drop$_r$ in System F.
The translation of typed substitutions to untyped substitutions follows the same idea.

```
⊢σ⇝σ : ∀ {σ : F^O.Sub F^O.S₁ F^O.S₂} {Γ₁ : F^O.Ctx F^O.S₁} {Γ₂ : F^O.Ctx F^O.S₂} →
  σ F^O.: Γ₁ ⇒ₛ Γ₂ →
  F.Sub (Γ⇝S Γ₁) (Γ⇝S Γ₂)
⊢σ⇝σ (⊢typeₛ {τ = τ} ⊢σ) = F.singleₛ (⊢σ⇝σ ⊢σ) (τ⇝τ τ)
⊢σ⇝σ (⊢ext-cstrₛ ⊢σ) = F.extₛ (⊢σ⇝σ ⊢σ)
⊢σ⇝σ (⊢drop-cstrₛ ⊢σ) = F.dropₛ (⊢σ⇝σ ⊢σ)
- ...
```

Cases $\vdash$id$_s$, $\vdash$ext$_s$, $\vdash$drop$_s$, $\vdash$ext-cstr$_s$ and $\vdash$drop-cstr$_s$ are analogous to the cases for renamings. The typed introduction of a type $\vdash$type$_s$ translated to the untyped introduction of a term single$_s$.

Terms

Types and kinds can be translated without typing information. Kind $\star$ translates to direct counterpart in System F. Furthermore, all System $\mathrm{F_O}$ types translate to their direct counterparts in System F, except the constraint type [o : τ]⇒ τ'.

```
τ⇝τ : ∀ {Γ : F^O.Ctx F^O.S} →
  F^O.Type F^O.S →
  F.Type (Γ⇝S Γ)
τ⇝τ ([ o : τ ]⇒ τ') = τ⇝τ τ ⇒ τ⇝τ τ'
- ...
```

Constraint types [o : τ]⇒ τ' translate to function types $\tau \Rightarrow \tau$'. The translation from constraint types to function types corresponds directly to the translation of constraint abstractions to normal abstractions. The implicitly resolved constraint will now be taken as higher order function argument.
Arbitrary terms can only be translated using typing information. The typing carries information about the instances that were resolved, for all usages of overloaded variables. The unique variable name for the resolved instance can then be substituted for the overloaded variable. We only look at the translation of System $\mathrm{F_O}$ expressions that do not have a direct counterpart in System F.

```
⊢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 Fᴼ.⊢ t ∶ T →
  F.Term (Γ⇝S Γ) (s⇝s Fᴼ.s)
⊢t⇝t (⊢‘o o∶τ∈Γ) = ‘ o∶τ∈Γ⇝x o∶τ∈Γ
⊢t⇝t (⊢ƛ ⊢e) = λ‘x→ (⊢t⇝t ⊢e)
⊢t⇝t (⊢⊘ ⊢e o∶τ∈Γ) = ⊢t⇝t ⊢e · ‘ o∶τ∈Γ⇝x o∶τ∈Γ
⊢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
- ...
```

Typed overloaded variables ⊢‘o carry information about the instance that was resolved for o. We translate the resolved instance to the unique variable in System F, that points to the former instance, now let binding. Constraint abstractions translate to normal abstractions. An implicitly resolved constraint translates to a explicit application, that passes the resolved instance as argument. The decl expressions could be translated to nothing, as seen in the example at the beginning. Instead decl expressions are translated to useless let bindings, binding a unit value. Because decl expressions bind a new overloaded variable in System F_O, removing them would result in a variable binding less in System F and hence, more complex proofs. As already discussed, inst expressions translate to let bindings.

5.3 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 (⊢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))
⊢t⇝⊢t {Γ = Γ} (⊢‘x {x = x} Γxᴼ≡τ) = ⊢‘x (Γx≡τ⇝Γx≡τ x Γxᴼ≡τ)
⊢t⇝⊢t ⊢⊤ = ⊢⊤
⊢t⇝⊢t (⊢λ {τ’ = τ’} ⊢e) = ⊢λ (subst (_ F.⊢ ⊢t⇝t ⊢e ∶_) τ⇝wk·τ≡wk·τ⇝τ (⊢t⇝⊢t ⊢e))
⊢t⇝⊢t (⊢Λ ⊢e) = ⊢Λ (⊢t⇝⊢t ⊢e)
⊢t⇝⊢t (⊢· ⊢e₁ ⊢e₂) = ⊢· (⊢t⇝⊢t ⊢e₁) (⊢t⇝⊢t ⊢e₂)
⊢t⇝⊢t (⊢• {τ’ = τ’} {τ = τ} ⊢e) = subst (_ F.⊢ ⊢t⇝t ⊢e • τ⇝τ τ ∶_) (τ’⇝τ’[τ⇝τ]≡τ⇝τ’[τ] τ τ’) (⊢• (⊢t⇝⊢t ⊢
⊢t⇝⊢t (⊢let ⊢e₂ ⊢e₁) = ⊢let (⊢t⇝⊢t ⊢e₂) (subst (_ F.⊢ ⊢t⇝t ⊢e₁ ∶_) τ⇝wk·τ≡wk·τ⇝τ (⊢t⇝⊢t ⊢e₁))
⊢t⇝⊢t (⊢decl ⊢e) = ⊢let ⊢⊤ (subst (_ F.⊢ ⊢t⇝t ⊢e ∶_) τ⇝wk·τ≡wk·τ⇝τ (⊢t⇝⊢t ⊢e))
⊢t⇝⊢t (⊢inst {o = o} ⊢e₂ ⊢e₁) = ⊢let
  (⊢t⇝⊢t ⊢e₂)
  (subst (_ F.⊢ ⊢t⇝t ⊢e₁ ∶_) τ⇝wk-inst·τ≡wk-inst·τ⇝τ (⊢t⇝⊢t ⊢e₁))
```

Variables

```
Γx≡τ⇝Γx≡τ : ∀ {Γ : F^O.Ctx F^O.S} {τ : F^O.Type F^O.S} (x : F^O.Var F^O.S e_s) →
  F^O.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 F^O.⊢wk_r 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 F^O.⊢wk_r (F^O.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)
o:τ∈Γ⇝Γx≡τ {τ = τ} {Γ = Γ F^O.▸ c@(‘ o : τ)} (here {Γ = Γ}) =
  begin
    F.lookup (Γ⇝Γ Γ ▶ τ⇝τ τ) (here refl)
  ≡⟨ ⊢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 = Γ ▶ _} (under-bind {τ = τ} x) = trans
  (cong F.wk (o:τ∈Γ⇝Γx≡τ x))
  (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ F^O.⊢wk_r τ)
o:τ∈Γ⇝Γx≡τ {τ = τ} {Γ = Γ ▸ c@(‘ o : τ')} (under-cstr {c' = _ : τ'} o:τ∈Γ) =
  begin
    F.wk (F.lookup (Γ⇝Γ Γ) (o:τ∈Γ⇝x o:τ∈Γ))
  ≡⟨ cong F.wk (o:τ∈Γ⇝Γx≡τ o:τ∈Γ) ⟩
    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
  ∎
```

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.S₁ F^O.S₂} {Γ₁ : F^O.Ctx F^O.S₁} {Γ₂ : F^O.Ctx F^O.S₂} →
  (⊢σ : σ F^O.: Γ₁ ⇒ₛ Γ₂) →
  (x : F^O.Var F^O.S₁ τₛ) →
  F.sub (⊢σ⇝σ ⊢σ) (' x⇝x x) ≡ τ⇝τ (F^O.sub σ (' x))
⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢idₛ x = refl
⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢extₛ ⊢σ) (here refl) = refl
⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢extₛ {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.⊢wkᵣ (σ x))
⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢dropₛ {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.⊢wkᵣ (σ x))
⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢typeₛ ⊢σ) (here refl) = refl
⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢typeₛ ⊢σ) (there x) = ⊢σ⇝σ·x⇝x≡τ⇝σ·x ⊢σ x
⊢σ⇝σ·x⇝x≡τ⇝σ·x (⊢ext-cstrₛ {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-instᵣ (σ x)) ⟩
    τ⇝τ (F^O.ren F^O.idᵣ (σ x))
  ≡⟨ cong τ⇝τ (idᵣτ≡τ (σ x)) ⟩
    τ⇝τ (σ x)
  □)
⊢σ⇝σ·x⇝x≡τ⇝σ·x(⊢drop-cstrₛ {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-instᵣ (σ x) ⟩
    τ⇝τ (F^O.ren F^O.idᵣ (σ x))
  ≡⟨ cong τ⇝τ (idᵣτ≡τ (σ x)) ⟩
    τ⇝τ (σ x)
  □)
```

```
⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ : ∀ {σ : F^O.Sub F^O.S₁ F^O.S₂} {Γ₁ : F^O.Ctx F^O.S₁} {Γ₂ : F^O.Ctx F^O.S₂} →
  (⊢σ : σ F^O.: Γ₁ ⇒ₛ Γ₂) →
  (τ : F^O.Type F^O.S₁) →
```

F.sub (⊢σ⇝σ ⊢σ) (τ⇝τ τ) ≡ τ⇝τ (F^O.sub σ τ)

```
τ'⇝τ'[τ⇝τ]≡τ⇝τ'[τ] τ τ' = ⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ ⊢single-typeₛ 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 based system (HM_O) and our target language would be Hindley-Milner (HM). HM is a restricted form of System F. HM would require two new sorts m_s and p_s for mono and poly types in favour of arbitrary types τ_s. Poly types can include quantification over type variables, 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. In consequence, constraint abstractions would only be allowed to introduce constraints for overloaded variables with mono types. Instance expression bodies would be allowed to have poly types, because they translate to let bindings after all. But instances would need to be restricted as well. For each overloaded variable o, all instances would need to differ in the type of their first argument. With these two restrictions, type inference, using an extended version of Algorithm W, should be preserved [CITE]. Formalizing the changes and restrictions mentioned above should be a fairly straight forward adjustment to the formalization of System F and System F_O.

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. The correct instance for o needs to be resolved based on the types of arguments e_1 .. e_n. More specifically, to formalize small step semantics we would need to apply the restriction mentioned above, that all instances for the same overloaded variable o must differ in the type of their first argument. In consequence, the resolved instance for single application step ` o · e would be decidable. Let $\vdash e \hookrightarrow \vdash e'$ be such a typed small step semantic for System F_O. We would need to prove something similar to: If $\vdash e \hookrightarrow \vdash e'$ then $\exists$ [e''] (⊢e↪e'⇝e↪e' $\vdash e \hookrightarrow^* e''$) × (⊢e↪e'⇝e↪e' $\vdash e' \hookrightarrow^* e''$), 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 $\vdash e$ and $\vdash e'$ reduce to some System F expression e'' using finite many reduction steps. This more general formulation is needed because there might be more reduction steps in the translated System F expression than in the System F_O expression. For example, an implicitly resolved constraint in System F_O needs to be explicitly passed using a additional application step in System F. For now it is unclear, if semantic preservation can be shown using induction over the semantic rules or if logical relations are needed.

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 overloading. Using Agda we formalized the Dictionary Passing Transform between System F and System F_O. Finally, we proved the System F formalization to be sound and the Dictionary Passing Transform to be type preserving.

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