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 F_O, a minimal language extension to System F, with support for overloading. Furthermore, we proof the Dictionary Passing Transform from System F_O to System F to be type preserving using Agda.

Table of Contents

1 Introduction

1.1 Overloading in Programming Languages

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. Python, for example, 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 Typeclasses in Haskell

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 concrete types. Instantiating a typeclass gives concrete implementations 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` $\mapsto$ `Nat`$\}$ and $\{$`α` $\mapsto$ `[β]`$\}$, where `β` is a concrete type and there exists an instance `Eq β`. The intuition here is that we want to be able to compare natural numbers `Nat` and lists `[β]`, given 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` $\mapsto$ `Nat`$\}$. After that, `Eq` is instantiated for $\{$`α` $\mapsto$ `[β]`$\}$, given that an instance `Eq β` can be found. Hence we can call `eq` on expressions with type `Nat` and `[Nat]`. In the latter case, the type constraint `Eq β ⇒ ..` in the instance for lists resolves to the instance for natural numbers.

1.3 Desugaring Typeclass Functionality to System F_O

System F_O is a minimal calculus with support for overloading and polymorphism, based on System F [CITE]. In System F_O we give up high level language constructs and instead desugar a subset of the typeclass functionality.
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 Haskell instances must comply with the generic type signature defined by the typeclass. Such a signature is not present in System F_O and overloaded variables can be overloaded for arbitrary types. Locally shadowing other instances of the same type is allowed. Constraints can be introduced on the expression level using the constraint abstraction `ƛ (o : τ). e'`. Constraint abstractions result in constraint types `[o : τ] ⇒ τ'`. We introduce constraints on the expression level, because instance expressions do not have an explicit type annotation in System F_O. Expressions with constraint types `[o : τ] ⇒ τ'` are implicitly treated as expressions of type τ', given the constraint `o : τ` can be resolved.

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 would potentially need to recursively call `eq` for sublists but System F_O's formalization does not actually support recursion. Extending System F_O with recursive let bindings and thus recursive instances is known to be straight forward.

1.4 Translating System F_O back to System F

System F_O can be translated back to System F. Hence, System F_O is not more expressive or powerful than System F. Overloading is a convenience feature after all. We could just use let bindings with unique variable names and check constraints by ourselves.
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` o_τ `= e in e'` expressions, where o_τ is an unique name with respect to type `τ` of expression `e`. Constraint

abstractions `λ (o : τ). e'` translate to normal abstractions λo_τ`. 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 o_τ bound by the translated instance. Implicitly resolved constraints in System F_O must be explicitly passed as arguments in System F. The translation becomes more intuitive when looking at an example.

Example: Dicitionary Passing Transform

Recall the System F_O example from above. We use indices to represent new unique names. Applying the Dictionary Passing Transform to the example above results in well formed 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 instance for lists, the constraint abstraction translates to a normal lambda abstraction. The lambda abstraction now takes the constraint, that was implicitly resolved in System F_O, as explicit higher order function argument. Invocations of `eq` are translated to the correct unique variables eq_i. When eq_2 is invoked, we must pass the correct instance to eliminate the former constraint abstraction, now higher order function binding, by explicitly passing instance eq_1 as argument.

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 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, well formed Agda programs 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 syntaxes in Agda we use a single data type `Term` indexed by sorts s to represent the syntax. Sorts distinguish between different categories of terms. For example, sort e_s represents expressions e, τ_s represents τ and κ_s represents 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 support self indexed data types. In the actual implementation Term has another index S, that we will ignore for now.

2.3 Overview 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 $\rightsquigarrow : (\Gamma \vdash_{F_O} t : T) \rightarrow \mathsf{Term}_F\ s$ be the Dictionary Passing Transform that translates well typed System $\mathrm{F_O}$ terms to untyped System F terms. Further let $\rightsquigarrow_\Gamma : \mathsf{Ctx}_{F_O} \rightarrow \mathsf{Ctx}_F$ be the transform of contexts and $\rightsquigarrow_T : \mathsf{Term}_{F_O}\ s' \rightarrow \mathsf{Term}_F\ s'$ 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 $(\rightsquigarrow_\Gamma \Gamma) \vdash_F (\rightsquigarrow \vdash t) : (\rightsquigarrow_T\ T)$.
We begin by formalizing System F and prove its soundness [3]. Then System $\mathrm{F_O}$ is formalized, although without semantics and soundness proof [4]. In the end, we formalize the translation of the Dictionary Passing Transform and prove it to be type preserving [5].

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 $\mathsf{Ctxable}$. Index $\top^C$ indicates that variables for terms of sort s can be bound. In contrast, $\bot^C$ says that variables for terms of sort s cannot be bound. In this case, System F supports abstracting over expressions and types, but not over kinds. Going forward we also use shorthand S for lists of contextable sorts: $\mathsf{Sorts} = \mathsf{List}\ (\mathsf{Sort}\ \top^C)$.

Syntax

The syntax of System F is represented in a single data type Term, indexed by sorts S and sort s. The 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 to the current amount of bound variables.

Thus, unbound variables cannot be referenced by definition. But indexing Term with a number is not sufficient, since System F has both expression and type variables, that need to be distinguished. To solve this problem, we need to extend the idea of Debruijn indices and store the corresponding sort for each variable. Thus, 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. The 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 membership proofs $s \in S$. In consequence we can only reference already bound variables. Membership proofs of type $s \in S$ are inductively defined, similar to natural numbers. Membership proofs can be here refl, where refl is prove that the last element in S is the element we searched for. Alternatively, memberships proofs can be there x, where x is another membership proof for S with one element less.
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'. Both bindings introduce an additional sort e_s, or τ_s respectively, to the index of the body.
To eliminate abstractions we use application $e_1 \cdot e_2$.
Similarly, type application $e \bullet \tau$ eliminates type abstractions.
Let bindings let‘x= e_2 ‘in e_1 combine abstraction and application.
All types τ have kind $\star$.
We use shorthands Var S s = $s \in S$, Expr S = Term S e_s, Type S = Term S τ_s and variable names x, e and τ respectively. Further, we use t as variable 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 to elements in S_2.

```
ren : Ren S₁ S₂ → (Term S₁ s → Term S₂ s)
ren ρ (‘ x) = ‘ (ρ x)
ren ρ (λ‘x→ e) = λ‘x→ (ren (ext_r ρ) e)
ren ρ (τ₁ ⇒ τ₂) = ren ρ τ₁ ⇒ ren ρ τ₂
- ...
```

The renaming is applied to all variables x.
When we encounter a binder for a term of sort s, the renaming is extended using ext_r : $\mathsf{Ren}\ S_1\ S_2 \to \mathsf{Ren}\ (S_1 \triangleright s)\ (S_2 \triangleright s)$.
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 membership proofs, shifting variables by one binder is accomplished by wrapping them in the there constructor.

Substitution

Substitutions σ of type $\mathsf{Sub}\ S_1\ S_2$ 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. If we encounter a binder in sub, the substitution must be extended using function ext_s.

```
ext_s : Sub S₁ S₂ → Sub (S₁ ▷ s) (S₂ ▷ s)
ext_s σ (here refl) = ‘ here refl
ext_s σ (there x) = wk (σ x)
```

The extension of a substitution is defined as the weakening of the term that results in substitution being applied to variable x.
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 (single_s id_s t') t
```

A single substitution $\mathsf{single}_s : \mathsf{Sub}\ S_1\ S_2 \to \mathsf{Term}\ S_2\ s \to \mathsf{Sub}\ (S_1 \triangleright s)\ S_2$ takes an existing substitution σ' and introduces an additional binding, that is substituted with t'. In the case of _[_] we let σ' be the identity substitution $\mathsf{id}_s : \mathsf{Sub}\ S\ S$.

Context

Similar to terms, typing contexts Γ of type $\mathsf{Ctx}\ S$ are also indexed by the list of bound variables. In consequence only types and kinds for bound variables can be stored in Γ by definition.

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

Contexts are inductively defined and can either be empty $\emptyset$ or extended with one element T, using constructor $\Gamma \blacktriangleright T$. Variable T represents terms of sort kind-of s. The function kind-of maps contextable sorts s to the sort of the term that is stored in Γ for variables of sort s. Thus, if we extend a context with a term of sort kind-of s, the list of bound variables is extended by s.

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

Expressions variables require Γ to store the corresponding type and for type variables Γ stores the corresponding kind.
The lookup function resolves the type or kind for a variable x in Γ.

```
lookup : Ctx S → Var S s → Term S (kind-of s)
lookup (Γ ▶ T) (here refl) = wk T
lookup (Γ ▶ T) (there x) = wk (lookup Γ x)
```

Both the base and induction case wrap the looked up constraint in a weakening. Thus, the looked up T has index S that is compatible with the current amount of bound variables. The lookup function cannot fail by definition, because we only allow to lookup bound variables, that must have an entry in Γ.

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 the looked up type for x in Γ is τ.
All unit expressions tt have the type ‘⊤. This is expressed by the rule ⊢⊤.
The rule for abstractions ⊢λ introduces an expression variable of type τ to body e. Because the body type τ’ cannot use the newly 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 $\tau \Rightarrow \tau$’.
The 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 the rule ⊢· and says that, if e_1 is a function from τ_1 to τ_2 and e_2 has type τ_1, then $e_1 \cdot e_2$ has type τ_2.
Similarly, the type application rule ⊢• states that, if e has type ∀‘α τ, then α can be substituted with another type τ’ in τ.
The rule ⊢let combines the abstraction and application rule.
For the typing of types, the 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 of Renaming & Substitution

Because of extrinsic typing, both renamings and substitutions need to have typed counterparts. We formalize typed renamings ⊢ρ 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
  ⊢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, that is used when a binder is encountered.
Further, the constructor ⊢drop_r allows 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 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 σ x in Γ_2. The typing result of σ 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 ↪ 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 type into the abstraction body.
Reduction β-let is equivalent to β-λ and substitutes e_2 into e_1.
Rules ξ-$\cdot_i$ 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
```

The xi reduction cases ξ-·₁, ξ-·₂, ξ-• and ξ-let follow directly from the induction hypothesis.
For the beta reduction cases β-λ, β-Λ and β-let we need to prove that substitutions preserve typing. We have two cases for substitutions in reduction rules: 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 the 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 in System F_O 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 introduces an additional instance for o.
Constraints c can be constructed using constructor $o : \tau$.
A constraint c can be part of both constraint abstractions ƛ $c \Rightarrow e$ and constraint types [c]⇒ τ.
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, constraints are stored 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. The lookup method is defined analogously to lookup in System F and simply ignores constraints stored in the context.

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 in 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 desired constraint c.
If the last item in the context is not the constraint c, c must be further inside the context, either behind a item stored in Γ (under-bind) or a 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
  ⊢‘o :
    [ ‘ o : τ ]∈ Γ →
    Γ ⊢ ‘ o : τ
```

```
⊢λ :
  Γ ▸ c ⊢ e : τ →
  Γ ⊢ λ 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
⊢inst :
  Γ ⊢ e₂ : τ →
  Γ ▸ (‘ o : τ) ⊢ e₁ : τ’ →
  Γ ⊢ inst‘ ‘ o ‘= e₂ ‘in e₁ : τ’
- ...
```

Rule ⊢‘o for overloaded variables says that, if we can resolve the constraint o : τ in Γ, then o can take on type τ.
The rule for constraint abstraction ⊢λ appends constraint c to Γ and thus assumes c to be valid in body e. Constraint abstraction result in the corresponding constraint type, that lifts the constraint onto the type level.
Expressions e with constraint type [c]⇒ τ’ have the constraint implicitly eliminated using the ⊢⊘ rule, given constraint c can be resolved in Γ.
The rule ⊢decl introduces a new overloaded variable o to e. To introduce o in Γ, we only need to store the information that o exists. Thus, Γ is extended by the single kind ⋆, to denote the existence of o, similar to type variables. Similar to τ’ inside the abstraction rule ⊢λ,τ is weakened to be compatible in S with Γ, not extended by ⋆, to act as the resulting type of the typing.
A instance for an overloaded variable o is typed using the rule ⊢inst. Given the instance body e_2 has type τ, we extend Γ with constraint o : τ inside e_1.

Typing Renaming & Substitution

Typed renamings are identical to the typed renamings in System F, except there is an additional case for the weakening by a constraint.

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

Constraint o : τ can only be introduced to Γ_2 using the constructor ⊢drop-cstr$_r$. Dropping a constraint corresponds to a typed weakening, similar to ⊢drop$_r$, but instead of introducing an unused variable we introduce an unused constraint.
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
  - ...
```

The constructor $\vdash$typeₛ allows to substitute the last binder with type τ by extending Γ_1 with kind $\star$ and leaving Γ_2 unchanged. Thus, $\vdash$typeₛ complements the single-typeₛ function. The intuition here is that, if we would allow all terms to be introduced using a $\vdash$termₛ constructor, typed substitutions in System F_O would be arbitrary again. Constructors $\vdash$extₛ, $\vdash$dropₛ and$\vdash$drop-cstrₛ are not shown. All of them function the same way as their counterparts in typed renamings.

5 Dictionary Passing Transform

5.1 Translation

Sorts

The translation of System 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 : Fᴼ.Sort ⊤ᶜ → F.Sort ⊤ᶜ
s⇝s eₛ = eₛ
s⇝s oₛ = eₛ
s⇝s τₛ = τₛ
```

Sort e_s and τ_s translate to their corresponding counterparts in System F.
Overloaded variables in System 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 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 F_O.

```
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
```

The empty context $\emptyset$ corresponds to the empty list [].
For each constraint in Γ an additional sort e_s is appended to S, to complement the new binding construct that will be introduced by the translation.
If we find that a normal item is stored in the context, s is directly translated to s⇝s s.

Variables

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

```
x⇝x : ∀ {Γ : F^O.Ctx F^O.S} →
  F^O.Var F^O.S F^O.s → F.Var (Γ⇝S Γ) (s⇝s F^O.s)
x⇝x {Γ = Γ ▶ τ} (here refl) = here refl
x⇝x {Γ = Γ ▶ τ} (there x) = there (x⇝x x)
x⇝x {Γ = Γ ▸ c} x = there (x⇝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.

```
o:τ∈Γ⇝x : ∀ {Γ : F^O.Ctx F^O.S} →
  [ ‘ F^O.o : F^O.τ ]∈ Γ → F.Var (Γ⇝S Γ) F.e_s
o:τ∈Γ⇝x here = here refl
o:τ∈Γ⇝x (under-bind o:τ∈Γ) = there (o:τ∈Γ⇝x o:τ∈Γ)
o:τ∈Γ⇝x (under-cstr o:τ∈Γ) = there (o:τ∈Γ⇝x o:τ∈Γ)
```

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.

```
Γ⇝Γ : (Γ : F^O.Ctx F^O.S) → F.Ctx (Γ⇝S Γ)
Γ⇝Γ (Γ ▸ (‘ o : τ)) = (Γ⇝Γ Γ) ▶ τ⇝τ τ
- ...
```

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 τ⇝τ τ required by the constraint. Again, whenever we pick up a constraint in System F_O there will be a new binder 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 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-cstr_r ⊢ρ) = F.ext_r (⊢ρ⇝ρ ⊢ρ)
⊢ρ⇝ρ (⊢drop-cstr_r ⊢ρ) = F.drop_r (⊢ρ⇝ρ ⊢ρ)
- ...
```

Typed renamings ⊢id$_r$, ⊢ext$_r$ and ⊢drop$_r$ translate to their untyped counterparts. Because constraints in contexts translate to actual bindings, both ⊢ext-cstr$_r$ and ⊢drop-cstr$_r$ translate to normal ⊢ext$_r$ and ⊢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)
- ...
```

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

Terms

Types and kinds can be translated without typing information. Kind ⋆ translates to direct counterpart in System F. Furthermore, all System 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 be taken as higher order function argument in System F.
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 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 F^O.⊢ t : T →
  F.Term (Γ⇝S Γ) (s⇝s F^O.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.
All inst expressions translate to let bindings.

5.2 Type Preservation

Terms

We first look at the final proof of type preservation for the Dictionary Passing Transform to motivate all necessary lemmas. Type preservation is proven by induction over the typing rules of System F_O. Given a typed System F_O term $\vdash t$, the function ⊢t⇝⊢t produces a typed System F term. The untyped translated System F_O term ⊢t⇝t t gets typed in translated context Γ⇝Γ Γ and has typing result T⇝T T. The function T⇝T translates untyped types and kinds from System F_O to System F.

```
⊢t⇝⊢t : {Γ : Fᴼ.Ctx Fᴼ.S} {t : Fᴼ.Term Fᴼ.S Fᴼ.s}
           {T : Fᴼ.Term Fᴼ.S (Fᴼ.kind-of Fᴼ.s)} →
  (⊢t : Γ Fᴼ.⊢ t : T) →
  (Γ⇝Γ Γ) F.⊢ (⊢t⇝t ⊢t) : (T⇝T T)
⊢t⇝⊢t (⊢`x {x = x} Γx≡τ) = ⊢`x (Γx≡τ⇝Γx≡τ x Γx≡τ)
⊢t⇝⊢t (⊢`o o:τ∈Γ) = ⊢`x (o:τ∈Γ⇝Γx≡τ o:τ∈Γ)
⊢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 (⊢ƛ {c = (` o : τ)} ⊢e) = ⊢λ
  (subst (_ F.⊢ ⊢t⇝t ⊢e :_) τ⇝wk·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 (⊢• {τ = τ} {τ' = τ'} ⊢e) = subst (_ F.⊢ ⊢t⇝t ⊢e • τ⇝τ τ' :_)
  (τ'⇝τ'[τ⇝τ]≡τ⇝τ'[τ] τ' τ) (⊢• (⊢t⇝⊢t ⊢e))
- ...
```

Proof $\Gamma x{\equiv}\tau$ that a variable x has type τ in Γ translates to proof that x⇝x x has type τ⇝τ τ in Γ⇝Γ Γ using lemma Γx≡τ⇝Γx≡τ. With lemma Γx≡τ⇝Γx≡τ the typing rule ⊢`x can be translated to the type rule for variables in System F.
Similarly, Lemma o:τ∈Γ⇝Γx≡τ translates proof that an instance $o : \tau$ was resolved for a overloaded variable o to proof that unique variable o:τ∈Γ⇝x o:τ∈Γ has type τ⇝τ τ in Γ⇝Γ Γ. Using lemma o:τ∈Γ⇝Γx≡τ the typing rule for overloaded variables ⊢`o can be translated to the typing rule for normal variables ⊢`x.
Typed let bindings ⊢let $\vdash e_2$ $\vdash e_1$ translate to typed let bindings in System F. Rule $\vdash e_2$ is translated directly using the induction hypothesis. Because the typing for e_1 in $\vdash e_1$ results in wk τ', proof is needed that τ' weakened in System F_O and translated to System F is equivalent to the weakening of translated τ' in System F. Lemma τ⇝wk·τ≡wk·τ⇝τ is used to substitute the required equivalence into the translated typing rule ⊢t⇝⊢t $\vdash e_1$.

Typed constraint abstractions ⊢λ translate to normal abstractions in System F. Inside the typing for ⊢e, the result type τ for body e does not need to be weakened, because the constraint abstraction only introduced a constraint to context Γ and no actual binding. After the translation, the former constraint will be bound by a binding and thus a new item in Γ⇝Γ Γ will exist. To ignore the binding, τ is weakened in the abstraction rule ⊢λ. Lemma τ⇝wk·τ≡wk-inst·τ⇝τ proves that translating τ in Γ extended by a constraint is equivalent to weakening τ after the translation. This is true, because in the first case, the constraint translates to an actual binding and thus both side have an additional unnecessary expression binding, that τ cannot use.
Typed implicitly resolved constraints ⊢⊘ carry the information about the instance resolved. In System F the former constraint is now explicitly passed as variable pointing to the correct translated instance. Thus, ⊢⊘ results in typed application ⊢·. We apply the correct instance using lemma o:τ∈Γ⇝Γx≡τ to resolve the correct unique variable for the resolved constraint.
Type application rule ⊢• contains type in type substitution. Hence, we need proof that it is irrelevant, if τ' is substituted into τ and then translated or both τ and τ' are translated and substitution happens in System F. Using lemma τ'⇝τ'[τ⇝τ]≡τ⇝τ'[τ] we can substitute the equivalence into the translated typing rule ⊢t⇝⊢t ⊢e.
The translation of ⊢⊤, ⊢λ, ⊢·, ⊢decl and ⊢inst is either a direct translation or does not use other lemmas than the ones discussed.

Renaming

Both τ⇝wk·τ≡wk·τ⇝τ and τ⇝wk·τ≡wk-inst·τ⇝τ directly follow from a more general lemma ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ for arbitrary renamings. Lemma ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ proves that translating both the typed renaming ⊢ρ and type τ and then apply the renaming in System F is equivalent to applying the renaming ρ in System F_O and then translating renamed τ. The lemma can be proven by induction over System F_O types τ.

```
⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ : {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 : ρ F^O.∶ Γ₁ ⇒ᵣ Γ₂) →
  (τ : F^O.Type F^O.S₁) →
  F.ren (⊢ρ⇝ρ ⊢ρ) (τ⇝τ τ) ≡ τ⇝τ (F^O.ren ρ τ)
⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢ρ (‘ x) = cong ‘_ (⊢ρ⇝ρ·x⇝x≡x⇝ρ·x ⊢ρ x)
⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢ρ ([ ‘ o : τ ]⇒ τ’) = cong₂ _⇒_
  (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢ρ τ) (⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ ⊢ρ τ’)
- ...
```

The case for type variables needs an additional lemma ⊢ρ⇝ρ·x⇝x≡x⇝ρ·x specifically for variables. Lemma ⊢ρ⇝ρ·x⇝x≡x⇝ρ·x proves the exact same statement, but for type variables applied to a renamings: (⊢ρ⇝ρ ⊢ρ) (x⇝x x) ≡ x⇝x (ρ x). This statement can be proven via straight forward induction over typed System F_O renamings ⊢ρ.
All other cases follow directly from the induction hypothesis. The only small exception is the constraint type, where we need to respect that it translates to a function type.

Substitution

Similar to renamings, the lemma for single substitution on types τ'⇝τ'[τ⇝τ]≡τ⇝τ'[τ] follows from a more general lemma about substitutions: τ'⇝τ'[τ⇝τ]≡τ⇝τ'[τ] τ τ' = ⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ ⊢single-type$_s$ τ'. The more general lemma ⊢σ⇝σ·τ⇝τ≡τ⇝σ·τ also follows by straight forward induction over System F_O types, except the case for type variables. Other than with renamings, lemma ⊢σ⇝σ·x⇝x≡τ⇝σ·x does not follow directly. To understand why, we at look at case ⊢ext$_s$.

```
⊢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 (⊢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))
```

Case ⊢ext$_s$ is proven via induction over variable x, similar to how ext$_s$ is defined. The base case holds by definition. In the induction case, we use the weakening of the outer induction hypothesis and combine it with proof that weakenings preserve the translation, using transitivity. The intuition here is that we need the renaming lemma ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ, because ext$_s$ is defined by weakening the result of the substitution σ applied to variable x.
Both ⊢id$_s$ and ⊢type$_s$ follow directly from the induction hypothesis. The cases for ⊢drop$_s$, ⊢drop-cstr$_s$ and ⊢ext-cstr$_s$ are similar to ⊢ext$_s$.

Variables

We first look at the proof for lemma Γx≡τ⇝Γx≡τ. Lemma Γx≡τ⇝Γx≡τ is proven via induction over the System F_O context Γ.

```
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 eₛ) →
  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.⊢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 F^O.⊢wkᵣ (F^O.lookup Γ x))
- ...
```

Exemplarily we will look at case Γ ▶ τ, that is proven via induction over variables x. The prove follows the same reasoning as the ⊢ext$_s$ case for substitutions above. Because the function lookup weakens the looked up type τ in both the base case and induction step, both use lemma ⊢ρ⇝ρ·τ⇝τ≡τ⇝ρ·τ.
The case Γ ▸ c is a little more complicated but uses similar concepts. Additional complexity arieses, because we need to deal with the fact, that constraints were ignored by the lookup method in System F_O, but translate to actual context items in System F.
Lemma o:τ∈Γ⇝Γx≡τ can proven via induction over the type for resolved constraints [c]∈ Γ. The proof is analogous to the proof shown for Γx≡τ⇝Γx≡τ, since the type for resolved constraint has the exact same structure as context Γ.

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. The inference algorithm would treat instance expressions similar to let bindings and could infer the type of an overloaded identifier via the type of the first argument applied. 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 Proving Semantic Preservation

For now System F_O does not have semantics formalized. Semantics for System F_O would need to be typed semantics, because applications ‘ $o \cdot e_1 \,..\, \cdot\, e_n$ 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 ‘ $o \cdot e$ 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 $\exists\, [\ e''\ ]\ (\vdash\!\mathsf{e}\!\hookrightarrow\!\mathsf{e}'\!\rightsquigarrow\!\mathsf{e}\!\hookrightarrow\!\mathsf{e}'\ \vdash e \hookrightarrow^* e'') \times (\vdash\!\mathsf{e}\!\hookrightarrow\!\mathsf{e}'\!\rightsquigarrow\!\mathsf{e}\!\hookrightarrow\!\mathsf{e}'\ \vdash e' \hookrightarrow^* e'')$, where $\vdash\!\mathsf{e}\!\hookrightarrow\!\mathsf{e}'\!\rightsquigarrow\!\mathsf{e}\!\hookrightarrow\!\mathsf{e}'$ 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 Related Work

System F_O is heavily inspired by System O [CITE]. System O is a language extension to the Hindley-Milner System and preserves full type inference. Aside from using Hindley-Milner instead of System F as base system, System O differs from System F_O by tieing constraint introductions to forall types. Constraints can not be introduced everywhere using a expression level construct, instead constraints are introduced via explicit type annotations of instances inside forall types.

6.4 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. We proved the System F formalization to be sound and the Dictionary Passing Transform to be type preserving. The full formalization of System F, System F_O and the Dictionary Passing Transform can be found as Agda code files on GitHub [CITE]. A reasonable next step would be to prove the Dictionary Passing Transform to be semantic 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