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 prove the Dictionary Passing Transform from System F_O to System F to be type-preserving using Agda.

Table of Contents

1 Introduction 3
  1.1 Overloading in Programming Languages 3
  1.2 Typeclasses in Haskell 3
  1.3 Desugaring Typeclass Functionality to System $\mathrm{F_O}$ 4
  1.4 Translating System $\mathrm{F_O}$ back to System F 5
2 Preliminary 6
  2.1 Dependently Typed Programming in Agda 6
  2.2 Design Decisions for the Agda Formalization 6
  2.3 Overview of the Type Preservation Proof 6
3 System F 7
  3.1 Specification 7
  3.2 Soundness 13
4 System $\mathrm{F_O}$ 14
  4.1 Specification 14
5 The Dictionary Passing Transform 18
  5.1 Translation 18
  5.2 Type Preservation 21
6 Further Work and Conclusion 24
  6.1 Hindley Milner with Overloading 24
  6.2 Proving Semantic Preservation 24
  6.3 Related Work 25
  6.4 Conclusion 26

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 a function name helps to solve the problem of having to define similar but differing names and operators for different type combinations. The idea of overloading is also sometimes referred to as ad-hoc polymorphism. An ad-hoc polymorphic function is allowed to have multiple type-specific meanings for all types that it is defined for. In contrast, a parametric polymorphic (or generic) function only defines abstract behavior that works for all types.
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. When we instantiate a typeclass, we must provide an actual implementation to all functions defined by the typeclass based on the concrete types that the typeclass is instantiated for. 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 that were applied to the overloaded function name. Furthermore, Haskell allows to constrain a type variable α via type constraints `Tc α ⇒ τ'` to only be substituted by a concrete type τ if there exists an instance `Tc τ`. Type constraints allow to abstract over all types that inherit a shared behavior, but the actual implementation of the behavior can differ per type. Hence, type constraints are a powerful formalism in addition to parametric polymorphism.

Example: Overloading Equality in Haskell

In this example the function `eq : α → α → Bool` is overloaded with different meanings for different substitutions $\{\alpha \mapsto \tau\}$. We want to be able to call `eq` on both $\{\alpha \mapsto$ `Nat`$\}$ and $\{\alpha \mapsto$ `[β]`$\}$, where β is a type and there exists an instance that gives meaning to `eq : β → β → Bool`. 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` is declared with a single generic function signature `eq :: α → α → Bool`. Next, we instantiate `Eq` for {α ↦ `Nat`}. After that, `Eq` is instantiated for {α ↦ `[β]`}, given that an instance `Eq β` can be resolved. 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. In System F_O we give up high level language constructs and instead work with simple overloaded identifiers.
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 signatures defined by the typeclass. Such signatures are not present in System F_O and overloaded variables can be instantiated for arbitrary types. Locally shadowing other instances of the same type is allowed. Constraints can be introduced on the expression level using constraint abstractions `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 a type annotation in System F_O. Expressions with constraint types `[o : τ] ⇒ τ'` are implicitly treated as expressions of type τ' by the type system, given that 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 that 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 instance for lists would potentially need to recursively call `eq` for sublists but the formalization of System F_O 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. After all, overloading is more of a convenience feature. We simply could use let bindings with unique variable names and pass constraints as higher order functions.

The Dictionary Passing Transform translates well typed System F_O expressions to well typed System F expressions. The translation requires knowledge acquired during type checking. More specifically, we need to know the instances that were resolved for invocations of overloaded identifiers and the instances that constraints were implicitly resolved to.

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 a unique name with respect to the type τ of the expression `e`. Constraint abstractions `ƛ (o : τ). e'` translate to normal abstractions λo_τ`. e'`. The implicitly resolved constraint in System F_O will be taken as higher-order function argument in System F. Hence, constraint types `[o : τ] ⇒ τ'` translate to function types `τ → τ'`. Invocations of overloaded function names `o` translate to the correct unique variable name o_τ that is bound by the let binding that got introduced by the translation for the instance resolved at that invocation. 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 a well formed System F expression.

```
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] ..
```

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 takes the constraint that was implicitly resolved in System F_O as explicit higher-order function argument. Invocations of `eq` translate to the correct unique variables $\texttt{eq}_i$. When $\texttt{eq}_2$ is invoked for lists of numbers, we must pass the correct instance to eliminate the former constraint abstraction, now higher-order function binding, by explicitly passing instance $\texttt{eq}_1$ as argument.

2 Preliminary

2.1 Dependently Typed Programming in Agda

Agda is a dependently typed programming language and proof assistant [3]. Agda's type system is based on intuitionistic type theory and allows to construct proofs based on the Curry-Howard correspondence. The Curry-Howard correspondence is an isomorphic relationship between programs written in dependently typed programming languages and mathematical proofs written in first order logic. Because of the Curry-Howard correspondence, programs correspond to proofs and formulae correspond to types. Thus, type checked Agda programs imply the correctness of the corresponding proofs, assuming we do not use unsafe Agda features and Agda is implemented correctly. We will use Agda to formalize the type preservation proof for the Dictionary Passing Transform from System F_O to System F.

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, the sort e_s represents expressions e, τ_s represents types τ and κ_s represents kinds. In System F and System F_O there only exists a single kind $\star$. The idea of sorts originates from the theory of pure type systems [2], but neither System F nor System F_O allow any interesting dependencies between terms of the sort e_s, τ_s, and κ_s. Using a single data type to formalize the syntax yields more elegant proofs involving contexts, substitutions and renamings. In consequence of using a single data type, we must use extrinsic typing because intrinsically typed terms Term e_s ⊢ Term τ_s would need to be indexed by themselves and Agda does not support self-indexed data types. In the actual implementation, the data type Term has another index S that we will ignore for now.

2.3 Overview of the Type Preservation Proof

The overall goal will be to prove that the Dictionary Passing Transform is type-preserving. Let $\vdash t$ be any well formed System F_O term $\Gamma \vdash_{F_O} t : T$, where Γ is a typing environment of type Ctx_{F_O}, t is a Term_{F_O} s, T is a Term_{F_O} s' and s' is the sort of the typing result for terms of the 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 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 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 the syntax, typing and semantic of System F in Agda and prove its soundness in section 3. In section 4, we formalize System F_O's syntax and typing. In the end, we formalize the translation of the Dictionary Passing Transform and prove it to be type-preserving in section 5. We do not formalize semantics and soundness for System F_O. In a way, correct semantics for System F_O are already given by the type-preserving translation from System F_O to System F. This is because we can simply apply the semantics of System F after translating. And furthermore, the semantics of System F are proven to be sound in combination with the type system that System F_O is translated to.

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 : Bindable → Set where
  eₛ : Sort var
  τₛ : Sort var
  κₛ : Sort no-var
```

Sorts are indexed by the boolish data type Bindable. The index var in a sort constructor indicates that variables for terms of that sort can be bound. In contrast, no-var says that variables for terms of a sort cannot be bound. In this case, System F supports abstracting over expressions and types, but not over kinds. Going forward, we will use the variable s for sorts and the variable S for lists of bindable sorts with type Sorts = List (Sort var).

Syntax

The syntax of System F is represented in a single data type Term that is 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 at. The Debruijn representation of a term is unambiguous and all syntaxes with variables names and potentially shadowed variable names can be translated into Debruijn representation. In Agda, terms are often indexed by the amount of bound variables. The variable constructor then only accepts Debruijn indices, instead of variable names, that are smaller or equal to the current amount of bound variables. As a result, unbound variables cannot be referenced by definition. This technique is also referred to as intrinsically scoped. But indexing Term with a number is not sufficient because System F has both expression variables 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 bindable sorts with type 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 debruijn index i.
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 of type $s \in S$. Membership proofs are inductively defined, similar to the definition of natural numbers. Membership proofs can be constructed using the constructor here refl, where refl is proof that the last element in S is the element we searched for. Alternatively, membership proofs can be constructed via the constructor there x, where x is another membership proof for the sublist S' that has one element less. As discussed, this representation of variables has the advantage that only already bound variables can be referenced by definition.
The unit element tt and unit type '⊤ represent base expressions and types respectively. 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 abstractions and forall types introduce an additional sort e_s, or τ_s respectively, to the index S of their corresponding body to account for the additional new variable.
The application constructor $e_1 \cdot e_2$ applies the argument e_2 to the function e_1.
Similarly, type application $e \bullet \tau$ eliminates type abstractions.
Let bindings let'x= e_2 'in e_1 combine abstraction and application.
The kind ⋆ is kind of all types.
We use abbreviations Var S s = $s \in S$, Expr S = Term S e_s, Type S = Term S τ_s and variables x, e and τ respectively. Furthermore, we use the variable t for an arbitrary Term S s.

Renaming

Renamings ρ of type Ren S_1 S_2 are defined as total functions that map variables Var S_1 s to variables Var S_2 s. Renamings preserve 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 represented as references to elements in S_2 instead of S_1.

```
ren : Ren S₁ S₂ → (Term S₁ s → Term S₂ s)
ren ρ (' x) = ' (ρ _ x)
ren ρ (λ'x→ e) = λ'x→ (ren (extᵣ 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 function ext$_r$. If we want to use the function ext$_r$, the sort argument s can usually be inferred by Agda. Inferring a function argument is denoted with _.

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

The extension of a renaming introduces an additional variable of sort s. Thus, if we encounter the new binding here refl in the extended renaming, then we return the

variable for the new binding here refl. The variables x of the original renaming ρ are weakened by wrapping them in an additional there constructor. The sort arguments are ignored inside the function body by using the wildcard pattern _.
Similar to variables, terms can be weakened using the function wk that shifts all variables present in the term by one recursively.

```
wk : Term S s → Term (S ▷ s') s
wk = ren wkᵣ
```

The function wk_r generates a weakening by wrapping all variables in an additional there constructor.

```
wkᵣ : Ren S (S ▷ s)
wkᵣ _ = there
```

Substitution

The definition of substitutions σ with type Sub S_1 S_2 is similar to the definition of 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 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.

```
extₛ : Sub S₁ S₂ → (s : Sort var) → Sub (S₁ ▷ s) (S₂ ▷ s)
extₛ σ s _ (here refl) = ` here refl
extₛ σ s _ (there x) = wk (σ _ x)
```

For the newly introduced binder of variable here refl, we return the variable term ` here refl. Furthermore, all terms originally present in the substitution σ are weakened using the function wk, that itself uses renaming to shift all variables in the term by one recursively.
The substitution operator t [t'] substitutes the last bound variable in t with t', given that the sort of the last binder corresponds to the sort of term t'.

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

A single substitution single_s : Sub S_1 S_2 → Term S_2 s → Sub (S_1 ▷ s) S_2 takes an existing substitution σ' and substitutes t' for an additional new binding. In the case of _[_], we let σ' be the identity substitution id_s : Sub S S.

Context

Similar to terms, typing contexts Γ of type Ctx S are indexed by the list of bound variables as well. 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 (type-of s) → Ctx (S ▷ s)
```

Contexts are inductively defined and can either be empty ∅ or extended with one element T, using the constructor Γ ▶ T. The variable T represents terms of the sort type-of s. The function type-of maps bindable sorts s to the sort of the term that is stored in Γ for variables of the sort s.

```
type-of eₛ = τₛ
type-of τₛ = κₛ
```

Expression variables require Γ to store the corresponding type. For type variables, Γ stores the corresponding kind. Thus, if we bind a new variable for a term of the sort s, then Γ needs to be extended by a term of the sort type-of s.
The lookup function resolves the type or kind T for a variable x in Γ.

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

In the case split on variables x, both cases wrap the looked up T in a weakening. Thus, T always has index S that aligns 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 by definition.

Typing

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

```
data _⊢_:_ : Ctx S → Term S s → Term S (type-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 : ⋆
```

The rule ⊢'x says that a variable ' x has type τ, if the type for x in Γ is τ.
All unit expressions tt have type '⊤. This is expressed by the rule ⊢⊤.
The rule for abstractions ⊢λ introduces an expression variable of type τ to the body e. Because the resulting body type τ' cannot use the newly introduced expression variable, we let τ' have one variable bound less and weaken it to align with the context Γ ▶ τ. Hence, τ' aligns with τ in the list of bound variables to form the resulting function type $\tau \Rightarrow \tau$'.
The type abstraction rule ⊢Λ introduces a type variable to the body e and results in the forall type ∀'α τ, where τ is the type of e. The type variable in e is introduced by extending Γ with the kind ⋆.
Application is handled by the rule ⊢·. The rule 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, the type application rule ⊢• states that if e has type ∀'α τ, then a can be substituted with another type τ' inside τ.
The rule ⊢let combines the abstraction and application rule.
Regarding 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. Hence, all types are well typed.

Typing of Renaming & Substitution

Because of extrinsic typing, both renamings and substitutions need to have typed counterparts.
We formalize typed renamings ⊢ρ inductively as order preserving embeddings. Thus, if a 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 renamings would allow swapping items and potentially violate the telescoping. Telescoping allows types stored in the context to depend on type variables bound before them.
Interestingly, because of the intrinsically scoped definition of terms, all renamings must be order preserving embeddings by definition. Thus, it should be possible to prove order preservation in the form of lemmas. Instead we choose to represent the rules for order preserving embeddings as constructors of a data type, such that we can access the property of order preservation by matching on the data type.

```
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 (type-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 (type-of s)} →
```

```
ρ : Γ₁  ⇒ᵣ Γ₂ →
(dropᵣ ρ) : Γ₁ ⇒ᵣ (Γ₂ ▶ T')
```

The identity renaming $\vdash$id$_r$ is typed by definition.
The typed extension of a renaming $\vdash$ext$_r$ allows to extend both Γ_1 and Γ_2 by T' and renamed T' respectively. The constructor $\vdash$ext$_r$ corresponds to the typed version of the function ext$_r$ that is used when a binder is encountered.
The constructor $\vdash$drop$_r$ allows 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 total functions, 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 the term $\sigma\ x$ in Γ_2. The type of the term $\sigma\ x$ must be the original type of x in Γ_1 applied to the substitution σ.

Semantics

The semantics of System F are formalized as 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 the unit value v-tt.
We formalize small step semantics, where each constructor represents a single reduction step $e \hookrightarrow e'$. Small step semantics distinguish between β and ξ rules. Meaningful computation in the form of substitution is done by β rules while ξ rules only reduce subexpressions.

```
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
```

The rules β-λ and β-Λ give meaning to application and type application by substituting the applied expression, or type respectively, into the abstraction body. In both cases, we make sure that the abstraction and applied argument must be irreducible.
The reduction rule β-let is equivalent to β-λ and substitutes value e_2 into e_1.
The rules ξ-·$_i$ and ξ-• evaluate subexpressions of applications until e_1 and e_2, or e respectively, are values.
The rule 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 by showing that a typed expression e can either be further reduced to another expression 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)
```

The cases ⊢⊤, ⊢λ and ⊢Λ result in values. The 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 ξ reduction cases ξ-·₁, ξ-·₂, ξ-• and ξ-let follow directly from the induction hypothesis.
For the β reduction cases β-λ, β–Λ and β-let we need to prove that substitutions preserve the typing. We have two different types of substitution present inside the reduction rules: e [e] and e [τ]. Both e[e]-preserves and e[τ]-preserves follow from a more general lemma ⊢σ-preserves. The lemma ⊢σ-preserves proves that applying a typed substitution preserves the typing.

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

The lemma ⊢σ-preserves follows by induction over the typing rules and lemmas about the type preservation of substitutions and renamings. More specifically, we also need to prove that all operations on substitutions preserve the typing as well. For instance, we need to prove the lemma ⊢σ↑ that says that the typed extension of a substitution ⊢extₛ is type-preserving. Because extₛ uses renaming under the hood, we also need to prove the lemma ⊢ρ-preserves that says that applying a renaming preserves the typing. Furthermore we need to prove the lemmas assoc-sub-ren, assoc-ren-ren, assoc-ren-sub and assoc-sub-sub that prove the operations of applying a renaming and substitution to be associative in all combinations. [1]. The soundness property of System F 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.

[1] Considering the fact that the soundness proof for System F is not the main part of this work and resources can be found online [5], the overview of the proof itself is rather short. The full proof can be found on GitHub: `https://github.com/Mari-W/System-Fo/blob/main/proofs/SystemF.agda`

```
data Sort : Bindable → Set where
  oₛ : Sort var
  cₛ : Sort no-var
  - ...
```

Terms of sort o_s can only be constructed using the variable constructor ‘_. Thus, terms of sort o_s are called overloaded variables and sort o_s is indexed by var. We use the variable o for overloaded variables and variable c for constraints. Variables for constraints do not exist in System $\mathrm{F_O}$ and thus c_s is indexed by no-var.

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 the sort o_s inside the body e.
The expression inst‘ o = e_2 ‘in e_1 introduces an additional instance for o. The actual meaning for the instance is given by e_2. Instance expressions do not introduce new bindings and thus, the index S is never extended.
Constraints c can be constructed using constructor o : τ.
A constraint c can be part of a constraint abstraction ƛ c ⇒ e. Constraint abstractions assume the constraint c to be valid inside the body e and result in constraint types [c]⇒ τ. The constraint type lifts the constraint from the expression level to the type level, where it will be implicitly eliminated by the typing rules.
Going forward, we will use the abbreviation Cstr S = Term S c_s.

Renaming & Substitution

Renamings and substitutions in System $\mathrm{F_O}$ 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
```

The single-type$_s$ function only introduces a new binding for types and not arbitrary terms. Because we do not formalize direct semantics for System $\mathrm{F_O}$, only substitutions of types in types are necessary. Type in type substitution appears in the typing rule for type application.

Context

In addition to types and kinds, the existence of overloaded variables is stored inside the context. Overloaded variables act as normal context items. Because overloaded variables themselves do not have a type, but rather multiple types that they can take on, we only need to store their existence in Γ. Thus, similar to type variables, we store kind $\star$ in Γ to denote the existence of an overloaded variable. As a result, the type-of function is extended and returns the sort κ_s when applied to the sort o_s.
The types that an overloaded variable can take on are stored in the form of constraints. Constraints can be introduced to the context by both constraint abstractions and instance expressions.

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

We write Γ ▸ c to pick up a constraint c. Because constraints give an additional meaning to an overloaded variable that is already bound, the index S is not modified. The lookup function in System F_O 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 reference constraints in Γ and not in S. The constraint solving type does need to search in Γ because S does not know about the existence of constraints.

```
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 desired constraint c, then c must be further inside the context. The constraint can either behind an item stored in Γ (under-bind) or a constraint (under-cstr). In the case that c is under a binder, the constraint needs to be weakened, to align in S with the position it is resolved for.
We use the constraint solving type inside the type system to resolve the instance for usages of overloaded variables and to implicitly eliminate constraints.

Typing

We only discuss typing rules not already discussed in the System F specification.

```
data _⊢_:_ : Ctx S → Term S s → Term S (type-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’
  - ...
```

The rule for overloaded variables ⊢‘o 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 the constraint c to Γ and thus assumes c to be valid inside the body e. Constraint abstractions result in the corresponding constraint type [c]⇒ τ that lifts the constraint onto the type level.
Expressions e with constraint type [c]⇒ τ’ have the constraint implicitly eliminated using the rule ⊢⊘, given c can be resolved in Γ. Because the rule can be applied to arbitrary e, it is not syntax directed.
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 as overloaded variable. The existence of o is denoted by extending Γ with kind ⋆. Analogous to the type τ’ inside the abstraction rule ⊢λ, the resulting type τ is weakened to align in S with Γ not extended by ⋆, such that it can act as the resulting type of the typing.
An instance for an overloaded variable o is typed using the rule ⊢inst. We extend Γ with constraint o : τ inside e_1, where τ is the type of e_2.

Typing Renaming & Substitution

Typed renamings are identical to 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 be introduced only to Γ_2 using the ⊢drop-cstr$_r$ constructor. Dropping a constraint corresponds to a typed weakening, similar to constructor ⊢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 F$_O$. Similar to the substitution operator we restrict typed substitutions in System F$_O$ to substitutions of types in types.

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

The constructor ⊢single-type$_s$ allows to introduce an additional new type variable binder that is substituted with type τ. Thus, the constructor ⊢single-type$_s$ complements the single-type$_s$ function. The intuition here is that if we would allow all terms to be introduced using a ⊢term$_s$ constructor, then typed substitutions in System F_O would be arbitrary again. The restriction to type in type substitutions simplifies the type preservation proof for the Dictionary Passing Transform by eliminating cases for non-type terms that would otherwise needed to be proven. Constructors ⊢id$_s$, ⊢ext$_s$, ⊢drop$_s$ and ⊢drop-cstr$_s$ are not shown. All of them function the same way as their counterparts in typed renamings.

5 The Dictionary Passing Transform

5.1 Translation

Sorts

The translation of System F_O sorts to System F sorts only considers sorts that are bindable. The two missing non-bindable sorts c$_s$ and κ$_s$ do not need to be translated. Intuitively there does not even exist a sensible translation for c$_s$.

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

Sorts e$_s$ and τ$_s$ translate to their corresponding counterparts in System F.
Overloaded variables in System F_O translate to normal variables in System F. Thus the sort o$_s$ translates to e$_s$.
Translating the index 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 an additional 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 all positions of new bindings 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 ∅ corresponds to the empty list [].
For each constraint in Γ an additional sort e$_s$ is appended to S.
If we find that a normal item is stored in the context, the sort s is directly translated using the function s⇝s.

Variables

Similar to the translation of sorts 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 correct unique expression variables. We can apply the same idea as seen in the translation for variables because the type for resolved constraints [c]∈ Γ preserves the structure of the context perfectly.

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

Inside the case here we found the correct instance, now variable.
When we encounter a normal binding in the case under-bind, we wrap the variable in the there constructor to respect the binding.
In the case under-cstr we again wrap the variable in an additional there that was not present before.

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 that was originally required by the constraint. This is exactly what we want because for each constraint in System F_O there will be an additional binder in System F that accepts the constraint as higher-order function. Thus, the corresponding function type for that binding is expected in Γ at that position.

Renaming & Substitution

Typed renamings in System F_O translate to untyped renamings in System F.

```
⊢ρ⇝ρ : ∀ {ρ : Fᴼ.Ren Fᴼ.S₁ Fᴼ.S₂} {Γ₁ : Fᴼ.Ctx Fᴼ.S₁} {Γ₂ : Fᴼ.Ctx Fᴼ.S₂} →
  ρ Fᴼ.: Γ₁ ⇒ᵣ Γ₂ →
  F.Ren (Γ⇝S Γ₁) (Γ⇝S Γ₂)
⊢ρ⇝ρ (⊢drop-cstrᵣ ⊢ρ) = F.dropᵣ (⊢ρ⇝ρ ⊢ρ)
- ...
```

Because constraints translate to actual bindings, the constructor ⊢drop-cstr$_r$ translates to drop$_r$ in System F.
Typed renamings ⊢id$_r$, ⊢ext$_r$ and ⊢drop$_r$ translate to their untyped counterparts.
The translation of typed substitutions to untyped substitutions follows similarly.

```
⊢σ⇝σ : ∀ {σ : Fᴼ.Sub Fᴼ.S₁ Fᴼ.S₂} {Γ₁ : Fᴼ.Ctx Fᴼ.S₁} {Γ₂ : Fᴼ.Ctx Fᴼ.S₂} →
  σ Fᴼ.: Γ₁ ⇒ₛ Γ₂ →
  F.Sub (Γ⇝S Γ₁) (Γ⇝S Γ₂)
⊢σ⇝σ (⊢single-typeₛ {τ = τ} ⊢σ) = F.singleₛ (⊢σ⇝σ ⊢σ) (τ⇝τ τ)
- ...
```

The typed renaming ⊢single-type$_s$ translates to its untyped counterpart for arbitrary terms single$_s$.
The cases ⊢id$_s$, ⊢ext$_s$, ⊢drop$_s$ and ⊢drop-cstr$_s$ are analogous to the cases for renamings.

Terms

Types and kinds can be translated without typing information using the function T⇝T. Kind ⋆ translates to its direct counterpart in System F. Furthermore, all System F$_O$ types translate to their direct counterpart in System F, except the constraint type [o : τ]⇒ τ'.

```
τ⇝τ : ∀ {Γ : Fᴼ.Ctx Fᴼ.S} →
  Fᴼ.Type Fᴼ.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 τ ⇒ τ'. 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 of type τ.
Arbitrary terms can only be translated using typing information. The typing carries information about the instances that were resolved for usages of overloaded variables and the instances that were implicitly resolved for constraints. We only look at the translation of System F$_O$ expressions that do not have a direct counterpart in System F.

```
⊢t⇝t : ∀ {Γ : Fᴼ.Ctx Fᴼ.S} {t : Fᴼ.Term Fᴼ.S Fᴼ.s}
             {T : Fᴼ.Term Fᴼ.S (Fᴼ.type-of Fᴼ.s)} →
  Γ Fᴼ.⊢ t : T →
  F.Term (Γ⇝S Γ) (s⇝s Fᴼ.s)
⊢t⇝t (⊢`o o:τ∈Γ) = ` o⇝x o:τ∈Γ
⊢t⇝t (⊢λ ⊢e) = λ`x→ (⊢t⇝t ⊢e)
```

```
⊢t⇝t (⊢⊘ ⊢e o:τ∈Γ) = ⊢t⇝t ⊢e · ` o⇝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 the information $o{:}\tau{\in}\Gamma$ about the instance that was resolved for o. We translate the resolved instance to the unique variable in System F using the o⇝x function from above.
Constraint abstractions translate to normal abstractions with an additional new binding.
An implicitly resolved constraint translates to an explicit application that passes the resolved instance as argument. We again use function o⇝x to translate the resolved instance to the corresponding unique variable.
The decl expression could be removed by the translation as seen in the example at the beginning. Instead decl expressions are translated to useless let bindings that bind a unit value. Because decl expressions bind a new overloaded variable in System $\mathrm{F_O}$, removing them would result in a variable binding less in System F and hence, more complex proofs.
We translate inst expressions to let expressions that introduce an additional binding not present in System $\mathrm{F_O}$.

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 $\mathrm{F_O}$. The function ⇝-pres-⊢ produces a typed System F term for an arbitrary typed System $\mathrm{F_O}$ term ⊢t. The untyped translated System $\mathrm{F_O}$ term ⊢t⇝t t is typed in the translated context Γ⇝Γ Γ and has the typing result T⇝T T.

```
⇝-pres-⊢ : {Γ : Fᴼ.Ctx Fᴼ.S} {t : Fᴼ.Term Fᴼ.S Fᴼ.s}
  {T : Fᴼ.Term Fᴼ.S (Fᴼ.type-of Fᴼ.s)} →
  (⊢t : Γ Fᴼ.⊢ t : T) →
  (Γ⇝Γ Γ) F.⊢ (⊢t⇝t ⊢t) : (T⇝T T)
⇝-pres-⊢ (⊢`x {x = x} Γx≡τ) = ⊢`x (⇝-pres-lookup x Γx≡τ)
⇝-pres-⊢ (⊢`o o:τ∈Γ) = ⊢`x (⇝-pres-cstr-solve o:τ∈Γ)
⇝-pres-⊢ (⊢let ⊢e₂ ⊢e₁) = ⊢let (⇝-pres-⊢ ⊢e₂)
  (subst (_ F.⊢ ⊢t⇝t ⊢e₁ :_) ⇝-dist-wk-type(⇝-pres-⊢ ⊢e₁))
⇝-pres-⊢ (⊢λ {c = (` o : τ)} ⊢e) = ⊢λ
  (subst (_ F.⊢ ⊢t⇝t ⊢e :_) ⇝-dist-wk-inst-type (⇝-pres-⊢ ⊢e))
⇝-pres-⊢ (⊢⊘ ⊢e o:τ∈Γ) = ⊢· (⇝-pres-⊢ ⊢e) (⊢`x (⇝-pres-cstr-solve o:τ∈Γ))
⇝-pres-⊢ (⊢• {τ = τ} {τ' = τ'} ⊢e) = subst (_ F.⊢ ⊢t⇝t ⊢e • τ⇝τ τ' :_)
  (⇝-dist-τ[τ'] τ' τ) (⊢• (⇝-pres-⊢ ⊢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 ⇝-pres-lookup. The lemma produces an equality proof of type F.lookup (Γ⇝Γ Γ) (x⇝x x) ≡ (τ⇝τ τ). With the lemma ⇝-pres-lookup the typing rule ⊢\`x can be translated to the typing rule for variables in System F.

Similarly, the lemma ⇝-pres-cstr-solve translates the proof that an instance o : τ was resolved for an overloaded variable o to proof that the unique variable o⇝x o:τ∈Γ has type τ⇝τ τ in Γ⇝Γ Γ: F.lookup (Γ⇝Γ Γ) (o⇝x o:τ∈Γ) ≡ (τ⇝τ τ). Using lemma ⇝-pres-cstr-solve the typing rule for overloaded variables ⊢‘o can be translated to the typing rule for normal variables ⊢‘x.
Typed let bindings ⊢let ⊢e_2 ⊢e_1 translate to typed let bindings in System F. The typing rule ⊢e_2 is translated directly using the induction hypothesis. Because the typing for 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 the translated τ’ in System F. Lemma ⇝-dist-wk has type τ⇝τ {Γ = Γ ▶ T} (F^O.wk τ’) ≡ F.wk (τ⇝τ τ’) and is used to substitute the required equivalence into the translated typing rule ⇝-pres-⊢ ⊢e_1.
Typed constraint abstractions ⊢λ translate to normal abstractions in System F. Inside the typing ⊢e, the result type τ for 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 ⇝-dist-wk-inst-type proves that translating τ in Γ extended by a constraint is equivalent to weakening τ after the translation: τ⇝τ {Γ = Γ ▸ (‘ o : τ’)} τ ≡ F.wk (τ⇝τ τ). The lemma follows because the constraint translates to an actual binding and consequently, both sides have an additional unnecessary expression binding that τ cannot use.
Implicitly resolved constraints ⊢⊘ carry the information o:τ∈Γ about the instance that was resolved. In System F the former constraint is explicitly passed as variable pointing to the correct translated instance. Thus, ⊢⊘ results in typed application ⊢·. We apply the correct instance using lemma ⇝-pres-cstr-solve to get the correct unique variable for the resolved constraint.
The 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 is applied in System F. Using lemma ⇝-dist-τ[τ’] of type (τ⇝τ {Γ = Γ ▶ ⋆} τ’ F.[τ⇝τ τ]) ≡ τ⇝τ (τ’ F^O.[τ]) we can substitute the equivalence into the System F typing rule ⊢• (⇝-pres-⊢ ⊢e).
The translation of ⊢⊤, ⊢λ, ⊢·, ⊢decl and ⊢inst is either a direct translation or uses similar ideas and no other lemmas than the ones discussed.

Renaming

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

```
⇝-dist-ren-type : {ρ : Fᴼ.Ren Fᴼ.S₁ Fᴼ.S₂}
                    {Γ₁ : Fᴼ.Ctx Fᴼ.S₁} {Γ₂ : Fᴼ.Ctx Fᴼ.S₂} →
  (⊢ρ : ρ Fᴼ.: Γ₁ ⇒ᵣ Γ₂) →
  (τ : Fᴼ.Type Fᴼ.S₁) →
  F.ren (⊢ρ⇝ρ ⊢ρ) (τ⇝τ τ) ≡ τ⇝τ (Fᴼ.ren ρ τ)
⇝-dist-ren-type ⊢ρ (‘ x) = cong ‘_ (⇝-dist-ren-var ⊢ρ x)
⇝-dist-ren-type ⊢ρ ([ ‘ o : τ ]⇒ τ’) = cong₂ _⇒_
```

```
  (⇝-dist-ren-type ⊢ρ τ) (⇝-dist-ren-type ⊢ρ τ')
- ...
```

The case for type variables needs an additional lemma ⇝-dist-ren-var specifically for type variables. Lemma ⇝-dist-ren-var proves an analogous statement, but for type variables applied to a renaming: (⊢ρ⇝ρ ⊢ρ) _ (x⇝x x) ≡ x⇝x (ρ x). This statement can be proven via straight forward induction over typed System $\mathrm{F_O}$ 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 ⇝-dist-τ[τ'] follows from a more general lemma about type in type substitutions ⇝-dist-sub-type. The lemma ⇝-dist-sub-type follows by straight forward induction over System $\mathrm{F_O}$ types as well, except the case for type variables. Other than with renamings, the cases for lemma ⇝-dist-sub-type-var do not follow directly from the induction hypothesis. To understand why, we at look at the induction case ⊢ext$_s$.

```
⇝-dist-sub-var : {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))
⇝-dist-sub-var (⊢extₛ ⊢σ) (here refl) = refl
⇝-dist-sub-var (⊢extₛ {σ = σ} ⊢σ) (there x) = trans
  (cong F.wk (⇝-dist-sub-var ⊢σ x)) (⇝-dist-ren-type F^O.⊢wkᵣ (σ x))
```

The 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 weaken both sides of the equality that results from the outer induction hypothesis. We then combine the weakened induction hypothesis with proof that weakenings preserve the translation using transitivity. The intuition here is that we need the renaming lemma ⇝-dist-ren-type because ext$_s$ is defined by weakening the term that result of the substitution σ being applied to the variable x.
Both ⊢id$_s$ and ⊢single-type$_s$ follow directly from the induction hypothesis. The cases for ⊢drop$_s$ and ⊢drop-cstr$_s$ are similar to the case ⊢ext$_s$ and align with their definition.

Variables

We first look at the proof for lemma ⇝-pres-lookup. Lemma ⇝-pres-lookup is proven via induction over the System $\mathrm{F_O}$ context Γ.

```
⇝-pres-lookup : ∀ {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)
⇝-pres-lookup {Γ = Γ ▶ τ} (here refl) refl = ⇝-dist-ren-type F^O.⊢wkᵣ τ
⇝-pres-lookup {Γ = Γ ▶ _} {τ'} (there x) refl = trans
  (cong F.wk (⇝-pres-lookup x refl))
```

```
    (⇝-dist-ren-type Fᴼ.⊢wkᵣ (Fᴼ.lookup Γ x))
  - ...
```

As an example we will look at case Γ ▶ τ. The case 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 type τ that is looked up in Γ in both cases here and there, both use lemma ⇝-dist-ren-type to account for the weakening.
The case Γ ▸ c follows analogous and case $\emptyset$ is impossible because by definition, there must exist a context item if there exists a variable.
Lemma ⇝-pres-cstr-solve can proven via induction over the type for resolved constraints [c]∈ Γ. The proof is analogous to the proof shown for ⇝-pres-lookup because the type for resolved constraints preserves the structure of the context Γ perfectly.
This finishes up the type preservation proof for the Dictionary Passing Transform from System $\mathrm{F_O}$ to System F.

6 Further Work and Conclusion

6.1 Hindley Milner with Overloading

In this scenario the source language for the Dictionary Passing Transform would be an extended Hindley-Milner [8] based system $\mathrm{HM_O}$ and the target language would be Hindley-Milner. The Hindley-Milner system is a restricted form of System F and $\mathrm{HM_O}$would be a restricted form of System $\mathrm{F_O}$.
Formalizing Hindley-Milner would require two new sorts, m$_s$ and p$_s$ for mono and poly types, in favour of the sort for 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 Hindley-Milner is also called let polymorphism. As a result, constraints must have the form o : m, where m is a mono type. Because there are no expression level constructs to introduce type variables in Hindley-Milner, we would need to embed constraints into explicit type annotation of instances instead of introducing them on the expression level. The explicit type annotation for instances would allow poly types because instance expressions 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 full type inference for instances and overloaded variables 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. For now it remains unclear, if the inference algorithm can be extended to work for arbitrary mono type constraints and how constraints should be handled by the inference algorithm in general.

6.2 Proving Semantic Preservation

For now System $\mathrm{F_O}$ does not have direct semantics formalized. In Section 2.3 we already discussed that correct semantics are already implicitly given by the translation, but it could also be interesting to investigate direct semantics on the System $\mathrm{F_O}$ and $\mathrm{HM_O}$ syntaxes.

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 could introduce a reduction rule β-o-* that reduces ` o · e_1 · .. · e_n to e · e_1 · .. · e_n, where e is the resolved instance based on the types of e_1 .. e_n. The drawback would be that partial application to overloaded variables would not be possible. Alternately, we could apply the restriction mentioned above that restricts all instances for an overloaded variable o to differ in the type of their first argument. In consequence, the resolved instance for o in a single application step ` o · e would be decidable.

Let ⊢e ↪ ⊢e' be such a typed small step semantic for System F_O. We would need to prove something similar to: If ⊢e ↪ ⊢e' then ∃ [e"] (⊢e↪e'⇝e↪e' ⊢e ↪* e") × (⊢e↪e'⇝e↪e' ⊢e' ↪* e"), where ⊢e↪e'⇝e↪e' translates typed System F_O reductions to a untyped System F reductions. Instead of translating reduction steps directly, we prove that both translated ⊢e and translated ⊢e' reduce to a 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 an additional application step in System F. For now it remains unclear if semantic preservation can be proven using induction over the typed semantic rules or if logical relations are needed [1].

6.3 Related Work

The ideas for the required restrictions to preserve the inference algorithm in Section 6.1 originate from System O [9]. System O is a language extension to the Hindley-Milner System. In contrast to System $F_O$and similar to HM_O, constraints are not introduced on the expression level and instead are introduced via explicit type annotations of instances as part of forall types.

For instance, the valid System F_O and HM_Otype `∀α. ∀β. [a : α → α → α] => [b : β → β → β] => ..` would be expressed as `∀α. (a : α → α → α) => ∀β. (b : β → β → β) => ..` in System O. Inside the System O type, we only introduce one constraint per type variable, but a list of constraints would be allowed. The part about the inference algorithm that remained unclear in section [?] is solved in System O by restricting constraints to begin their type with the type variable bound by the quantifier that they are introduced in. In HM_O such a a connection does not exist.

Originally the plan was to formalize System O in Agda, but multiple issues arose in the type preservation proof. First, because we have a list of n constraints for each forall type, translating them results in n new lambda bindings (and n applications when constraints are explicitly passed) in one induction step. Furthermore, the translation of the System O type above must pull out forall quantifiers, because translating the constraints directly to higher oder functions would break the rule that function types are only allowed to be built from mono types. Thus, the translated System O type from above should not be `∀α. (α → α → α) → ∀β. (β → β → β) → ..`, but rather `∀α. ∀β. (α → α → α) → (β → β → β) → ..` to be a valid Hindley-Milner type. Including the additional transform on types complicates the type preservation proof immensely, because the transform affects the type of the next n expressions and thus straight forward induction cannot be used.

There exist other formalizations that are more similar to actual typeclasses in Haskell [10] [4]. A more general approach to constraint types is presented by the theory of qualified types [6].

6.4 Conclusion

We have formalized both System F and System F_O in Agda. In the process, we explored the technique of using an intrinsically scoped and sorted data type to represent syntaxes. The essence of System F_O was to act as a core calculus that captures the idea of ad-hoc polymorphism. Furthermore, 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 from System F_O to System F to be type-preserving. The full formalization of the Dictionary Passing Transform, System F_O and System F can be found as Agda code files [2].
One trick used in the formalization was to detach constraints from forall types such that each constraint is bound using a single constraint abstraction. As a result, we were able to translate constraint abstractions directly to lambda abstractions in System F. Another trick that we used was to preserve the structure of the context in the type for resolved constraints. In consequence, the translation of resolved constraints to unique variables was straight forward because the position of new bindings introduced by the translation was perfectly known.
A reasonable next step would be to define direct semantics for System F_O and prove semantic preservation for the Dictionary Passing Transform.

[2] Formalizations and proofs as Agda code files: `https://github.com/Mari-W/System-Fo/tree/main/proofs`

References

1. Abel, A., Allais, G., Hameer, A., Pientka, B., Momigliano, A., Schäfer, S. & Stark, K. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal Of Functional Programming.* **29** pp. e19 (2019), http://dx.doi.org/10.1017/S0956796819000170
2. Barendregt, H. Introduction to generalized type systems. *Journal Of Functional Programming.* **1**, 125-154 (1991), https://doi.org/10.1017%2Fs0956796800020025
3. Bove, A., Dybjer, P., Norell, U. (2009). A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds) Theorem Proving in Higher Order Logics. TPHOLs 2009. Lecture Notes in Computer Science, vol 5674. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03359-9_6
4. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. ACM Trans. Program. Lang. Syst. 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700
5. Chapman, J., Kireev, R., Nester, C. & Wadler, P. System F in Agda, for Fun and Profit. (2019,10)
6. Jones, M.P. (1992). A theory of qualified types. In: Krieg-Brückner, B. (eds) ESOP '92. ESOP 1992. Lecture Notes in Computer Science, vol 582. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-55253-7_17
7. Martin-Löf, P. & Sambin, G. Intuitionistic Type Theory. (Bibliopolis, 1984), https://www.cse.chalmers.se/~peterd/papers/MartinL%C3%B6f1984.pdf
8. Milner, R. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences.* **17**, 348-375 (1978), https://www.sciencedirect.com/science/article/pii/0022000078900144
9. Odersky, M., Wadler, P. & Wehr, M. A Second Look at Overloading. *Proceedings Of The Seventh International Conference On Functional Programming Languages And Computer Architecture.* pp. 135-146 (1995), https://doi.org/10.1145/224164.224195
10. P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89). Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

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