



# 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 $F_O$ .....	4
1.4	Translating System $F_O$ back to System $F$ .....	4
2	Preliminary .....	5
2.1	Dependently Typed Programming in Agda .....	5
2.2	Design Decisions for the Agda Formalization .....	6
2.3	Overview of the Type Preservation Proof .....	6
3	System $F$ .....	6
3.1	Specification .....	6
3.2	Soundness .....	12
4	System $F_O$ .....	13
4.1	Specification .....	13
5	The Dictionary Passing Transform .....	16
5.1	Translation .....	16
5.2	Type Preservation .....	19
6	Further Work and Conclusion .....	22
6.1	Hindley Milner with Overloading .....	22
6.2	Proving Semantic Preservation .....	23
6.3	Related Work .....	23
6.4	Conclusion .....	24

## 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 helps overcome this verbosity. 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 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  $\alpha$  via type constraints  $\text{Eq } \alpha \Rightarrow \tau$  to only be substituted by a concrete type  $\tau$  if there exists an instance  $\text{Eq } \tau$ .

#### Example: Overloading Equality in Haskell

In this example the function  $\text{eq} : \alpha \rightarrow \alpha \rightarrow \text{Bool}$  is overloaded with different meanings for different substitutions  $\{\alpha \mapsto \tau\}$ . We want to be able to call  $\text{eq}$  on both  $\{\alpha \mapsto \text{Nat}\}$  and  $\{\alpha \mapsto [\beta]\}$ , where  $\beta$  is a type and there exists an instance that gives meaning to  $\text{eq} : \beta \rightarrow \beta \rightarrow \text{Bool}$ . The intuition here is that we want to be able to compare natural numbers  $\text{Nat}$  and lists  $[\beta]$ , given the elements of type  $\beta$  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  $\text{Eq}$  is declared with a single generic function signature  $\text{eq} :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$ . Next, we instantiate  $\text{Eq}$  for  $\{\alpha \mapsto \text{Nat}\}$ . After that,  $\text{Eq}$  is instantiated for  $\{\alpha \mapsto [\beta]\}$ , given that an instance  $\text{Eq } \beta$  can be found. Hence we can call  $\text{eq}$  on expressions with type  $\text{Nat}$  and  $[\text{Nat}]$ . In the latter case, the type constraint  $\text{Eq } \beta \Rightarrow \dots$  in the instance for lists resolves to the instance for natural numbers.

### 1.3 Desugaring Typeclass Functionality to System F<sub>O</sub>

System F<sub>O</sub> is a minimal calculus with support for overloading and polymorphism based on System F. In System F<sub>O</sub> 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  $\tau$  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<sub>O</sub> 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  $\lambda (o : \tau). e'$ . Constraint abstractions result in constraint types  $[o : \tau] \Rightarrow \tau'$ . We introduce constraints on the expression level because instance expressions do not have a type annotation in System F<sub>O</sub>. Expressions with constraint types  $[o : \tau] \Rightarrow \tau'$  are implicitly treated as expressions of type  $\tau'$  by the type system, given that the constraint  $o : \tau$  can be resolved.

#### Example: Overloading Equality in System F<sub>O</sub>

Recall the Haskell example from above. The same functionality can be expressed in System F<sub>O</sub>. 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. λ(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  $eq : \beta \rightarrow \beta \rightarrow \text{Bool}$  introduced by the constraint abstraction  $\lambda$  is satisfied. Because System F<sub>O</sub> is based on System F, we are required to bind type variables using type abstractions  $\Lambda$  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<sub>O</sub> does not actually support recursion. Extending System F<sub>O</sub> with recursive let bindings and thus recursive instances is known to be straight forward.

### 1.4 Translating System F<sub>O</sub> back to System F

System F<sub>O</sub> can be translated back to System F. Hence, System F<sub>O</sub> 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<sub>O</sub> 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 oτ = e in e'` expressions, where `oτ` is a unique name with respect to the type  $\tau$  of the expression `e`. Constraint abstractions  $\lambda (o : \tau) . e'$  translate to normal abstractions  $\lambda o_{\tau} . e'$ . Hence, constraint types  $[o : \tau] \Rightarrow \tau'$  translate to function types  $\tau \rightarrow \tau'$ . Invocations of overloaded function names `o` translate to the correct unique variable name `oτ` 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: Dictionary 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 `eqi`. When `eq2` 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 `eq1` 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 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  $e$ ,  $\tau_s$  represents types  $\tau$  and  $\kappa_s$  represents kind. Although, 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  $e_s$ ,  $\tau_s$ , and  $\kappa_s$ . 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  $e_s$   $\vdash$  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

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  $\Gamma$  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  $\rightsquigarrow : (\Gamma \vdash_{F_O} t : T) \rightarrow \text{Term}_F s$  be the Dictionary Passing Transform that translates well typed System  $F_O$  terms to untyped System F terms. Further, let  $\rightsquigarrow_\Gamma : \text{Ctx}_{F_O} \rightarrow \text{Ctx}_F$  be the transform of contexts and  $\rightsquigarrow_T : \text{Term}_{F_O} s' \rightarrow \text{Term}_F s'$  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.

## 3 System F

### 3.1 Specification

#### Sorts

The formalization of System F requires three sorts:  $e_s$  for expressions,  $\tau_s$  for types and  $\kappa_s$  for kinds.

```
data Sort : Ctxable → Set where
  es : Sort  $\top^C$ 
   $\tau_s$  : Sort  $\top^C$ 
   $\kappa_s$  : Sort  $\perp^C$ 
```

Sorts are indexed by the boolean data type `Ctxable`. The index  $\top^C$  indicates that variables for terms of a sort can be bound. In contrast,  $\perp^C$  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 contextable sorts with type `Sorts = List (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 at. 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 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 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 e_s
  λ'x→ _   : Term (S ▷ e_s) e_s → Term S e_s
  Λ'α→ _   : Term (S ▷ τ_s) e_s → Term S e_s
  _ · _    : Term S e_s → Term S e_s → Term S e_s
  _ • _    : Term S e_s → Term S τ_s → Term S e_s
  let'x= _ 'in _ : Term S e_s → Term (S ▷ e_s) e_s → Term S e_s
  'T       : Term S τ_s
  _ ⇒ _    : Term S τ_s → Term S τ_s → Term S τ_s
  ∀'α _    : Term (S ▷ τ_s) τ_s → Term S τ_s
  ★       : Term S κ_s
```

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  $S$  with one element less. In consequence, we can only reference already bound variables. The unit element `tt` and unit type `'T` 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  $\forall' \alpha \tau'$ . Both abstractions introduce an additional sort  $e_s$ , or  $\tau_s$  respectively, to the index  $S$  of the body  $e'$  to account for the newly bound 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= e2 'in e1` combine abstraction and application.

The kind `★` is kind of all types.

We use abbreviations `Var S s = s ∈ S`, `Expr S = Term S e_s`, `Type S = Term S τ_s` and variables  $x$ ,  $e$  and  $\tau$  respectively. Furthermore, we use the variable  $t$  for an arbitrary `Term S s`.

## Renaming

Renamings  $\rho$  of type  $\text{Ren } S_1 \ S_2$  are defined as total functions mapping variables  $\text{Var } S_1$   $s$  to variables  $\text{Var } S_2 \ s$ . 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  $\text{Ren } S_1 \ S_2$  to a term  $\text{Term } S_1 \ s$  yields a new term  $\text{Term } S_2 \ s$ , where variables are now 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  $\text{ext}_r$ :  $\text{Ren } S_1 \ S_2 \rightarrow \text{Ren } (S_1 \triangleright s) \ (S_2 \triangleright s)$ .

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
```

Because variables are represented as membership proofs, shifting variables by one binder is accomplished by wrapping them in an additional  $\text{there}$  constructor.

## Substitution

The definition of substitutions  $\sigma$  with type  $\text{Sub } S_1 \ S_2$  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  $\text{sub}$  function is analogous to applying a renaming using  $\text{ren}$ . If we encounter a binder in  $\text{sub}$ , the substitution must be extended using function  $\text{ext}_s$ .

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

The extension of a substitution is defined as the weakening of all terms that result in the substitution being applied to variables  $x$ .

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  $\text{single}_s : \text{Sub } S_1 \ S_2 \rightarrow \text{Term } S_2 \ s \rightarrow \text{Sub } (S_1 \triangleright s) \ S_2$  takes an existing substitution  $\sigma'$  and substitutes  $t'$  for an additional new binding. In the case of  $\_ [ \_ ]$ , we let  $\sigma'$  be the identity substitution  $\text{id}_s : \text{Sub } S \ S$ .



## Context

Similar to terms, typing contexts  $\Gamma$  of type  $\text{Ctx } S$  are indexed by the list of bound variables as well. In consequence, only types and kinds for bound variables can be stored in  $\Gamma$  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 the constructor  $\Gamma \blacktriangleright T$ . The variable  $T$  represents terms of the sort  $\text{kind-of } s$ . The function  $\text{kind-of}$  maps contextable sorts  $s$  to the sort of the term that is stored in  $\Gamma$  for variables of the sort  $s$ .

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

Expression variables require  $\Gamma$  to store the corresponding type. For type variables,  $\Gamma$  stores the corresponding kind. Thus, if we bind a new variable for a term of the sort  $s$ , the context  $\Gamma$  needs to be extended by a term of the sort  $\text{kind-of } s$ .

The `lookup` function resolves the type or kind for a variable  $x$  in  $\Gamma$ .

```
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 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  $\Gamma$ .

## Typing

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

```
data _⊢_ : Ctx S → Term S s → Term S (kind-of s) → Set where
  ⊢'x :
    lookup Γ x ≡ τ →
    Γ ⊢ ' x : τ
  ⊢T :
    Γ ⊢ tt : 'T
  ⊢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 →
```

$$\begin{array}{l}
\Gamma \vdash e_2 : \tau_1 \rightarrow \\
\Gamma \vdash e_1 \cdot e_2 : \tau_2 \\
\vdash \bullet : \\
\Gamma \vdash e : \forall' \alpha \tau \rightarrow \\
\Gamma \vdash e \bullet \tau' : \tau [ \tau' ] \\
\vdash \text{let} : \\
\Gamma \vdash e_2 : \tau \rightarrow \\
\Gamma \blacktriangleright \tau \vdash e_1 : \text{wk } \tau' \rightarrow \\
\Gamma \vdash \text{let}' x = e_2 \text{ 'in } e_1 : \tau' \\
\vdash \tau : \\
\Gamma \vdash \tau : \star
\end{array}$$

The rule  $\vdash'x$  says that a variable  $'x$  has type  $\tau$ , if the type for  $x$  in  $\Gamma$  is  $\tau$ .

All unit expressions  $\text{tt}$  have type  $'\top$ . This is expressed by the rule  $\vdash \top$ .

The rule for abstractions  $\vdash \lambda$  introduces an expression variable of type  $\tau$  to the body  $e$ . Because the resulting body type  $\tau'$  cannot use the newly introduced expression variable, we let  $\tau'$  have one variable bound less and weaken it to align with the context  $\Gamma \blacktriangleright \tau$ . Hence,  $\tau'$  aligns with  $\tau$  in the list of bound variables to form the resulting function type  $\tau \Rightarrow \tau'$ .

The type abstraction rule  $\vdash \Lambda$  introduces a type variable to the body  $e$  and results in the forall type  $\forall' \alpha \tau$ , where  $\tau$  is the type of  $e$ . The type variable in  $e$  is introduced by extending  $\Gamma$  with the kind  $\star$ .

Application is handled by the rule  $\vdash \cdot$ . The rule says that if  $e_1$  is a function from  $\tau_1$  to  $\tau_2$  and  $e_2$  has type  $\tau_1$ , then  $e_1 \cdot e_2$  has type  $\tau_2$ .

Similarly, the type application rule  $\vdash \bullet$  states that if  $e$  has type  $\forall' \alpha \tau$ , then  $a$  can be substituted with another type  $\tau'$  inside  $\tau$ .

The rule  $\vdash \text{let}$  combines the abstraction and application rule.

Regarding the typing of types, the rule  $\vdash \tau$  indicates that all types  $\tau$  are well formed and have kind  $\star$ . Type variables are correctly typed per definition and type constructors  $\forall' \alpha$  and  $\Rightarrow$  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  $\vdash \rho$  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.

$$\begin{array}{l}
\text{data } \_ : \_ \Rightarrow_r \_ : \text{Ren } S_1 \ S_2 \rightarrow \text{Ctx } S_1 \rightarrow \text{Ctx } S_2 \rightarrow \text{Set where} \\
\vdash \text{id}_r : \forall \{ \Gamma \} \rightarrow \_ : \_ \Rightarrow_r \_ \{ S_1 = S \} \{ S_2 = S \} \text{id}_r \Gamma \Gamma \\
\vdash \text{ext}_r : \forall \{ \rho : \text{Ren } S_1 \ S_2 \} \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \} \\
\quad \{ T' : \text{Term } S_1 \text{ (kind-of } s) \} \rightarrow \\
\quad \rho : \Gamma_1 \Rightarrow_r \Gamma_2 \rightarrow \\
\quad (\text{ext}_r \rho) : (\Gamma_1 \blacktriangleright T') \Rightarrow_r (\Gamma_2 \blacktriangleright \text{ren } \rho \ T') \\
\vdash \text{drop}_r : \forall \{ \rho : \text{Ren } S_1 \ S_2 \} \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \} \\
\quad \{ T' : \text{Term } S_2 \text{ (kind-of } s) \} \rightarrow
\end{array}$$

$$\begin{aligned} \rho : \Gamma_1 &\Rightarrow_r \Gamma_2 \rightarrow \\ (\text{drop}_r \rho) : \Gamma_1 &\Rightarrow_r (\Gamma_2 \blacktriangleright T') \end{aligned}$$

The identity renaming  $\text{id}_r$  is typed by definition.

The typed extension of a renaming  $\text{ext}_r$  allows to extend both  $\Gamma_1$  and  $\Gamma_2$  by  $T'$  and renamed  $T'$  respectively. The constructor  $\text{ext}_r$  corresponds to the typed version of the function  $\text{ext}_r$  that is used when a binder is encountered.

The constructor  $\text{drop}_r$  allows to introduce  $T'$  only in  $\Gamma_2$ . Hence,  $\text{drop}_r \text{id}_r$  corresponds to the typed weakening of a term.

Typed Substitutions are defined as total functions, similar to untyped substitutions.

$$\begin{aligned} \_ : \_ \Rightarrow_s \_ : \text{Sub } S_1 \ S_2 \rightarrow \text{Ctx } S_1 \rightarrow \text{Ctx } S_2 \rightarrow \text{Set} \\ \_ : \_ \Rightarrow_s \_ \{S_1 = S_1\} \ \sigma \ \Gamma_1 \ \Gamma_2 = \forall \{s\} \ (x : \text{Var } S_1 \ s) \rightarrow \\ \Gamma_2 \vdash \sigma \ \_ \ x : (\text{sub } \sigma \ (\text{lookup } \Gamma_1 \ x)) \end{aligned}$$

Typed substitutions  $\vdash \sigma$  map variables  $x \in S_1$  to the corresponding typing of  $\sigma x$  in  $\Gamma_2$ . The typing result of  $\sigma x$  is the original type of  $x$  in  $\Gamma_1$  applied to  $\sigma$ .

## 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.

$$\begin{aligned} \text{data Val} : \text{Expr } S \rightarrow \text{Set} \text{ where} \\ \text{v-}\lambda : \text{Val } (\lambda'x \rightarrow e) \\ \text{v-}\Lambda : \text{Val } (\Lambda' \alpha \rightarrow e) \\ \text{v-tt} : \forall \{S\} \rightarrow \text{Val } (\text{tt } \{S = S\}) \end{aligned}$$

System F has three values. The two closure values  $\text{v-}\lambda$  and  $\text{v-}\Lambda$  and the unit value  $\text{v-tt}$ . We formalize small step semantics where each constructor represents a single reduction step  $e \hookrightarrow e'$ . We distinguish between  $\beta$  and  $\xi$  rules. Meaningful computation in the form of substitution is done by  $\beta$  rules while  $\xi$  rules only reduce subexpressions.

$$\begin{aligned} \text{data } \_ \hookrightarrow \_ : \text{Expr } S \rightarrow \text{Expr } S \rightarrow \text{Set} \text{ where} \\ \beta\text{-}\lambda : \\ \text{Val } e_2 \rightarrow \\ (\lambda'x \rightarrow e_1) \cdot e_2 \hookrightarrow e_1 [e_2] \\ \beta\text{-}\Lambda : \\ (\Lambda' \alpha \rightarrow e) \bullet \tau \hookrightarrow e [\tau] \\ \beta\text{-let} : \\ \text{Val } e_2 \rightarrow \\ \text{let}'x = e_2 \text{ 'in } e_1 \hookrightarrow (e_1 [e_2]) \\ \xi\text{-}\cdot_1 : \\ e_1 \hookrightarrow e \rightarrow \\ e_1 \cdot e_2 \hookrightarrow e \cdot e_2 \\ \xi\text{-}\cdot_2 : \\ e_2 \hookrightarrow e \rightarrow \\ \text{Val } e_1 \rightarrow \\ e_1 \cdot e_2 \hookrightarrow e_1 \cdot e \\ \xi\text{-}\bullet : \end{aligned}$$

$$\begin{array}{l}
e \hookrightarrow e' \rightarrow \\
e \bullet \tau \hookrightarrow e' \bullet \tau \\
\xi\text{-let} : \\
e_2 \hookrightarrow e \rightarrow \\
\text{let}'x = e_2 \text{ 'in } e_1 \hookrightarrow \text{let}'x = e \text{ 'in } e_1
\end{array}$$

The rules  $\beta\text{-}\lambda$  and  $\beta\text{-}\Lambda$  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 a irreducible.

The reduction rule  $\beta\text{-let}$  is equivalent to  $\beta\text{-}\lambda$  and substitutes value  $e_2$  into  $e_1$ .

The rules  $\xi\text{-}\cdot$  and  $\xi\text{-}\bullet$  evaluate subexpressions of applications until  $e_1$  and  $e_2$ , or  $e$  respectively, are values.

The rule  $\xi\text{-let}$  reduces the bound expression  $e_2$  until  $e_2$  is a value and  $\beta\text{-let}$  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.

$$\begin{array}{l}
\text{progress} : \\
\emptyset \vdash e : \tau \rightarrow \\
(\exists [e'] (e \hookrightarrow e')) \uplus \text{Val } e \\
\text{progress } \vdash \top = \text{inj}_2 \text{ v-tt} \\
\text{progress } (\vdash \lambda \_) = \text{inj}_2 \text{ v-}\lambda \\
\text{progress } (\vdash \Lambda \_) = \text{inj}_2 \text{ v-}\Lambda \\
\text{progress } (\vdash \{e_1 = e_1\} \{e_2 = e_2\} \vdash e_1 \vdash e_2) \text{ with progress } \vdash e_1 \mid \text{progress } \vdash e_2 \\
\ldots \mid \text{inj}_1 (e_1', e_1 \hookrightarrow e_1') \mid \_ = \text{inj}_1 (e_1' \cdot e_2, \xi\text{-}\cdot e_1 \hookrightarrow e_1') \\
\ldots \mid \text{inj}_2 v \mid \text{inj}_1 (e_2', e_2 \hookrightarrow e_2') = \text{inj}_1 (e_1 \cdot e_2', \xi\text{-}\cdot e_2 \hookrightarrow e_2' v) \\
\ldots \mid \text{inj}_2 (\text{v-}\lambda \{e = e_1\}) \mid \text{inj}_2 v = \text{inj}_1 (e_1 [e_2], \beta\text{-}\lambda v) \\
\text{progress } (\vdash \bullet \{\tau' = \tau'\} \vdash e) \text{ with progress } \vdash e \\
\ldots \mid \text{inj}_1 (e', e \hookrightarrow e') = \text{inj}_1 (e' \bullet \tau', \xi\text{-}\bullet e \hookrightarrow e') \\
\ldots \mid \text{inj}_2 (\text{v-}\Lambda \{e = e\}) = \text{inj}_1 (e [\tau'], \beta\text{-}\Lambda) \\
\text{progress } (\vdash \text{let } \{e_2 = e_2\} \{e_1 = e_1\} \vdash e_2 \vdash e_1) \text{ with progress } \vdash e_2 \\
\ldots \mid \text{inj}_1 (e_2', e_2 \hookrightarrow e_2') = \text{inj}_1 ((\text{let}'x = e_2' \text{ 'in } e_1), \xi\text{-let } e_2 \hookrightarrow e_2') \\
\ldots \mid \text{inj}_2 v = \text{inj}_1 (e_1 [e_2], \beta\text{-let } v)
\end{array}$$

The cases  $\vdash \top$ ,  $\vdash \lambda$  and  $\vdash \Lambda$  result in values. The application cases  $\vdash \cdot$ ,  $\vdash \bullet$  and  $\vdash \text{let}$  follow directly from the induction hypothesis.

#### Subject Reduction

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

$$\begin{array}{l}
\text{subject-reduction} : \forall \{F : \text{Ctx } S\} \rightarrow \\
F \vdash e : \tau \rightarrow
\end{array}$$

```


$$e \hookrightarrow e' \rightarrow$$


$$\Gamma \vdash e' : \tau$$


$$\text{subject-reduction } (\vdash \cdot (\beta\text{-}\lambda \vdash e_1) \vdash e_2) (\beta\text{-}\lambda v_2) = \text{e[e]-preserves } \vdash e_1 \vdash e_2$$


$$\text{subject-reduction } (\vdash \cdot \vdash e_1 \vdash e_2) (\xi\text{-}\cdot_1 e_1 \hookrightarrow e) = \vdash \cdot (\text{subject-reduction } \vdash e_1 e_1 \hookrightarrow e) \vdash e_2$$


$$\text{subject-reduction } (\vdash \cdot \vdash e_1 \vdash e_2) (\xi\text{-}\cdot_2 e_2 \hookrightarrow e x) = \vdash \cdot \vdash e_1 (\text{subject-reduction } \vdash e_2 e_2 \hookrightarrow e)$$


$$\text{subject-reduction } (\vdash \bullet (\vdash \Lambda \vdash e)) \beta\text{-}\Lambda = \text{e[\tau]-preserves } \vdash e \vdash \tau$$


$$\text{subject-reduction } (\vdash \bullet \vdash e) (\xi\text{-}\bullet e \hookrightarrow e') = \vdash \bullet (\text{subject-reduction } \vdash e e \hookrightarrow e')$$


$$\text{subject-reduction } (\vdash \text{let } \vdash e_2 \vdash e_1) (\beta\text{-let } v_2) = \text{e[e]-preserves } \vdash e_1 \vdash e_2$$


$$\text{subject-reduction } (\vdash \text{let } \vdash e_2 \vdash e_1) (\xi\text{-let } e_2 \hookrightarrow e') = \vdash \text{let}$$


$$(\text{subject-reduction } \vdash e_2 e_2 \hookrightarrow e') \vdash e_1$$


```

The  $\xi$  reduction cases  $\xi\text{-}\cdot_1$ ,  $\xi\text{-}\cdot_2$ ,  $\xi\text{-}\bullet$  and  $\xi\text{-let}$  follow directly from the induction hypothesis.

For the  $\beta$  reduction cases  $\beta\text{-}\lambda$ ,  $\beta\text{-}\Lambda$  and  $\beta\text{-let}$  we need to prove that substitutions preserve the typing. We have two different types of substitution present inside the reduction rules:  $e [ e ]$  and  $e [ \tau ]$ . Both  $\text{e[e]-preserves}$  and  $\text{e[\tau]-preserves}$  follow from a more general lemma  $\vdash\sigma\text{-preserves}$ . The lemma  $\vdash\sigma\text{-preserves}$  proves that all typed substitutions preserve typing.

```


$$\vdash\sigma\text{-preserves} : \forall \{ \sigma : \text{Sub } S_1 S_2 \} \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \}$$


$$\{ t : \text{Term } S_1 s \} \{ T : \text{Term } S_1 (\text{kind-of } s) \} \rightarrow$$


$$\sigma : \Gamma_1 \Rightarrow_s \Gamma_2 \rightarrow$$


$$\Gamma_1 \vdash t : T \rightarrow$$


$$\Gamma_2 \vdash (\text{sub } \sigma t) : (\text{sub } \sigma T)$$


```

The lemma  $\vdash\sigma\text{-preserves}$  follows by induction over the typing rules and lemmas about the interaction between renamings and substitutions<sup>1</sup>. The soundness property of System F follows as a consequence of  $\text{progress}$  and  $\text{subject-reduction}$ .

## 4 System F<sub>O</sub>

### 4.1 Specification

#### Sorts

In addition to the sorts of System F, System F<sub>O</sub> introduces two new sorts:  $\mathbf{o}_s$  for overloaded variables and  $\mathbf{c}_s$  for constraints.

```

data Sort : Ctxable → Set where
  os : Sort  $\top^C$ 
  cs : Sort  $\perp^C$ 
  - ...

```

Terms of sort  $\mathbf{o}_s$  can only be constructed using the variable constructor  $\_$ . Variables for constraints do not exist in System F<sub>O</sub> and thus  $\mathbf{c}_s$  is indexed by  $\perp^C$ .

<sup>1</sup> Because the soundness proof for System F is not the main part of this work, the explanation is rather short. The full proof along all necessary lemmas can be found on GitHub: <https://github.com/Mari-W/System-Fo/blob/main/proofs/SystemF.agda#L327>

## Syntax

We only discuss additions to the syntax of System F.

```

data Term : Sorts → Sort → 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
  λ _ ⇒ _           : 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  $e$ .

The expression `inst' o = e2 'in e1` introduces an additional instance for  $o$ . The actual meaning for the instance is given by  $e_2$ .

Constraints  $c$  can be constructed using constructor  $o : \tau$ .

A constraint  $c$  can be part of a constraint abstraction  $\lambda c \Rightarrow e$ . Constraint abstractions assume the constraint  $c$  to be valid inside the body  $e$  and result in constraint types  $[c] \Rightarrow \tau$ .

Going forward, we will use the abbreviation `Cstr S = Term S cs`.

## Renaming & Substitution

Renamings and substitutions in System F<sub>O</sub> 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<sub>O</sub>, 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  $F \blacktriangleright c$  to pick up a constraint  $c$ . Constraints give an additional meaning to a overloaded variable that is already bound. Hence index  $S$  is not modified. The `lookup` function in System F<sub>O</sub> 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  $\Gamma$  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  $\Gamma$  is the desired constraint  $c$ .

If the last item in the context is not the desired constraint  $c$ ,  $c$  must be further inside the context. The constraint can either be behind a item stored in  $\Gamma$  (`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.

## Typing

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'
  - ...
```

The rule for overloaded variables `⊢'o` says that if we can resolve the constraint  $o : \tau$  in  $\Gamma$ , then  $o$  can take on type  $\tau$ .

The rule for constraint abstraction `⊢λ` appends the constraint  $c$  to  $\Gamma$  and thus assumes  $c$  to be valid inside the body  $e$ . Constraint abstractions result in the corresponding constraint type  $[c] \Rightarrow \tau$  that lifts the constraint onto the type level.

Expressions  $e$  with constraint type  $[c] \Rightarrow \tau'$  have the constraint implicitly eliminated using the `⊢⊙` rule, given  $c$  can be resolved in  $\Gamma$ .

The rule `⊢decl` introduces a new overloaded variable  $o$  to  $e$ . To introduce  $o$  in  $\Gamma$ , we only need to store the information that  $o$  exists as overloaded variable. Thus,  $\Gamma$  is extended

by the single kind  $\star$  to denote the existence of  $o$ , similar to type variables. Analogous to the type  $\tau'$  inside the abstraction rule  $\vdash\lambda$ , the resulting type  $\tau$  is weakened to align in  $S$  with  $\Gamma$  not extended by  $\star$ , such that it can act as the resulting type of the typing. An instance for an overloaded variable  $o$  is typed using the rule  $\vdash\text{inst}$ . We extend  $\Gamma$  with constraint  $o : \tau$  inside  $e_1$ , where  $\tau$  is the type of the actual additional meaning  $e_2$ .

## 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 _ :  $\Rightarrow_r$  : Ren  $S_1 S_2 \rightarrow$  Ctx  $S_1 \rightarrow$  Ctx  $S_2 \rightarrow$  Set where
   $\vdash\text{drop-cstr}_r : \forall \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \} \{ \tau \} \{ o \} \rightarrow$ 
     $\rho : \Gamma_1 \Rightarrow_r \Gamma_2 \rightarrow$ 
     $\rho : \Gamma_1 \Rightarrow_r (\Gamma_2 \blacktriangleright (o : \tau))$ 
  - ...
```

Constraint  $o : \tau$  can be introduced only to  $\Gamma_2$  using the  $\vdash\text{drop-cstr}_r$  constructor. Dropping a constraint corresponds to a typed weakening similar to  $\vdash\text{drop}_r$  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. This restriction simplifies proofs for the type preservation of the Dictionary Passing Transform.

```
data _ :  $\Rightarrow_s$  : Sub  $S_1 S_2 \rightarrow$  Ctx  $S_1 \rightarrow$  Ctx  $S_2 \rightarrow$  Set where
   $\vdash\text{type}_s : \forall \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \} \{ \tau : \text{Type } S_2 \} \rightarrow$ 
     $\sigma : \Gamma_1 \Rightarrow_s \Gamma_2 \rightarrow$ 
     $\text{single-type}_s \sigma \tau : \Gamma_1 \blacktriangleright \star \Rightarrow_s \Gamma_2$ 
  - ...
```

The constructor  $\vdash\text{type}_s$  allows to introduce an additional new type variable binder that is substituted with type  $\tau$ . Thus,  $\vdash\text{type}_s$  complements the  $\text{single-type}_s$  function. The intuition here is that if we would allow all terms to be introduced using a  $\vdash\text{term}_s$  constructor, then typed substitutions in System  $F_O$  would be arbitrary again. Constructors  $\vdash\text{ext}_s$ ,  $\vdash\text{drop}_s$  and  $\vdash\text{drop-cstr}_s$  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 contextable. The two missing non-contextable sorts  $c_s$  and  $\kappa_s$  do not need to be translated. Intuitively there does not even exist a sensible translation for  $c_s$ .

```
 $\rightsquigarrow_s : F^O.\text{Sort } T^C \rightarrow F.\text{Sort } T^C$ 
 $\rightsquigarrow_s e_s = e_s$ 
```



$$\begin{aligned} \textcolor{blue}{s} \rightsquigarrow \textcolor{blue}{S} \text{ } o_s &= e_s \\ \textcolor{blue}{s} \rightsquigarrow \textcolor{blue}{S} \text{ } \tau_s &= \tau_s \end{aligned}$$

Sorts  $e_s$  and  $\tau_s$  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 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 = e2 'in e1` expression does not bind a new variable in  $e_1$ , but translates to a `let 'x = e2 'in e1` 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  $\Gamma$  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  $F_O$ .

$$\begin{aligned} \Gamma \rightsquigarrow S &: F^O.\text{Ctx } F^O.S \rightarrow F.\text{Sorts} \\ \Gamma \rightsquigarrow S \text{ } \emptyset &= [] \\ \Gamma \rightsquigarrow S (\Gamma \triangleright c) &= \Gamma \rightsquigarrow S \Gamma \triangleright F.e_s \\ \Gamma \rightsquigarrow S \{S \triangleright s\} (\Gamma \triangleright x) &= \Gamma \rightsquigarrow S \Gamma \triangleright \textcolor{blue}{s} \rightsquigarrow \textcolor{blue}{S} s \end{aligned}$$

The empty context  $\emptyset$  corresponds to the empty list  $[]$ .

For each constraint in  $\Gamma$  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 to  $\textcolor{blue}{s} \rightsquigarrow \textcolor{blue}{S} s$ .

## Variables

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

$$\begin{aligned} x \rightsquigarrow x &: \forall \{ \Gamma : F^O.\text{Ctx } F^O.S \} \rightarrow \\ &F^O.\text{Var } F^O.S \text{ } F^O.s \rightarrow F.\text{Var } (\Gamma \rightsquigarrow S \Gamma) (\textcolor{blue}{s} \rightsquigarrow \textcolor{blue}{S} F^O.s) \\ x \rightsquigarrow x \{ \Gamma = \Gamma \triangleright \tau \} &(\text{here refl}) = \text{here refl} \\ x \rightsquigarrow x \{ \Gamma = \Gamma \triangleright \tau \} &(\text{there } x) = \text{there } (x \rightsquigarrow x) \\ x \rightsquigarrow x \{ \Gamma = \Gamma \triangleright c \} &x = \text{there } (x \rightsquigarrow x) \end{aligned}$$

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. We can apply the same translation as seen in the function  $x \rightsquigarrow x$  because the type for resolved constraints  $[c] \in \Gamma$  preserves the structure of the context along its constraints.

$$\begin{aligned} o:\tau \in \Gamma \rightsquigarrow x &: \forall \{ \Gamma : F^O.\text{Ctx } F^O.S \} \rightarrow \\ &[ \text{ } F^O.o : F^O.\tau ] \in \Gamma \rightarrow F.\text{Var } (\Gamma \rightsquigarrow S \Gamma) F.e_s \\ o:\tau \in \Gamma \rightsquigarrow x \text{ here} &= \text{here refl} \\ o:\tau \in \Gamma \rightsquigarrow x (\text{under-bind } o:\tau \in \Gamma) &= \text{there } (o:\tau \in \Gamma \rightsquigarrow x \text{ } o:\tau \in \Gamma) \\ o:\tau \in \Gamma \rightsquigarrow x (\text{under-cstr } o:\tau \in \Gamma) &= \text{there } (o:\tau \in \Gamma \rightsquigarrow x \text{ } o:\tau \in \Gamma) \end{aligned}$$

Inside the base base case we found the correct instance, now variable. In the induction case `under-cstr` we again wrap the applied induction hypothesis in an additional `there`.

## Context

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

$$\begin{aligned} \Gamma \rightsquigarrow \Gamma &: (\Gamma : \mathbf{F}^O.\text{Ctx } F^O.S) \rightarrow \mathbf{F}.\text{Ctx } (\Gamma \rightsquigarrow S \Gamma) \\ \Gamma \rightsquigarrow \Gamma (I \triangleright ('o : \tau)) &= (\Gamma \rightsquigarrow \Gamma I) \triangleright \tau \rightsquigarrow \tau \tau \\ - \dots \end{aligned}$$

Following the idea from above, constraints  $o : \tau$  stored inside  $\Gamma$  translate to normal items in the translated  $\Gamma$ . The item introduced is the translated type  $\tau \rightsquigarrow \tau \tau$  that was originally required by the constraint. Again, for each constraint in System  $\mathbf{F}_O$  there will be a new binder in System  $\mathbf{F}$  that accepts the constraint as higher order function. Thus, the corresponding function type for that binding is expected in  $\Gamma$  at that position.

## Renaming & Substitution

Typed renamings in System  $\mathbf{F}_O$  translate to untyped renamings in System  $\mathbf{F}$ .

$$\begin{aligned} \vdash \rho \rightsquigarrow \rho &: \forall \{\rho : \mathbf{F}^O.\text{Ren } F^O.S_1 F^O.S_2\} \{\Gamma_1 : \mathbf{F}^O.\text{Ctx } F^O.S_1\} \{\Gamma_2 : \mathbf{F}^O.\text{Ctx } F^O.S_2\} \rightarrow \\ &\rho F^O : \Gamma_1 \Rightarrow_r \Gamma_2 \rightarrow \\ &\mathbf{F}.\text{Ren } (\Gamma \rightsquigarrow S \Gamma_1) (\Gamma \rightsquigarrow S \Gamma_2) \\ \vdash \rho \rightsquigarrow \rho (\vdash \text{drop-ctr}_r \vdash \rho) &= \mathbf{F}.\text{drop}_r (\vdash \rho \rightsquigarrow \rho \vdash \rho) \\ - \dots \end{aligned}$$

Because constraints in contexts translate to actual bindings, the constructor  $\vdash \text{drop-ctr}_r$  translates to  $\text{drop}_r$  in System  $\mathbf{F}$ .

Typed renamings  $\vdash \text{id}_r$ ,  $\vdash \text{ext}_r$  and  $\vdash \text{drop}_r$  translate to their untyped counterparts.

The translation of typed substitutions to untyped substitutions follows similarly.

$$\begin{aligned} \vdash \sigma \rightsquigarrow \sigma &: \forall \{\sigma : \mathbf{F}^O.\text{Sub } F^O.S_1 F^O.S_2\} \{\Gamma_1 : \mathbf{F}^O.\text{Ctx } F^O.S_1\} \{\Gamma_2 : \mathbf{F}^O.\text{Ctx } F^O.S_2\} \rightarrow \\ &\sigma F^O : \Gamma_1 \Rightarrow_s \Gamma_2 \rightarrow \\ &\mathbf{F}.\text{Sub } (\Gamma \rightsquigarrow S \Gamma_1) (\Gamma \rightsquigarrow S \Gamma_2) \\ \vdash \sigma \rightsquigarrow \sigma (\vdash \text{type}_s \{\tau = \tau'\} \vdash \sigma) &= \mathbf{F}.\text{single}_s (\vdash \sigma \rightsquigarrow \sigma \vdash \sigma) (\tau \rightsquigarrow \tau \tau) \\ - \dots \end{aligned}$$

The typed renaming  $\vdash \text{type}_s$  translates to its untyped counterpart for arbitrary terms  $\text{single}_s$ .

The cases  $\vdash \text{id}_s$ ,  $\vdash \text{ext}_s$ ,  $\vdash \text{drop}_s$  and  $\vdash \text{drop-ctr}_s$  are analogous to the cases for renamings.

## Terms

Types and kinds can be translated without typing information. Kind  $\star$  translates to its direct counterpart in System  $\mathbf{F}$ . Furthermore, all System  $\mathbf{F}_O$  types translate to their direct counterpart in System  $\mathbf{F}$ , except the constraint type  $[o : \tau] \Rightarrow \tau'$ .

$$\begin{aligned} \tau \rightsquigarrow \tau &: \forall \{\Gamma : \mathbf{F}^O.\text{Ctx } F^O.S\} \rightarrow \\ &\mathbf{F}^O.\text{Type } F^O.S \rightarrow \\ &\mathbf{F}.\text{Type } (\Gamma \rightsquigarrow S \Gamma) \\ \tau \rightsquigarrow \tau ([o : \tau] \Rightarrow \tau') &= \tau \rightsquigarrow \tau \tau \Rightarrow \tau \rightsquigarrow \tau \tau' \\ - \dots \end{aligned}$$

Constraint types  $[o : \tau] \Rightarrow \tau'$  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 of type  $\tau$ .

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.

$$\begin{aligned}
\vdash_{\rightsquigarrow} t : T &: \forall \{ \Gamma : F^O.\text{Ctx } F^O.S \} \{ t : F^O.\text{Term } F^O.S \ F^O.s \} \\
&\{ T : F^O.\text{Term } F^O.S \ (F^O.\text{kind-of } F^O.s) \} \rightarrow \\
&\Gamma F^O.\vdash t : T \rightarrow \\
&F.\text{Term } (\Gamma \rightsquigarrow S \Gamma) (s \rightsquigarrow F^O.s) \\
\vdash_{\rightsquigarrow} (\vdash' o \ o : \tau \in \Gamma) &= \vdash' o : \tau \in \Gamma \rightsquigarrow x \ o : \tau \in \Gamma \\
\vdash_{\rightsquigarrow} (\vdash \lambda \vdash e) &= \lambda' x \rightarrow (\vdash_{\rightsquigarrow} t \vdash e) \\
\vdash_{\rightsquigarrow} (\vdash \odot \vdash e \ o : \tau \in \Gamma) &= \vdash_{\rightsquigarrow} t \vdash e \cdot \vdash' o : \tau \in \Gamma \rightsquigarrow x \ o : \tau \in \Gamma \\
\vdash_{\rightsquigarrow} (\vdash \text{decl} \vdash e) &= \text{let}' x = \text{tt} \text{ in } \vdash_{\rightsquigarrow} t \vdash e \\
\vdash_{\rightsquigarrow} (\vdash \text{inst} \vdash e_2 \vdash e_1) &= \text{let}' x = \vdash_{\rightsquigarrow} t \vdash e_2 \text{ in } \vdash_{\rightsquigarrow} t \vdash e_1 \\
&- \dots
\end{aligned}$$

Typed overloaded variables  $\vdash' o$  carry information about the instance that was resolved for  $o$ . We translate the resolved instance to the unique variable in System F using the  $o : \tau \in \Gamma \rightsquigarrow x$  function.

Constraint abstractions translate to normal abstractions.

An implicitly resolved constraint translates to a explicit application that passes the resolved instance as argument.

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  $F_O$ , 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  $F_O$ . The function  $\vdash_{\rightsquigarrow} t \vdash t$  produces a typed System F term for an arbitrary typed System  $F_O$  term  $\vdash t$ . The untyped translated System  $F_O$  term  $\vdash_{\rightsquigarrow} t$  gets typed in the translated context  $\Gamma \rightsquigarrow \Gamma \Gamma$  and has the typing result  $\vdash_{\rightsquigarrow} T \vdash T$ . Untyped types and kinds translate from System  $F_O$  to System F using function  $\vdash_{\rightsquigarrow} T \vdash T$ .

$$\begin{aligned}
\vdash_{\rightsquigarrow} t \vdash t &: \{ \Gamma : F^O.\text{Ctx } F^O.S \} \{ t : F^O.\text{Term } F^O.S \ F^O.s \} \\
&\{ T : F^O.\text{Term } F^O.S \ (F^O.\text{kind-of } F^O.s) \} \rightarrow \\
&(\vdash t : \Gamma F^O.\vdash t : T) \rightarrow \\
&(\Gamma \rightsquigarrow \Gamma \Gamma) F.\vdash (\vdash_{\rightsquigarrow} t \vdash t) : (\vdash_{\rightsquigarrow} T \vdash T) \\
\vdash_{\rightsquigarrow} t \vdash t \ (\vdash' x \ \{ x = x \} \ \Gamma x \equiv \tau) &= \vdash' x \ (\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau \ x \ \Gamma x \equiv \tau)
\end{aligned}$$

$$\begin{aligned}
& \vdash_{\sim} \text{let } o : \tau \in \Gamma \vdash e_1 = \vdash_{\sim} x \text{ (} o : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau \text{ } o : \tau \in \Gamma \text{)} \\
& \vdash_{\sim} \text{let } \vdash_{e_2} \vdash_{e_1} = \vdash_{\sim} \text{let } (\vdash_{\sim} \vdash_{e_2}) \\
& \text{(subst } (\_ F. \vdash_{\sim} \vdash_{e_1} : \_) \tau \rightsquigarrow \text{wk} \cdot \tau \equiv \text{wk} \cdot \tau \rightsquigarrow \tau \text{ (} \vdash_{\sim} \vdash_{e_1} \text{)}) \\
& \vdash_{\sim} \text{let } (\vdash_{\sim} \{c = (\_ o : \tau)\} \vdash e) = \vdash_{\sim} \lambda \\
& \text{(subst } (\_ F. \vdash_{\sim} \vdash_{e_1} \vdash e : \_) \tau \rightsquigarrow \text{wk} \cdot \tau \equiv \text{wk} \cdot \text{inst} \cdot \tau \rightsquigarrow \tau \text{ (} \vdash_{\sim} \vdash_{e_1} \text{)}) \\
& \vdash_{\sim} \text{let } (\vdash_{\sim} \vdash_{e_2} \vdash_{e_1} o : \tau \in \Gamma) = \vdash_{\sim} (\vdash_{\sim} \vdash_{e_2}) (\vdash_{\sim} x \text{ (} o : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau \text{ } o : \tau \in \Gamma \text{)}) \\
& \vdash_{\sim} \text{let } (\vdash_{\sim} \{ \tau = \tau' \} \{ \tau' = \tau' \} \vdash e) = \text{subst } (\_ F. \vdash_{\sim} \vdash_{e_1} \vdash e \bullet \tau \rightsquigarrow \tau' : \_) \\
& \text{(} \tau' \rightsquigarrow \tau' [\tau \rightsquigarrow \tau] \equiv \tau \rightsquigarrow \tau' [\tau] \text{ } \tau' \text{ } \tau \text{) (} \vdash_{\sim} \vdash_{e_1} \text{)} \\
& - \dots
\end{aligned}$$

Proof  $\Gamma x \equiv \tau$  that a variable  $x$  has type  $\tau$  in  $\Gamma$  translates to proof that  $x \rightsquigarrow x$  has type  $\tau \rightsquigarrow \tau$  in  $\Gamma \rightsquigarrow \Gamma$  using lemma  $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau$ . With the lemma  $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau$  the typing rule  $\vdash_{\sim} x$  can be translated to the rule for variables in System F.

Similarly, lemma  $o : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau$  translates the proof that an instance  $o : \tau$  was resolved for an overloaded variable  $o$  to proof that unique variable  $o : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau$  has type  $\tau \rightsquigarrow \tau$  in  $\Gamma \rightsquigarrow \Gamma$ . Using lemma  $o : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau$  the typing rule for overloaded variables  $\vdash_{\sim} o$  can be translated to the typing rule for normal variables  $\vdash_{\sim} x$ .

Typed let bindings  $\vdash_{\sim} \text{let } \vdash_{e_2} \vdash_{e_1}$  translate to typed let bindings in System F. The rule  $\vdash_{e_2}$  is translated directly using the induction hypothesis. Because the typing for  $e_1$  in  $\vdash_{e_1}$  results in  $\text{wk } \tau'$ , proof is needed that  $\tau'$  weakened in System  $F_O$  and translated to System F is equivalent to the weakening of the translated  $\tau'$  in System F. Lemma  $\tau \rightsquigarrow \text{wk} \cdot \tau \equiv \text{wk} \cdot \tau \rightsquigarrow \tau$  is used to substitute the required equivalence into the translated typing rule  $\vdash_{\sim} \text{let } \vdash_{e_2} \vdash_{e_1}$ .

Typed constraint abstractions  $\vdash_{\sim} \lambda$  translate to normal abstractions in System F. Inside the typing for  $\vdash_{\sim} e$ , the result type  $\tau$  for  $e$  does not need to be weakened because the constraint abstraction only introduced a constraint to context  $\Gamma$  and no actual binding. After the translation the former constraint will be bound by a binding and thus a new item in  $\Gamma \rightsquigarrow \Gamma$  will exist. To ignore the binding  $\tau$  is weakened in the abstraction rule  $\vdash_{\sim} \lambda$ . Lemma  $\tau \rightsquigarrow \text{wk} \cdot \tau \equiv \text{wk} \cdot \text{inst} \cdot \tau \rightsquigarrow \tau$  proves that translating  $\tau$  in  $\Gamma$  extended by a constraint is equivalent to weakening  $\tau$  after the translation. The lemma follows because in the first case, the constraint translates to an actual binding and consequently, both sides have an additional unnecessary expression binding that  $\tau$  cannot use.

Implicitly resolved constraints  $\vdash_{\sim} \odot$  carry the information about the instance that was resolved. In System F the former constraint is now explicitly passed as variable pointing to the correct translated instance. Thus,  $\vdash_{\sim} \odot$  results in typed application  $\vdash_{\sim} \cdot$ . We apply the correct instance using lemma  $o : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau$  to get the correct unique variable for the resolved constraint.

The Type application rule  $\vdash_{\sim} \bullet$  contains type in type substitution. Hence, we need proof that it is irrelevant, if  $\tau'$  is substituted into  $\tau$  and then translated or both  $\tau$  and  $\tau'$  are translated and substituted in System F. Using lemma  $\tau' \rightsquigarrow \tau' [\tau \rightsquigarrow \tau] \equiv \tau \rightsquigarrow \tau' [\tau]$  we can substitute the equivalence into the System F typing rule  $\vdash_{\sim} \bullet$  ( $\vdash_{\sim} \vdash_{e_1} \vdash_{e_2}$ ).

The translation of  $\vdash_{\sim} \top$ ,  $\vdash_{\sim} \lambda$ ,  $\vdash_{\sim} \cdot$ ,  $\vdash_{\sim} \text{decl}$  and  $\vdash_{\sim} \text{inst}$  is either a direct translation or does not use other lemmas than the ones discussed.

## Renaming

Both  $\tau \rightsquigarrow \text{wk} \cdot \tau \equiv \text{wk} \cdot \tau \rightsquigarrow \tau$  and  $\tau \rightsquigarrow \text{wk} \cdot \tau \equiv \text{wk} \cdot \text{inst} \cdot \tau \rightsquigarrow \tau$  directly follow from a more general lemma  $\vdash_{\sim} \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau$  for arbitrary renamings. The lemma  $\vdash_{\sim} \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau$  proves that translating both the typed renaming  $\vdash_{\sim} \rho$  and type  $\tau$  and then applying the

renaming in System F is equivalent to applying the renaming  $\rho$  in System  $F_O$  and then translating renamed  $\tau$ . The lemma can be proven by induction over System  $F_O$  types  $\tau$ .

$$\begin{aligned}
& \vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau : \{ \rho : F^O.\text{Ren } F^O.S_1 F^O.S_2 \} \\
& \quad \{ \Gamma_1 : F^O.\text{Ctx } F^O.S_1 \} \{ \Gamma_2 : F^O.\text{Ctx } F^O.S_2 \} \rightarrow \\
& \quad (\vdash \rho : \rho F^O. : \Gamma_1 \Rightarrow_r \Gamma_2) \rightarrow \\
& \quad (\tau : F^O.\text{Type } F^O.S_1) \rightarrow \\
& \quad F.\text{ren } (\vdash \rho \rightsquigarrow \rho \vdash \rho) (\tau \rightsquigarrow \tau) \equiv \tau \rightsquigarrow \tau (F^O.\text{ren } \rho \tau) \\
& \quad \vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau \vdash \rho ('x) = \text{cong } ' \_ (\vdash \rho \rightsquigarrow \rho \cdot x \rightsquigarrow x \equiv x \rightsquigarrow \rho \cdot x \vdash \rho x) \\
& \quad \vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau \vdash \rho ([ ' o : \tau ] \Rightarrow \tau') = \text{cong}_2 \_ \Rightarrow \_ \\
& \quad (\vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau \vdash \rho \tau) (\vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau \vdash \rho \tau') \\
& \quad - \dots
\end{aligned}$$

The case for type variables needs an additional lemma  $\vdash \rho \rightsquigarrow \rho \cdot x \rightsquigarrow x \equiv x \rightsquigarrow \rho \cdot x$  specifically for type variables. Lemma  $\vdash \rho \rightsquigarrow \rho \cdot x \rightsquigarrow x \equiv x \rightsquigarrow \rho \cdot x$  proves an analogous statement, but for type variables applied to a renaming:  $(\vdash \rho \rightsquigarrow \rho \vdash \rho) (x \rightsquigarrow x) \equiv x \rightsquigarrow x (\rho x)$ . This statement can be proven via straight forward induction over typed System  $F_O$  renamings  $\vdash \rho$ . 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  $\tau' \rightsquigarrow \tau'[\tau \rightsquigarrow \tau] \equiv \tau \rightsquigarrow \tau'[\tau]$  follows from a more general lemma about substitutions:  $\tau' \rightsquigarrow \tau'[\tau \rightsquigarrow \tau] \equiv \tau \rightsquigarrow \tau'[\tau] \tau \tau' = \vdash \sigma \rightsquigarrow \sigma \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \sigma \cdot \tau F^O.\vdash \text{single-type}_s \tau'$ . The more general lemma  $\vdash \sigma \rightsquigarrow \sigma \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \sigma \cdot \tau$  also follows by straight forward induction over System  $F_O$  types, except the case for type variables. Other than with renamings, the cases for lemma  $\vdash \sigma \rightsquigarrow \sigma \cdot x \rightsquigarrow x \equiv \tau \rightsquigarrow \sigma \cdot x$  do not follow directly from the induction hypothesis. To understand why, we at look at the case  $\vdash \text{ext}_s$ .

$$\begin{aligned}
& \vdash \sigma \rightsquigarrow \sigma \cdot x \rightsquigarrow x \equiv \tau \rightsquigarrow \sigma \cdot x : \{ \sigma : F^O.\text{Sub } F^O.S_1 F^O.S_2 \} \\
& \quad \{ \Gamma_1 : F^O.\text{Ctx } F^O.S_1 \} \{ \Gamma_2 : F^O.\text{Ctx } F^O.S_2 \} \rightarrow \\
& \quad (\vdash \sigma : \sigma F^O. : \Gamma_1 \Rightarrow_s \Gamma_2) \rightarrow \\
& \quad (x : F^O.\text{Var } F^O.S_1 \tau_s) \rightarrow \\
& \quad F.\text{sub } (\vdash \sigma \rightsquigarrow \sigma \vdash \sigma) ('x \rightsquigarrow x) \equiv \tau \rightsquigarrow \tau (F^O.\text{sub } \sigma ('x)) \\
& \quad \vdash \sigma \rightsquigarrow \sigma \cdot x \rightsquigarrow x \equiv \tau \rightsquigarrow \sigma \cdot x (\vdash \text{ext}_s \vdash \sigma) (\text{here refl}) = \text{refl} \\
& \quad \vdash \sigma \rightsquigarrow \sigma \cdot x \rightsquigarrow x \equiv \tau \rightsquigarrow \sigma \cdot x (\vdash \text{ext}_s \{ \sigma = \sigma \} \vdash \sigma) (\text{there } x) = \text{trans} \\
& \quad (\text{cong } F.\text{wk } (\vdash \sigma \rightsquigarrow \sigma \cdot x \rightsquigarrow x \equiv \tau \rightsquigarrow \sigma \cdot x \vdash \sigma x)) (\vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau F^O.\vdash \text{wk}_r (\sigma x))
\end{aligned}$$

The case  $\vdash \text{ext}_s$  is proven via induction over variable  $x$ , similar to how  $\text{ext}_s$  is defined. The base case holds by definition. In the induction case we use the weakening of the applied 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  $\vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau$  because  $\text{ext}_s$  is defined by weakening the result of the substitution  $\sigma$  applied to the variable  $x$ .

Both  $\vdash \text{id}_s$  and  $\vdash \text{type}_s$  follow directly from the induction hypothesis. The cases for  $\vdash \text{drop}_s$ ,  $\vdash \text{drop-cstr}_s$  and  $\vdash \text{ext-cstr}_s$  are similar to  $\vdash \text{ext}_s$ .

## Variables

We first look at the proof for lemma  $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau$ . Lemma  $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau$  is proven via induction over the System  $F_O$  context  $\Gamma$ .

```

 $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau : \forall \{ \Gamma : F^O.Ctx \ F^O.S \} \{ \tau : F^O.Type \ F^O.S \} (x : F^O.Var \ F^O.S \ e_s) \rightarrow$ 
   $F^O.lookup \ \Gamma \ x \equiv \tau \rightarrow$ 
   $F.lookup \ (\Gamma \rightsquigarrow \Gamma \ \Gamma) \ (x \rightsquigarrow x \ x) \equiv (\tau \rightsquigarrow \tau \ \tau)$ 
 $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau \ \{ \Gamma = \Gamma \triangleright \tau \} \text{ (here refl) } \text{ refl} = \vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau \ F^O.\vdash wk_r \ \tau$ 
 $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau \ \{ \Gamma = \Gamma \triangleright \_ \} \{ \tau' \} \text{ (there } x \text{) } \text{ refl} = \text{trans}$ 
   $(\text{cong } F.wk \ (\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau \ x \ \text{refl}))$ 
   $(\vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau \ F^O.\vdash wk_r \ (F^O.lookup \ \Gamma \ x))$ 
  - ...

```

As an example we will look at case  $\Gamma \triangleright \tau$ . The case is proven via induction over variables  $x$ . The prove follows the same reasoning as the  $\vdash ext_s$  case for substitutions above. Because the function `lookup` weakens the looked up type  $\tau$  in both the base case and induction step, both use lemma  $\vdash \rho \rightsquigarrow \rho \cdot \tau \rightsquigarrow \tau \equiv \tau \rightsquigarrow \rho \cdot \tau$  applied to the typed weakening and  $\tau$ .

The case  $\Gamma \triangleright c$  is a little more complicated but uses similar concepts. Additional complexity arises because we need to deal with the fact that constraints were ignored by the `lookup` method in System  $F_O$  but then they are translated to actual context items in System  $F$ .

Lemma  $\mathbf{o} : \tau \in \Gamma \rightsquigarrow \Gamma x \equiv \tau$  can proven via induction over the type for resolved constraints  $[c] \in \Gamma$ . The proof is analogous to the proof shown for  $\Gamma x \equiv \tau \rightsquigarrow \Gamma x \equiv \tau$  because the type for resolved constraints preserves the structure of context  $\Gamma$ .

This finishes up the type preservation proof for the Dictionary Passing Transform from System  $F_O$  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 [7] based system  $HM_O$  and the target language would be Hindley-Milner. Hindley-Milner is a restricted form of System  $F$ . 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  $\tau_s$ . 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 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 arbitrary constraints using type variables still could be resolved deterministically by the inference algorithm.

## 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 ' $o \cdot e_1 \dots e_n$ ' need type information to reduce properly. The correct instance for  $o$  needs to be resolved based on the types of arguments  $e_1 \dots e_n$ . More specifically, to formalize small step semantics we would need to apply the restriction mentioned above and restrict all instances for an overloaded variable  $o$  to 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  $\exists [e''] (\vdash e \hookrightarrow e' \rightsquigarrow \vdash e' \hookrightarrow e'' \times (\vdash e \hookrightarrow e' \rightsquigarrow \vdash e' \hookrightarrow e'') \times (\vdash e \hookrightarrow e' \rightsquigarrow \vdash e' \hookrightarrow e''))$ , where  $\vdash e \hookrightarrow e' \rightsquigarrow \vdash e' \hookrightarrow e''$  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 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 [8]. System O is a language extension to the Hindley-Milner System. In contrast to System  $F_O$ , 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  $HM_O$  type  $\forall \alpha. \forall \beta. [\text{add} : \alpha \rightarrow \alpha \rightarrow \alpha] \Rightarrow [\text{mul} : \beta \rightarrow \beta \rightarrow \beta] \Rightarrow [\alpha] \rightarrow [\beta] \rightarrow (\alpha, \beta)$  would be  $\forall \alpha. (\text{add} : \alpha \rightarrow \alpha \rightarrow \alpha) \Rightarrow \forall \beta. (\text{mul} : \beta \rightarrow \beta \rightarrow \beta) \Rightarrow [\alpha] \rightarrow [\beta] \rightarrow (\alpha, \beta)$  in System O. Here we only introduce one constraint per type variable, but a list of constraints is allowed. To preserve the type inference in System O, all constraints must have the type variable that they are tied to as the first type of the function type they require. 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 in one induction step. Furthermore, the translation of the System O type above must pull out forall types, because translating the constraints directly to higher order functions would break the rule that function types are only allowed to be built from mono types. Thus the translated System O type should not be  $\forall \alpha. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \forall \beta. (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \dots$  but rather  $\forall \alpha. \forall \beta. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \dots$ . 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 [9] [4]. A more general approach to constraint types is presented by the theory of qualified types [5].

## 6.4 Conclusion

We have formalized both System F and System  $F_O$  in Agda. System  $F_O$  acts as a core calculus that 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 from System  $F_O$  to System F 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 <sup>2</sup>. A reasonable next step would be to prove semantic preservation for the Dictionary Passing Transform.

---

<sup>2</sup> 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](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. 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](https://doi.org/10.1007/3-540-55253-7_17)
6. Martin-Löf, P. & Sambin, G. *Intuitionistic Type Theory*. (Bibliopolis, 1984), <https://www.cse.chalmers.se/~peterd/papers/MartinL%C3%B6f1984.pdf>
7. 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>
8. 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>
9. 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