



# 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 and polymorphism. Furthermore, we prove the Dictionary Passing Transform from System  $F_O$  to System F to be type preserving using Agda.

**Abstract. (german)** Viele stark getypte Programmiersprachen unterstützen das Überladen von Funktionsnamen. In Kombination mit Polymorphismus ergeben sich essenzielle Sprachkonstrukte, beispielsweise Typklassen in Haskell, oder Traits in Rust. Wir erarbeiten System  $F_O$ , eine minimale Spracherweiterung von System F, die Überladung und Polymorphismus unterstützt. Anschließend beweisen wir formal in Agda, dass die Dictionary Passing Transform von System  $F_O$  zu System F typerhaltend ist.

# 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 $F$ .....	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 $F_O$ .....	15
4.1	Specification .....	15
5	The Dictionary Passing Transform .....	19
5.1	Translation .....	19
5.2	Type Preservation .....	22
6	Discussion and Conclusion .....	25
6.1	Hindley Milner with Overloading .....	25
6.2	Proving Semantic Preservation .....	26
6.3	Related Work .....	26
6.4	Conclusion .....	27

## 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. 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 function only defines abstract behavior that must work 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 for 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 type variables via type constraints to only be substituted by a concrete type, if there exists an instance for the concrete type. Type constraints allow to abstract over all types that inherit a shared behavior, but the actual implementation of the behavior can differ per type. Type constraints are a powerful formalism in addition to parametric polymorphism.

#### Example: Overloading Equality in Haskell

In this example the function `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 `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 `eq :  $\beta \rightarrow \beta \rightarrow \text{Bool}$` . The intuition here is that we want to be able to compare natural numbers `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 `Eq` is declared with a single generic function signature `eq :: α → α → Bool`. We then instantiate `Eq` for  $\{\alpha \mapsto \text{Nat}\}$ . After that, `Eq` is instantiated for  $\{\alpha \mapsto [\beta]\}$ , given that an instance `Eq β` can be resolved for some concrete type  $\beta$ . As a result, we can invoke `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  $F$ . System  $F$  is also sometimes referred to as polymorphic lambda calculus or second-order lambda calculus. 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 the type  $\tau$  of the 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  $\lambda (o : \tau). e'$ . A constraint `o : τ` requires the type system to search for an instance for the overloaded variable `o` that has type  $\tau$ . If the constraint cannot be resolved, then the program is invalid. Constraint abstractions result in constraint types  $[o : \tau] \Rightarrow \tau'$  that lift constraints onto the type level. We introduce constraints on the expression level because instance expressions do not have an explicit type annotation in System  $F_O$ . All 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 : 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. λ(eq : β → β → Bool). λxs. λys. .. in

.. eq 42 0 .. eq Nat [42, 0] [42, 0] ..

```

We first 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 is satisfied. Because System  $F_O$  is based on System F, a calculus without type inference, we are required to bind type variables using type abstractions `Λα. ..` 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. This implies that System  $F_O$  is no more expressive or powerful than System F. After all, overloading is more of a convenience feature.

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 with.

The translation removes all `decl` 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 `τ` of the expression `e`. Implicitly resolved constraints in System  $F_O$  will be taken as explicit higher-order function arguments in System F. As a result, constraint abstractions `λ (o : τ). e'` translate to normal abstractions `λoτ. e'` and constraint types `[o : τ] ⇒ τ'` translate to function types `τ → τ'`. A invocation of an overloaded function name `o` translates to the correct unique variable name `oτ`, that is bound by the let binding that got introduced for the corresponding resolved instance. 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 unique names `oτ`. Applying the Dictionary Passing Transform to the example above results in a well typed System F expression. Type annotations for let bindings are given for convenience.

```

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 remove the `decl` expression and transform both `inst` expressions to `let` bindings with unique names `eqi`. 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 an explicit higher-order function argument. Invocations of `eq` translate to correct unique variables `eqi` that are bound by the `let` bindings that got introduced for the former resolved instances. Because `eq2` is invoked for a list of numbers, we must pass the correct instance to eliminate the new 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 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 that 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 the syntaxes of System  $F$  and System  $F_O$  in Agda we use a single data type `Term` indexed by sorts  $s$ . Sorts distinguish between different categories of terms. For example, the sort `es` represents expressions  $e$ , `τs` represents types  $τ$  and `κs` represents kinds. In System  $F$  and System  $F_O$  there only exists a single kind  $*$ . The name 'sort' originates from the theory of pure type systems [2], but neither System  $F$  nor System  $F_O$  allow any interesting dependencies between terms of sort `es`, `τs`, and `κs`. Using a single data type to formalize syntaxes yields more elegant proofs involving contexts, renamings and substitutions. In consequence of using a single data type, we must use extrinsic typing because intrinsically typed terms 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  $\Gamma$  is a typing

context of type  $\text{Ctx}_{F_O}$ ,  $t$  is a  $\text{Term}_{F_O} s$ ,  $T$  is a  $\text{Term}_{F_O} s'$  and  $s'$  is the sort of the typing result for terms of the sort  $s$ . There are 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 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 a given System  $F_O$  program. And furthermore, the semantics of System F are proven to be sound in combination with the type system that System  $F_O$  is safely translated to.

### 3 System F

#### 3.1 Specification

##### Sorts

The formalization of System F requires three sorts:  $\mathbf{e}_s$  for expressions,  $\mathbf{\tau}_s$  for types and  $\mathbf{\kappa}_s$  for kinds.

```
data Sort : Bindable → Set where
  e_s : Sort var
  τ_s : Sort var
  κ_s : Sort no-var
```

Sorts are indexed by the boolish data type `Bindable`. The index `var` indicates that variables for terms of a sort can be bound. In contrast, the index `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  $r$  for elements of type `Bindable`, the variable  $s$  for elements of type `Sort` 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. 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 than 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, 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 a list  $S$  is the element we searched for. Alternatively, membership proofs for a list  $S$  can be constructed via the constructor `there x`, where  $x$  is a membership proof for the sublist  $S'$  of  $S$  that has one element less. As discussed, the Debruijn representation of variables has the advantage that only already bound variables can be referenced by the variable constructor by definition.

The unit element `tt` and type `'T` represent base expressions and types respectively.

Lambda abstractions  $\lambda'x \rightarrow e$  result in function types  $\tau_1 \Rightarrow \tau_2$  and type abstractions  $\Lambda' \alpha \rightarrow e$  result in forall types  $\forall' \alpha \tau$ . Both abstractions and forall types introduce an additional sort  $e_s$ , or  $\tau_s$  respectively, to the index  $S$  of their corresponding body to account for the additional new binding.

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  $\text{Var } S \ s = s \in S$ ,  $\text{Expr } S = \text{Term } S \ e_s$ ,  $\text{Type } S = \text{Term } S \ \tau_s$  and variables  $x$ ,  $e$  and  $\tau$  respectively. Furthermore, we use the variable  $t$  for an arbitrary  $\text{Term } S \ s$ .

## Renaming

Renamings  $\rho$  of type  $\text{Ren } S_1 \ S_2$  are defined as total functions that map 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 represented as references to elements in  $S_2$  instead of  $S_1$ . The function  $\text{ren}$  applies a renaming to a term.

```

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
- . . .

```

In the first case, the renaming is applied to all variables  $x$ .

When we encounter a binder for a term of sort  $s$ , as seen in the second case, the renaming is extended using function  $\text{ext}_r$ . If we want to use a renaming as a function or use the function  $\text{ext}_r$ , 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 binding of sort  $s$ . Thus, if we encounter the new binding  $\text{here refl}$  in the extended renaming, then we return the variable for the new binding  $\text{here refl}$ . The variables  $x$  of the original renaming  $\rho$  are weakened by wrapping them in an additional  $\text{there}$  constructor. The sort arguments are ignored inside the function body of  $\text{ext}_r$  by using the wildcard pattern  $\_$ .

Similar to variables, terms can be weakened using the function  $\text{wk}$  that shifts all variables present in the term by one recursively.

```

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

```

The function  $\text{wk}_r$  generates a weakening by wrapping all variables in an additional  $\text{there}$  constructor.

```

wkr : Ren S (S ▷ s)
wkr _ = there

```

## 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 var) → Sub (S1 ▷ s) (S2 ▷ s)
exts σ s _ (here refl) = ' here refl
exts σ s _ (there x) = wk (σ _ x)

```

For the newly bound variable `here refl`, we return the variable term `' here refl`. Furthermore, all terms  $\sigma \_ x$  originally present in the substitution  $\sigma$  are weakened using the function `wk`.

The substitution operator  $t \llbracket t' \rrbracket$  substitutes the last bound variable in  $t$  with  $t'$ , given that the sort of the last binder corresponds to the sort of  $t'$ .

$$\begin{aligned} \_ \llbracket \_ \rrbracket &: \text{Term } (S \triangleright s') \ s \rightarrow \text{Term } S \ s' \rightarrow \text{Term } S \ s \\ t \llbracket t' \rrbracket &= \text{sub } (\text{single}_s \text{ id}_s \ t') \ t \end{aligned}$$

A single substitution `singles` takes an existing substitution  $\sigma'$  and term  $t'$ . The term  $t'$  is then introduced for an additional new binding `here refl`.

$$\begin{aligned} \text{single}_s &: \text{Sub } S_1 \ S_2 \rightarrow \text{Term } S_2 \ s \rightarrow \text{Sub } (S_1 \triangleright s) \ S_2 \\ \text{single}_s \ \sigma \ t' \ \_ &(\text{here refl}) = t' \\ \text{single}_s \ \sigma \ t' \ \_ &(\text{there } x) = \sigma \_ x \end{aligned}$$

In the case of  $\_ \llbracket \_ \rrbracket$ , we let  $\sigma'$  be the identity substitution  $\text{id}_s : \text{Sub } S \ S$ .

## Context

Similar to terms, typing contexts  $\Gamma$  are indexed by a list of bound variables. In consequence, only types and kinds for bound variables can be stored in  $\Gamma$  by definition.

$$\begin{aligned} \text{data Ctx} &: \text{Sorts} \rightarrow \text{Set where} \\ \emptyset &: \text{Ctx } [] \\ \_ \blacktriangleright \_ &: \text{Ctx } S \rightarrow \text{Term } S \ (\text{type-of } s) \rightarrow \text{Ctx } (S \triangleright s) \end{aligned}$$

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 `type-of`  $s$ . The function `type-of` maps bindable sorts  $s$  to the sort of the term that is stored in  $\Gamma$  for variables of the sort  $s$ . Thus, if we bind a new variable for a term of the sort  $s$ , then  $\Gamma$  needs to be extended by a term of the sort `type-of`  $s$ .

$$\begin{aligned} \text{type-of } e_s &= \tau_s \\ \text{type-of } \tau_s &= \kappa_s \end{aligned}$$

Expression variables require  $\Gamma$  to store the corresponding type. Similarly, we store the corresponding kind for all types in  $\Gamma$ .

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

$$\begin{aligned} \text{lookup} &: \text{Ctx } S \rightarrow \text{Var } S \ s \rightarrow \text{Term } S \ (\text{type-of } s) \\ \text{lookup } (\Gamma \blacktriangleright T) \ (\text{here refl}) &= \text{wk } T \\ \text{lookup } (\Gamma \blacktriangleright T) \ (\text{there } x) &= \text{wk } (\text{lookup } \Gamma \ x) \end{aligned}$$

Inside both cases of the case split on variables, we wrap the looked up  $T$  in a weakening. As a result,  $T$  always has the index  $S$  that aligns with the current required 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  $\Gamma$  by definition.

## Typing

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

```

data  $\vdash_{\text{type}} : \text{Ctx } S \rightarrow \text{Term } S \rightarrow \text{Term } S \text{ (type-of } s) \rightarrow \text{Set where}$ 
 $\vdash'x :$ 
  lookup  $\Gamma x \equiv \tau \rightarrow$ 
   $\Gamma \vdash' x : \tau$ 
 $\vdash\top :$ 
   $\Gamma \vdash \text{tt} : \top$ 
 $\vdash\lambda :$ 
   $\Gamma \triangleright \tau \vdash e : \text{wk } \tau' \rightarrow$ 
   $\Gamma \vdash \lambda'x \rightarrow e : \tau \Rightarrow \tau'$ 
 $\vdash\Lambda :$ 
   $\Gamma \triangleright \star \vdash e : \tau \rightarrow$ 
   $\Gamma \vdash \Lambda' \alpha \rightarrow e : \forall' \alpha \tau$ 
 $\vdash\cdot :$ 
   $\Gamma \vdash e_1 : \tau_1 \Rightarrow \tau_2 \rightarrow$ 
   $\Gamma \vdash e_2 : \tau_1 \rightarrow$ 
   $\Gamma \vdash e_1 \cdot e_2 : \tau_2$ 
 $\vdash\bullet :$ 
   $\Gamma \vdash e : \forall' \alpha \tau \rightarrow$ 
   $\Gamma \vdash e \bullet \tau' : \tau [ \tau' ]$ 
 $\vdash\text{let} :$ 
   $\Gamma \vdash e_2 : \tau \rightarrow$ 
   $\Gamma \triangleright \tau \vdash e_1 : \text{wk } \tau' \rightarrow$ 
   $\Gamma \vdash \text{let}'x = e_2 \text{ 'in } e_1 : \tau'$ 
 $\vdash\tau :$ 
   $\Gamma \vdash \tau : \star$ 

```

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 in the list of bound variables with the context  $\Gamma \triangleright \tau$ . As a result,  $\tau'$  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 inside the context 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  $\_ : \_ \Rightarrow_r \_ : \text{Ren } S_1 \ S_2 \rightarrow \text{Ctx } S_1 \rightarrow \text{Ctx } S_2 \rightarrow \text{Set}$  where
   $\vdash \text{id}_r : \forall \{I\} \rightarrow \_ : \_ \Rightarrow_r \_ \{S_1 = S\} \{S_2 = S\} \text{id}_r \ I \ I$ 
   $\vdash \text{ext}_r : \forall \{\rho : \text{Ren } S_1 \ S_2\} \{I_1 : \text{Ctx } S_1\} \{I_2 : \text{Ctx } S_2\}$ 
     $\{T' : \text{Term } S_1 \ (\text{type-of } s)\} \rightarrow$ 
     $\rho : I_1 \Rightarrow_r I_2 \rightarrow$ 
     $(\text{ext}_r \ \rho \ \_) : (I_1 \blacktriangleright T') \Rightarrow_r (I_2 \blacktriangleright \text{ren } \rho \ T')$ 
   $\vdash \text{drop}_r : \forall \{\rho : \text{Ren } S_1 \ S_2\} \{I_1 : \text{Ctx } S_1\} \{I_2 : \text{Ctx } S_2\}$ 
     $\{T' : \text{Term } S_2 \ (\text{type-of } s)\} \rightarrow$ 
     $\rho : I_1 \Rightarrow_r I_2 \rightarrow$ 
     $(\text{drop}_r \ \rho) : I_1 \Rightarrow_r (I_2 \blacktriangleright T')$ 

```

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

The typed extension of a renaming  $\vdash \text{ext}_r$  allows to extend both  $I_1$  and  $I_2$  by  $T'$  and renamed  $T'$  respectively. The constructor  $\vdash \text{ext}_r$  corresponds to the typed version of the function  $\text{ext}_r$  that is used when a binder is encountered.

The constructor  $\vdash \text{drop}_r$  allows to introduce  $T'$  only in  $I_2$ . Hence,  $\vdash \text{drop}_r \ \vdash \text{id}_r$  corresponds to the typed weakening  $\vdash \text{wk}_r$  of a term.

The absence of a constructor that allows to introduce some  $T$  only to  $I_1$  is exactly the restriction needed for typed renamings to be order preserving embeddings.

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

```

 $\_ : \_ \Rightarrow_s \_ : \text{Sub } S_1 \ S_2 \rightarrow \text{Ctx } S_1 \rightarrow \text{Ctx } S_2 \rightarrow \text{Set}$ 
 $\_ : \_ \Rightarrow_s \_ \{S_1 = S_1\} \ \sigma \ I_1 \ I_2 = \forall \{s\} \ (x : \text{Var } S_1 \ s) \rightarrow$ 
   $I_2 \vdash \sigma \ \_ \ x : (\text{sub } \sigma \ (\text{lookup } I_1 \ x))$ 

```

Typed substitutions  $\vdash \sigma$  map variables  $x \in S_1$  to the corresponding typing of the term  $\sigma \ \_ \ x$  in  $I_2$ . The type of the term  $\sigma \ \_ \ x$  must be the original type of  $x$  in  $I_1$  applied to the substitution  $\sigma$ .

## 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.

```

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

```

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

```

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 the applied argument are values.

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

The rules  $\xi\text{-}\cdot_i$  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.

```

progress :
  ∅ ⊢ e : τ →
  (∃[ e' ] (e ↦ e')) ⊔ Val e
progress ⊢ T = 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  $\vdash T$ ,  $\vdash \lambda$  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.

```

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

```

The  $\xi$  reduction cases  $\xi_{-1}$ ,  $\xi_{-2}$ ,  $\xi_{-•}$  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 **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.

$$\begin{aligned}
&\vdash\sigma\text{-preserves} : \forall \{ \sigma : \text{Sub } S_1 S_2 \} \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \} \\
&\quad \{ t : \text{Term } S_1 s \} \{ T : \text{Term } S_1 (\text{type-of } s) \} \rightarrow \\
&\quad \sigma : \Gamma_1 \Rightarrow_s \Gamma_2 \rightarrow \\
&\quad \Gamma_1 \vdash t : T \rightarrow \\
&\quad \Gamma_2 \vdash (\text{sub } \sigma t) : (\text{sub } \sigma T)
\end{aligned}$$

The lemma  $\vdash\sigma\text{-preserves}$  follows by induction over the typing rules and lemmas about the interaction of substitutions and renamings. More specifically, we also need to prove that all operations on substitutions preserve the typing. For instance, we need to prove the lemma  $\vdash\sigma\uparrow$  that says that the typed extension of a substitution  $\vdash\text{ext}_s$  is type preserving. Because  $\text{ext}_s$  uses renaming under the hood, we also need to prove the lemma  $\vdash\rho\text{-preserves}$  that says that applying a typed renaming preserves the typing. Furthermore, we need to prove the lemmas  $\text{assoc-sub-ren}$ ,  $\text{assoc-ren-ren}$ ,  $\text{assoc-ren-sub}$  and  $\text{assoc-sub-sub}$  that prove the operations of applying a renaming and substitution to be associative in all combinations.<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:  $\text{o}_s$  for overloaded variables and  $\text{c}_s$  for constraints.

```

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

```

Terms of sort  $\text{o}_s$  can only be constructed using the variable constructor  $\_$ . Thus, terms of sort  $\text{o}_s$  are called overloaded variables and sort  $\text{o}_s$  is indexed by  $\text{var}$ . Variables for constraints do not exist in System F<sub>O</sub> and thus  $\text{c}_s$  is indexed by  $\text{no-var}$ . We use the variable symbol  $o$  for overloaded variables and the variable symbol  $c$  for constraints.

#### Syntax

We only discuss additions to the syntax of System F.

<sup>1</sup> 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 as Agda code file: <https://github.com/Mari-W/System-Fo/blob/main/proofs/SystemF.agda>

```

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
  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 the 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  $e_2$ . Instance expressions do not introduce new bindings and thus, the index  $S$  is never extended.

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  $[c] \Rightarrow \tau$ . 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  $\text{Cstr } S = \text{Term } S \text{ c}_s$ .

## Renaming & Substitution

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

```

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

```

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

## Context

In addition to 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  $\Gamma$ . Thus, similar to type variables, we store kind  $\star$  in  $\Gamma$  to denote the existence of an overloaded variable.

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  $\Gamma \blacktriangleright c$  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 analogous to the `lookup` function in System F and simply ignores constraints stored in the context.

### Constraint Solving

The search for constraints in a context is inductively formalized, similar to membership proofs  $s \in S$ . The subtle difference is that we reference constraints in  $\Gamma$  and not in  $S$ . The constraint solving type does need to search in  $\Gamma$  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  $\Gamma$  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 be behind an 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 that 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. The typing for overloaded variables results in a type. As a result, the `type-of` function returns the sort  $\tau_s$  for the sort  $o_s$  in the case of typings.

```
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  $\vdash^o$  says that if we can resolve the constraint  $o : \tau$  in  $\Gamma$ , then  $o$  can take on type  $\tau$ .

The rule for constraint abstractions  $\vdash^\lambda$  appends the constraint  $c$  to  $\Gamma$  and thus assumes  $c$  to be valid inside the body  $e$ . Constraint abstractions result in expressions of 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  $\vdash^\emptyset$ , given  $c$  can be resolved in  $\Gamma$ .

The rule  $\vdash^{\text{decl}}$  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. The existence of  $o$  is denoted by extending  $\Gamma$  with kind  $\star$ . Analogous to the type  $\tau'$  inside the abstraction rule  $\vdash^\lambda$ , the resulting type  $\tau$  cannot use the introduced overloaded variable and thus is weakened to align in  $S$  with  $\Gamma$ . In consequence,  $\tau$  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  $e_2$ .

## 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
   $\vdash^{\text{drop-ctr}_r} : \forall \{ \Gamma_1 : \text{Ctx } S_1 \} \{ \Gamma_2 : \text{Ctx } S_2 \} \{ \tau \} \{ o \} \rightarrow$ 
     $\rho : \Gamma_1 \Rightarrow_r \Gamma_2 \rightarrow$ 
     $\rho : \Gamma_1 \Rightarrow_r (\Gamma_2 \blacktriangleright (o : \tau))$ 
  - ...
```

A constraint  $o : \tau$  can be introduced only to  $\Gamma_2$  using the  $\vdash^{\text{drop-ctr}_r}$  constructor. Dropping a constraint corresponds to a typed weakening, similar to constructor  $\vdash^{\text{drop}_r}$ , but instead of introducing an unused variable we introduce an unused constraint. In consequence, the typed weakening by a constraint  $\vdash^{\text{wk-ctr}_r}$  is defined by  $\vdash^{\text{drop-ctr}_r} \vdash^{\text{id}_r}$ .

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

The constructor  $\vdash^{\text{single-type}_s}$  allows to introduce an additional new type variable binder that is substituted with type  $\tau$ . Thus, the constructor  $\vdash^{\text{single-type}_s}$  complements the  $\text{single-type}_s$  function.

Constructors  $\vdash^{\text{id}_s}$ ,  $\vdash^{\text{ext}_s}$ ,  $\vdash^{\text{drop}_s}$  and  $\vdash^{\text{drop-ctr}_s}$  are not shown. All of them function the same way as their counterparts in typed renamings.

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. 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. In hindsight arbitrary substitutions would not have produced an unreasonable amount of additional work, but the restriction does not effect the type preservation proof in any way, so it remained as is.

## 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  $\kappa_s$  do not need to be translated. Intuitively there does not even exist a sensible translation for  $c_s$ .

$$\begin{aligned} \rightsquigarrow_S &: F^O.\text{Sort } \text{var} \rightarrow F.\text{Sort } \text{var} \\ \rightsquigarrow_S e_s &= e_s \\ \rightsquigarrow_S o_s &= e_s \\ \rightsquigarrow_S \tau_s &= \tau_s \end{aligned}$$

Sorts  $e_s$  and  $\tau_s$  translate to their corresponding counterparts in System F.

The sort  $o_s$  translates to  $e_s$ . This is because, instead of removing `decl` expressions as seen in the example at the beginning, we keep them as redundant `let` expressions that bind a unit value `tt`. Because `decl` expressions bind a new overloaded variable in System  $F_O$ , removing them would result in a variable binding less in System F. To prevent more complex proofs involving the index  $S$  of terms, we keep all `decl` expressions as redundant `let` bindings. In consequence, all bindings for overloaded variables translate to normal expression bindings in System F and the sort  $o_s$  needs to translate 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, an `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 an additional entry  $e_s$  at the corresponding position to further function as valid index for the translated  $e_1$ . Similarly, an additional sort  $e_s$  must be appended to  $S$  inside the body of translated constraint abstractions. 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 all positions of new bindings that were not present in System  $F_O$ .

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

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, then the sort  $s$  is directly translated using the function  $\rightsquigarrow_S$ .

## Variables

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

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

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.

Resolved constraints translate to correct unique expression variables using function  $o \rightsquigarrow x$ . We can apply a symmetric argumentation as seen in the translation for variables because the type for resolved constraints  $[c] \in \Gamma$  preserves the structure of the context perfectly. The subtle difference to  $x \rightsquigarrow x$  is that we have the two cases **here** and **under-ctr** for constraints, instead of the two cases **here** and **there** for normal variables. Furthermore, we only have one case for bindings **under-bind** in  $o \rightsquigarrow x$ , instead of one case for constraints in  $x \rightsquigarrow x$ .

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

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 a **there** constructor to respect the binding.

In the case **under-ctr**, we again wrap the variable in an additional **there** that was not present before to respect the new binding introduced by the translation.

## Context

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

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

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. This is exactly what we want because for each constraint in System  $F_O$  there will either be an additional lambda binding in System F that accepts the constraint as higher-order function or a let binding that binds a variable with type  $\tau \rightsquigarrow \tau \ \tau$ .

## Renaming & Substitution

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

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

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

The 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 follows similarly.

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

The typed renaming  $\vdash \text{single-type}_s$  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  $\vdash \text{drop-cstr}_s$  are analogous to the cases for renamings.

## Terms

Types and kinds can be translated without typing information using the function  $\mathbb{T} \rightsquigarrow \mathbb{T}$ . We only look at the untyped translation of types  $\tau \rightsquigarrow \tau$  that is used inside the function  $\mathbb{T} \rightsquigarrow \mathbb{T}$  because the kind  $\star$  translates to its direct counterpart in System F.

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

Constraint types  $[\ o : \tau ] \Rightarrow \tau'$  translate to function types  $\tau \Rightarrow \tau'$ . The translation from constraint types to function types directly corresponds to the translation of constraint abstractions to normal abstractions. The implicitly resolved constraint will be taken as higher-order function argument of type  $\tau \rightsquigarrow \tau \tau$ .

All other System  $F_O$  types translate to their direct counterparts in System F.

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.

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

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

Typed constraint abstractions  $\vdash' \lambda$  translate to normal abstractions with an additional new binding.

An implicitly resolved constraint  $\vdash' \odot$  translates to an explicit application that passes the resolved instance  $o : \tau \in \Gamma$  as argument. We again use function  $o \rightsquigarrow x$  to translate the resolved instance to the corresponding unique variable.

As discussed, we translate typed declaration expressions  $\vdash' \text{decl}$  to redundant  $\text{let}$  bindings that bind a unit value.

We translate typed instance expressions  $\vdash' \text{inst}$  to  $\text{let}$  expressions that introduce an additional binding not present in System  $F_O$ .

## 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  $F_O$ . The function  $\rightsquigarrow\text{-pres-}\vdash$  produces a typed System F term for an arbitrary typed System  $F_O$  term  $\vdash t$ . The translated System  $F_O$  term  $\vdash_{\rightsquigarrow} t$  is typed in the translated context  $\Gamma \rightsquigarrow \Gamma$  and has the typing result  $T \rightsquigarrow T$ .

$$\begin{aligned}
& \rightsquigarrow\text{-pres-}\vdash : \{ \Gamma : F^O.\text{Ctx } F^O.S \} \{ t : F^O.\text{Term } F^O.S \ F^O.s \} \\
& \quad \{ T : F^O.\text{Term } F^O.S \ (F^O.\text{type-of } F^O.s) \} \rightarrow \\
& \quad (\vdash t : \Gamma F^O.\vdash t : T) \rightarrow \\
& \quad (\Gamma \rightsquigarrow \Gamma) F.\vdash (\vdash_{\rightsquigarrow} t \vdash t) : (T \rightsquigarrow T) \\
& \rightsquigarrow\text{-pres-}\vdash (\vdash' x \ \Gamma x \equiv \tau) = \vdash' x \ (\rightsquigarrow\text{-pres-lookup } \Gamma x \equiv \tau) \\
& \rightsquigarrow\text{-pres-}\vdash (\vdash' o \ o : \tau \in \Gamma) = \vdash' x \ (\rightsquigarrow\text{-pres-cstr-solve } o : \tau \in \Gamma) \\
& \rightsquigarrow\text{-pres-}\vdash (\vdash' \text{let} \vdash e_2 \vdash e_1) = \vdash' \text{let} \ (\rightsquigarrow\text{-pres-}\vdash \vdash e_2) \\
& \quad (\text{subst } (\_ F.\vdash \vdash_{\rightsquigarrow} t \vdash e_1 : \_) \rightsquigarrow\text{-dist-wk-type}(\rightsquigarrow\text{-pres-}\vdash \vdash e_1)) \\
& \rightsquigarrow\text{-pres-}\vdash (\vdash' \lambda \{ c = (\vdash' o : \tau) \} \vdash e) = \vdash' \lambda \\
& \quad (\text{subst } (\_ F.\vdash \vdash_{\rightsquigarrow} t \vdash e : \_) \rightsquigarrow\text{-dist-wk-inst-type}(\rightsquigarrow\text{-pres-}\vdash \vdash e)) \\
& \rightsquigarrow\text{-pres-}\vdash (\vdash' \odot \vdash e \ o : \tau \in \Gamma) = \vdash' \cdot (\rightsquigarrow\text{-pres-}\vdash \vdash e) (\vdash' x \ (\rightsquigarrow\text{-pres-cstr-solve } o : \tau \in \Gamma)) \\
& \rightsquigarrow\text{-pres-}\vdash (\vdash' \bullet \{ \tau = \tau' \} \{ \tau' = \tau' \} \vdash e) = \text{subst } (\_ F.\vdash \vdash_{\rightsquigarrow} t \vdash e \bullet \tau \rightsquigarrow \tau' : \_) \\
& \quad (\rightsquigarrow\text{-dist-}\tau[\tau'] \ \tau' \ \tau) (\vdash' \bullet (\rightsquigarrow\text{-pres-}\vdash \vdash e)) \\
& - \dots
\end{aligned}$$

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  $\rightsquigarrow\text{-pres-lookup}$ . The lemma produces an equality proof

of type  $\mathbf{F.lookup} (\Gamma \rightsquigarrow \Gamma' I) (x \rightsquigarrow x) \equiv (\tau \rightsquigarrow \tau)$  when given an equality proof  $\mathbf{F}^O.lookup \Gamma x \equiv \tau$ . With the lemma  $\rightsquigarrow\text{-pres-lookup}$  the typing rule  $\vdash'x$  can be translated to the typing rule for variables in System F.

Similarly, the lemma  $\rightsquigarrow\text{-pres-cstr-solve}$  translates the proof  $o:\tau \in \Gamma$  that an instance  $o:\tau$  was resolved for an overloaded variable  $o$  to proof that the unique variable  $o \rightsquigarrow x : o:\tau \in \Gamma$  has type  $\tau \rightsquigarrow \tau$  in  $\Gamma \rightsquigarrow \Gamma' I: \mathbf{F.lookup} (\Gamma \rightsquigarrow \Gamma' I) (o \rightsquigarrow x : o:\tau \in \Gamma) \equiv (\tau \rightsquigarrow \tau)$ . Using lemma  $\rightsquigarrow\text{-pres-cstr-solve}$  the typing rule for overloaded variables  $\vdash'o$  can be translated to the typing rule for normal variables  $\vdash'x$ .

Typed let bindings  $\vdash\text{let} \vdash e_2 \vdash e_1$  translate to typed let bindings in System F. The typing rule  $\vdash e_2$  is translated directly using the induction hypothesis. Because the expression  $e_1$  results in type  $\mathbf{wk} \tau'$  inside typing rule  $\vdash e_1$ , 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  $\rightsquigarrow\text{-dist-wk-type}$  produces the required equality proof of type  $\tau \rightsquigarrow \tau \{ \Gamma = \Gamma \triangleright T \} (\mathbf{F}^O.wk \tau') \equiv \mathbf{F.wk} (\tau \rightsquigarrow \tau)$ . On the left hand side of the equality we need to specify that the otherwise implicit argument  $\Gamma = \Gamma$  needs to be an extended  $\Gamma$  in that case. We then substitute the equivalence into the translation of the typing rule  $\vdash e_1$ .

Typed constraint abstractions  $\vdash\lambda$  translate to normal abstractions in System F. Inside the typing  $\vdash e$ , the result type  $\tau$  of the body  $e$  does not need to be weakened because the constraint abstraction only introduced a constraint and no actual binding to context  $\Gamma$ . After the translation the former constraint will be bound by a binding and thus a new item in  $\Gamma \rightsquigarrow \Gamma' I$  will exist. To ignore the binding,  $\tau$  is weakened in the abstraction rule  $\vdash\lambda$ . Lemma  $\rightsquigarrow\text{-dist-wk-inst-type}$  proves that translating  $\tau$  in  $\Gamma$  extended by a constraint is equivalent to weakening  $\tau$  after the translation:  $\tau \rightsquigarrow \tau \{ \Gamma = \Gamma \triangleright ('o : \tau') \} \tau \equiv \mathbf{F.wk} (\tau \rightsquigarrow \tau)$ . The lemma follows because the constraint translates to an actual binding and consequently, both sides of the equivalence have an additional expression binding that  $\tau$  does not care about.

Implicitly resolved constraints  $\vdash\odot$  carry the information  $o:\tau \in \Gamma$  about the instance that was resolved. In System F the former constraint is explicitly passed as variable pointing to the correct translated instance. Thus, the typing  $\vdash\odot$  results in typed application  $\vdash\cdot$ . We apply the correct equality proof to the typing rule  $\vdash'x$  using lemma  $\rightsquigarrow\text{-pres-cstr-solve}$ .

The type application rule  $\vdash\bullet$  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 substitution is applied in System F. Using lemma  $\rightsquigarrow\text{-dist-}\tau[\tau']$  of type  $(\tau \rightsquigarrow \tau \{ \Gamma = \Gamma \triangleright \star \} \tau' \mathbf{F}.[\tau \rightsquigarrow \tau \tau]) \equiv \tau \rightsquigarrow \tau (\tau' \mathbf{F}^O.[\tau])$  we can substitute the equivalence into the System F typing rule  $\vdash\bullet (\rightsquigarrow\text{-pres-}\vdash e)$ .

The translation of  $\vdash\top$ ,  $\vdash\lambda$ ,  $\vdash\cdot$ ,  $\vdash\text{decl}$  and  $\vdash\text{inst}$  is either a direct translation or uses similar ideas and no other lemmas than the ones discussed.

## Renaming

Both  $\rightsquigarrow\text{-dist-wk-type}$  and  $\rightsquigarrow\text{-dist-wk-inst-type}$  directly follow from a more general lemma  $\rightsquigarrow\text{-dist-ren-type}$  for arbitrary renamings. The lemma  $\rightsquigarrow\text{-dist-ren-type}$  proves that translating both the typed renaming  $\vdash\rho$  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 the renamed type  $\tau$ . The lemma can be proven by induction over System  $F_O$  types  $\tau$ .

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

The case for type variables needs an additional lemma  $\rightsquigarrow\text{-dist-ren-var-type}$  specifically for type variables. Lemma  $\rightsquigarrow\text{-dist-ren-var-type}$  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  $\rightsquigarrow\text{-dist-}\tau[\tau']$  follows from a more general lemma about type in type substitutions  $\rightsquigarrow\text{-dist-sub-type}$ . The lemma  $\rightsquigarrow\text{-dist-sub-type}$  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  $\rightsquigarrow\text{-dist-sub-var-type}$  do not follow directly from the induction hypothesis. To understand why, we at look at the case  $\vdash\text{ext}_s$ .

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

The case  $\vdash\text{ext}_s$  is proven via a case split on variables, similar to how  $\text{ext}_s$  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  $\rightsquigarrow\text{-dist-ren-type}$  applied to the typed weakening  $\vdash\text{wk}_r$  because  $\text{ext}_s$  is defined by weakening the terms that result of the substitution  $\sigma$  being applied to variables  $x$ .

Both  $\vdash\text{id}_s$  and  $\vdash\text{single-type}_s$  follow directly from the induction hypothesis. The cases for  $\vdash\text{drop}_s$  and  $\vdash\text{drop-cstr}_s$  are similar to the case  $\vdash\text{ext}_s$  and also align with their corresponding definition.



## Variables

We first look at the proof for lemma  $\rightsquigarrow\text{-pres-lookup}$ . The lemma is proven via induction over the System  $F_O$  context  $\Gamma$ .

$$\begin{aligned} \rightsquigarrow\text{-pres-lookup} &: \forall \{ \Gamma : F^O.\text{Ctx } F^O.S \} \{ \tau : F^O.\text{Type } F^O.S \} \{ x : F^O.\text{Var } F^O.S \ e_s \} \rightarrow \\ &F^O.\text{lookup } \Gamma \ x \equiv \tau \rightarrow \\ &F.\text{lookup } (\Gamma \rightsquigarrow \Gamma) \ (x \rightsquigarrow x) \equiv (\tau \rightsquigarrow \tau) \\ \rightsquigarrow\text{-pres-lookup } \{ \Gamma = \Gamma \blacktriangleright \tau \} \{ x = \text{here } \text{refl} \} \text{refl} &= \rightsquigarrow\text{-dist-ren-type } F^O.\vdash_{w_r} \tau \\ \rightsquigarrow\text{-pres-lookup } \{ \Gamma = \Gamma \blacktriangleright \_ \} \{ \tau' \} \{ x = \text{there } x' \} \text{refl} &= \text{trans} \\ &(\text{cong } F.wk \ (\rightsquigarrow\text{-pres-lookup } \{ x = x' \} \text{refl})) \\ &(\rightsquigarrow\text{-dist-ren-type } F^O.\vdash_{w_r} (F^O.\text{lookup } \Gamma \ x')) \\ &- \dots \end{aligned}$$

As an example we will look at the case  $\Gamma \blacktriangleright \tau$ . The case is proven via a case split on variables. The prove follows the same reasoning as the  $\vdash\text{-ext}_s$  case for substitutions above. Because the function `lookup` weakens the type  $\tau$  that is looked up in  $\Gamma$  in both cases `here` and `there`, both use lemma  $\rightsquigarrow\text{-dist-ren-type}$  applied to the typed weakening  $\vdash_{w_r}$  to account for the weakening.

The case  $\Gamma \blacktriangleright c$  follows analogously, but uses  $\vdash_{w_r}\text{-cstr}_r$  applied to the induction hypothesis instead of  $\vdash_{w_r}$ . Furthermore, the case  $\emptyset$  is impossible because there must be a context item if a variable exists.

The lemma  $\rightsquigarrow\text{-pres-cstr-solve}$  can be proven via induction over the type for resolved constraints  $[c] \in \Gamma$ . The lemma follows symmetrically to the lemma  $\rightsquigarrow\text{-pres-lookup}$  because the type for resolved constraints preserves the structure of  $\Gamma$  perfectly.

$$\begin{aligned} \rightsquigarrow\text{-pres-cstr-solve} &: \forall \{ \Gamma : F^O.\text{Ctx } F^O.S \} \rightarrow \\ &(\sigma : \tau \in \Gamma : [c] : F^O.o : F^O.\tau] \in \Gamma) \rightarrow \\ &F.\text{lookup } (\Gamma \rightsquigarrow \Gamma) \ (\sigma \rightsquigarrow x \ \sigma : \tau \in \Gamma) \equiv (\tau \rightsquigarrow \tau \ F^O.\tau) \end{aligned}$$

Similar to the case split on variables with constructors `here` and `there` inside the case  $\Gamma \blacktriangleright \tau$  of lemma  $\rightsquigarrow\text{-pres-lookup}$ , we have the two cases `here` and `under-cstr` in lemma  $\rightsquigarrow\text{-pres-cstr-solve}$ . Both cases work similarly to cases in lemma  $\rightsquigarrow\text{-pres-lookup}$ , except that they use  $\vdash_{w_r}\text{-cstr}_r$  instead of  $\vdash_{w_r}$ .

Furthermore, we have the case `under-bind` that works similar to the case  $\Gamma \blacktriangleright c$  and always uses the induction hypothesis applied to  $\vdash_{w_r}$  because we search for a former constraint and not a variable.

## 6 Discussion 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  $HM_O$  and the target language would be Hindley-Milner. The Hindley-Milner system is a restricted form of System F that allows for full type inference. Similarly,  $HM_O$  would be a restricted form of System  $F_O$  with support for full type inference.

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. To separate the type logic from the expression logic in Hindley-Milner fashion, we would need to embed constraints into explicit type annotations 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  $F_O$  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  $F_O$  syntax.

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 substituted based on the types of the arguments  $e_1 \dots e_n$ . More specifically, we could introduce a reduction rule  $\beta\text{-}\alpha^*$  that reduces  $o \cdot e_1 \dots e_n$  to  $e \cdot e_1 \dots e_n$ , where  $e$  is the resolved instance based on the types of  $e_1 \dots e_n$ . 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  $o \cdot e$  would be decidable.

Let  $\vdash e \hookrightarrow \vdash e'$  be a typed small step semantic for System  $F_O$ . We would need to prove something similar to: If  $\vdash e \hookrightarrow \vdash e'$  then  $\exists [e''] (\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e' \vdash e \hookrightarrow^* e'') \times (\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e' \vdash e' \hookrightarrow^* e'')$ , where  $\vdash e \hookrightarrow e' \rightsquigarrow e \hookrightarrow e'$  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 translated  $\vdash e'$  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 as argument using an additional application step in System F. For now it remains unclear if semantic preservation can be proven by induction over 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  $F_O$  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  $HM_O$  type  $\forall\alpha. \forall\beta. [a : \alpha \rightarrow \alpha \rightarrow \alpha] \Rightarrow [b : \beta \rightarrow \beta \rightarrow \beta] \Rightarrow \dots$  would be expressed as  $\forall\alpha. (a : \alpha \rightarrow \alpha \rightarrow \alpha) \Rightarrow \forall\beta. (b : \beta \rightarrow \beta \rightarrow \beta) \Rightarrow \dots$  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 6.1 is solved in System O by restricting constraints to begin their type with the type variable that is bound by the quantifier that they are part of. In  $HM_O$  such a connection would 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 in each forall type, translating them results in  $n$  new lambda bindings in one induction step. While the problem above can be handled, another problem complicated the proof of type preservation via induction immensely. The translation of System O types must pull out forall quantifiers, because translating constraints directly to higher-order functions would break the rule that function types are only allowed to be built from mono types. For instance, the translated System O type from above should not be  $\forall\alpha. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \forall\beta. (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \dots$ , but rather  $\forall\alpha. \forall\beta. (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \dots$  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 did not work out. Besides System O, there exist other formalizations that are closer related to 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 overloading and type constraints. We formalized the Dictionary Passing Transform between System F and System  $F_O$ . Furthermore, 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 <sup>2</sup>.

One trick used in the formalization was to introduce constraints individually using constraint abstractions. 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 inside the type for resolved constraints. In consequence, the translation of resolved constraints to Debruijn 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.

<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. 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](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.

Freiburg i. Br, 27.03.2023

---

Place, Date

M. Weidner

---

Signature