# Abstractions for Multi-Sorted Substitutions

**Hannes Saffrich**
University of Freiburg, Germany

## Abstract

Formalizing a typed programming language in a proof assistant requires to choose representations for variables and typing. Variables are often represented as de Bruijn indices, where substitution is usually defined in terms of renamings to allow for proofs by structural induction. Typing can be represented extrinsically by defining untyped terms and a typing relation, or intrinsically by combining syntax and typing into a single definition of well-typed terms. For extrinsic typing, there is again a choice between extrinsic scoping, where terms and the notion of free variables are defined separately, and intrinsic scoping, where terms are indexed by their free variables.

This paper describes an Agda framework for formalizing programming languages with extrinsic typing, intrinsic scoping, and de Bruijn Indices for variables. The framework supports object languages with arbitrary many variable sorts and dependencies, making it suitable for polymorphic languages and dependent types. Given an Agda definition of syntax and typing, the framework derives substitution operations and lemmas for untyped terms, and provides an abstraction to prove type preservation of these operations with just a single lemma. The key insights behind the framework are the use of multi-sorted syntax definitions, which enable parallel substitutions that replace all variables of all sorts simultaneously, and abstractions that unify the definitions, compositions, typings, and type preservation lemmas of multi-sorted renamings and substitutions. Case studies have been conducted to prove subject reduction for System F with subtyping, dependently typed lambda calculus, and lambda calculus with pattern matching.

**2012 ACM Subject Classification** Theory of computation → Type theory; Software and its engineering → Syntax; Theory of computation → Logic and verification

**Keywords and phrases** Agda, Metatheory, Framework

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2024.32

**Supplementary Material**
*Software (Source Code)*: `https://github.com/m0rphism/kitty-supplement`
archived at `swh:1:dir:8ee2a0aeb901498fca00f6d379099386fab74c60`

## 1 Introduction

Formalizing programming languages in proof assistants quickly gets repetitive. Almost every programming language supports variables with static binding, and hence requires numerous definitions and lemmas related to variable substitution.

Additionally, repetition can also occur within a single formalization. This can be seen with polymorphic languages, where multiple sorts of variables are present. Consider for example System F, which supports both expression- and type-variables. With a naive approach, the whole substitution machinery needs to be duplicated three times! We need to substitute expression-variables in expressions, type-variables in types, but also type-variables in expressions. Even worse, having two substitutions acting on expressions requires to also prove lemmas about their interactions. If we would additionally introduce kind-variables, we would end up with a total of six duplications of the substitution machinery and corresponding interaction lemmas!


© Hannes Saffrich;
licensed under Creative Commons License CC-BY 4.0

Further repetition can occur due to the choice of variable representation. For example, for de Bruijn indices, substitution is usually defined in terms of renamings[1] to allow for structural induction. With a naive approach, this would again double the amount of substitution machinery, as all definitions and lemmas need to be written first for renamings and then again for substitutions.

Similarly, for a typed language formalized via extrinsic typing we need type preservation lemmas for each substitution and renaming operation, which again doubles the number of definitions.

If one is not careful, a formalization of a language with type and kind polymorphism can easily end up with 24 slightly changed copies of the whole substitution machinery! Clearly, this situation is in need of automation.

Our framework approaches this problem by using a combination of abstractions and reflection in the context of extrinsically typed, intrinsically scoped syntax with de Bruijn indices. The user can write a language specification using regular Agda definitions (no encodings via generic programming[2]) and our framework derives definitions and lemmas for untyped substitution, and provides an abstraction to prove type preservation for all substitution operations with only a single lemma. For System F, our framework allows to prove subject reduction with only a single handwritten lemma for type preservation.

Compared to standard practices, we do not derive substitutions for each of the variable sorts and syntactic categories, but instead use a novel approach for defining syntax, which directly supports substitutions that replace all variable sorts in parallel and can be applied to all syntactic categories. By further unifying renamings and substitutions, we gain the ability to talk abstractly about any kind of renaming or substitution that can occur in the formalization. This generality is key to then define abstractions for typing and type preservation on the same level of generality, allowing to prove type preservation for renamings and substitutions of all variable sorts and syntactic categories in a single lemma for many typing relations, including those of our case studies.

## 1.1 Structure

The rest of this paper introduces our framework using System F as a running example for a substitution-preserves-typing proof.

Code of the framework is displayed in gray boxes. Code of examples is displayed in yellow boxes. Code of the System F formalization is displayed without boxes. The latter is the only code a user of our framework has to write.

In this paper, we present a simplified version of the full framework, focusing on the core concepts. We present all necessary definitions and lemmas, but omit some proofs for the sake of space. The omitted proofs can be found in our supplementary material, which includes the simplified framework (365 lines of code) and the System F formalization (212 lines of code, where 79 can be derived by using the full framework). The full framework including the case studies is available on Github: `https://github.com/m0rphism/kitty`

The rest of this paper is structured as follows: Section 2 introduces the multi-sorted syntax and compares it to the more common unsorted syntax. Section 3 introduces multi-sorted substitutions and renamings, and an abstraction to unify them. Section 4 introduces composition of multi-sorted substitutions and renamings, and an abstraction to unify all four compositions. Section 5 shows how to define types, type contexts, and typing relations, and

[1] *Renamings* are substitutions that are only allowed to replace variables with variables

presents an abstraction for unifying type preservation lemmas for renamings and substitutions. Section 6 describes the class of object languages covered by our reflection algorithm. Section 7 discusses our case studies. Section 8 discusses related work. Section 9 concludes.

### 1.2 Contributions

1. a novel approach for formalizing intrinsically-scoped syntax with multiple variable sorts as a special kind of intrinsically-typed syntax, we call *multi-sorted syntax*;
2. a novel abstraction for composition and its metatheory, unifying the four compositions between renamings and substitutions;
3. a novel abstraction for typing, unifying type preservation of renaming and substitution;
4. a formalized specification of a large class of object languages for which untyped substitution and lemmas can be derived generically.
5. an implementation as an Agda framework featuring a reflection algorithm, representation independence for substitutions and type contexts, heterogeneous equality between renamings and substitutions, and absence of postulated axioms.
6. three case studies in using our approach to prove subject reduction for System F with subtyping, a dependently-typed lambda calculus, and pattern matching.

## 2 Syntax

### 2.1 Unsorted Syntax

The following shows a typical intrinsically-scoped syntax of System F:

```
data Kind : Set where
  ⋆ : Kind                                  -- Type Kind
data Type (n : ℕ) : Set where
  ‘_       : Fin n → Type n                 -- Type variable
  ∀[α:_]_ : Kind → Type (suc n) → Type n    -- Universal quantification
  _⇒_      : Type n → Type n → Type n       -- Function type
data Expr (n m : ℕ) : Set where
  ‘_     : Fin m → Expr n m                 -- Expression variable
  λx_    : Expr n (suc m) → Expr n m        -- Expression abstraction
  Λα_    : Expr (suc n) m → Expr n m        -- Type abstraction
  _·_    : Expr n m → Expr n m → Expr n m   -- Expression application
  _•_    : Expr n m → Type n → Expr n m     -- Type application
```

Types are indexed by the number of free type variables n. Expressions are additionally indexed by the number of free expression variables m. Variables ‘_ are represented as de Bruijn indices, where Fin n is the type of n elements.

We identify two drawbacks with this style of syntax:

1. the syntactic categories (Kind, Type, and Expr) have different types, which makes it difficult to treat them uniformly; and
2. the different sorts of variables are modeled separately, which requires to define not just type-in-type and expression-in-expression substitution, but also type-in-expression substitution. Consequently, interaction lemmas between the substitutions are required.

To avoid these drawbacks, we instead use a multi-sorted syntax.

## 2.2 Multi-Sorted Syntax

A multi-sorted syntax is defined by a single type of sort-indexed terms.

A sort describes to which syntactic category a term belongs and is itself indexed by a sort type, which describes whether the syntax permits variables of this sort:

```
data SortTy : Set where Var NoVar : SortTy

data Sort : SortTy → Set where -- Our syntax supports:
  ★ : Sort Var    -- expressions and expression variables;
  ≈ : Sort Var    -- types and type variables; and
  ¬ : Sort NoVar  -- kinds, but no kind variables.
```

The term type `S ⊢ s` is indexed by its sort `s` and the sorts of its free variables `S`. For example, `[≈, ≈] ⊢ ★` is the type of expressions `(★ ★)` with two free type-variables `(≈)`.

```
data _⊢_ : List (Sort Var) → Sort st → Set where
  `_       : S ∋ s → S ⊢ s                    -- Expression and type variables
  λx_      : (★ :: S) ⊢ ★ → S ⊢ ★             -- Expression abstraction
  Λα_      : (≈ :: S) ⊢ ★ → S ⊢ ★             -- Type abstraction
  ∀[α:_]_  : S ⊢ ¬ → (≈ :: S) ⊢ ≈ → S ⊢ ≈     -- Universal quantification
  _·_      : S ⊢ ★ → S ⊢ ★ → S ⊢ ★            -- Expression application
  _•_      : S ⊢ ★ → S ⊢ ≈ → S ⊢ ★            -- Type application
  _⇒_      : S ⊢ ≈ → S ⊢ ≈ → S ⊢ ≈            -- Function type
  ★        : S ⊢ ¬                            -- Type kind
```

The notation `_⊢_` is often used for terms in intrinsically-typed languages. This is no accident: in effect, we defined an intrinsically-typed language with the twist that the typing relation assures exactly that the syntactic categories are followed. Sorts `s` correspond to types, and lists of sorts `S` correspond to type environments.

As it is typical in intrinsic typing, variables are represented as typed (in our case sorted) de Bruijn indices `S ∋ s`, i.e. values of the usual proof-relevant list-membership relation:

```
data _∋_ {ℓ} {A : Set ℓ} : List A → A → Set ℓ where
  zero : ∀ {xs x} → (x :: xs) ∋ x
  suc  : ∀ {xs x y} → xs ∋ x → (y :: xs) ∋ x
```

Note that there is no straightforward way to construct a multi-sorted syntax with intrinsic typing: in a direct translation, the type of terms `_⊢_` would be indexed by itself, which most proof assistants forbid to avoid breaking logical consistency.

## 2.3 A Structure for Multi-Sorted Syntax

The regularity of the multi-sorted syntax makes it easy to define a structure for arbitrary syntaxes, i.e. syntaxes with arbitrarily many syntactic categories and variable types:

```
record Syntax : Set₁ where
  field Sort       : SortTy → Set
        _⊢_        : ∀ {st} → List (Sort Var) → Sort st → Set
        `_         : ∀ {S} {s : Sort Var} → S ∋ s → S ⊢ s
        `-injective : ∀ {S s} {x y : S ∋ s} → ` x ≡ ` y → x ≡ y
```

The first three fields record the definitions of sorts, terms, and variable introduction. The last field records that variable introduction `` `_ `` is injective, which is trivially true for constructors. `Syntax` has type `Set₁`, because the `Sort` and `_⊢_` fields have type `Set`.

The instantiation for our System F syntax is straightforward:

```
SystemF-Syntax : Syntax
SystemF-Syntax = record { Sort = Sort ; _⊢_ = _⊢_ ; `_ = `_ ; `-injective = λ { refl → refl } }
```

# 3 Renamings & Substitutions

## 3.1 Multi-Sorted Renamings & Substitutions

Working with a sort-indexed syntax allows us to define renamings and substitutions that replace all variables of all sorts simultaneously:

```
_→ᵣ_ _→ₛ_ : List (Sort Var) → List (Sort Var) → Set
S₁ →ᵣ S₂ = ∀ s → S₁ ∋ s → S₂ ∋ s
S₁ →ₛ S₂ = ∀ s → S₁ ∋ s → S₂ ⊢ s
```

A renaming $S_1 \to_r S_2$ maps variables from $S_1$ to variables from $S_2$. A substitution $S_1 \to_s S_2$ maps variables from $S_1$ to terms with free variables from $S_2$.

This representation has the benefit that there is no combinatory explosion of substitutions and renamings, e.g. no extra lemmas have to be proved between an expression-in-expression and a type-in-expression substitution, because both are simply substitutions.

## 3.2 Unifying Renamings & Substitutions

To avoid the duplication between renamings and substitutions, McBride[6, 15] introduced the *kit* abstraction. A kit is a structure that allows to abstract over whether something is a term or a variable. The intention is to instantiate this structure exactly twice (once for variables and once for terms), and then write definitions, which are parameterized over a kit and consequently can be used for both variables and terms.

```
record Kit (_∋/⊢_ : List (Sort Var) → Sort Var → Set) : Set where
  field id/`          : S ∋ s → S ∋/⊢ s
        `/id          : S ∋/⊢ s → S ⊢ s
        wk            : ∀ s' → S ∋/⊢ s → (s' :: S) ∋/⊢ s
        `/`-is-`      : ∀ (x : S ∋ s) → `/id (id/` x) ≡ ` x
        id/`-injective : id/` x₁ ≡ id/` x₂ → x₁ ≡ x₂
        `/id-injective : ∀ {x/t₁ x/t₂ : S ∋/⊢ s} → `/id x/t₁ ≡ `/id x/t₂ → x/t₁ ≡ x/t₂
        wk-id/`       : ∀ s' (x : S ∋ s) → wk s' (id/` x) ≡ id/` (suc x)
```

As we intend to have exactly two Kit instances, we choose names of the form x/y, where x is the name we choose for the variable instance, and y is the name we choose for the term instance. For example the parameter type _∋/⊢_ will be instantiated to _∋_ for the variable kit, and to _⊢_ for the term kit.

A kit consists of the following components:

- id/` converts a variable S ∋ s into a S ∋/⊢ s. For the variable kit, _∋/⊢_ is instantiated to _∋_, so this operation is the identity. For the term kit, _∋/⊢_ is instantiated to _⊢_, so this operation is the variable constructor `_.
- `/id converts a S ∋/⊢ s into a term S ⊢ s and is analogous to the id/` operation.
- wk shifts the de Bruin indices in a variable or term. The new, unused variable zero can assume any sort s'. For variables, wk is the successor suc. For terms, wk means applying a shifting renaming to the term.

```
_→ₖ_ : List (Sort Var) → List (Sort Var) → Set
S₁ →ₖ S₂ = ∀ s → S₁ ∋ s → S₂ ∋/⊢ s

_&_ : S₁ ∋ s → S₁ →ₖ S₂ → S₂ ∋/⊢ s
x & ϕ = ϕ _ x

_↑_ : S₁ →ₖ S₂ → ∀ s → (s :: S₁) →ₖ (s :: S₂)
(ϕ ↑ s) _ zero    = id/' zero
(ϕ ↑ s) _ (suc x) = wk _ (ϕ _ x)

⟪_⟫ : S ∋/⊢ s → (s :: S) →ₖ S
⟪ x/t ⟫ _ zero    = x/t
⟪ x/t ⟫ _ (suc x) = id/' x

weaken : ∀ s → S →ₖ (s :: S)
weaken s _ x = wk s (id/' x)

_~_ : (ϕ₁ ϕ₂ : S₁ →ₖ S₂) → Set
_~_ {S₁} ϕ₁ ϕ₂ = ∀ s (x : S₁ ∋ s) →
                  ϕ₁ s x ≡ ϕ₂ s x

postulate ~-ext : ∀ {ϕ₁ ϕ₂ : S₁ →ₖ S₂}
                  → ϕ₁ ~ ϕ₂ → ϕ₁ ≡ ϕ₂

id : S →ₖ S
id s x = id/' x

id↑~id : (id {S} ↑ s) ~ id {s :: S}
```

**Figure 1** Map Operations.

- '/'-is-' states that converting a variable first to a "variable-or-term" and then further to a term is the same as converting it directly to a term using the variable constructor '_.
- '/id-injective and id/'-injective state that '/id and id/' are injective.
- wk-id/' characterizes the behaviour of the wk function by how it acts on variables: injecting a variable and then shifting it, is the same as injecting a shifted variable.

Figure 1 shows the usual operations for renamings and substitutions. The definitions are included directly in the record module of Kit, so they are implicitly parameterized over a kit. The type $S_1 \to_k S_2$ unifies renamings $S_1 \to_r S_2$ and substitutions $S_1 \to_s S_2$. We call a value of type $S_1 \to_k S_2$ a *map* and use the meta-variable $\phi$ for it. The operation $\phi$ & x applies a map to a variable. The operation $\phi \uparrow s$ lifts a map under a binder of sort s. The operation ⟪ x/t ⟫ constructs a singleton map that replaces zero with x/t and decreases all other variables by one. The weaken map increases all variables by one. The $\phi_1 \sim \phi_2$ type expresses extensional equality of maps. For simplicity, we postulate functional extensionality ~-ext.[2] There is an identity map id. The lemma id↑-id states that a lifted identity map is again an identity map.

To make it easier to talk about a specific kit, we introduce the following notations:

- we write $S_1$ −[ K ]→ $S_2$ for the $S_1 \to_k S_2$ of some specific Kit K; and
- we write S ∋/⊢[ K ] s for the S ∋/⊢ s of some specific Kit K.

The operation of applying a map to a term depends on the concrete object language. It is captured by the following structure:

```
record Traversal : Set₁ where
  field _⋯_ : ∀ ⦃ K : Kit _∋/⊢_ ⦄ → S₁ ⊢ s → S₁ −[ K ]→ S₂ → S₂ ⊢ s
        ⋯-var : ∀ ⦃ K : Kit _∋/⊢_ ⦄ (x : S₁ ∋ s) (ϕ : S₁ −[ K ]→ S₂) → (' x) ⋯ ϕ ≡ '/id (x & ϕ)
        ⋯-id  : ∀ ⦃ K : Kit _∋/⊢_ ⦄ (t : S ⊢ s) → t ⋯ id ⦃ K ⦄ ≡ t
```

[2] The actual implementation does *not* use any postulates. Functional extensionality can be avoided by proving that $\phi_1 \sim \phi_2$ implies $(t \cdots \phi_1) \equiv (t \cdots \phi_2)$, i.e. that if two maps are extensionally equal, then their applications (⋯) to the same term are intensionally equal. The downside of this approach is boilerplate, because for each operation on maps, it needs to be proved that the operation preserves extensional equality, e.g. that $\phi_1 \sim \phi_2$ implies $(\phi_1 \uparrow s) \sim (\phi_2 \uparrow s)$. None of those lemmas are necessary with functional extensionality.

The fields of this structure have the following meaning:

- `t ··· ϕ` applies the map `ϕ` (a renaming or substitution) to the term `t`.
- `···-var` states that applying a map `ϕ` to a variable term `‘ x`, is the same as applying `ϕ` to the variable `x`, and then converting the result to a term via `id/‘`.
- `···-id` states that applying the identity map `id` to a term does not change the term.

Finally, we define the actual kit instances. The variable kit definition is straightforward:

```
Kᵣ : Kit _∋_
Kᵣ = record { id/‘          = λ x → x          ; ‘/id          = ‘_
            ; wk            = λ s’ x → suc x   ; ‘/‘-is-‘      = λ x → refl
            ; id/‘-injective = λ eq → eq       ; ‘/id-injective = ‘-injective
            ; wk-id/‘       = λ s’ x → refl    }
```

The term kit requires both the variable kit and the Traversal to be defined, because shifting a term with wk means applying the shifting renaming to the term. Hence, we define the term kit in the record module of Traversal:

```
Kₛ : Kit _⊢_
Kₛ = record { id/‘          = ‘_                                  ; ‘/id          = λ t → t
            ; wk            = λ s’ t → t ··· weaken {{ Kᵣ }} s’    ; ‘/‘-is-‘      = λ x → refl
            ; id/‘-injective = ‘-injective                         ; ‘/id-injective = λ eq → eq
            ; wk-id/‘       = λ s’ x → ···-var x (weaken {{ Kᵣ }} s’) }
```

## 3.3 Instantiation for System F

In this subsection, we show how to instantiate the Traversal abstraction for System F. In practice, this is done by our reflection algorithm automatically (Section 6), but it can be instructive to see what happens under the hood.

First, we define the operation of applying a map to a term:

```
_···_ : ∀ {{ K : Kit _∋/⊢_ }} → S₁ ⊢ s → S₁ –[ K ]→ S₂ → S₂ ⊢ s
(‘ x)          ··· ϕ = ‘/id (x & ϕ)
(λx t)         ··· ϕ = λx (t ··· (ϕ ↑ ))
(Λα t)         ··· ϕ = Λα (t ··· (ϕ ↑ ≈))
(∀[α: t₁ ] t₂) ··· ϕ = ∀[α: t₁ ··· ϕ ] (t₂ ··· (ϕ ↑ ≈))
(t₁ · t₂)      ··· ϕ = (t₁ ··· ϕ) · (t₂ ··· ϕ)
(t₁ • t₂)      ··· ϕ = (t₁ ··· ϕ) • (t₂ ··· ϕ)
(t₁ ⇒ t₂)      ··· ϕ = (t₁ ··· ϕ) ⇒ (t₂ ··· ϕ)
⋆              ··· ϕ = ⋆
```

The interesting cases are those with variables and binders:

- In the variable case `(‘ x) ··· ϕ`, we first apply the map `ϕ` to the variable `x`. If `ϕ` is a renaming, we get back a variable and need to apply the variable constructor `‘_`. If `ϕ` is a substitution, we get back a term that we can use directly. This is exactly what `‘/id` does.
- In cases where the operation needs to go under a binder, like `(λx e) ··· ϕ`, we lift the map using `_↑_` to account for the bound variable before we apply it to the subterm.

We then prove that applying an identity map does not change the term:

```
···-id : ∀ {{ K : Kit _∋/⊢_ }} (t : S ⊢ s) → t ··· id ≡ t
```

```
···-id (` x)   = `/`-is-` x
···-id (λx e) = λx (e ··· (id ↑ )) ≡⟨ cong (λ ϕ → λx (e ··· ϕ)) (~-ext id↑~id) ⟩
                λx (e ··· id)      ≡⟨ cong (λ e → λx e) (···-id e) ⟩
                λx e               ■
```

We only display and discuss the interesting cases:

- in the variable case ` x, the id/` from the identity map meets the `/id from the traversal operation, so we need to use `/`-is-`.
- in the lambda abstraction case λx e, the traversal lifts the identity under its binder. Here we need to use id↑-id to show that a lifted identity map is again an identity map.

Finally, we instantiate the Traversal structure:

```
SystemF-Traversal : Traversal
SystemF-Traversal = record { _···_ = _···_ ; ···-id = ···-id ; ···-var = λ x ϕ → refl }
```

## 3.4 Extension Kits

As we defined the Kit structure before the Traversal structure, the fields of Kit could not use map application _···_ in their types. This prevented us to include another useful axiom into the Kit structure. As this axiom also needs to be proved separately for variables and terms, we define a new structure for it which extends a Kit:

```
record WkKit (K : Kit _∋/⊢_): Set₁ where
field wk-`/id : ∀ s {S s'} (x/t : S ∋/⊢ s') → `/id x/t ··· weaken s ≡ `/id (wk s x/t)
```

The wk-`/id axiom explains the wk function by how it acts on *terms*. It is the counterpiece to the Kit axiom wk-id/`, which explains the wk function by how it acts on *variables*. This lemma is useful for proving extensional equalities between maps involving weakening, where `/id-injective allows to add `/id on both sides of the equation, such that wk-`/id can be used to make further progress. The instantiations of the WkKit are straightforward:

```
Wᵣ : WkKit Kᵣ ; Wₛ : WkKit Kₛ
Wᵣ = record { wk-`/id = λ s x → ···-var x (weaken s) }
Wₛ = record { wk-`/id = λ s t → refl }
```

As the variable and term Kits are the only two Kits, and both have WkKit instances, it is always safe to assume that a Kit also supports the WkKit extension.

# 4 Map Composition

In this section, we extend our framework with an abstraction for the composition of arbitrary maps. The core property of composition is the fusion lemma, which states that applying two maps $\phi_1$ and $\phi_2$ in sequence to a term t, is the same as applying their composition $(\phi_1 \cdot_k \phi_2)$ to t, i.e. $(t \cdots \phi_1) \cdots \phi_2 \equiv t \cdots (\phi_1 \cdot_k \phi_2)$. This property gives our framework the ability to reason about the application of multiple maps by reasoning about the application of a single map. As such it forms the basis for all lemmas involving multiple maps, e.g. that applying a weakening and then a singleton substitution cancel each other out.

As we defined substitution in terms of renamings, we need to consider all four compositions between renamings and substitutions. While the composition operations can be defined independently of each other, the fusion lemma for two substitutions, depends on the fusion lemmas for a renaming and a substitution, which in turn depend on the fusion lemma for two renamings.

Previous work on kits[6] addresses this issue by duplicating the definitions and using tactics to reduce boilerplate in proofs. In contrast, we define structures similar to Kit and Traversal, which allow us to abstract over all four compositions and use the same trick as before to eliminate the dependencies. With the help of a general map composition, we can prove lemmas about the interactions of general maps, which is crucial for proving a type preservation lemma for general map application instead of individual lemmas for renamings and substitutions.

## 4.1 An Examination of Composition

To motivate our abstraction, we first look at the four compositions individually:

```
_r·r_ : (S₁ →r S₂) → (S₂ →r S₃) → (S₁ →r S₃)
_r·s_ : (S₁ →r S₂) → (S₂ →s S₃) → (S₁ →s S₃)
_s·r_ : (S₁ →s S₂) → (S₂ →r S₃) → (S₁ →s S₃)
_s·s_ : (S₁ →s S₂) → (S₂ →s S₃) → (S₁ →s S₃)
```

```
(ϕ₁ r·r ϕ₂) _ x = (x & ϕ₁) & ϕ₂
(ϕ₁ r·s ϕ₂) _ x = (x & ϕ₁) & ϕ₂
(ϕ₁ s·r ϕ₂) _ x = (x & ϕ₁) ⋯ ϕ₂
(ϕ₁ s·s ϕ₂) _ x = (x & ϕ₁) ⋯ ϕ₂
```

The definitions reveal two interesting properties:

1. If we compose two maps $\phi_1$ and $\phi_2$, then the resulting map is a renaming, iff both $\phi_1$ and $\phi_2$ are renamings. In other words: if $\phi_1$ is a $K_1$-map and $\phi_2$ is a $K_2$-map, then the result is a $(K_1 \sqcup K_2)$-map, where $\sqcup$ refers to the lattice for $\{ K_r , K_s \}$ generated by $K_r < K_s$.
2. All four compositions first apply $\phi_1$ to $x$, and then apply $\phi_2$ to the result. If $\phi_1$ is a renaming, this result is another variable, but if $\phi_1$ is a substitution, this result is a term.

With the Kit abstraction, we can easily abstract over $\phi_2$ being a renaming or a substitution:

```
_r·_ : (S₁ →r S₂) → (S₂ –[ K ]→ S₃) → (S₁ –[ K ]→ S₃)
_s·_ : (S₁ →s S₂) → (S₂ –[ K ]→ S₃) → (S₁ →s   S₃)
```

```
(ϕ₁ r· ϕ₂) _ x = (x & ϕ₁) & ϕ₂
(ϕ₁ s· ϕ₂) _ x = (x & ϕ₁) ⋯ ϕ₂
```

But to abstract over $\phi_1$, the Kit abstraction is not sufficient: while it allows us to abstract over what we are applying, i.e. a renaming or a substitution, it does not allow us to abstract over what we are applying it to, i.e. a variable or a term. For the latter we have two distinct operations _&_ and _⋯_.

To fill this gap, we introduce a new abstraction that we call a *compose kit* (CKit), which provides an operation _&/⋯_ that unifies _&_ and _⋯_. This allows us to define a general composition as follows:

```
_·k_ : S₁ –[ K₁ ]→ S₂ → S₂ –[ K₂ ]→ S₃ → S₁ –[ K₁⊔K₂ ]→ S₃
(ϕ₁ ·k ϕ₂) _ x = (x & ϕ₁) &/⋯ ϕ₂
```

## 4.2 An Abstraction for Composition

A compose kit CKit $K_1$ $K_2$ $K_1 \sqcup K_2$ describes the operations necessary for definining the composition of a $K_1$-map with a $K_2$-map that results in a $K_1 \sqcup K_2$-map:

```
record CKit (K₁ : Kit _∋/⊢₁_) (K₂ : Kit _∋/⊢₂_) (K₁⊔K₂ : Kit _∋/⊢_) : Set where
field _&/⋯_    : S₁ ∋/⊢[ K₁ ] s → S₁ –[ K₂ ]→ S₂ → S₂ ∋/⊢[ K₁⊔K₂ ] s
      &/⋯-⋯    : (x/t : S₁ ∋/⊢[ K₁ ] s) (ϕ : S₁ –[ K₂ ]→ S₂) →
                 ‘/id (x/t &/⋯ ϕ) ≡ ‘/id x/t ⋯ ϕ
      &/⋯-wk-↑ : (x/t : S₁ ∋/⊢[ K₁ ] s) (ϕ : S₁ –[ K₂ ]→ S₂) →
                 wk s’ (x/t &/⋯ ϕ) ≡ wk s’ x/t &/⋯ (ϕ ↑ s’)
```

The third parameter $K_1 \sqcup K_2$ can be seen as a functional dependency[12] and is determined by the choice of $K_1$ and $K_2$. The fields of a compose kit have the following meaning:

- The _&/⋯_ operation takes a variable or term x/t (according to $K_1$) and a renaming or substitution $\phi$ (according to $K_2$) and applies $\phi$ to x/t resulting in a variable or term (according to $K_1 \sqcup K_2$). From this operation we derive map composition _ ·ₖ _ as shown in the previous subsection.
- The &/⋯-⋯ lemma describes the behavior of _&/⋯_ in terms of _⋯_, allowing subsequent lemmas to make use of the lemmas that we have already proved for _⋯_.
- The &/⋯-wk-↑ lemma states that applying a map and then weakening is the same as weakening first and then lifting the map over the variable introduced by the weakening. From this lemma, we can derive that lifting distributes over composition:

```
dist-↑-· : ∀ s (ϕ₁ : S₁ –[ K₁ ]→ S₂) (ϕ₂ : S₂ –[ K₂ ]→ S₃) →
           ((ϕ₁ ·ₖ ϕ₂) ↑ s) ~ ((ϕ₁ ↑ s) ·ₖ (ϕ₂ ↑ s))
```

A CTraversal provides a fusion lemma that works for the composition of any CKit:

```
record CTraversal : Set₁ where
  field fusion :
        ∀ {{ K₁ : Kit _∋/⊢₁_ }} {{ K₂ : Kit _∋/⊢₂_ }} {{ K : Kit _∋/⊢_ }} {{ W₁ : WkKit K₁ }}
          {{ C : CKit K₁ K₂ K }} (t : S₁ ⊢ s) (ϕ₁ : S₁ –[ K₁ ]→ S₂) (ϕ₂ : S₂ –[ K₂ ]→ S₃) →
        (t ⋯ ϕ₁) ⋯ ϕ₂ ≡ t ⋯ (ϕ₁ ·ₖ ϕ₂)
```

Given a CTraversal, we can prove the usual lemmas about interactions of multiple maps:

- A map $\phi$ followed by a weakening is equivalent to a weakening followed by $\phi$ that has been lifted over the weakened variable:

```
⋯-↑-wk : ∀ {{ K : Kit _∋/⊢_ }} {{ W : WkKit K }} {{ C₁ : CKit K Kᵣ K }} {{ C₂ : CKit Kᵣ K K }}
           (t : S₁ ⊢ s) (ϕ : S₁ –[ K ]→ S₂) s →
         t ⋯ ϕ ⋯ weaken s ≡ t ⋯ weaken s ⋯ (ϕ ↑ s)
```

- A weakening followed by a singleton substitution act as an identity map:

```
wk-cancels-⟪⟫-⋯ : ∀ {{ K : Kit _∋/⊢_ }} (t : S ⊢ s') (x/t : S ∋/⊢[ K ] s) →
                  t ⋯ weaken s ⋯ ⟪ x/t ⟫ ≡ t
```

- A singleton map can be swapped with any map $\phi$:

```
dist-↑-⟪⟫-⋯ : ∀ {{ K₁ : Kit _∋/⊢₁_ }} {{ K₂ : Kit _∋/⊢₂_ }} {{ K : Kit _∋/⊢_ }}
                {{ W₁ : WkKit K₁ }} {{ W₂ : WkKit K₂ }}
                {{ C₁ : CKit K₁ K₂ K }} {{ C₂ : CKit K₂ K K }}
                (t : (s :: S₁) ⊢ s') (x/t : S₁ ∋/⊢[ K₁ ] s) (ϕ : S₁ –[ K₂ ]→ S₂) →
              t ⋯ ⟪ x/t ⟫ ⋯ ϕ ≡ t ⋯ (ϕ ↑ s) ⋯ ⟪ x/t &/⋯ ϕ ⟫
```

Similarly, as it was the case for the Kit and Traversal structures, the idea is that we instantiate the CTraversal for our object language, and in return the framework defines the concrete CKit instances for us. Hence, we define the CKit instances in the record module of CTraversal:

```
Cᵣ : {{ K : Kit _∋/⊢_ }} → CKit Kᵣ K K
Cᵣ = record { _&/⋯_   = _&_
            ; &/⋯-⋯   = λ x ϕ → sym (⋯-var x ϕ)
            ; &/⋯-wk-↑ = λ x ϕ → refl }
Cₛ : {{ K : Kit _∋/⊢_ }} {{ C : CKit K Kᵣ K }} {{ W : WkKit K }} → CKit Kₛ K Kₛ
Cₛ = record { _&/⋯_   = _⋯_
            ; &/⋯-⋯   = λ t ϕ → refl
            ; &/⋯-wk-↑ = λ t ϕ → ⋯-↑-wk t ϕ _ }
```

$C_r$ is the compose kit between a renaming and another kit $K$. $C_s$ is the compose kit between a substitution and another kit $K$, and requires that we already know how to compose a $K$-map with a renaming. The following verifies that $C_r$ and $C_s$ indeed get us all four compositions:

```
Crr : CKit Kr Kr Kr
Crr = Cr ⦃ K = Kr ⦄
```
```
Crs : CKit Kr Ks Ks
Crs = Cr ⦃ K = Ks ⦄
```
```
Csr : CKit Ks Kr Ks
Csr = Cs ⦃ C = Crr ⦄
```
```
Css : CKit Ks Ks Ks
Css = Cs ⦃ C = Csr ⦄
```

## 4.3 Instantiation for System F

In this subsection, we show how to instantiate the CTraversal abstraction for System F. In practice, this is done by our reflection algorithm automatically (Section 6), but it can be instructive to see, as it motivates the axioms of the CKit.

```
fusion : ∀ ⦃ K₁ : Kit _∋/⊢₁_ ⦄ ⦃ K₂ : Kit _∋/⊢₂_ ⦄ ⦃ K : Kit _∋/⊢_ ⦄ ⦃ W₁ : WkKit K₁ ⦄
           ⦃ C : CKit K₁ K₂ K ⦄ (t : S₁ ⊢ s) (ϕ₁ : S₁ –[ K₁ ]→ S₂) (ϕ₂ : S₂ –[ K₂ ]→ S₃) →
         (t ⋯ ϕ₁) ⋯ ϕ₂ ≡ t ⋯ (ϕ₁ ·ₖ ϕ₂)
fusion (` x)   ϕ₁ ϕ₂ = sym (&/⋯-⋯ (ϕ₁ _ x) ϕ₂)
fusion (λx t) ϕ₁ ϕ₂ =
  λx ((t ⋯ (ϕ₁ ↑)) ⋯ (ϕ₂ ↑))  ≡⟨ cong (λ t → λx t) (fusion t (ϕ₁ ↑) (ϕ₂ ↑)) ⟩
  λx (t ⋯ ((ϕ₁ ↑) ·ₖ (ϕ₂ ↑))) ≡⟨ cong (λ ϕ → λx (t ⋯ ϕ)) (sym (~-ext (dist-↑-· ϕ₁ ϕ₂))) ⟩
  λx (t ⋯ ((ϕ₁ ·ₖ ϕ₂) ↑))     ∎
```

We only show the interesting cases, which are:

- variables, where we need to use the &/⋯-⋯ lemma provided by the CKit; and
- bindings, where the traversal operation _⋯_ needs to lift the map via _↑_, requiring us to distribute the lifting over the composition using dist-↑-·.

# 5 Types & Typing

## 5.1 Types

In the context of multi-sorted syntax, the notion of a type can be described as a mapping between sorts. For System F, the expression sort  maps to the type sort ≈, which in turn maps to the kind sort ⌝. The following structure is used to teach our framework about types:

```
record Types : Set₁ where
  field ↑ᵗ : ∀ {st} → Sort st → ∃[ st' ] Sort st'
```

For System F, the instantiation is

```
SystemF-Types : Types
SystemF-Types = record { ↑ᵗ = λ {  → _ , ≈ ; ≈ → _ , ⌝ ; ⌝ → _ , ⌝ } }
```

There are two things to discuss:

1. The ↑ᵗ function maps a sort of arbitrary sort type, to a sort of a potentially different sort type, which is expressed by the use of an existential. For System F we require this generality, as the sort ≈ can have variables, whereas its corresponding type sort ⌝ cannot.
2. Some sorts, like ⌝, do not have corresponding type sorts, but we still need to provide one, as ↑ᵗ is a total function. For such sorts, we can simply use arbitrary sort types, as the formalization will have no typing rules that use them.

To hide the existential, we define S :⊢ s, which represents the type for a term S ⊢ s.

```
_:⊢_ : ∀ {t} → List (Sort Var) → Sort t → Set
S :⊢ s = S ⊢ proj₂ (↑ᵗ s)
```

## 5.2 Type Contexts

Equipped with a notion of types, we are ready to define type contexts. As we want our framework to support dependent types, we allow a type in the context to use all variables bound previously in the context:

```
data Ctx : List (Sort Var) → Set where
  []    : Ctx []
  _::_ : S ∣⊢ s → Ctx S → Ctx (s :: S)
```

When looking up the type of a variable, we need to weaken it for each binding that comes after the variable[3]

```
lookup : Ctx S → S ∋ s → S ∣⊢ s
lookup (t :: Γ) zero    = t ⋯ weaken ⦃ Kᵣ ⦄ _
lookup (t :: Γ) (suc x) = lookup Γ x ⋯ weaken ⦃ Kᵣ ⦄ _
```

Finally, a variable typing $\Gamma \ni x : t$ states that looking up $x$ in $\Gamma$ yields $t$:

```
_∋_:_ : Ctx S → S ∋ s → S ∣⊢ s → Set
Γ ∋ x : t = lookup Γ x ≡ t
```

## 5.3 Typing

Now that we have a notion of types and type contexts, we are ready to define the multi-sorted typing relation for System F, which describes both typing and kinding:

```
data _⊢_:_ : Ctx S → S ⊢ s → S ∣⊢ s → Set where
  ⊢`  : ∀ {x : S ∋ s} {T : S ∣⊢ s} → Γ ∋ x : T → Γ ⊢ ` x : T
  ⊢λ  : ∀ {e : (_ :: S) ⊢ _} → (t₁ :: Γ) ⊢ e : (wk t₂) → Γ ⊢ λx e : t₁ ⇒ t₂
  ⊢Λ  : (k :: Γ) ⊢ e : t₂ → Γ ⊢ Λα e : ∀[α: k ] t₂
  ⊢·  : Γ ⊢ e₁ : t₁ ⇒ t₂ → Γ ⊢ e₂ : t₁ → Γ ⊢ e₁ · e₂ : t₂
  ⊢•  : ∀ {Γ : Ctx S} → (k₂ :: Γ) ⊢ t₁ : k₁ → Γ ⊢ t₂ : k₂ → Γ ⊢ e₁ : ∀[α: k₂ ] t₁ →
        Γ ⊢ e₁ • t₂ : t₁ ⋯ ⦇ t₂ ⦈
  ⊢τ  : Γ ⊢ t : ⋆
```

The interesting cases are:

- the variable rule $\vdash`$, which covers both expression- and type-variables, analogously to the variable term constructor;
- the lambda rule $\vdash\lambda$, which weakens the codomain type $t_2$. This is necessary, because multi-sorted syntax allows types to depend on expressions, so the typing derivation for $e$ has to account for a variable, which is not used by the type; and
- the kinding rule $\vdash\tau$ states that all types have kind $\star$. This is sufficient for System F as types are automatically well-kinded due to intrinsic scoping.

To teach the framework about typing, we create a structure analogously to Syntax:

```
record Typing : Set₁ where
  field _⊢_:_ : ∀ {s : Sort st} → Ctx S → S ⊢ s → S ∣⊢ s → Set
        ⊢`    : ∀ {Γ : Ctx S} {x : S ∋ s} {t} → Γ ∋ x : t → Γ ⊢ ` x : t
```

The instantiation for System F is straightforward:

```
SystemF-Typing : Typing
SystemF-Typing = record { _⊢_:_ = _⊢_:_ ; ⊢` = ⊢` }
```

[3] This includes the variable itself, which is not allowed to appear in its own type.

## 5.4 An Abstraction for Type Preservation

By now, the reader probably knows what comes next: we build an abstraction to unify type preservation for renamings and substitutions, eliminating the dependencies by yet another type of kits.

We start with TKits, which abstract over variable and term *typing*, and then define a TTraversal, which provides substitution-preserves-typing for all TKits.

```
record TKit (K : Kit _∋/⊢_) : Set₁ where
  field _∋/⊢_:_ : Ctx S → S ∋/⊢ s → S :⊢ s → Set
        id/⊢‘   : ∀ {t : S :⊢ s} {Γ : Ctx S} → Γ ∋ x : t → Γ ∋/⊢ id/‘ x : t
        ⊢‘/id   : ∀ {e : S ∋/⊢ s} {t : S :⊢ s} {Γ : Ctx S} → Γ ∋/⊢ e : t → Γ ⊢ ‘/id e : t
        ⊢wk     : ∀ (Γ : Ctx S) (t’ : S :⊢ s) (e : S ∋/⊢ s’) (t : S :⊢ s’) →
                  Γ ∋/⊢ e : t → (t’ :: Γ) ∋/⊢ wk _ e : (t ⋯ weaken _)
```

The first field abstracts over variable and term typing. The other fields express typings for the fields of a Kit. Building on the fields of a TKit, we define map typing and type preservation for map lifting and the singleton map in the record module of TKit:

- Using the variable/term typing, we can define a renaming/substitution typing:

  ```
  _:_⇒ₖ_ : S₁ –[ K ]→ S₂ → Ctx S₁ → Ctx S₂ → Set
  _:_⇒ₖ_ {S₁} {S₂} ϕ Γ₁ Γ₂ = ∀ {s₁} (x : S₁ ∋ s₁) (t : S₁ :⊢ s₁) →
    Γ₁ ∋ x : t → Γ₂ ∋/⊢ (x & ϕ) : (t ⋯ ϕ)
  ```

  $\phi : \Gamma_1 \Rightarrow_k \Gamma_2$ states that $\phi$ is a map that takes terms from $\Gamma_1$ to terms in $\Gamma_2$.

- Lifting a map preserves its typing:

  ```
  _⊢↑_ : ∀ ⦃ W : WkKit K ⦄ ⦃ C₁ : CKit K Kᵣ K ⦄
           {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {ϕ : S₁ –[ K ]→ S₂} →
         ϕ : Γ₁ ⇒ₖ Γ₂ → (t : S₁ :⊢ s) → (ϕ ↑ s) : (t :: Γ₁) ⇒ₖ ((t ⋯ ϕ) :: Γ₂)
  ```

- If a variable/term has a typing, then so does its singleton renaming/substitution:

  ```
  ⊢⦇_⦈ : ∀ {s S} {Γ : Ctx S} {x/t : S ∋/⊢ s} {T : S :⊢ s} →
         Γ ∋/⊢ x/t : T → ⦇ x/t ⦈ : (T :: Γ) ⇒ₖ Γ
  ```

We then define a TTraversal analogously to Traversal, but instead of defining the application of maps, it defines that the application of a typed map to a typed term yields a typed term:

```
record TTraversal : Set₁ where
  field _⊢⋯_ : ∀ ⦃ K : Kit _∋/⊢_ ⦄ ⦃ W : WkKit K ⦄ ⦃ TK : TKit K ⦄
                ⦃ C₁ : CKit K Kᵣ K ⦄ ⦃ C₂ : CKit K K K ⦄ ⦃ C₃ : CKit K Kₛ Kₛ ⦄
                {S₁ S₂ st} {Γ₁ : Ctx S₁} {Γ₂ : Ctx S₂} {s : Sort st}
                {e : S₁ ⊢ s} {t : S₁ :⊢ s} {ϕ : S₁ –[ K ]→ S₂} →
              Γ₁ ⊢ e : t →
              ϕ : Γ₁ ⇒ₖ Γ₂ →
              Γ₂ ⊢ (e ⋯ ϕ) : (t ⋯ ϕ)
```

Given a term e with typing $\vdash$e and a renaming/substitution $\phi$ with typing $\vdash\phi$, the term $\vdash$e $\vdash\cdots$ $\vdash\phi$ is a typing for e $\cdots$ $\phi$.

As before, we define the TKit instances in the record module of TTraversal:

```
TKᵣ : TKit Kᵣ ; TKₛ : TKit Kₛ
TKᵣ = record { _∋/⊢_:_ = _∋_:_      ; ⊢‘/id = ⊢‘
             ; id/⊢‘   = λ ⊢x → ⊢x ; ⊢wk   = λ { Γ t’ x t refl → refl } }
TKₛ = record { _∋/⊢_:_ = _⊢_:_ ; ⊢‘/id = λ ⊢x → ⊢x
             ; id/⊢‘   = ⊢‘     ; ⊢wk   = λ Γ t’ e t ⊢e → ⊢e ⊢⋯ ⊢wk Γ t’ }
```

The large amount of kit-parameters of _⊢⋯_ does not impose any restriction, as both our Kits support the WkKit extension and can be composed arbitrarily. Agda's instance resolution allows us to easily instantiate a concrete substitution-preserves-typing lemma:

```
_⊢⋯ₛ_ : Γ₁ ⊢ e : t → σ : Γ₁ ⇒ₛ Γ₂ → Γ₂ ⊢ (e ⋯ σ) : (t ⋯ σ)
_⊢⋯ₛ_ = _⊢⋯_
```

## 5.5 Instantiation for System F

In this subsection, we show how to instantiate the TTraversal abstraction for System F. This is the only structure that is *not* instantiated automatically via reflection, as typing relations can be arbitrary complex in general.

```
⊢‘ ⊢x                                  ⊢⋯ ⊢φ = ⊢‘/id (⊢φ _ _ ⊢x)
⊢λ {t₂ = t₂} ⊢e                        ⊢⋯ ⊢φ = ⊢λ (subst (λ t → _ ⊢ _ : t)
                                                    (sym (⋯-↑-wk t₂ _ _))
                                                    (⊢e ⊢⋯ (⊢φ ⊢↑ _)))
⊢Λ ⊢e                                  ⊢⋯ ⊢φ = ⊢Λ (⊢e ⊢⋯ (⊢φ ⊢↑ _))
⊢ · ⊢e₁ ⊢e₂                            ⊢⋯ ⊢φ = ⊢ · (⊢e₁ ⊢⋯ ⊢φ) (⊢e₂ ⊢⋯ ⊢φ)
⊢• {t₁ = t₁} {t₂ = t₂} ⊢t₁ ⊢t₂ ⊢e₁      ⊢⋯ ⊢φ = subst (λ t → _ ⊢ _ : t)
                                                    (sym (dist-↑-⦅⦆⋯ t₁ t₂ _))
                                                    (⊢• (⊢t₁ ⊢⋯ (⊢φ ⊢↑ _))
                                                        (⊢t₂ ⊢⋯ ⊢φ) (⊢e₁ ⊢⋯ ⊢φ))
⊢τ                                     ⊢⋯ ⊢φ = ⊢τ
```

The type of _⊢⋯_ is the same as in the record definition and hence omitted. The interesting parts of the proof are:

- There is a strong similarity to the instantiation of map traversal _⋯_: where _⋯_ used ‘/id or _↑_, our _⊢⋯_ uses their preservation lemmas ⊢‘/id or _⊢↑_.
- The lambda typing constructor ⊢λ weakens the type $t_2$ to shield it from expression-substitution, requiring us to use ⋯-↑-wk to move the map under the weakening.
- The type application constructor ⊢• substitutes $t_1$ into $t_2$, requiring us to use dist-↑-⦅⦆ to move the map under the singleton substitution.

# 6 Reflection & Generics

We use Agda's reflection mechanism to derive instantiations related to all structures for untyped substitution, i.e. Syntax, Traversal and CTraversal. The user only needs to define syntax and typing, and can then move on to proving that substitution preserves typing, where all substitution lemmas are already available.

To gain insight into the class of object languages supported by our reflection algorithm, we have instantiated the structures for a generic syntax similar to the one in Allais et al.[2]. Our reflection algorithm derives proofs with the same structure as the generic proofs, giving high confidence that it covers the same class of languages.

Informally, all objects languages with multi-sorted syntax are supported that

1. have a variable constructor of type ∀ {S s} → S ∋ s → S ⊢ s;
2. use subterms only directly (e.g. not in lists); and
3. only extend the scope-context of subterms, but never modify it otherwise.

The formal definition of the generic syntax can be found in the supplementary material. Restriction 2 is purely technical and can be lifted with a more sophisticated reflection

algorithm. In the current system, this restriction can be worked around by inlining the data structure constructors into the syntax definition as terms of a new sort. For example, to allow lists of terms, we can add a polymorphic list sort

```
⋖⊐∼≈ : Sort st → Sort NoVar
```

and corresponding syntax constructors for lists

```
[]    : S ⊢ ⋖⊐∼≈ s
_::_ : S ⊢ s → S ⊢ ⋖⊐∼≈ s → S ⊢ ⋖⊐∼≈ s
```

This allows us to model, e.g., a multi-argument function call expression as

```
call : S ⊢  → S ⊢ ⋖⊐∼≈  → S ⊢
```

Function types cannot be inlined, but require an extension of the reflection algorithm.

## 7 Case Studies

Using our full implementation of the framework, we proved subject reduction for the following object languages:

- For lambda calculus with dependent function types, the framework works out of the box for both deriving untyped substitution and instantiating the TTraversal. As types are terms, only a single sort is required. In the proof of confluence, the CKit abstraction allowed us to unify lemmas about the reduction of renamings and substitutions.
- For System F with subtyping, the main challenges are how to represent subtyping constraints and how to deal with the fact that substitution-preserves-typing is not generally true, as type variables have subtyping bounds that need to be respected. While it would be possible to use our framework only to derive untyped substitution and define typing contexts and type preservation lemmas by hand, we instead used an encoding that allows us to use the the TKit abstraction directly. Instead of binding a type variable with a subtyping constraint $\alpha <: \mathsf{t}$, we first bind the type variable as $\alpha : \star$, and then bind the constraint as $\mathsf{c} : (\alpha <: \mathsf{t})$. This description is similar to first-class constraints, but restricted enough to be isomorphic to the original formalization, as constraint variables cannot be accessed by the user. With this encoding, substitution-preserves-typing is generally true again: replacing a type variable $\alpha <: \mathsf{t}$ with a type $\mathsf{t}'$, which is not a subtype of $\mathsf{t}$, results in a term that is still well-typed, but in a context with an unsatisfiable constraint $\mathsf{t}' <: \mathsf{t}$.
- Object languages with pattern matching can be modeled by adding the sorts of the variables bound by a pattern to the pattern sort $\mathsf{Pat}$ itself. A pattern matching clause $\mathsf{p} \Rightarrow \mathsf{e}$ can then be defined as

  $$\_\Rightarrow\_ : \mathsf{S} \vdash_{\mathsf{Pat}} \mathsf{S}' \to (\mathsf{S}' +\!\!+ \mathsf{S}) \vdash  \to \mathsf{S} \vdash$$

  where $\mathsf{S}'$ describes the variables bound by the pattern and  is the sort of a clause.

## 8 Related Work

As the amount of related work is rather large, we focus on work that is closely related to ours and refer to other papers for the broader picture[21, 2].

## 8.1 Variable Binding

There is a plethora of different methods for representing variable binders: de Bruijn indices[10], co de Bruijn indices[16], locally nameless[8], locally named[17], higher order abstract syntax[18] and its parametric variant[9], nominal logic[23], shifted names[11], nameless painless[19], and scope graphs[24]. Many of these representations have been studied in solutions to the POPLMark challenges[4, 1].

## 8.2 Unifying Renamings & Substitution

The kit abstraction for unifying renamings and substitutions appeared first in an unpublished manuscript by McBride[15], and later in Benton et al.[6]. Wood and Atkey[25] propose an extension to kits that supports linear types via resource vectors. In all three cases kits are formulated for intrinsic typing and scenarios with polymorphism are not considered.

## 8.3 Extrinsically Typed Approaches

*Autosubst*[20] is a Coq framework, which derives parallel substitution definitions and lemmas for languages from annotated Coq syntax definitions using extrinsic typing, extrinsic scoping, and de Bruijn indices. The framework is implemented in Coq's tactic language Ltac and comes with a decision procedure for all assumption-free, equational substitution-lemmas. The implementation of Autosubst deals with multiple variable sorts by generating multiple substitutions and corresponding interaction lemmas.

*Autosubst 2*[21] is a standalone code generator, which translates second-order HOAS specifications into mutual inductive term sorts. Compared to Autosubst 1, it features mutually recursive object languages, intrinsic scoping, and vectorized substitutions. Compared to our work, the syntax they generate takes the form of what we described as *unsorted syntax* in Section 2, i.e. different syntactic categories are described by different types with different amount of indices for variable counts. To eliminate the need for interaction lemmas, they define the notion of *vectorized substitution*, which combines the individual substitutions by putting them in a vector. We believe their great work of creating a decision procedure for vectorized substitutions should also translate to our setting with multi-sorted substitutions.

*Needle and Knot*[13] is a code generator for unscoped syntax with de Bruijn indices. They generate substitution and interaction lemmas for single-pointed substitution for languages with multiple variable sorts and binders that bind lists of variables.

All of the above work does not provide machinery to model typing and type preservation and does not unify renaming and substitution and their compositions. Hence, type preservation needs to be modeled manually and individually for renamings and substitutions.

## 8.4 Intrinsically Typed Approaches

Allais et al.[3] propose a powerful abstraction for denotational semantics and semantic fusion lemmas. In later work[2], they use generic programming to instantiate this abstraction for a class of object languages comparable to ours. They demonstrate how both renamings and substitutions can be described as semantics, how the four composition lemmas follow from their generic fusion lemma, and also provide an abstraction to unify renamings and substitutions. They show how to use their framework for both intrinsic and extrinsic typing, but are missing a story for polymorphism.

With only a slight modification to their framework, we can instantiate it for multi-sorted syntax, enabling the definition of polymorphic languages. However, as the intrinsic typing is then used to describe syntactic categories (and not the actual typing), the semantic

abstractions then refer to untyped terms, so typing and type preservation lemmas have to be modeled entirely manually. We believe it would be worthwhile to explore how their semantic abstractions can be lifted to typing relations similar to how our typing kits lift regular kits from terms to typing relations.

## 8.5 Pure Type Systems

Pure Type Systems[5, 7, 22] describe a class of typed lambda calculi parameterized over a set of sorts, dependencies between sorts, and rules for quantification. While pure type systems may seem very similar to multi-sorted syntax, they are actually quite different:

- In multi-sorted syntax, sorts describe syntactic categories of terms. Terms of different sorts are kept syntactically different, e.g. the set of expressions $S \vdash$ and types $S \vdash \approx$ in our System F example.
- In pure type systems, sorts are universe types, e.g. like Set in Agda or Prop in Coq. Terms which have different sorts as types, do still belong to the same syntactic category.

We can model pure type systems as a multi-sorted syntax with a single sort, where the sorts of the pure type system occur as terms representing universe types.

# 9 Conclusion

We have presented an Agda framework, which automatically derives definitions and lemmas for untyped substitution, and provides an abstraction for proving type preservation of renaming and substitution for all syntactic categories with a single lemma (_⊢⋯_).

Compared to many extrinsically typed approaches, our framework also models typing and type preservation. Compared to many intrinsically typed approaches, our framework gracefully extends to polymorphic scenarios.

The main limitation of our framework is the shape of typing relations, similarly as it is the case with approaches based on intrinsic typing: we can only model classical ternary typing relations. To adapt our framework to more complicated typing relations, the machinery for untyped substitution can be reused, but the abstractions related to typing need to be modified. We found that this works surprisingly well in practice, where we have already made extended typing abstractions that support linear typing ala Wood and Atkey[25] and general substructural typing ala Licata et al[14]. In both cases, the typing relation is extended with a fourth component that models usage restrictions on the type context.

While intrinsic typing allows to unify definitions with their type preservation proofs, extrinsic typing allows to unify substitutions across different syntactic categories, as we have demonstrated. We believe this makes our framework particularly suited for polymorphic languages, where the downside of extrinsic typing is automated away, and where we have variables across multiple syntactic categories, so the benefits of a unified substitution bear fruits.

## References

1 Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. Poplmark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019. doi:10.1017/S0956796819000170.

2 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1--90:30, 2018. doi:10.1145/3236785.

3 Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 195--207. ACM, 2017. doi:10.1145/3018610.3018613.

4 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50--65. Springer, 2005. doi:10.1007/11541868_4.

5 Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 1--35. Springer, 1990. doi:10.1007/3-540-52592-0_53.

6 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141--159, 2012. doi:10.1007/s10817-011-9219-0.

7 S Berardi. Towards a mathematical analysis of the coquand-huet calculus of constructions and the other systems in barendregt's cube. dept. *Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, Italy*, 1988.

8 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363--408, 2012. doi:10.1007/S10817-011-9225-2.

9 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143--156. ACM, 2008. doi:10.1145/1411204.1411226.

10 Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381--392. Elsevier, 1972.

11 Stephen Dolan and Leo White. Syntax with shifted names. In *TyDe Workshop*, 2019. URL: http://tydeworkshop.org/2019-abstracts/paper16.pdf.

12 Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, pages 230--244. Springer, 2000. doi:10.1007/3-540-46425-5_15.

13 Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & knot: Binder boilerplate tied up. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 419--445. Springer, 2016. doi:10.1007/978-3-662-49498-1_17.

14 Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 25:1--25:22. Schloss Dagstuhl -- Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.FSCD.2017.25.

15 Conor McBride. Type-preserving renaming and substitution. Unpublished manuscript, 2005. URL: http://strictlypositive.org/ren-sub.pdf.

16 Conor McBride. Everybody's got to be somewhere. In Robert Atkey and Sam Lindley, editors, *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming,*

*MSFP@FSCD 2018, Oxford, UK, 8th July 2018*, volume 275 of *EPTCS*, pages 53--69, 2018. doi:10.4204/EPTCS.275.6.

17 James McKinna and Robert Pollack. Pure type systems formalized. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 289--305. Springer, 1993. doi:10.1007/BFB0037113.

18 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199--208. ACM, 1988. doi:10.1145/53990.54010.

19 Nicolas Pouillard. Nameless, painless. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 320--332. ACM, 2011. doi:10.1145/2034773.2034817.

20 Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359--374. Springer, 2015. doi:10.1007/978-3-319-22102-1_24.

21 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166--180. ACM, 2019. doi:10.1145/3293880.3294101.

22 Jan Terlouw. Een nadere bewijstheoretische analyse van gstt's. *Manuscript (in Dutch)*, 1989.

23 Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473--497, 2004. doi:10.1016/J.TCS.2004.06.016.

24 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA):114:1--114:30, 2018. doi:10.1145/3276484.

25 James Wood and Robert Atkey. A linear algebra approach to linear metatheory. In Ugo Dal Lago and Valeria de Paiva, editors, *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020*, volume 353 of *EPTCS*, pages 195--212, 2020. doi:10.4204/EPTCS.353.10.