Formalisation in Constructive Type Theory of Barendregt's Variable Convention for Generic Structures with Binders

Ernesto Copello
Department of Computer Science
The University of Iowa, USA
ernesto-copello@uiowa.edu

Nora Szasz
szasz@ort.edu.uy

Álvaro Tasistro
tasistro@ort.edu.uy

Facultad de Ingeniería
Universidad ORT Uruguay

We introduce a universe of regular datatypes with variable binding information, for which we define generic formation and elimination (i.e. induction /recursion) operators. We then define a generic α-equivalence relation over the types of the universe based on name-swapping, and derive iteration and induction principles which work modulo α-conversion capturing Barendregt's Variable Convention. We instantiate the resulting framework so as to obtain the λ-Calculus and System F, for which we derive substitution operations and substitution lemmas for α-conversion and substitution composition. The whole work is carried out in Constructive Type Theory and machine-checked by the system Agda.

1 Introduction

The definition of functions by recursion on the description of datatypes is the basic idea of generic programming. This method is based on defining a datatype, introduced as the *universe* [12], which contains datatype descriptions, such as "a list is either empty or a pair consisting of a parameter and a sublist". Indeed, the universe constructors correspond to the common notions "either", "pair", "parameter" and "substructure" abstracted out of informal descriptions such as the preceding one. Then a decoding function is introduced, which interprets instances of the universe, usually called universe *codes*, into actual datatypes. In a dependently typed setting we can define generic functions over the universe of codes and the associated decoded datatypes. In other words, the codes provide information enough to properly traverse the structure of decoded datatype instances. Thereby, traversal becomes an operation that can be described for any recursive structure by means of generic iteration and recursion principles and, thus, code duplication is avoided by abstracting out basic operations on the datatypes. One can wonder how many other practical behaviors can undergo such kind of generalisation.

In this work we introduce a universe of *regular trees* [13] extended with variables (i.e. names) and binding information. We first define generic formation and elimination (i.e. induction/recursion) operators over this universe. The inclusion of names and the notion of locality allow us to introduce a generic α-equivalence relation, which we choose to base on name-swapping. Then we derive α-iteration and induction principles that capture Barendregt's Variable Convention (BVC) which allows to proceed in proofs and definitions by conveniently choosing bound names so as to avoid conflict. At this generic level we are able to prove several properties, mainly concerning the interaction of the iteration and recursion principles with the swapping operation and α-equivalence relation. We next obtain λ-calculus and System F as instances of our universe and define corresponding substitution operations as instances of the generic α-iteration principle. We are thereby able to derive the lemmas on compatibility of substitution with α-equivalence by direct instantiation of the generic properties referred to above. Finally, we prove the substitution composition lemma for the System F showing how our approach allows us to mimic the BVC, proving particular results on instances of the framework as it is usually done in pen-and-paper style.

Submitted to:
LFMTP 2018

1.1 Related work

Programming languages supporting native constructions to declare and manipulate abstract syntax with binders are presented by Shinwell, et. al in [17, 16], where an ML extension *FreshML*, and an OCaml extension *Fresh O'Caml* are respectively developed. These languages allow to deconstruct datatypes with binders in a safe way, that is, in the case of an abstraction inspection, a renaming with a freshly generated binder is computed for the abstraction body. In this way, the language user has access only to a fresh binder, and the renamed body of the opened abstraction. This mechanism grants that values with binders are operationally equivalent if they represent α-equivalent objects. This result is proved in [17] by introducing a denotational semantics of the object language FreshML into FM-sets (Fraenkel and Mostowski's sets). They prove that this denotational semantics matches the operational one. In this way, they are able to prove that values of the introduced abstract syntax with binders properly represent α-equivalence classes of the object-level syntax. In [5] Cheney carries out a similar work, but instead of developing a language extension, he implements a Haskell library called *FreshLib*. As the author does not implement a language from scratch, this work introduces generic programming techniques in its implementation to support the required level of genericity.

All previous works address common operations dealing with general structures with binders. Although some of these developments give proofs about the soundness of their approaches, their main concern is the implementation of meta-programs. In [10], Lee et al. use generic programming techniques to develop mechanisations of formal meta-theory in the Coq proof assistant. This work allows the user to choose between nominal, locally nameless or de Bruijn first-order syntax. For each of these representations, they offer several infrastructure operations and their associated lemmas. For instance, for the locally nameless setting, two different substitutions are needed for bound and free variables respectively. In the case of System F, where terms and type variables have binding constructions, this representation involves six different substitution operations. Hence, as the number of syntactic sorts supporting binding constructions increases in the object language, there is a combinatorial explosion of the number of operations and lemmas involved in its formalisation for the locally nameless and de Bruijn first-order syntaxes. They manage to address these issues defining these operations and associated lemmas in a generic reusable way. Moreover, they provide a small annotation language to describe the binding structure of the object language, from which they can automatically derive an isomorphism between the object language and their generic universe syntax. However, introducing inductive relations in this framework requires the user to provide a mapping between the concrete relation, defined at the object language level, and the generic relation. They are able to instantiate some cases of the POPLmark challenge [2] in their framework, validating their approach both for the locally nameless and the de Bruijn first-order syntax, and comparing some metrics of their approach against other solutions. However, their particular choice of universe makes it impossible to have more than one sort of binder per datatype. Hence, they cannot represent in their setting a language such as Session Types [19], where there exist three distinct sort of binders: parameters, channels and ports within a concurrent calculus. We believe their work addresses reusing and usability in great manner, but lacks in extensibility and abstraction. By using this framework it is possible to reuse several operations and lemmas that hide some of the work required by the underlying binders representation. However, in order to introduce new operations and prove results, the user may have to deal with the underlying generic abstract syntax language. Their work seems to support the nominal syntax, although no α-conversion relation, neither any other classic relation, property or function over named terms is presented. Indeed, they do not further develop the nominal syntax, beyond the basic definitions of a nominal abstract syntax.

In [11], Licata and Harper codify a universe that mixes binding and computation constructions in

Agda, where computations are represented as meta-level functions injected in the universe constructions, i.e., they embed a HOL syntax in their development. Their representation is based on a well-scoped de Bruijn representation, that is, de Bruijn terms associated with a context indexing the free variables. For this universe, they provide a generic substitution operation, and prove context weakening and strengthening lemmas. In 2002 Pitts and Gabbay introduced the *Nominal Logic* [9], a first-order many-sorted logic with equality, containing primitives for renaming via name-swapping, for freshness of names, and for name-binding. The swapping operation has much nicer logical properties than the more general, non-bijective forms of renaming. This operation provides a sufficient foundation for a theory of structural induction/recursion for the syntax modulo α. In [18], Urban and Tasson use ideas from Nominal Logic to construct a set of λ-calculus terms modulo alpha, that is, identifying α-convertible terms. The construction is based on a HOAS syntax on top of Isabelle/HOL, deriving recursion and induction principles over this quotient set.

Our main motivation is to show it is feasible to formalise within constructive type theory α-iteration/ induction principles for a classical named syntax, deriving these principles from just simple structural induction on fist-order terms, where equality remains the simple definitional one, and not performing any kind of quotient on terms. Then, we want to study how feasible is to use this development in practical examples. This work is structured as follows: in Section 2 we present our regular tree universe, in Section 3 we introduce name-swapping, in Section 4 we give an α-conversion relation and iteration/induction schemes modulo α, which allow us to define the caputre-avoiding subtitution operation, and automatically derive some of its basic properties. In Section 5 we introduce the proof technique that mimics the BVC, and we apply it in the substitution composition lemma. Finally, in the last section we discuss related work and conclusions. We carry out the whole development within Constructive Type Theory as implemented in the system Agda [14]. We will show fragments of the Agda code, the complete version being available at: https://github.com/ernius/genericBindingFramework.

2 Universe of Regular Trees with Binders

2.1 Universe of (Codes of) Functors

We choose to build up a universe whose objects are codes to be interpreted as functions from Set to Set[1], i.e. as *functors*. The actual datatypes generated by this mechanism arise as fixed points of such functors. To this effect, we introduce in Figure 1:

- the datatype Functor of codes, and
- the (mutually recursive-inductive) definition of the decoding function $[\![\_]\!]$ and of the actual datatype $\mu\, F$ associated to any given functor code F.

Notice first that the (inductive) definition of $\mu\, F$ by means of the constructor $\langle\_\rangle$ indeed introduces it as the least fixed point of the functor corresponding to the code F. Now let us examine the codes and corresponding functors. The first three constructors of datatype Functor in Figure 1 represent the embedding of: the unity type, a recursive position, and an arbitrary (i.e. externally given) datatype. The fourth constructor embeds a datatype representable in our universe, while the next two constructors represent the sum and product of types. Finally, the last two introduction rules are specific to our desired domain of abstract syntaxes with binders. As our framework supports different sorts of names, the variables and binders constructors receive as parameters an identifier of the sort of variables that they respectively

[1] Set is the type of (small) datatypes.

```
                                                     mutual
data Functor : Set₁ where                              ⟦_⟧ : Functor → Set → Set
  |1|     :                         Functor            ⟦ |1|        ⟧ _  = ⊤
  |R|     :                         Functor            ⟦ |R|        ⟧ A  = A
  |E|     : Set                   → Functor            ⟦ |E|   B    ⟧ _  = B
  |Ef|    : Functor               → Functor            ⟦ |Ef|  F    ⟧ _  = μ F
  _|+|_   : Functor → Functor → Functor                ⟦ F |+| G    ⟧ A  = ⟦ F ⟧ A ⊎ ⟦ G ⟧ A
  _|x|_   : Functor → Functor → Functor                ⟦ F |x| G    ⟧ A  = ⟦ F ⟧ A × ⟦ G ⟧ A
  |v|     : Sort                  → Functor            ⟦ |v|  S     ⟧ _  = V
  |B|     : Sort    → Functor     → Functor            ⟦ |B|  S G   ⟧ A  = V        × ⟦ G ⟧ A
                                                       data μ (F : Functor) : Set where
                                                         ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Figure 1: Regular tree universe with binders.

introduce or bind. The binder constructor also receives the descriptor of the structure serving as scope of the bound variable. In many cases of interest (e.g. the λ-calculus and System F to be examined shortly) this subterm descriptor will be just a recursive position. However, a compound structure is often needed, as it is the case in e.g. languages with a `letrec` primitive. We can observe how the variable and binder constructions inject a fixed set of variables (names) V into the interpreted datatype. This set V is assumed to be infinite with a decidable equality. Notice too that in the cases of the variable injection and the binder functors the sort argument S has no impact on the interpreted set. Indeed, we have only one kind of names V. The sort identifier will be relevant to implement generic operations related to binding issues, as shown in subsequent sections.

For example, the types of natural numbers and of lists of natural numbers can be defined as follows:

$$\text{FNat} = |1| \;|{+}|\; |\text{R}| \qquad\qquad \text{FListNat} = |1| \;|{+}|\; (|\text{Ef}|\ \text{FNat}) \;|{\times}|\; |\text{R}|$$

$$\text{Nat} = \mu\ \text{FNat} \qquad\qquad \text{ListNat} = \mu\ \text{FListNat}$$

In Figure 2 we illustrate the use of the variables and binders constructions by encoding the λ-calculus. We show the corresponding classical concrete syntax definition using comments, that are written following a dash to the right of each line. This definition has only one sort of variables identified with the sort SortλTermVars.

```
λF : Functor                          - M,N :-              λTerm : Set
λF =  |v| SortλTermVars               - x                   λTerm = μ λF
  |+|  |R| |x| |R|                    - | M N
  |+|  |B| SortλTermVars |R|          - | λ x M
```

Figure 2: λ-calculus.

We next introduce notation resembling the concrete syntax of the λ-calculus and hiding away our universe code constructions.

```
v : V → λTerm          _·_ : λTerm → λTerm → λTerm           ƛ : V → λTerm → λTerm
v = ⟨_⟩ ∘ inj₁         M · N = ⟨ inj₂ (inj₁ (M , N)) ⟩        ƛ n M = ⟨ inj₂ (inj₂ (n , M)) ⟩
```

Next we present the codification of the System F. As this language also needs bindings at the type level, this encoding illustrates the use of two distinct sorts of identifiers, namely SortFTypeVars and SortλTermVars:

```
tyF : Functor                               - t,r :-              FType : Set
tyF =  |v| SortFTypeVars                    - α                   FType = μ tyF
   |+|  |R| |x| |R|                         - | t → r
   |+|  |B| SortFTypeVars |R|               - | ∀ α .  t

tF : Functor                                - M,N :-              FTerm : Set
tF =  |v| SortFTermVars                     - x                   FTerm = μ tF
   |+|  |R| |x| |R|                         - | M N
   |+|  |Ef| tyF |x| |B| SortFTermVars |R|  - | λ x :  t .  M
   |+|  |R| |x| |Ef| tyF                    - | M t
   |+|  |B| SortFTypeVars |R|               - | Λ α .  M
```

In the preceding constructions we have chosen a simplification of the universe of regular tree datatypes presented in [13], where recursive types are represented using μ-types (from [15]). However, instead of the nominal approach traditionally used with recursive type binders, they use a well-scoped de Bruijn representation. Therefore, in order to properly interpret the full universe, a definition indexed by a context with the multiple μ-recursive positions definitions is required. Our representation in Agda (Figure 1), simplifies this burden at the expense of not being able to represent mutually recursive datatypes. In other words, our universe construction has expressive power equivalent to admitting only a single top-level μ-recursive type binder.

2.2 Map and Fold

The classical definition of *fold* based on *map* that is usually introduced in category theory does not pass Agda's termination checker. The recursive call to fold is hidden inside a call to map, and because of this the termination checker cannot determine how map is using it. To make the fold operation pass the termination checker we have to fuse map and fold into a single function, as done in [14] for a similar regular tree universe. In Figure 3 we show our implementation of the function foldmap. We make use of Agda's implicit arguments feature, denoted by curly braces, to omit terms that the type checker can figure out for itself. For instance, we declare the A set argument as an implicit argument. The presented foldmap function needs to keep two functors, since the fold (recursive) part works always over the same functor argument F, while, for the map part, the auxiliary functor argument G gives the position of functor F during the traversal of the structure. Therefore, this function only uses the functor F in the recursive case rule (the |R| case) in which the right hand side expression basically begins a new traversal of the functor F, in a way similar to the original definition of fold. It does so by providing with a fresh copy of F in the position of the auxiliary argument G. The rest of the rules are equivalent to a map over the functor G. Note that this definition terminates because the argument of type ⟦ G ⟧ (μ F) decreases in each recursive call. The new fold operation is defined as a recursive instance of foldmap.

As an example, we define a function vars that counts the number of variable occurrences in a term of the λ-calculus. We do so by instantiating it as a case of the fold operation in fig. 4

Next we present a particular useful instantiation of the fold operator, named foldCtx. This instantiation aims at reproducing some techniques related to the nominal syntax considered in our work. We introduce an extra argument with type μ C, which is used by the folded function f. This function is partially applied to this extra argument, and then passed as an argument of fold. Hence, this argument acts as an explicit invariant context for the function f through the entire fold operation. Another difference with the original fold operation is that the result of this instance is a datatype μ H encoded in our universe instead of an arbitrary set.

```
foldCtx : {C H : Functor}(F : Functor) → (μ C → ⟦ F ⟧ (μ H) → μ H) → μ C → μ F → μ H
foldCtx F f c = fold F (f c)
```

```
foldmap : {A : Set}(F G  : Functor) → (⟦ F ⟧ A → A) → ⟦ G ⟧ (μ F) → ⟦ G ⟧ A
foldmap F |1|            f tt         =  tt
foldmap F |R|            f ⟨ e ⟩      =  f    (foldmap F F   f e)
foldmap F (|E|  A)       f e          =  e
foldmap F (|Ef|  G)      f e          =  e
foldmap F (G₁ |+|  G₂)   f (inj₁ e)   =  inj₁  (foldmap F G₁  f e)
foldmap F (G₁ |+|  G₂)   f (inj₂ e)   =  inj₂  (foldmap F G₂  f e)
foldmap F (G₁ |x|  G₂)   f (e₁ , e₂)  =  foldmap F G₁ f e₁  , foldmap F G₂  f e₂
foldmap F (|v|  S)       f x          =  x
foldmap F (|B| S   G)    f (x , e)    =  x                   , foldmap F G   f e

fold : {A : Set}(F : Functor) → (⟦ F ⟧ A → A) → μ F → A
fold F f e = foldmap F |R| f e
```

Figure 3: Terminating fold operation.

```
varsaux : ⟦ λF ⟧ ℕ → ℕ                             vars : μ λF → ℕ
varsaux (inj₁ _)                 = 1               vars = fold λF varsaux
varsaux (inj₂ (inj₁ (m , n)))    = m + n
varsaux (inj₂ (inj₂ (_ , m)))    = m
```

Figure 4: Fold application example.

From this fold instance we can directly derive the naive substitution operation for the λ-calculus. In order to do this, we next give the functor descriptor cF for the context argument. It represents the pair formed by the variable to be replaced and the substituted term:

```
cF =  |v| SortλTermVars |x| |Ef| λF
```

Next we define the function substaux (Figure 5) to be folded which, given a term structure with the results of the recursive calls in its recursive positions, constructs the final result of the substitution. For the variable case, we check, as usual, whether the substitution is to be applied, whereas the application and abstraction cases directly reconstruct the corresponding terms from the recursive call. Note that in the abstraction case we do not check whether the abstracted variable is different from the one being replaced, as in Barendregt's substitution definition in [3]. In fact, this comparison would be pointless because, as we are using an iteration principle, we do not have access to the original abstraction body subterm. Note that we hide the universe codes on the right side of this definition by using the previously introduced λ-calculus constructors.

```
substaux : μ cF → ⟦ λF ⟧ (μ λF) → μ λF
substaux _          (inj₂ (inj₁ (t₁ , t₂)))  = t₁ · t₂
substaux _          (inj₂ (inj₂ (y , t)))    = ƛ y t
substaux ⟨ x ,  N ⟩  (inj₁ y) with x ≟v y
... | yes _                                  = N
... | no _                                   = v y
```

Figure 5: Naive substitution auxiliary function.

Finally, we instantiate the foldCtx function with substaux, and its appropriate context pair to get the naive substitution operation.

```
_[_:=_]ₙ : λTerm → V → λTerm → λTerm
M [ x := N ]ₙ  = foldCtx λF substaux (⟨ x , N ⟩) M
```

2.3 Primitive Induction

We now develop a more generic elimination rule than the fold operation defined above. This elimination rule captures proof by induction, and is based on the recursion rule given by Benke et al. in [4]. However, our development departs from their work in the following points: Firstly, they derive an elimination rule for a simpler universe construction, based on one-sorted term algebras, and defined through signatures instead of functors. For instance, their universe does not allow the injection of previously defined datatypes, which is necessary for defining e.g. lists of natural numbers so as natural numbers become parts of the lists. Secondly, their induction principle would not pass Agda's termination checker due to reasons similar to the ones discussed for the first version of the fold operation. To define the desired function, to be called foldInd, we first introduce the auxiliary function fih (Figure 6).

```
fih  : {F : Functor}(G : Functor)(P : μ F → Set) → ⟦ G ⟧ (μ F) → Set
fih |1|            P tt          = ⊤
fih |R|            P e           = P e
fih (|E|  B)       P e           = ⊤
fih (|Ef|  G)      P e           = ⊤
fih (G₁ |+|  G₂)  P (inj₁  e)   = fih G₁  P e
fih (G₁ |+|  G₂)  P (inj₂  e)   = fih G₂  P e
fih (G₁ |x|  G₂)  P (e₁ ,  e₂)  = fih G₁  P e₁ × fih G₂ P e₂
fih (|v|  S)       P x           = ⊤
fih (|B| S    G)   P (x ,   e)   = fih G   P e
```

Figure 6: fih function.

This function receives a predicate P over the fixed point of a functor F, and an auxiliary functor G (with a similar functionality to the one used in the foldmap function). It returns a corresponding predicate of type ⟦ G ⟧ (μ F) → Set. This resulting predicate represents P holding for every recursive position μ F in an element of type ⟦ G ⟧ (μ F).

We can now present our induction principle. We proceed in a similar way as we did above for the fold function. First, we introduce the fold-map fusion function foldmapFh (Figure 7). Then, we use this function to directly derive the induction principle as a recursive instance of the fold-map fusion.

```
foldmapFh :  {F : Functor}(G : Functor)(P : μ F → Set)
               → ((e : ⟦ F ⟧ (μ F)) → fih F P e →  P ⟨ e ⟩) → (x : ⟦ G ⟧ (μ F)) → fih G P x
foldmapFh |1|               P hi tt          =  tt
foldmapFh {F} |R|           P hi ⟨ e ⟩       =  hi e (foldmapFh {F} F P hi e)
foldmapFh (|E|   B)         P hi b           =  tt
foldmapFh (|Ef|  F)         P hi b           =  tt
foldmapFh (G₁ |+|  G₂)     P hi (inj₁  e)   =  foldmapFh G₁  P hi e
foldmapFh (G₁ |+|  G₂)     P hi (inj₂  e)   =  foldmapFh G₂  P hi e
foldmapFh (G₁ |x|   G₂)    P hi (e₁   , e₂) =  foldmapFh G₁  P hi e₁ , foldmapFh G₂  P hi e₂
foldmapFh (|v|   S)         P hi n           =  tt
foldmapFh (|B| S    G)      P hi (x    , e)  =  foldmapFh G   P hi e

foldInd : (F : Functor)(P : μ F → Set) → ((e : ⟦ F ⟧ (μ F)) → fih F P e →  P ⟨ e ⟩) → (e : μ F) → P e
foldInd F P hi e = foldmapFh {F} |R| P hi e
```

Figure 7: Induction principle.

We next give an example of the use of this induction principle, namely proving that the application of the function vars to any lambda term is greater than zero. We introduce the predicate Pvars representing the property to be proved and an auxiliary lemma plus>0, stating that the sum of two positive numbers is also positive.

```
PVars : μ λF → Set                plus>0 : {m n : ℕ} → m > 0 → n > 0 → m + n > 0
PVars M = vars M > 0              plus>0 {m} {n} m>0 n>0 = ≤-steps m n>0
```

The proof that Pvars holds for every term M is a direct application of the induction principle. The variable case is direct, while the application case is the application of the lemma plus>0 to the induction hypotheses. Finally, the abstraction case is a direct application of the induction hypothesis.

```
proof : (e : ⟦ λF ⟧ (μ λF)) → fih λF PVars e → PVars ⟨ e ⟩     provePVars : (M : μ λF) → PVars M
proof (inj₁ x)              tt          = s≤s z≤n           provePVars = foldInd λF PVars proof
proof (inj₂ (inj₁ (M , N))) (ihM , ihN) = plus>0 ihM ihN
proof (inj₂ (inj₂ (_ , M))) ihM         = ihM
```

3 Name-Swapping

We now turn to considering a very basic primitive of name-swapping, which will be used for defining α-conversion without a mention to substitution. This constitutes the foundation of the implementation of the general idea that principles of recursion and induction ought to be defined so as to work modulo α-conversion, thus allowing to mimic the usual pen-an-paper conventions that allow the choice of convenient representatives of the terms involved in a definition or proof. The name-swapping operation completely traverses a data structure, swapping occurrences (either free, bound or binding) of two given names of some sort. Its implementation (Figure 8) is similar to that of fold.

```
swapF : {F : Functor}(G : Functor) → Sort → V → V → ⟦ G ⟧ (μ F) → ⟦ G ⟧ (μ F)
swapF |1|              S a b tt        = tt
swapF {F} |R|          S a b ⟨ e ⟩     = ⟨ swapF F S a b e ⟩
swapF (|E|  _)         S a b e         = e
swapF (|Ef|  G)        S a b ⟨ e ⟩     = ⟨ swapF G S a b e ⟩
swapF (G₁ |+| G₂)      S a b (inj₁ e)  = inj₁ (swapF G₁ S a b e)
swapF (G₁ |+| G₂)      S a b (inj₂ e)  = inj₂ (swapF G₂ S a b e)
swapF (G₁ |x| G₂)      S a b (e₁ , e₂) = swapF G₁ S a b e₁ , swapF G₂ S a b e₂

swapF (|v|  S')        S a b c with S' ≟S S
... | yes _                            = ( a • b )ₐ c
... | no  _                            = c

swapF (|B| S'  G)      S a b (c , e) with S' ≟S S
... | yes _  =                           ( a • b )ₐ c        , swapF G S a b e
... | no  _  =                           c                   , swapF G S a b e

swap : {F : Functor} → Sort → V → V → μ F → μ F
swap S a b e = swapF |R| S a b e
```

Figure 8: Name-swapping operation.

We use an auxiliary function swapF, that takes functors F and G, and traverses the G structure until a recursive or embedded position is reached, from where we restart the G argument with either the original recursive functor F or the embedded functor respectively. Note that this treatment differs from the one in the definition of fold, where this case is a base case. Here we must also traverse the embedded functor instance, as we are swapping all the variables in the structure, including the variables present in any embedded structure. Because of this, we cannot derive name-swapping as an instance of fold. In the variable and abstraction cases we use name-swapping over variables [2], denoted by the (mixfix) operator $(\_\bullet\_)_a\_$ as in [7].

[2]Requiring the decidable equality over names.

We prove a generic lemma about the interaction between name-swapping and the iteration principle. This lemma is presented in Figure 9, and states that the fold instance with context information is well-behaved with respect to name-swapping, given that the respectively folded operation is also well-behaved. Its proof goes by a direct induction on terms. This example shows how we are able to develop generic proofs over our universe with binders.

```
lemmaSwapFoldCtxEquiv : {C H F  : Functor}{S : Sort}{x y : V}
        {e : μ F}{f : μ C → ⟦ F ⟧ (μ H) → μ H}{c : μ C}
  → ({c : μ C}{S  : Sort}{x y : V}{e : ⟦ F ⟧ (μ H)}
                        → f (swap S x y c) (swapF F S x y e) ≡ swap S x y (f c e))
  → foldCtx F f (swap {C} S x y c) (swap {F} S x y e) ≡ swap {H} S x y (foldCtx F f c e)
```

Figure 9: Fold with context is well-behaved with respect to name-swapping.

We are able to directly apply the preceding lemma to the λ-calculus case in order to prove the result in Figure 10. This states that name-swapping commutes with substitution, which is particularly useful. We introduce the operator (_•_)_ to denote the swapping of variables in terms. In the proof we use of the auxiliary lemma lemma-substauxSwap which states that the function substaux, used to define substitution, is well-behaved with respect to name-swapping. This example shows how feasible it is in our framework to instantiate generic proofs for deriving useful lemmas holding for particular instances of the generic universe.

```
lemma-[]Swap : {x y z : V}{M N : λTerm}
  → (( y • z ) M) [ ( y • z )ₐ x := ( y • z ) N ]ₙ ≡ ( y • z ) (M [ x := N ]ₙ)
lemma-[]Swap {x} {y} {z} {M} {⟨ N ⟩}
  = lemmaSwapFoldCtxEquiv  {cF} {λF} {λF} {SortλTermVars} {y} {z} {M} {substaux} {⟨ x , ⟨ N ⟩ ⟩}
    (λ {c} {S} {x} {y} {e} → lemma-substauxSwap {c} {S} {x} {y} {e})
```

Figure 10: Substitution is well-behaved with respect to name-swapping.

In a similar manner we introduce a generic function returning the free variables of terms, and prove several properties about its interaction with swapping, fold, and α-conversion.

4 Alpha Equivalence Relation.

In Figure 11 we introduce the generic definition of the α-equivalence relation over our universe, named ∼α. Its definition follows a process similar to the one used before to implement generic functions over our universe. First, we define an auxiliary relation ∼αF, which is inductively defined introducing an auxiliary functor G, used to traverse the functor F structure. For the interesting binder case, we follow an idea similar to the one used in [7], that is, we define that two abstractions are α-equivalent if there exists some list of variables xs, such that for any given variable z not in xs, the result of swapping the corresponding binders with z in the abstraction bodies is α-equivalent. Note that the swapping is performed only over the sort of variables bound by this binder position, leaving any other sort of variables unchanged. We are able to prove that this is an equivalence relation, and also that it is preserved under name-swapping in a similar way as done in our previous work [7].

As we did before with name-swapping, we study how the iteration principle interacts with the introduced α-equivalence relation. We begin proving that the fold operation is α-compatible if it is applied to an also α-compatible function. We say a function is α-compatible iff it returns α-convertible results when it is applied to α-convertible arguments. In Figure 12 we state this lemma, whose proof goes by induction on terms. The only interesting case is the binders case, where we make use of the preservation of α-equivalence under name-swapping.

```
data ∼αF {F : Functor} : (G : Functor) → ⟦ G ⟧ (μ F) →  ⟦ G ⟧ (μ F) → Set where
  ∼α1   :                               ∼αF |1|           tt          tt
  ∼αR   :  {e e' : ⟦ F ⟧ (μ F)}
        →  ∼αF F e e'          →  ∼αF |R|           ⟨ e ⟩       ⟨ e' ⟩
  ∼αE   :  {B : Set}{b : B}    →  ∼αF (|E| B)        b           b
  ∼αEf  :  {G : Functor}{e e' : ⟦ G ⟧ (μ G)}
        →  ∼αF G e e'          →  ∼αF (|Ef| G)       ⟨ e ⟩       ⟨ e' ⟩
  ∼α+1  :  {F1 F2 : Functor}{e e' : ⟦ F1 ⟧ (μ F)}
        →  ∼αF F1 e e'         →  ∼αF (F1 |+| F2)  (inj1 e)   (inj1 e')
  ∼α+2  :  {F1 F2 : Functor}{e e' : ⟦ F2 ⟧ (μ F)}
        →  ∼αF F2 e e'         →  ∼αF (F1 |+| F2)  (inj2 e)   (inj2 e')
  ∼αx   :  {F1 F2 : Functor}{e1 e1' : ⟦ F1 ⟧ (μ F)}
           {e2 e2' : ⟦ F2 ⟧ (μ F)}
        →  ∼αF F1 e1 e1'       →  ∼αF F2 e2 e2'
                               →  ∼αF (F1 |x| F2)  (e1 , e2)  (e1' , e2')
  ∼αV   :  {x : V}{S : Sort}   →  ∼αF (|v| S)        x           x
  ∼αB   :  (xs : List V){S : Sort}{x y : V}{G : Functor}{e e' : ⟦ G ⟧ (μ F)}
        →  ((z : V) → z ∉ xs   →  ∼αF G  (swapF G S x z e) (swapF G S y z e'))
                               →  ∼αF (|B| S G)      (x , e)     (y  , e')

_∼α_ : {F : Functor} → μ F → μ F → Set
_∼α_ = ∼αF |R|
```

Figure 11: Alpha equivalence relation.

```
lemma-fold-alpha  : {F H : Functor}{f f' : ⟦ F ⟧ (μ H) → μ H}
  → ({e e' :  ⟦ F ⟧ (μ H)} → ∼αF F e e' → f e ∼α f' e')
  → (e : μ F) → fold F f e ∼α fold F f' e
```

Figure 12: Fold function α-compatibility property.

As a direct corollary we get that the fold with context instance is α-compatible in its context argument provided the folded function is also α-compatible on its arguments (Figure 13).

```
lemma-foldCtx-alpha-Ctx  : {F H C : Functor}{f : μ C → ⟦ F ⟧ (μ H) → μ H}{c c′ : μ C}
  →  ({e e′ :  ⟦ F ⟧ (μ H)}{c c′ : μ C} → c ∼α c′ →  ∼αF F e e′ → f c e ∼α f c′ e′)
  →  c ∼α c′ →  (e : μ F) → foldCtx F f c e ∼α foldCtx F f c′ e
lemma-foldCtx-alpha-Ctx {F} {f = f} {c} {c′} p c∼c′ e = lemma-fold-alpha (p c∼c′) e
```

Figure 13: Fold context function α-compatibility corollary.

We define other relations over our universe in a similar way as we have done for the α-equivalence relation. For instance, the notOccurBind relation holds if some given variable does not occur in any binder position within a term. In this relation we discard the name sort information. We do so to simplify our next development as we will explain later. We find useful to extend this relation to lists of variables, named as ListNotOccurBind, which holds if all the variables in a given list do not occur in any binder position (associated with any sort) in a term. Using this relation we are able to prove the lemma stated in Figure 14. This lemma states that the fold with context principle is α-compatible on its two arguments if the provided function is α-compatible and well-behaved with respect to name-swapping. Note that this lemma extends the one given before in Figure 13, although it requires extra freshness premises, and that the folded function is preserved under name-swapping.

```
lemma-foldCtx-alpha : {F H C : Functor}{f : μ C → ⟦ F ⟧ (μ H) → μ H}{c c' : μ C}{e e' : μ F}
  → ({e e' : ⟦ F ⟧ (μ H)}{c c′ : μ C} → c ∼α c′ → ∼αF F e e′ → f c e ∼α f c′ e′)
  → ({c : μ C}{S : Sort}{x y : V}{e : ⟦ F ⟧ (μ H)}
                    → f (swap S x y c) (swapF F S x y e) ≡ swap S x y (f c e))
  → ListNotOccurBind (fv c) e → ListNotOccurBind (fv c') e'
  → c ∼α c' → e ∼α e' → foldCtx F f c e ∼α foldCtx F f c' e'
```

Figure 14: Fold context α-compatibility property.

4.1 Alpha Fold

We are now able to introduce a fold operation that works at the level of α-equivalence classes of terms, that is, it only defines α-compatible functions. First, we introduce the function bindersFreeElem that takes a list of variables *xs* and an element *e*, and returns an element α-equivalent to *e* whose binders are not in the given list. This function will be useful to reproduce the BVC, which basically states that we can always pick a term with its binders fresh from a given context, which in this function is represented as a list of variables. We prove that this function has the important property of being strongly α-compatible, i.e. that it returns the same result for α-convertible terms.

```
bindersFreeElem : {F : Functor}(xs : List V)(e : μ F) → ∃ (λ e' → ListNotOccurBind {F} xs e')
```

Based on this function, we next directly implement the α-fold principle as an instance of the fold with context function.

```
foldCtx-alpha : {C H : Functor}(F : Functor) → (μ C → ⟦ F ⟧ (μ H) → μ H) → μ C → μ F → μ H
foldCtx-alpha F f c e = foldCtx F f c (proj₁ (bindersFreeElem (fv c) e))
```

This iteration principle first finds a fresh term for a given context *c*, and then directly applies the fold operation over it. We developed this iteration principle following a different approach from the one taken in our previous work [7], where we renamed the binders during the fold traverse. Instead, we chose to separate these two stages in order to reuse the previously defined fold operation and its properties.

We can now properly justify the name "alpha" given to the introduced iteration principle. Firstly, as bindersFreeElem returns syntactical equal terms when applied to α-convertible terms, we have that our function is trivially strong α-compatible on its last term argument. Secondly, as a direct consequence from the lemma already proved for our iteration principle foldCtx in Figure 13, this new principle inherits its α-compatibility in its context argument from foldCtx, given that the function received is also α-compatible on its arguments. Thus, the presented iteration principle works at the α-equivalence classes level when the given function works at the same level.

Now we can derive the capture avoiding substitution operation for the lambda calculus example by a direct application of the introduced α-fold principle. In fact this definition is exactly the same as the one given before for the naive substitution, but using now the α-fold operation instead of the fold one.

```
_[_:=_] : λTerm → V → λTerm → λTerm
M [ x := N ] = foldCtx-alpha λF substaux (⟨ x , N ⟩) M
```

Substitution lemmas stating that substitution is well-behaved with respect to α-conversion are inherited from the α-compatibility of the iteration principle, only requiring the α-compatibility property of the substaux function. As this auxiliary function is not recursively defined, this proof is just a simple case analysis, while the proof involving the recursive data-type traversal is resolved at the generic level.

Next lemma in Figure 15 relates the presented α-fold principle with the previously defined one, giving sufficient conditions under which the two principles return α-convertible terms. First, the folded function must be α-compatible on its two arguments, and also well-behaved with respect to name-swapping. Secondly, we need a freshness premise stating that the free variables in the context do not occur bound in the applied term.

$$
\begin{array}{l}
\mathsf{lemma\text{-}foldCtxAlpha\text{-}foldCtx} : \{C\, H : \mathsf{Functor}\}(F : \mathsf{Functor})\\
\qquad \{f : \mu\, C \to [\![\, F\,]\!]\ (\mu\, H) \to \mu\, H\}\{c : \mu\, C\}\{e : \mu\, F\}\\
\quad \to\ (\{e\, e' \ :\ [\![\, F\,]\!]\ (\mu\, H)\}\{c\, c' : \mu\, C\} \to c \sim\!\alpha\ c' \to \sim\!\alpha\mathsf{F}\ F\ e\ e' \to f\, c\, e \sim\!\alpha\, f\, c'\, e')\\
\quad \to\ (\{c : \mu\, C\}\{S\ : \mathsf{Sort}\}\{x\, y : \mathsf{V}\}\{e : [\![\, F\,]\!]\ (\mu\, H)\}\\
\qquad\qquad\qquad \to f\,(\mathsf{swap}\ S\ x\ y\ c)\ (\mathsf{swapF}\ F\ S\ x\ y\ e) \equiv \mathsf{swap}\ S\ x\ y\ (f\ c\ e))\\
\quad \to\ \mathsf{ListNotOccurBind}\ (\mathsf{fv}\ c)\ e \to \mathsf{foldCtx\text{-}alpha}\ F\, f\, c\ e \sim\!\alpha\ \mathsf{foldCtx}\ F\, f\, c\ e
\end{array}
$$

Figure 15: Fold and α-fold relations.

We can instantiate this lemma to the λ-calculus to get sufficient conditions under which the two presented substitution operations are α-convertible. Its proof requires two lemmas about the substaux function, one stating that it is α-compatible, and the other one stating that it is well-behaved under name-swapping. Both lemmas were already used in previous proofs.

4.2 Alpha Induction Principle

In this section we generalise previous works [7, 6], developing an α-induction principle for α-compatible predicates. Our presentation introduces an explicit premise about the α-compatibility of the predicate being proved, which in general is not explicitly mentioned in informal developments, but is certainly required to ensure that the predicate in question is actually about *abstract* terms, i.e. not dependent on the choice of bound names. Hence, to prove some property over any term, this principle requires the user to prove the property only for terms with fresh enough binders, i.e. distinct from the variables in some given context, as is usually done under the BVC. We derive this principle following a procedure similar to the one used to infer the principle in Section 2.3. We show below the interesting recursive and binder cases, the remaining ones being equivalent to those presented in Figure 6. We define an auxiliary function fihalpha, which transforms a given predicate P over a datatype μ F to a predicate over the datatype $[\![$ G $]\!]$ (μ F). This predicate states that P holds for every recursive position μ F in a datatype $[\![$ G $]\!]$ (μ F). Besides, it adds freshness premises with respect to some given variables list xs in the recursive and binder cases of its definition: In the binder case, it states that the binder is not in the given list xs, while, in the recursive case, it states that no variable in xs occurs in a binder position in the recursive subterm e.

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathsf{fihalpha} :\ \{F : \mathsf{Functor}\}(G : \mathsf{Functor})(P : \mu\, F \to \mathsf{Set}) \to \mathsf{List}\ \mathsf{V} \to\ [\![\, G\,]\!]\ (\mu\, F) \to \mathsf{Set}}\\
\mathsf{fihalpha}\ |\mathsf{R}| & P\ xs\ e & = & P\ e & \times\ (\forall\ a \to a \in xs \to a\ \mathsf{notOccurBind}\ e)\\
\mathsf{fihalpha}\ (|\mathsf{B}|\ S\ \ G) & P\ xs\ (x\ ,\ e) & = & x \notin xs & \times\ \mathsf{fihalpha}\ G\ \ P\ xs\ e
\end{array}
$$

We state this principle in Figure 16. Its proof is similar to the α-fold principle's proof. We firstly use the function bindersFreeElem (from Section 4.1) over the parameter e and the freshness context xs to get an α-equivalent term e' with binders not occurring in the list xs. Then we apply the primitive induction principle (Figure 7) over the fresh term e' to prove the following predicate P':

$$P'(x) \equiv (\forall c \in xs \Rightarrow c\ notOccurrBind\ x) \Rightarrow P(x)$$

Finally, we apply the proof of predicate P' to the term e' and its freshness hypothesis to obtain that $P\ e'$ must hold. Hence, as the predicate P is α-compatible, and $e \sim_\alpha e'$, we get that $P\ e$ should also hold.

$$
\begin{array}{l}
\mathsf{alphaPrimInd} : \{F : \mathsf{Functor}\}(P : \mu\, F \to \mathsf{Set})(xs : \mathsf{List}\ \mathsf{V}) \to \alpha\mathsf{CompatiblePred}\ P\\
\quad \to ((e : [\![\, F\,]\!]\ (\mu\, F)) \to \mathsf{fihalpha}\ F\ P\ xs\ e \to P\ \langle\ e\ \rangle) \to \forall\ e \to P\ e
\end{array}
$$

Figure 16: Alpha induction principle.

The proof of P' is done using an auxiliary lemma which recursively reconstructs a proof of fihalpha $P\ xs\ e$ given that fih $P\ xs\ e$ holds and that the binders of e do not occur in the context xs. This proof is just a generalisation of the one already presented in [6] for an equivalent α-induction principle for λ-calculus.

In this previous work we were also able to prove the Church-Rosser theorem for the λ-calculus using this equivalent induction principle. Therefore, we conjecture that following the same procedure we would be able to achieve the confluence of β-reduction result within our generic framework. However, we sketch another approach in next section to prove the substitution composition lemma, a crucial lemma in the confluence proof.

5 Codification of a BVC proof technique.

In Figure. 17 we show a result that validates the BVC and usual practices in common pen-and-paper proofs within our generic framework. It states that for any α-compatible predicate P, we can prove $P\ e$ for any term e by just proving it for terms whose binders are all different from their own free variables and from the variables in an arbitrary list xs. As in previous induction principle, this technique requires the α-compatibility of the predicate being proved.

```
alphaProof : {F : Functor}(P : μ F → Set)(xs : List V) → αCompatiblePred P
  → ((e : μ F) → ListNotOccurBind xs e → ListNotOccurBind (fv e) e → P e )
  → ∀ e → P e
```

Figure 17: BVC proof principle.

To prove $P\ e$ for arbitrary e, we proceed as follows: We first find a fresh enough term e' such that $e' \sim_\alpha e$ using the function bindersFreeElem. Then, we can use the hypothesis for the fresh term e' to derive that $P\ e'$ holds. Finally, $P\ e$ must also hold, as P is α-compatible. We do not show the code of the proof, since it is similar to others previously presented.

Next we illustrate the use of this result to prove the substitution composition lemma for System F. First, we prove this lemma for the naive substitution operation. Next we introduce the property to be proved. Note that an extra freshness premise stating that x does not occur bound in the term L is required, since we use the naive substitution.

```
PSCn : {x y : V}{L : FTerm} → FTerm → FTerm → Set
PSCn {x} {y} {L} N M = x ∉ y :: fv L → x notOccurBind L
  → (M [ x := N ]n) [ y := L ]n ∼α (M [ y := L ]n)[ x := N [ y := L ]n ]n
```

The proof is done using the structural induction principle (Figure 7). We show below the interesting abstraction case:

```
lemma-substCompositionNAux (inj2 (inj2 (inj1 (t , z , M)))) (_ , hiM) xnotInyfvL xnotBL =
  begin
    (ƛ z t M) [ x := N ]n [ y := L ]n
  ≈⟨ refl ⟩
    ƛ z t (M [ x := N ]n [ y := L ]n)
  ∼⟨ ∼αR (∼α+2 (∼α+2 (∼α+1
    (∼αx ρF (lemma∼+B (hiM xnotInyfvL xnotBL)))))) ⟩
    ƛ z t (M [ y := L ]n [ x := N [ y := L ]n ]n)
  ≈⟨ refl ⟩
    (ƛ z t M) [ y := L ]n [ x := N [ y := L ]n ]n
  ■
```

This equational proof is constructed following the usual pen-and-paper practice: First we push the substitution inside the abstraction. Then, by the induction hypothesis we know that the composition of substitutions in the abstraction bodies are α-convertible, and hence we are able to prove that the entire abstraction is α-convertible too, using the auxiliary lemma lemma∼+B. Finally, we push back the substitutions outside the abstraction to conclude the proof.

Now we prove the substitution composition lemma for the capture-avoiding substitution operation using the introduced α-proof technique. We begin by defining the functor describing a triple of terms TreeTermF. Then, we introduce the predicate PSComp over triples, stating the composition lemma for the substitution.

```
TreeFTermF = |Ef| tF |x| |Ef| tF |x| |Ef| tF
TreeFTerm = μ TreeFTermF

PSComp : {x y : V} → TreeFTerm → Set
PSComp {x} {y} ⟨ M , N , L ⟩ = x ∉ y :: fv L
  → (M [ x := N ]) [ y := L ] ∼α (M [ y := L ])[ x := N [ y := L ] ]
```

We prove that PSComp is α-compatible with respect to triples of terms by a direct equational proof using basically the previous substitution lemmas. In Figure 18 we show the core of the proof. It uses the preceding substitution lemmas to replace the classical substitution operations with the naive ones. This can be done because we have freshness premises stating that in the introduced context of triples all binders are different from the free variables in the involved terms, and also from variables x and y. Finally, we work in very much the same way as we did at the beginning to recover the classical substitutions from the naive ones. There are many auxiliary lemmas and boilerplate code concerning the freshness premises involved in the last proof which we do not show in this presentation. These are hidden inside auxiliary lemmas as: y:fvL-NB-M[x:=N]$_n$ and y:fvL-NB-N occurring in the proof. The first of these lemmas, for instance, proves that neither the variable y nor the free variable binding in L occur bound in $M[x{:=}N]_n$, which is easy to verify from the freshness premises. However, we believe further work is necessary to automatise some of these proofs, or even rewriting the freshness relations in order to alleviate its handling.

Finally, we can use the introduced α-proof principle with the previous proof obligations to finish the proof. Note how, by applying the α-proof technique to a triple of terms, we were able to get sufficient freshness premises to develop a proof similar in structure to pen-and-paper ones in a direct manner. This is possible because in our generic framework we can state the α-equivalence of any structure (triples in this case), and not just language terms.

```
alpha-proof : {x y : V}(Ms : μ TreeFTermF)
  → ListNotOccurBind (x :: y :: []) Ms → ListNotOccurBind (fv Ms) Ms → PSComp {x} {y} Ms
alpha-proof {x} {y} ⟨ M , N , L ⟩ nOcc nOcc2 xnIny:fvL
    = begin
        (M [ x := N ]) [ y := L ]
        ≈⟨ lemma-subst-alpha {M [ x := N ]} (lemmaSubsts {x} {M} {N} x:fvN-NB-M)        ⟩
          M [ x := N ]_n [ y := L ]
        ∼⟨ lemmaSubsts {y} {M [ x := N ]_n} {L} y:fvL-NB-M[x:=N]_n                       ⟩
          M [ x := N ]_n [ y := L ]_n
        ∼⟨ lemma-substCompositionN {x} {y} {M} {N} {L} xnIny:fvL x-NB-L                  ⟩
          M [ y := L ]_n [ x := N [ y := L ]_n ]_n
        ∼⟨ lemma-substn-alpha {x} {M [ y := L ]_n} (σ (lemmaSubsts {y} {N} y:fvL-NB-N))  ⟩
          M [ y := L ]_n [ x := N [ y := L ] ]_n
        ∼⟨ σ (lemmaSubsts {x} {M [ y := L ]_n} {N [ y := L ]} x:fvN[y:=L]-NB-M[y:=L]_n)  ⟩
          M [ y := L ]_n [ x := N [ y := L ] ]
        ≈⟨ lemma-subst-alpha (σ (lemmaSubsts {y} {M} {L} y:fvL-NB-M))                     ⟩
        (M [ y := L ]) [ x := N [ y := L ] ]
```
■

Figure 18: Proof of Substitution composition lemma.

6 Conclusions

We address the formalisation of a general first order named syntax with multi-sorted binders by applying a combination of generic programming and nominal techniques to derive fold operations, name-swapping, the α-conversion relation, and α-induction/iteration principles for any language abstract syntax with binders. We derive the λ-calculus and System F as instances of the introduced general framework. For these examples we derive both the naive and the capture-avoiding substitutions as direct instances of the corresponding fold and α-fold principles. We directly inherit the classical substitution lemmas for the α-conversion, and the good behavior of substitution under the name-swapping from fold properties already proved at the generic level. We prove a lemma stating sufficient conditions under which the fold and α-fold functions are α-equivalent. Therefore, as substitution operations are direct instances of these iteration principles, we get in an almost free manner a result about the relation between the naive and the capture-avoiding substitution operations for the λ-calculus and System F. This result is particularly useful in the last proof, which is conducted using the introduced α-proof technique, that enables us to mimic the BVC in a generic setup.

Our work uses generic programming techniques to develop the meta-theory of abstract syntax with binders in a general way as in related works. But we choose to maintain names for binders like as usually done in informal practice. On the other hand, contrary to the historical standpoint, following ideas in [9], we give α-conversion a more fundamental role than that of the definition of substitution. Indeed, we verify that the name-swapping is powerful enough to define a theory of structural induction/recursion modulo α in a general way.

We generalise the α-recursion/induction principles developed in [7, 6]. In these previous works we renamed binders within the fold traversal. Instead, in this work we separate these stages, managing to reuse the fold operation and its properties. We also present an α-proof technique which is not based on an induction principle as in the previous works, and thus can be used to prove properties over relations or composite datatypes. This is the case in the proof of the substitution composition lemma, where it is used to obtain freshness premises over a triple of terms, which is possible because of the generic character of the approach, allowing us to state the α-equivalence or freshness premises over any composite datatype; that is, we are able to state freshness in any mathematical context, thus reflecting the BVC more accurately.

Generic programming techniques are capable of further improvements as the one considered in [8], where a more modular assembly is introduced, enabling a more structured approach to the reuse of meta-theory formalisations through the composition of modular inductive definitions and proofs. The present work does not directly support a modular reuse, but it would be interesting to explore this improvement.

In [1] Reynold's parametricity theory is used to prove the α-compatibility property of a big step semantics using reflection within Coq. They introduce a lambda calculus terms interface, and by a formalisation of Reynold's parametricity, they prove that polymorphic functions (over this interface) applied to related inputs produces related outputs. Then, given two concrete implementations of their lambda terms interface, one with de Brujin syntax (where α-convertible are syntaticaly equal), and another one nominal, they are able to get as a "free theorem" that on α-convertible inputs, the big step function produces α-convertible outputs in the nominal syntax. It remains as future work to study how our generic framework could be used to internalise this kind of "free theorems" by introducing a de Brujin interpretation to our universe, and then translate results between interpretations.

Acknowledgements. We gratefully acknowledge DoD support under award FA9550-16-1-0082 (MURI Program).

References

[1] Abhishek Anand & Greg Morrisett (2017): *Revisiting Parametricity: Inductives and Uniformity of Propositions*. *CoRR* abs/1705.01163. Available at `http://arxiv.org/abs/1705.01163`.

[2] B. Aydemir, A. Bohannon, M. Fairbairn, N. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich & S. Zdancewic (2005): *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In: *Proceedings of TPHOLs'05*, Springer-Verlag, pp. 50–65, doi:10.1007/11541868_4.

[3] Hendrik Barendregt (1984): *The λ-calculus Its Syntax and Semantics*, revised edition. *Studies in Logic and the Foundations of Mathematics* 103, North Holland.

[4] Marcin Benke, Peter Dybjer & Patrik Jansson (2003): *Universes for Generic Programs and Proofs in Dependent Type Theory*. *Nordic Journal of Computing* 10(4), pp. 265–289.

[5] James Cheney: *Scrap Your Nameplate: (Functional Pearl)*.

[6] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2017): *Machine-checked proof of the Church-Rosser Theorem for the Lambda Calculus using the Barendregt Variable Convention in Constructive Type Theory*.

[7] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove & Maribel Fernández (2016): *Alpha-Structural Induction and Recursion for the λ-calculus in Constructive Type Theory*. *ENTCS* 323, pp. 109 – 124, doi:10.1016/j.entcs.2016.06.008.

[8] Benjamin Delaware, Bruno C. d. S. Oliveira & Tom Schrijvers (2013): *Meta-theory à La Carte*. *SIGPLAN Not.* 48(1), pp. 207–218, doi:10.1145/2429069.2429094.

[9] Murdoch J. Gabbay & Andrew M. Pitts (2001): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing* 13(3—5), p. 341—363, doi:10.1007/s001650200016.

[10] Gyesik Lee, Bruno Oliveira, Sungkeun Cho & Kwangkeun Yi (2012): *GMeta: A Generic Formal Metatheory Framework for First-Order Representations*. Springer, doi:10.1.1.298.2957.

[11] Daniel R. Licata & Robert Harper (2009): *A universe of binding and computation*. In: *International Conference on Functional Programming (ICFP)*, pp. 123–134, doi:10.1145/1596550.1596571.

[12] Per Martin-Löf (1984): *Intuitionistic type theory*. *Studies in Proof Theory. Lecture Notes* 1, Naples.

[13] Peter Morris, Thorsten Altenkirch & Conor McBride (2006): *Exploring the Regular Tree Types*. In: *International Conference on Types for Proofs and Programs*, TYPES'04, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/11617990_16.

[14] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, Springer-Verlag, Berlin, doi:10.1145/1481861.1481862.

[15] Benjamin C. Pierce (2002): *Types and Programming Languages*, 1st edition. The MIT Press.

[16] Mark R. Shinwell (2006): *Fresh O'Caml: Nominal Abstract Syntax for the Masses*. *ENTCS* 148(2), doi:10.1016/j.entcs.2005.11.040.

[17] Mark R. Shinwell, Andrew M. Pitts & Murdoch James Gabbay (2003): *FreshML: programming with binders made simple*. *SIGPLAN Notices* 38(9), pp. 263–274, doi:10.1145/944746.944729.

[18] Christian Urban & Christine Tasson (2005): *Nominal Techniques in Isabelle/HOL*. In: *Automated Deduction – CADE-20*, *LNCS* 3632, Springer Berlin Heidelberg, doi:10.1007/s10817-008-9097-2.

[19] Nobuko Yoshida & Vasco T. Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. *ENTCS* 171(4), pp. 73 – 93, doi:10.1016/j.entcs.2007.02.056.