A Scope Safe Universe of Syntaxes with Binding, Their Semantics and Proofs*

Guillaume Allais[1], Robert Atkey[2], James Chapman[2], Conor McBride[2], and James McKinna[3]

1 **Radboud University Nijmegen, Nijmegen, The Netherlands**
`gallais@cs.ru.nl`

2 **University of Strathclyde, Glasgow, United Kingdom**
`firstname.lastname@strath.ac.uk`

3 **LFCS, University of Edinburgh, Edinburgh, United Kingdom**
`james.mckinna@ed.ac.uk`

Abstract

Syntaxes with binding are omnipresent in Programming Languages research but also in the more practical setting of Embedded Domain Specific Languages. The advanced features available in some languages' type systems has made it possible to statically enforce well-scopedness. However the user still has to write a lot of boilerplate code to get common scope safe programs (e.g. renaming, substitution, CPS transformation, printing with names, etc.) and the proof that they are well-behaved.

Building on an abstract but nonetheless expressive notion of semantics and a universe of syntaxes with binding, we demonstrate how to implement these traversals once and for all by generic programming, and how to derive their properties by generic proving. All of this work has been fully formalised in Agda and is available at `https://github.com/gallais/generic-syntax`.

1998 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Dummy keyword – please provide 1–5 keywords

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Nowadays the software programmer writing an embedded DSL [17] and the PL researcher formalising a calculus both know and leverage the host language's type system. Using Generalised Algebraic Data Types (GADTs) or the more general indexed families of Type Theory [15] for their deep embedding, they can *statically* enforce some of the invariants present in their language. Managing the scope is a popular use case [6] as directly manipulating raw de Bruijn indices is error-prone.

This paper starts with primers on scope safe terms, scope preserving programs acting on them and a generic way to represent data types. These introductory sections help us build an understanding of the problem at hand as well as a toolkit that leads us to the original content of this paper: a universe of scope safe syntaxes with binding together with a generic notion of scope safe semantics for these syntaxes. This give us the opportunity to write generic implementations of renaming, substitution but also elaboration of a surface language to a core one, and normalisation by evaluation. We also explore opportunities for generic proving by describing a framework to formally describe what it means for a semantics to be able to simulate another one.

* This work was partially supported by someone.

2 A Primer on Scope Safe Terms

Scope safe terms are following a strict discipline which enforces statically that they may only refer to variables introduced by a binder beforehand. A scope safe language is a programming language in which all the valid terms are guaranteed to be scope safe.

Bellegarde and Hook [8], Bird and Patterson [10], and Altenkirch and Reus [6] introduced the nowadays classic presentation of scope safe languages using inductive *families* [15] to track scoping information at the type level. Instead of describing the type of abstract syntax trees of the language as the fixpoint of an endofunctor on **Set**, they used an endofunctor on $\mathbf{Set}^{\mathbf{Set}}$ where the **Set** index corresponds to the set of variables in scope. Because the empty Set has no inhabitant, it is a natural representation of the empty scope. Conversely, the functor $M(X) = 1 + X$ is used to extend the running scope with an extra variable.

This generic presentation of scope safe languages leads to the following definition of the untyped λ-calculus. The endofunctor $T(F) = \lambda X \in \mathbf{Set}.X + (F(X) \times F(X)) + F(1 + X)$ offers a choice of three constructors. The first one corresponds to the variable case; it packages an inhabitant of X, the index **Set**. The second corresponds to an application node; both the function and its argument live in the same scope as the overall expression. Last but not least, the third corresponds to a λ-abstraction; it extends the current scope with a fresh variable. The language is obtained as the fixpoint of T:

$$Lam = \mu F \in \mathbf{Set}^{\mathbf{Set}}.\lambda X \in \mathbf{Set}.X + (F(X) \times F(X)) + F(1 + X)$$

The proof that the fixpoint is functorial then corresponds to renaming whilst the proof that it is monadic implements substitution: the variable constructor is return and bind defines parallel substitution.

2.1 A Mechanized Variant of Altenkirch and Reus' Untyped Calculus

There is no reason to restrict this technique to fixpoints of endofunctors on $\mathbf{Set}^{\mathbf{Set}}$ apart from the fact that renaming and substitution correspond to well-known structures in that specific case. The more general case of fixpoints of (strictly positive) endofunctors on $\mathbf{Set}^{I}$ can be endowed with similar operations by using what Altenkirch, Chapman and Uustalu [3, 4] refer to as relative monads.

In this paper, we pick $I = \mathbb{N}$ where the natural number used as an index is straightforwardly the number of (de Bruijn) variables in scope. This natural number can be seen as a list associating to each variable in scope an element of the unit type; in a typed setting, it would carry the variable's type instead.

Our implementation language is Agda [22] however these techniques are language independent: any dependently typed language whose logic is at least as powerful as Martin-Löf Type Theory [19] equipped with inductive families [15] ought to do.

In order to lighten the presentation, we weaponise the observation that the current scope is either threaded to subterms (e.g. in the application's case) or adjusted (e.g. in the λ-abstraction's case) by introducing combinators to build indexed types. Although it may seem surprising at first to define infix operators of arity three, they are meant to be used partially applied, surrounded by [_] which turns an indexed Set into a Set by implicitly quantifying over the index. The first two combinators are the pointwise liftings of the function space and the product type respectively, both silently threading the underlying scope. The third one is simply the constant function turning a Set into an indexed Set by ignoring the index. Finally, the last one makes explicit the *adjustment* made to the index by a function. We use Agda's mixfix operator notation (where underscores denote argument positions) to suggest its connection to the mathematical convention of only mentioning context *extensions* when presenting judgements (see e.g. [19]) and write $f \vdash T$ where f is the modification and T the indexed set is operates on.

```
_→̇_ : (I → Set) → (I → Set) → (I → Set)        _×̇_ : (I → Set) → (I → Set) → (I → Set)
(S →̇ T) i = S i → T i                           (S ×̇ T) i = S i × T i

κ : Set → (I → Set)     _⊢_ : (I → I) → (I → Set) → (I → Set)     [_] : (I → Set) → Set
κ S i = S               (f ⊢ T) i = T (f i)                        [ T ] = ∀ {i} → T i
```

Figure 1 Combinators to build indexed Sets

For instance, the fairly compact expression $[\ \mathsf{suc} \vdash (P \mathbin{\dot{\times}} Q) \mathbin{\dot{\to}} R\ ]$ corresponds to the more verbose type $\forall\ \{i\} \to (P\ (\mathsf{suc}\ i) \times Q\ (\mathsf{suc}\ i)) \to R\ i$. Using these combinators, the untyped λ-calculus can be represented using the following definitions:

```
data Var : ℕ → Set where                 data Lam : ℕ → Set where
  z : [          suc ⊢ Var ]               V : [ Var           →̇ Lam ]
  s : [ Var ⟶   suc ⊢ Var ]               A : [ Lam →̇ Lam    →̇ Lam ]
                                           L : [ suc ⊢ Lam     →̇ Lam ]
```

Figure 2 Scope Aware Variables and Untyped λ-Terms

The inductive family Var corresponds to well scoped de Bruijn [14] indices. Its first constructor (z for zero) states that we have a name to refer to the nearest binder in a non-empty scope. The second one (s for successor) lifts a name for a variable in a given scope into a name for it in the extended scope where an extra variable has been bound. Both of their types have been written using combinators, altought we will abstain from unfolding them in the future, we do so here in the hope it will help the reader get acquainted with them: they respectively normalise to $\forall n.\mathbf{Var}(suc(n))$ for z, and $\forall n.\mathbf{Var}(n) \to \mathbf{Var}(suc(n))$ for s.

The ℕ-indexed family Lam is the variant of Altenkirch and Reus' untyped λ-calculus. The two interesting constructors are the one lifting variables to terms and the λ-abstraction whose body lives in an extended context.

3 A Primer on Scope Safe Programs

This scope safe deep embedding of the untyped λ-calculus is naturally only a start: once the programmer has access to a good representation of the language she is interested in, she wants and needs to (re)implement standard traversals manipulating terms. Renaming and substitution are perhaps the two most iconic examples of such traversals. And now that well-scopedness is enforced in the terms' indices, all of these traversals have to be implemented in a scope safe manner. These constraints show up in the type of renaming and substitution which can be defined as follows:

Looking more closely at these two functions' code, it is quite evident that they have a very similar structure. In each case, we have used two (function specific) auxiliary definitions named ⟦V⟧ and extend respectively to highlight this fact. Abstracting away this shared structure would allow for these definitions to be refactored, and their common properties to be proved in one swift move.

Previous efforts in dependently typed programming [9, 2] have precisely achieved this goal and refactored renaming and substitution, but also normalisation by evaluation, printing with names or CPS conversion as various instances of a more general traversal. Unpublished results also demonstrate that typechecking in the style of Atkey [7] fits in that framework. To be able to make sense of

```
ren : (Var m → Var n) → Lam m → Lam n          sub : (Var m → Lam n) → Lam m → Lam n
ren ρ (V k)     = ⟦V⟧ (ρ k)                     sub ρ (V k)     = ⟦V⟧ (ρ k)
ren ρ (A f t)   = A (ren ρ f) (ren ρ t)         sub ρ (A f t)   = A (sub ρ f) (sub ρ t)
ren ρ (L b)     = L (ren (extend ρ) b)          sub ρ (L b)     = L (sub (extend ρ) b)
```

Figure 3 Scope Preserving Renaming and Substitution

this body of work, we need to introduce three new definitions:

A Thinning from m to n is a function from Var m to Var n. It is a special case of a notion of environment that stores values living in a scope n for each variable in a scope m. We introduce environment as records (this helps the host language's type inference reconstruct the type of values) and write $(m$ —Env) $\mathcal{V}$ n for such an environment with values in $\mathcal{V}$.

```
record _—Env (m : ℕ) (𝒱 : ℕ → Set) (n : ℕ) : Set where      Thinning : ℕ → ℕ → Set
  constructor pack; field lookup : Var m → 𝒱 n              Thinning m n = (m —Env) Var n
```

Figure 4 Environments of Well Scoped Values and Thinnings

Thinnings subsume more structured notions such as the Category of Weakenings [5] or Order Preserving Embeddings [12]. In particular, it does not prevent the user from defining arbitrary permutations or from introducing contractions although we will not use such instances. The fact that our representation is a function space grants us monoid laws "for free" as per Jeffrey's observation [18].

The □ combinator turns any ℕ-indexed Set into one that can absorb thinnings. It is akin to Kripke-style quantification over all possible future worlds and □ $(D \to D)$ indeed corresponds to the Kripke function space used in normalisation by evaluation via a domain D. Because we can define an identity Thinning and Thinnings do compose, □ is a comonad.

The notion of Thinnable is the property of being stable under thinnings, in other words: Thinnables are the coalgebras of □. It is a crucial property for values to have if one wants to be able to push them under binders. Unsurprisingly, from the comonadic structure we get that the □ combinator freely turns any ℕ-indexed Set into a Thinnable one.

```
□ : (ℕ → Set) → (ℕ → Set)                      Thinnable : (ℕ → Set) → Set
(□ T) m = [ Thinning m ∸→ T ]                   Thinnable T = [ T ∸→ □ T ]

extract   : [ □ T ∸→ T         ]                th□ : Thinnable (□ T)
duplicate : [ □ T ∸→ □ (□ T)   ]                th□ = duplicate
```

Figure 5 The □ comonad, Thinnable, and the cofree Thinnable.

Equipped with these new notions, we can define an abstract concept of semantics for our scope safe language. Broadly speaking, a semantics turns our deeply embedded abstract syntax trees into the shallow embedding of the corresponding parametrised higher order abstract syntax term. We get various semantics by using different 'host languages' for this shallow embedding. A semantics with values in $\mathcal{V}$ and computations in $\mathcal{C}$ is meant to give rise to a traversal which, provided a term and an environment of values for each one of the variables in scope, delivers a computation. The traversal sem realises this specification generically for all Semantics.

A Semantics is characterised by a set of constraints. First of all, values should be thinnable so that sem may push the environment under binders. Second, the set of computations needs to be closed under various combinators which are the semantical counterparts of the language's constructors. Here the semantical counterpart of application is not particularly interesting. However the interpretation of the λ-abstraction is of interest: it is a variant on the Kripke function space one can find in normalisation by evaluation. In all possible thinnings of the scope at hand, it promise to deliver a computation whenever it is provided with a value for its newly bound variable. This is concisely expressed by the type ($\square$ ($\mathcal{V} \to \mathcal{C}$)).

```
record Sem (𝒱 𝒞 : ℕ → Set) : Set where
  field  th^𝒱 : Thinnable 𝒱
         ⟦V⟧  : [ 𝒱              →̇ 𝒞 ]
         ⟦A⟧  : [ 𝒞 →̇ 𝒞          →̇ 𝒞 ]
         ⟦L⟧  : [ □ (𝒱 →̇ 𝒞)      →̇ 𝒞 ]
```

```
sem : (m ─Env) 𝒱 n → (Lam m → 𝒞 n)
sem ρ (V k)     = ⟦V⟧ (lookup ρ k)
sem ρ (A f t)   = ⟦A⟧ (sem ρ f) (sem ρ t)
sem ρ (L b)     = ⟦L⟧ (λ σ v → sem (extend σ ρ v) b)
```

Figure 6 Semantics for Lam and their Fundamental Lemma

Coming back to renaming and substitution, we can see that they do fit in the Sem framework. We also include the definition of a very basic printer relying on a name supply to highlight the fact that computations can very well be effectful. Both Printing and Renaming highlight the importance of having a distinct notion of values and computations: the type of values in their respective environments are distinct from their type of computations.

```
Renaming : Sem Var Lam
Renaming = record
  { th^𝒱  = th^Var
  ; ⟦V⟧   = V
  ; ⟦A⟧   = A
  ; ⟦L⟧   = λ b → L (b (pack s) z) }


Substitution : Sem Lam Lam
Substitution = record
  { th^𝒱  = λ t ρ → sem Renaming ρ t
  ; ⟦V⟧   = id
  ; ⟦A⟧   = A
  ; ⟦L⟧   = λ b → L (b (pack s) (V z)) }
```

```
Printing : Sem (λ _ → String) (λ _ → State ℕ String)
Printing = record
  { th^𝒱  = λ t _ → t
  ; ⟦V⟧   = return
  ; ⟦A⟧   =  λ mf mt → mf >>= λ f → mt >>= λ t →
             return $ f ++ " (" ++ t ++ ") "
  ; ⟦L⟧   =  λ mb → get >>= λ x → put (suc x) >>
             let x′ = show x in mb (pack s) x′ >>= λ b →
             return $ "λ" ++ x′ ++ " . " ++ b }
```

Figure 7 Renaming, Substitution and Printing as Instances of Sem

All of these examples are already desribed at length by Allais, Chapman, McBride and McKinna [2] so we will not spend any more time on them. They have also obtained the simulation and fusion theorems demonstrating that these traversals are well-behaved as corollaries of more general results expressed in terms of that generic traversal. We will come back to this in Section 8.

One important observation to make is the tight connection between the constraint described by Sem and the definition of Lam: the semantical combinators are related to the corresponding constructors where the recursive occurences of the inductive family have been replaced with either a computation or a Kripke function space whenever an extra variable was bound. This suggest that it ought to be possible to compute the definition of Sem from the one of the datatype.

4 A Primer on the Universe of Data Types

Chapman, Dagand, McBride and Morris [11] defined a universe of data types inspired by Dybjer and Setzer's finite axiomatisation of Inductive-Recursive definitions [16]. This explicit definition of *codes* for data types empowers the user to write generic programs tackling *all* of the data types one can obtain this way. In this section we recall the main aspects of this construction we are interested in to build up our generic representation of syntaxes with binding.

The first component of this universe's definition is an inductive type of Descriptions of strictly positive functors on **Set**. It has three constructors: one to store data (the rest of the description can depend upon this stored value), one to attach a recursive substructure and one to stop.

These constructors give the programmer the ability to build up the data types she is used to. For instance, the functor corresponding to lists of elements in A stores one bit of data: whether the current node is the empty list or not. Depending on that bit, the rest of the description is either the "stop" token or a pair of an element in A and a recursive substructure (i.e. the tail of the list).

```
data Desc : Set₁ where
  `σ : (A : Set) (d : A → Desc) → Desc
  `X : Desc                     → Desc
  `■ :                            Desc
```

```
listD : Set → Desc
listD A = `σ Bool $ λ isNil →
          if isNil then `■ else `σ A (λ _ → `X `■)
```

Figure 8 Data Descriptions and List Description

The recursive function ⟦_⟧ makes the interpretation of the Descriptions formal by associating a value in $\mathbf{Set}^{\mathbf{Set}}$ to each one of them. They essentially give rise to unit terminated right nested tuples. Together with ⟦_⟧, we can define the fmap recursive function witnessing the fact that the meaning of a description is indeed functorial. This is the first example of generic programming over all the functors one can obtain as the meaning of a description.

```
⟦_⟧ : Desc → (Set → Set)
⟦ `σ A d ⟧ X = Σ[ a ∈ A ] (⟦ d a ⟧ X)
⟦ `X d   ⟧ X = X × ⟦ d ⟧ X
⟦ `■     ⟧ X = ⊤
```

```
fmap : (d : Desc) → (X → Y) → (⟦ d ⟧ X → ⟦ d ⟧ Y)
fmap (`σ A d) f (a , v)  = (a , fmap (d a) f v)
fmap (`X d)   f (r , v)  = (f r , fmap d f v)
fmap `■       f t        = t
```

Figure 9 Meaning of Descriptions and Proof of Functoriality

All the functors obtained as meanings of Descriptions are strictly positive so we can build their least fixpoint μ which is the initial algebra corresponding to d's meaning as proven by fold d.

```
data μ (d : Desc) : Size → Set where
  `con : ⟦ d ⟧ (μ d i) → μ d (↑ i)
```

```
fold : (d : Desc) → (⟦ d ⟧ X → X) → μ d i → X
fold d alg (`con t) = alg (fmap d (fold d alg) t)
```

Figure 10 Fixpoint and Generic Fold

Here the Size [1] index added to the inductive definition of μ plays a crucial role in getting the termination checker to see that fold is a total function. Indeed the recursive calls to fold are performed indirectly via fmap and it's only because the compiler knows that i, the index at which the recursive calls are performed, is strictly smaller than ↑ i that it accepts the definition as total. Without this type based approach to termination checking the definition of fold would be rejected and our only recourse would be to manually inline fmap. However, in most definitions the Size does not matter in which case we will simply use the primitive limit Size ∞ which is characterised by ∞ = ↑ ∞.

This demonstrates that this approach allows us to define generically the iteration principle associated to all the datatypes which arise as the fixpoint of a description's meaning. It seems appropriate to base our universe of scope safe syntaxes on a similar construction so that we may be able to define generically a notion of semantics for all the syntaxes with binding one may come up with. Chapman, Dagand, McBride and Morris also give a more general universe which supports higher order branching (but still does not have a notion of variable). We decidedly stick to finitary constructors, thus sticking to the common understanding of 'syntax'.

5 A Universe of Scope Safe Syntaxes

Our universe of scope safe syntaxes follows the same principle as McBride's universe of datatypes except that we are not building endofunctors on **Set** but rather $\textbf{Set}^{\mathbb{N}}$. Descriptions can be built using three constructors: the first one makes it possible to store data (and, as usual, the rest of the description may depend upon the value stored), the second takes a natural number m and corresponds to a substructure with exactly m additional variable in scope and the last one ends the definition.

The meaning function ⟦_⟧ we associate to a description is not quite an endofunctor on $\textbf{Set}^{\mathbb{N}}$; it is more general than that. Given an X that interprets substructures with an extra m bound variables in a scope that has n bound variables already as $X\ m\ n$, we give a description a meaning as an $\textbf{Set}^{\mathbb{N}}$. The astute reader may have noticed that ⟦_⟧ is uniform in X and n; however refactoring ⟦_⟧ to use the partially applied X_n following this observation would lead to a definition harder to use with the combinators for indexed sets described in Figure 1 which make our types much more readable.

```
data Desc : Set₁ where                              ⟦_⟧ : Desc → (ℕ → ℕ → Set) → (ℕ → Set)
  `σ : (A : Set) (d : A → Desc)  →  Desc            ⟦ `σ A d ⟧ X n = Σ[ a ∈ A ] (⟦ d a ⟧ X n)
  `X : ℕ  → Desc                 →  Desc            ⟦ `X m d ⟧ X n = X m n × ⟦ d ⟧ X n
  `■ :                              Desc            ⟦ `■     ⟧ X n = ⊤
```

Figure 11 Syntax Descriptions and their Meaning as Functor

If we pre-compose the meaning function ⟦_⟧ with a notion of 'scope' (denoted Scope here) which turns any ℕ-indexed family into a function of type ℕ → ℕ → **Set** by simply summing the two indices, we recover an endofunctor on $\textbf{Set}^{\mathbb{N}}$ and we can take its fixpoint. This time, instead of considering the initial algebra, we opt for the free relative monad [4]: the `var constructor corresponds to return and we will define bind (also known as sub) in the next section. We have once more a Size index to get all the benefits of type based termination checking.

Coming back to the well-scoped untyped λ-calculus, we now have the ability to give its code. The variable constructor will be introduced by the free monad construction so we only have to describe two cases: the application case where we have two substructures which do not bind any extra argument and the λ-abstraction case which has exactly one substructure with precisely one extra bound variable.

We can then define constructors corresponding to the original ones in Figure 2: `V for V the variable constructor, `A for A the application one and `L for L the λ-abstraction. In Agda, we can

```
data Tm (d : Desc) : Size → ℕ → Set where
  `var :  [ Var                        ∙→ Tm d (↑ i) ]
  `con :  [ ⟦ d ⟧ (Scope (Tm d i))  ∙→ Tm d (↑ i) ]
```

```
Scope : (ℕ → Set) → (ℕ → ℕ → Set)
Scope T m = (m +_) ⊢ T
```

Figure 12 Term Trees: The Free Relative Monads on Descriptions

actually almost define these as pattern synonyms (the language currently does not allow the user to specify type annotations for her pattern synonyms), meaning that the end user can seemlessly write pattern-matching programs on encoded terms without dealing with the gnarly details of the encoding.

```
LCD : Desc
LCD =  `σ Bool $ λ isApp →
          if isApp then `X 0 (`X 0 `■) else `X 1 `■
```

```
LC : ℕ → Set
LC = Tm LCD ∞
```

```
`V : [ Var ∙→ LC ]
`V = `var
```

```
`A : [ LC ∙→ LC ∙→ LC ]
`A f t = `con (true , f , t , tt)
```

```
`L : [ suc ⊢ LC ∙→ LC ]
`L b = `con (false , b , tt)
```

Figure 13 Example: The Untyped Lambda Calculus

It is the second time (the first time being the definition of listD in Figure 8) that we use a Bool to distinguish between two constructors. In order to avoid re-encoding the same logic, the next section introduces combinators demonstrating that descriptions are closed under sums and products.

5.1 Common Combinators and Their Properties

As we have seen previously, we can take the coproduct of two descriptions by using a dependent pair whose first component stores a Boolean tagging which branch was taken whilst the second one uses that information to return the description corresponding to that branch. We can define an appropriate eliminator case which given two continuations picks the one corresponding to the chosen branch.

```
_`+_ : Desc → Desc → Desc
d `+ e =  `σ Bool $ λ isLeft →
             if isLeft then d else e
```

```
case : (⟦ d       ⟧ ρ n  → A) →
       (⟦ e       ⟧ ρ n  → A) →
       (⟦ d `+ e  ⟧ ρ n  → A)
```

Figure 14 Descriptions are closed under Sums

Closure under product is however a bit more technical: it is defined by induction on the first description of the product. The definition is purely structural except for the "stop" constructor which gets replaced by the second description. Because of the indirect nature of the definition of closure under products, it is convenient to have functions going back and forth between the interpretation of a product of descriptions and the product of their respective interpretations.

```
_`×_ : Desc → Desc → Desc
`σ A d  `× e = `σ A (λ a → d a `× e)
`X k d  `× e = `X k (d `× e)
`∎       `× e = e
```

```
pair   : [ ⟦ d ⟧ ρ ⇾ ⟦ e ⟧ ρ ⇾ ⟦ d `× e ⟧ ρ ]
unpair : [ ⟦ d `× e ⟧ ρ ⇾ ⟦ d ⟧ ρ ×̇ ⟦ e ⟧ ρ   ]
```

Figure 15 Descriptions are closed under Products

6 Generic Scope Safe Programs for Syntaxes

Based on the structure made explicit in the example worked out in Section 3, we can define a generic notion of semantics for all syntax descriptions. It is once more parametrised by two ℕ-indexed families $\mathcal{V}$ and C corresponding respectively to values associated to bound variables and computations delivered by evaluating terms. These two families have to abide by three constraints

- Values should be thinnable for us to be able to push the evaluation environment under binders;
- Values should embed into computations for us to be able to the return the value associated to a variable as the result of its evaluation;
- Last but not least, we should have an algebra turning a term where substructures have been replaced with either computations or kripke functional spaces (depending on whether extra bound variables have been introduced) into computations

Here we crucially use the fact that the meaning of a description is defined in terms of a function interpreting substructures which has the type ℕ → ℕ → **Set**, i.e. that gets access to the current scope but also the exact number of newly-bound variables. We define such a function, Kripke, by case analysis on the number of newly-bound variables: if it's 0 then we expect the substructure to simply be a computation (the result of the evaluation function's recursive call) but if there are newly bound variables then we expect a function which takes one value for each one of them and delivers a computation corresponding to the evaluation of the body of the binder in the extended environment.

```
Kripke : (𝒱 C : ℕ → Set) → (ℕ → ℕ → Set)
Kripke 𝒱 C 0 = C
Kripke 𝒱 C m = □ ((m ─Env) 𝒱 ⇾ C)
```

```
record Sem (d : Desc) (𝒱 C : ℕ → Set) : Set where
  field th^𝒱  : Thinnable 𝒱
        var   : [ 𝒱                    ⇾ C ]
        alg   : [ ⟦ d ⟧ (Kripke 𝒱 C)   ⇾ C ]
```

Figure 16 A Generic Notion of Semantics

It is once more the case that the abstract notion of Semantics comes with a fundamental lemma: all ℕ-indexed families $\mathcal{V}$ and C satisfying the three criteria we have put forward give rise to an evaluation function. Here the fundamental lemma is called sem and it is defined mutually with a function body turning a Scope (i.e. a substructure in a potentially extended context) into a Kripke (i.e. a subcomputation expecting a value for each newly bound variable).

Renaming can be defined generically for all syntax descriptions as a semantics with Var as values and Tm as computations. The two first constraints on Var described earlier are trivially satisfied. Because renaming strictly respects the structure of the term it goes through, the algebra is implemented using fmap. When dealing with the body a binder, we simply 'reify' the Kripke function by evaluating it in an extended context and feeding it dummy values corresponding to the extra variables introduced by that context. This is reminiscent both of what we did in Section 3 and the definition of reification in the setting of normalisation by evaluation (see e.g. Coquand's work [13]).

```
sem  : (m ─Env) 𝒱 n → Tm d i m → 𝒞 n
body : (m ─Env) 𝒱 n → ∀ k → Scope (Tm d i) k m → Kripke 𝒱 𝒞 k n

sem ρ (`var k) = var (lookup ρ k)             body ρ 0       t = sem ρ t
sem ρ (`con t) = alg (fmap d (body ρ) t)      body ρ (suc k) t = λ ren vs → sem (vs ≫ th^Env th^𝒱 ρ ren) t
```

Figure 17 Fundamental Lemma of Semantics

Substitution can be defined in a similar manner. Of the two constraints applying to terms as values, the first one corresponds precisely to renaming and the second one is trivial. The algebra can once more be defined by using fmap and reifying the bodies of binders. This reification process can be implemented generically for semantics which have “VarLike” values i.e. values that are thinnable and such that we can craft dummy ones in non-empty contexts.

```
record VarLike (𝒱 : ℕ → Set) : Set where        reify : VarLike 𝒱 → ∀ m → Kripke 𝒱 𝒞 m n → Scope 𝒞 m n
  field  new  : [ suc ⊢ 𝒱 ]                      reify vl^𝒱 zero          b = b
         th^𝒱 : Thinnable 𝒱                       reify vl^𝒱 m@(suc _)     b = b (fresh vl^Var m) (fresh vl^𝒱 m)

Renaming : ∀ d → Sem d Var (Tm d ∞)              Substitution : ∀ d → Sem d (Tm d ∞) (Tm d ∞)
Renaming d = record                              Substitution d = record
  { th^𝒱 = λ k ρ → lookup ρ k                      { th^𝒱 = λ t ρ → Sem.sem (Renaming d) ρ t
  ; var  = `var                                    ; var  = id
  ; alg  = `con ∘ fmap d (reify vl^Var) }           ; alg  = `con ∘ fmap d (reify vl^Tm) }

ren :  ∀ d → (m ─Env) Var n →                    sub :  ∀ d → (m ─Env) (Tm d ∞) n →
         Tm d ∞ m → Tm d ∞ n                              Tm d ∞ m → Tm d ∞ n
ren d = Sem.sem (Renaming d)                     sub d = Sem.sem (Substitution d)
```

Figure 18 Generic Renaming and Substitution for All Scope Safe Syntaxes with Binding

7 Other Generic Programs

7.1 Sugar and Desugaring as a Semantics

One of the advantages of having a universe of programming languages descriptions is the ability to concisely define an *extension* of an existing language by using Description transformers grafting extra constructors à la Swiestra [23]. This is made extremely simple by the disjoint sum combinator _`+_ which we have already seen in Section 5.1. An example of such an extension is the addition of let-bindings to an existing language. Let bindings allow the user to avoid repeating herself by naming sub-expressions and then using these names to refer to the associated terms.

In a dependently typed language a type may depend on a value which in the presence of let bindings may be a variable standing for an expression. The user naturally does not want it to make any difference whether she used a variable referring to a let-bound expression or the expression itself. Various typechecking strategies can accomodate this expectation: in Coq [20] let bindings

```
_times_ : ℕ → (A → A) → (A → A)
(zero  times f) d = d
(suc n  times f) d = f ((n times f) d)
```

```
Let : Desc
Let = `σ ℕ (λ n → (n times `X 0) `■ `× `X n `■)
```

Figure 19 Parallel Let Binding

are primitive constructs of the language and have their own typing and reduction rules whereas in Agda they are elaborated away to the core language by inlining.

This latter approach to extending a language d with let bindings by inlining them before type-checking can be implemented generically as a Semantics over (Let `+ d) where values in the environment and computations both are let-free terms. The algebra of that semantics can be defined by parts: the old constructors are simply interpreted using the algebra defined generically for the Substitution semantics whilst the newer one precisely provides the extra values to be added to the environment (we leave the definition of alg′ out because of a lack of space). The process of removing let binders is kickstarted with a dummy environment associating each variable to itself.

```
UnLet : ∀ d → Sem (Let `+ d) (Tm d ∞) (Tm d ∞)
UnLet d = record
  { th^𝒱  = th^Tm
  ; var  = id
  ; alg  = case alg′ (Sem.alg (Substitution d)) }
```

```
unlet : [ Tm (Let `+ d) ∞ ⇾ Tm d ∞ ]
unlet = Sem.sem (UnLet _) (pack `var)
```

Figure 20 Inlining Let Binding

In about 20 lines of code we have defined a generic extension of syntaxes with binding together with a semantics which corresponds to an elaborator translating away this new constructor. In their own setting working on a given language, Allais, Chapman, McBride and McKinna [2] have shown that it is similarly possible to implement a Continuation Passing Style transformation as a semantics.

The ease with which one can define such generic transformations suggests that this setup could be a good candidate to implement generic compilation passes.

7.2 (Unsafe) Normalisation by Evaluation

A key type of traversal we have not studied yet is a language's evaluator. Our universe of syntaxes with binding does not impose any typing discipline on the user defined languages and as such cannot guarantee their totality. This is embodied by our running example: the untyped λ-calculus. As a consequence there is no hope for a safe generic framework to define normalisation functions.

The clear connection between the Kripke functional space characteristic of our semantics and the one that shows up in normalisation by evaluation suggests we ought to manage to give a generic framework for normalisation by evaluation. By temporarily **disabling Agda's positivity checker**, we can define a generic reflexive domain Dm in which to interpret our syntaxes. It has three constructors corresponding respectively to a free variable, a constructor's counterpart where scopes have become Kripke functional spaces on Dm and an error token because the evaluation of untyped programs may (and usually does!) go wrong.

This datatype definition is utterly unsafe. The more conservative user will happily restrict herself to typed settings where the domain can be defined as a logical predicate or opt instead for a step-

```
{-# NO_POSITIVITY_CHECK #-}
data Dm (d : Desc) : Size → ℕ → Set where
  V : [ Var                                 ⇾  Dm d i      ]
  C : [ ⟦ d ⟧ (Kripke (Dm d i) (Dm d i))  ⇾  Dm d (↑ i)  ]
  ⊥ : [                                        Dm d (↑ i)  ]
```

Figure 21 Generic Reflexive Domain

indexed approach. But this domain does make it possible to define a generic nbe semantics as well as a reification function turning elements of the reflexive domain into terms. By composing them, we obtain the normalisation function which gives its name to normalisation by evaluation.

The user still has to explicitly pass an interpretation of the various constructors because there is no way for us to know what the binders are supposed to represent: they may stand λ-abstractions, Σ-types, fixpoints, or anything else she may want to define.

```
reify^Dm   : [ Dm d i ⇾ Maybe ∘ Tm d ∞ ]
nbe        : [ ⟦ d ⟧ (Kripke (Dm d ∞) (Dm d ∞)) ⇾ Dm d ∞ ] → Sem d (Dm d ∞) (Dm d ∞)

norm       : [ ⟦ d ⟧ (Kripke (Dm d ∞) (Dm d ∞)) ⇾ Dm d ∞ ] → [ Tm d ∞ ⇾ Maybe ∘ Tm d ∞ ]
norm alg   = reify^Dm ∘ Sem.sem (nbe alg) (refl vl^Dm)
```

Figure 22 Generic Normalisation by Evaluation Framework

Using this setup, we can write a normaliser for the untyped λ-calculus: we use case to distinguish between the semantical counterpart of the application constructor on one hand and the λ-abstraction one on the other. The latter is trivial: functions are already values! The semantical counterpart of application proceeds by case analysis on the function: if it corresponds to a λ-abstraction, we can fire the redex by using the kripke functional space; otherwise we grow the spine of stuck applications.

```
norm^LC : [ LC ⇾ Maybe ∘ LC ]
norm^LC = norm $ case app (C ∘ (false , _)) where

  app : [ ⟦ 'X 0 ('X 0 '■) ⟧ (Kripke (Dm LCD ∞) (Dm LCD ∞)) ⇾ Dm LCD ∞ ]
  app (C (false , f , _)  , t  , _) = f (refl vl^Var) (ε • t)  - redex
  app (f                  , t  , _) = C (true , f , t , _)     - stuck application
```

Figure 23 Normalisation by Evaluation for the Untyped λ-Calculus

8 Building Generic Proofs about Generic Programs

Allais, Chapman, McBride, and McKinna [2] have already shown that, for their specific language, introducing an abstract notion of Semantics not only reveals the shared structure of common traversals, it also allows them to give abstract proof frameworks for simulation or fusion lemmas. Their idea naturally extends to our generic presentation of semantics for all syntaxes.

The most important concept in this section is (Zip d), a relation transformer which characterises structurally equal layers such that their substructures are themselves related by the relation it is passed as an argument. It is defined by induction on the description and case analysis on the two layers which are meant to be equal. It inherits a lot of its relational arguments' properties: whenever R is reflexive (respectively symmetric or transitive) so is Zip d R.

```
Zip : (d : Desc) (R : (m : ℕ) → [ X m ∸→ Y m ∸→ κ Set ]) → [ ⟦ d ⟧ X ∸→ ⟦ d ⟧ Y ∸→ κ Set ]
Zip `∎         R x       y        = ⊤
Zip (`X k d)  R (r , x)  (r′ , y)  = R k r r′ × Zip d R x y
Zip (`σ A d)  R (a , x)  (a′ , y)  = Σ[ eq ∈ a′ ≡ a ] Zip (d a) R x (rew eq y)
  where rew = subst (λ a → ⟦ d a ⟧ _ _)
```

Figure 24 Zip: Characterising Structurally Equal Values with Related Substructures

8.1 Simulation Lemma

A Zip constraint appears naturally when we want to say that a semantics can simulate another one. Given a relation connecting values in $\mathcal{V}_1$ and $\mathcal{V}_2$, and a relation connecting computations in $\mathcal{C}_1$ and $\mathcal{C}_2$, we can define Kripke^R relating values Kripke $\mathcal{V}_1$ $\mathcal{C}_1$ and Kripke $\mathcal{V}_2$ $\mathcal{C}_2$ by stating that they send related inputs to related outputs. We use the relation transformer ∀[_] which lifts a relation on values to one on environments.

```
Kripkeᴿ : (m : ℕ) → [ Kripke 𝒱₁ 𝒞₁ m ∸→ Kripke 𝒱₂ 𝒞₂ m ∸→ κ Set ]
Kripkeᴿ zero     k₁ k₂ = ℛᶜ k₁ k₂
Kripkeᴿ (suc _)  k₁ k₂ = ∀[ ℛᵛ ] ρ₁ ρ₂ → ℛᶜ (k₁ σ ρ₁) (k₂ σ ρ₂)
```

Figure 25 Relational Kripke Function Spaces: From Related Inputs to Related Outputs

We can then use Zip together with Kripke^R to express the idea that two semantic objects of respective types ⟦ d ⟧ (Kripke $\mathcal{V}_1$ $\mathcal{C}_1$) and ⟦ d ⟧ (Kripke $\mathcal{V}_2$ $\mathcal{C}_2$) are synchronised. The simulation constraint on two Semantics' algebras then becomes: given synchronized objects, the algebras should yield related computations. Together with self-explanatory constraints on var and th$^{\mathcal{V}}$, this constitutes the whole Simulation constraint:

```
record Sim (d : Desc) (𝒮₁ : Sem d 𝒱₁ 𝒞₁) (𝒮₂ : Sem d 𝒱₂ 𝒞₂) : Set where
  field thᴿ   : (σ : Thinning m n) → ℛᵛ v₁ v₂ → ℛᵛ (Sem.thᵛ 𝒮₁ v₁ σ) (Sem.thᵛ 𝒮₂ v₂ σ)
        varᴿ  : ℛᵛ v₁ v₂ → ℛᶜ (Sem.var 𝒮₁ v₁) (Sem.var 𝒮₂ v₂)
        algᴿ  : Zip d Kripkeᴿ b₁ b₂ → ℛᶜ (Sem.alg 𝒮₁ b₁) (Sem.alg 𝒮₂ b₂)
```

Figure 26 A Generic Notion of Simulation

We can prove a generic theorem showing that for each pair of Semantics respecting the Simulation constraint, we get related computations given environments of related input values. This theorem is once more mutually proven with a statement about Scopes, and Sizes play a crucial role in ensuring that the function is indeed total.

```
sim   : (t : Tm d i m) → ℛᶜ (Sem.sem 𝒮₁ ρ₁ t) (Sem.sem 𝒮₂ ρ₂ t)
body  : (m : ℕ) (t : Scope (Tm d i) m n) →
        Kripkeᴿ m (Sem.body 𝒮₁ ρ₁ m t) (Sem.body 𝒮₂ ρ₂ m t)
```

```
rensub : (d : Desc) (ρ : Thinning m n) (t : Tm d ∞ m) →
         Sem.sem (Renaming d) ρ t ≡ Sem.sem (Substitution d) (`var <$> ρ) t
rensub d ρ = Sim.sim (RenSub d) (packᴿ (λ _ → _≡_.refl))
```

Figure 27 Fundamental Lemma of Simulations and Renaming as a Substitution as its Corollary

Instantiating this generic simulation lemma, we can for instance get that Renaming is a special case of Substitution. This example is the simplest of the abstract proof frameworks Allais, Chapman, McBride and McKinna introduce for their specific language. They also explain how a similar framework can be defined for fusion lemmas and deploy it for the renaming-substitution interactions but also their respective interactions with normalisation by evaluation.

9 Conclusion

We have started from an example of a scope safe language (the untyped λ-calculus), have studied various common traversals and noticed their similarity. After introducing a notion of semantics and refactoring these traversals as various instances of the same fundamental lemma, we have observed the tight connection between the abstract definition of semantics and the shape of the language. By extending a universe of datatype descriptions to support a notion of binding, we have managed to give a generic presentation of syntaxes with binding as well as a large class of scope safe programs acting on them: from Renaming and Substitution, to Normalisation by Evaluation, and the Desugaring of new constructors added by a language transformer. Last but not least, we have seen how to construct generic proofs about these generic programs. The diverse influences leading to this body of work suggest many opportunities for future research:

The question of a universe of syntaxes with binding which are not only well scoped but also intrinsically well typed by construction is an exciting challenge. The existing variation on the universe of datatypes giving a universe of inductive families [15] is a natural candidate.

Our example of the elaboration of an enriched language to a core one, and Allais, Chapman, McBride and McKinna's implementation of a Continuation Passing Style conversion function begs the question of how many such common compilation passes can be implemented generically. An extension of McBride's theory of ornaments [21] could provide an appropriate framework to highlight the connection between various languages, some being seen as the extension of others.

References

1 Andreas Abel. Miniagda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010.*, volume 43 of *EPTCS*, pages 14–28, 2010. doi:10.4204/EPTCS.43.2.

2 Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 195–207, New York, NY, USA, 2017. ACM. doi:10.1145/3018610.3018613.

3 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. *Monads Need Not Be Endofunctors*, pages 297–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. `doi:10.1007/978-3-642-12032-9_21`.

4 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Relative monads formalised. *Journal of Formalized Reasoning*, 7(1):1–43, 2014.

5 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *LNCS*, volume 530, pages 182–199. Springer, 1995.

6 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL*, pages 453–468. Springer, 1999.

7 Robert Atkey. An algebraic approach to typechecking and elaboration. 2015. URL: `http://bentnib.org/posts/2015-04-19-algebraic-approach-typechecking-and-elaboration.html`.

8 Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2):287 – 311, 1994.

9 Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in Coq. *JAR*, 49(2):141–159, 2012.

10 Richard S. Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

11 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. `doi:10.1145/1863543.1863547`.

12 James Maitland Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

13 Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15(1):57–90, 2002.

14 Nicolaas Govert de Bruijn. Lambda Calculus notation with nameless dummies. In *Indagationes Mathematicae*, volume 75, pages 381–392. Elsevier, 1972.

15 Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

16 Peter Dybjer and Anton Setzer. *A Finite Axiomatization of Inductive-Recursive Definitions*, pages 129–146. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. `doi:10.1007/3-540-48959-2_11`.

17 Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

18 Alan Jeffrey. Associativity for free! `http://thread.gmane.org/gmane.comp.lang.agda/3259`, 2011.

19 Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104:153–175, 1982.

20 The Coq Development Team. *The Coq proof assistant reference manual*. πr^2 Team, 2017. Version 8.6. URL: `http://coq.inria.fr`.

21 Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of functional programming*, 2017.

22 Ulf Norell. Dependently typed programming in Agda. In *AFP Summer School*, pages 230–266. Springer, 2009.

23 Wouter Swiestra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. `doi:10.1017/S0956796808006758`.