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

GUILLAUME ALLAIS, Radboud University, NL
ROBERT ATKEY, JAMES CHAPMAN, and CONOR MCBRIDE, University of Strathclyde, UK
JAMES MCKINNA, University of Edinburgh, UK

Almost every programming language's syntax includes a notion of binder and corresponding bound occurrences, along with the accompanying notions of α-equivalence, capture avoiding substitution, typing contexts, runtime environments, and so on. In the past, implementing and reasoning about programming languages required careful handling to maintain the correct behaviour of bound variables. Modern programming languages include features that enable constraints like scope safety to be expressed in types. Nevertheless, the programmer is still forced to write the same boilerplate over again for each new implementation of a scope safe operation (e.g., renaming, substitution, desugaring, printing, etc.), and then again for correctness proofs.

We present an expressive universe of syntaxes with binding and demonstrate how to (1) implement scope safe traversals once and for all by generic programming; and (2) how to derive properties of these traversals by generic proving. Our universe description, generic traversals and proofs, and our examples have all been formalised in Agda and are available in the accompanying material.

NB. we recommend printing the paper in colour to benefit from syntax highlighting in code fragments.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

ACM Reference Format:
Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding, Their Semantics and Proofs. *ACM Trans. Web* 9, 4, Article 39 (March 2018), 31 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In modern typed programming languages, programmers writing embedded DSLs [Hudak 1996] and researchers formalising them can now use the host language's type system to help them. Using Generalised Algebraic Data Types (GADTs) or the more general indexed families of Type Theory [Dybjer 1994] for representing their syntax, programmers can *statically* enforce some of the invariants in their languages. Managing variable scope is a popular use case [Altenkirch and Reus 1999] as directly manipulating raw de Bruijn indices is error-prone. Solutions range from enforcing well scopedness to ensuring full type and scope correctness. In short, we use types to ensure that "illegal states are unrepresentable", where illegal states are ill scoped or ill typed terms.

Despite the large body of knowledge in how to use types to define well formed syntax (see the Related Work in Section 9), it is still necessary for the working DSL designer or formaliser to redefine essential functions like renaming and substitution for each new syntax, and then to reprove essential lemmas about those functions. To reduce the burden of such repeated work

Authors' addresses: Guillaume Allais, Radboud University, NL; Robert Atkey; James Chapman; Conor McBride, University of Strathclyde, UK; James McKinna, University of Edinburgh, UK.

1559-1131/2018/3-ART39 $15.00
https://doi.org/0000001.0000001

and boilerplate, we apply the methodology of data-genericity to programming and proving with syntaxes with binding.

To motivate our approach, let us look at the formalisation of an apparently straightforward program transformation: the inling of let-bound variables by substitution. You have two languages: the source (S), which has let-bindings, and the target (T), which only differs in that it does not:

$$\mathsf{S} ::= x \mid \mathsf{S}\ \mathsf{S} \mid \lambda x.\mathsf{S} \mid \text{let } x = \mathsf{S} \text{ in } \mathsf{S} \qquad \mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Breaking the task down, you need to define an operational semantics for each language, define the program transformation itself, and prove a correctness lemma that states each step in the source language is simulated by zero or more steps of the transformed terms in the target language. In the course of doing this, you discover that you actually have a large amount of work to do:

(1) To define the operational semantics you need to define substitution, and hence renaming, for both the source and target languages, even though they are very similar;
(2) In the course of proving the correctness lemma, you discover that you need to prove eight lemmas about the interactions of renaming, substitution, and transformation that are all remarkably similar, but must be stated and proved separately (e.g, as in [Benton et al. 2012]).

Even after doing all of this work, you have only a result for a single pair of source and target languages. If you were to change your languages S or T, you would have to repeat the same work all over again (or at least do a lot of cutting, pasting, and editing).

Using the universe of syntaxes with binding we present in this paper, we are able to solve this repetition problem *once and for all.*

Content and Contributions. We start with primers on scoped and sorted terms (Section 2), scope and sort safe programs acting on them (Section 3), and programmable descriptions of data types (Section 4). These introductory sections help us build an understanding of the problem at hand as well as a toolkit that leads us to the novel content of this paper: a universe of scope safe syntaxes with binding (Section 5) together with a notion of scope safe semantics for these syntaxes (Section 6). This gives us the opportunity to write generic implementations of renaming and substitution (Section 6.2), a generic let-binding removal transformation (generalising the problem stated above) (Section 7.1), and normalisation by evaluation (Section 7.2). Further, we show how to construct generic proofs by formally describing what it means for a semantics to be able to simulate another one (Section 8.1), or for two semantics to be fusable (Section 8.2). This allows us to prove the lemmas required above for renaming and substitution generically, for *every* syntax in our universe.

Our implementation language is Agda [Norell 2009]. However, our techniques are language independent: any dependently typed language at least as powerful as Martin-Löf Type Theory [Martin-Löf 1982] equipped with inductive families [Dybjer 1994] such as Coq [The Coq Development Team 2017], Lean [de Moura et al. 2015] or Idris [Brady 2013] ought to do.

2 A PRIMER ON SCOPE AND SORT SAFE TERMS

Scope safe terms follow the discipline that every variable is either bound by some binder or is explicitly accounted for in a context. Bellegarde and Hook (1994), Bird and Patterson (1999), and Altenkirch and Reus (1999) introduced the classic presentation of scope safety using inductive *families* [Dybjer 1994] instead of inductive types to represent abstract syntax. Indeed, using a family indexed by a **Set**, we can track scoping information at the type level. The empty **Set** represents the empty scope. The functor $1 + (\_)$ extends the running scope with an extra variable.

An inductive type is the fixpoint of an endofunctor on **Set**. Similarly, an inductive family is the fixpoint of an endofunctor on **Set** → **Set**. Using inductive families to enforce scope safety, we get the following definition of the untyped λ-calculus: $T(F) = \lambda X \in \textbf{Set}.\ X + (F(X) \times F(X)) + F(1 + X)$.

This endofunctor 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. 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:

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

Since 'Lam' is a endofunction on Set, it makes sense to ask whether it is also a functor and a monad. Indeed it is, as Altenkirch and Reus have shown. The functorial action corresponds to renaming, the monadic 'return' corresponds to the use of variables, and the monadic 'join' corresponds to substitution. The functor and monad laws correspond to well known properties from the equational theories of renaming and subsitution. We will revisit these properties below in Section 8.2.

2.1 A Mechanized Typed Variant of Altenkirch and Reus' Calculus

There is no reason to restrict this technique to fixpoints of endofunctors on $\mathbf{Set}^{\mathbf{Set}}$. The more general case of fixpoints of (strictly positive) endofunctors on $\mathbf{Set}^{J}$ can be endowed with similar operations by using Altenkirch, Chapman and Uustalu's relative monads (2010; 2014).

We pick as our J the category whose objects are inhabitants of List I (I is a parameter of the construction) and whose morphisms are thinnings (see Section 3). This List I is intended to represent the list of the sort (/ kind / types depending on the application) of the de Bruijn variables in scope. We can recover an untyped approach by picking I to be the unit type. Given this typed setting, our functors take an extra I argument corresponding to the type of the expression being built. This is summed up by the large type I –Scoped:

$$\begin{aligned}&\_\text{–Scoped} : \mathsf{Set} \to \mathsf{Set}_1\\ &I\text{–Scoped} = I \to \mathsf{List}\ I \to \mathsf{Set}\end{aligned}$$

We use Agda's mixfix operator notation where underscores denote argument positions.

To lighten the presentation, we exploit the observation that the current scope is either passed unchanged to subterms (e.g. in the application case) or extended (e.g. in the λ-abstraction case) by introducing combinators to build indexed types.

$$\begin{aligned}&\_\dot{\to}\_ : (A \to \mathsf{Set}) \to (A \to \mathsf{Set}) \to (A \to \mathsf{Set}) && \_\dot{\times}\_ : (A \to \mathsf{Set}) \to (A \to \mathsf{Set}) \to (A \to \mathsf{Set})\\ &(S \mathrel{\dot{\to}} T)\ a = S\ a \to T\ a && (S \mathrel{\dot{\times}} T)\ a = S\ a \times T\ a\end{aligned}$$

$$\begin{aligned}&\_\vdash\_ : (A \to A) \to (A \to \mathsf{Set}) \to (A \to \mathsf{Set}) && \kappa : \mathsf{Set} \to (A \to \mathsf{Set}) && [\_] : (A \to \mathsf{Set}) \to \mathsf{Set}\ (\ell^{A})\\ &(f \vdash T)\ a = T\ (f\ a) && \kappa\ S\ a = S && [\ T\ ] = \forall\ \{a\} \to T\ a\end{aligned}$$

We lift the function space and the product type pointwise with $\_\dot{\to}\_$ and $\_\dot{\times}\_$ respectively, silently threading the underlying scope. The $\_\vdash\_$ makes explicit the *adjustment* made to the index by a function, conforming to the convention (see e.g. [Martin-Löf 1982]) of mentioning only context *extensions* when presenting judgements and write $f \vdash T$ where f is the modification and T the indexed Set it operates on. Although it may seem surprising at first to define binary infix operators as having 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. Lastly, κ is the constant combinator, ignoring the index.

We make $\dot{\to}$ associate to the right as one would expect and give it the highest precedence level as it is the most used combinator. These combinators lead to more readable type declarations. For instance, the compact expression $[\ \mathsf{suc} \vdash (P \mathrel{\dot{\times}} Q) \mathrel{\dot{\to}} R\ ]$ desugars to the more verbose type $\forall\ \{i\} \to (P\ (\mathsf{suc}\ i) \times Q\ (\mathsf{suc}\ i)) \to R\ i$.

As the context comes second in the definition of _−Scoped, we can readily use these combinators to thread, modify, or quantify over the scope when defining such families:

```
data Var : I −Scoped where                     data Lam : Type −Scoped where
  z :  [                (i ::_) ⊢ Var i ]        V :  [ Var σ                   ∸→ Lam σ             ]
  s :  [ Var i ∸→       (j ::_) ⊢ Var i ]        A :  [ Lam (σ ⇒ τ) ∸→ Lam σ    ∸→ Lam τ             ]
                                                 L :  [ (σ ::_) ⊢ Lam τ         ∸→ Lam (σ ⇒ τ)       ]
```

The inductive family Var represents well scoped and well kinded de Bruijn (1972) indices. Its z (for zero) constructor refers to the nearest binder in a non-empty scope. The s (for successor) constructor lifts a a variable in a given scope to the extended scope where an extra variable has been bound. Both of the constructors' types have been written using the combinators defined above. They respectively normalise to:

```
z : ∀ i xs → Var i (i :: xs)        s : ∀ i j xs → Var i xs → Var i (j :: xs)
```

The Type −Scoped family Lam is Altenkirch and Reus' simply typed λ-calculus representation.

3 A PRIMER ON TYPE AND SCOPE SAFE PROGRAMS

The scope-and-type safe representation described in the previous section is naturally only a start: once the programmer has access to a good representation of the language they are interested in, they will naturally want to (re)implement standard traversals manipulating terms. Renaming and substitution are the two most typical examples of such traversals. Now that well-typedness and well-scopedness are enforced statically, all of these traversals have to be implemented in a type and scope safe manner.

These constraints show up in the types of renaming and substitution defined as follows:

```
ren : (Γ −Env) Var Δ → Lam σ Γ → Lam σ Δ          sub : (Γ −Env) Lam Δ → Lam σ Γ → Lam σ Δ
ren ρ (V k)     = ⟦V⟧_ren (lookup ρ k)             sub ρ (V k)     = ⟦V⟧_sub (lookup 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_ren ρ) b)         sub ρ (L b)     = L (sub (extend_sub ρ) b)
```

Fig. 1. Type and Scope Preserving Renaming and Substitution

We have voluntarily hidden technical details behind some auxiliary definitions left abstract here: ⟦V⟧ and extend. Their implementations are distinct for ren and sub but they serve the same purpose: ⟦V⟧ is used to turn a value looked up in the evaluation environment into a term and extend is used to alter the environment when going under a binder. This presentation highlights the common structure between ren and sub which we will exploit later in this section, particularly in Figures 5 and 6 where we define an abstract notion of semantics and the corresponding generic traversal.

Both renaming and substitution are defined in terms of *environments* (Γ −Env) $\mathcal{V}$ Δ that describe how to associate a value $\mathcal{V}$ (variables for renaming, terms for substitution) well scoped and typed in Δ to every entry in Γ. Environments are defined as the following record structure (using a record helps Agda's type inference reconstruct the type of values $\mathcal{V}$ for us):

As we have already observed, the definitions of renaming and substitution have very similar structure. 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 [Allais et al. 2017; Benton et al. 2012] have achieved this goal and refactored renaming and substitution, but also normalisation by evaluation,

```
record _─Env (Γ : List I) (𝒱 : I ─Scoped) (Δ : List I) : Set where
  constructor pack
  field lookup : ∀ {i} → Var i Γ → 𝒱 i Δ
```

Fig. 2. Well Typed and Scoped Environments of Values

printing with names or CPS conversion as various instances of a more general traversal. As we will show in Section 7.3, typechecking in the style of Atkey (2015) also fits in that framework. To make sense of this body of work, we need to introduce three new notions: Thinning, a generalisation of renaming; Thinnables which are types that permit thinning; and the □ functor, which freely adds Thinnability to any indexed type. We use □, and our compact notation for the indexed function space between indexed types, to crisply encapsulate the additional quantification over environment extensions which is typical of Kripke semantics.

```
Thinning : List I → List I → Set
Thinning Γ Δ = (Γ ─Env) Var Δ
```

Fig. 3. Thinnings: A Special Case of Environments

Thinnings subsume more structured notions such as the Category of Weakenings [Altenkirch et al. 1995] or Order Preserving Embeddings [Chapman 2009]. In particular, they do not prevent the user from defining arbitrary permutations or from introducing contractions although we will not use such instances. However, such extra flexibility will not get in our way, and permits a representation as a function space which grants us monoid laws "for free" as per Jeffrey's observation (2011).

The □ combinator turns any (List I)-indexed Set into one that can absorb thinnings. This is accomplished by abstracting over all possible thinnings from the current scope, akin to an S4-style necessity modality. The axioms of S4 modal logic incite us to observe that the functor □ is a comonad: extract applies the identity Thinning to its argument, and duplicate is obtained by composing the two Thinnings we are given. The expected laws hold trivially thanks to Jeffrey's trick mentionned above.

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. From the comonadic structure we get that the □ combinator freely turns any (List I)-indexed Set into a Thinnable one.

```
□ : (List I → Set) → (List I → Set)          Thinnable : (List I → Set) → Set
(□ T) Γ = [ Thinning Γ →̇ T ]                 Thinnable T = [ T →̇ □ T ]

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

Fig. 4. The □ comonad, Thinnable, and the cofree Thinnable.

As Allais, Chapman, McBride and McKinna (ACMM) (2017) shows, equipped with these new notions we can define an abstract concept of semantics for our scope-and-type safe language (cf. Figures 5 and 6). 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 a choice of useful scope-and-sort safe traversals by using different ‘host languages’ for this shallow embedding.

Semantics, specified in terms of a record Sem, are defined in Figure 5 in terms of a choice of values $\mathcal{V}$ and computations $\mathcal{C}$. Realisation of a semantics will produce a computation in $\mathcal{C}$ for every term whose variables are assigned values in $\mathcal{V}$ as demonstrated in Figure 6. A semantics must satisfy constraints on the notions of values $\mathcal{V}$ and computations $\mathcal{C}$ at hand. 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. The semantical counterpart of application is an operation that takes a representation of a function and a representation of an argument and produces a representation of the result. The interpretation of the λ-abstraction is of particular 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 promises to deliver a computation whenever it is provided with a value for its newly bound variable. This is concisely expressed by the type ($\square$ ($\mathcal{V}\ \sigma \longrightarrow \mathcal{C}\ \tau$)).

```
record Sem (𝒱 𝒞 : Type ─Scoped) : Set where
  field th^𝒱  : ∀ {σ} → Thinnable (𝒱 σ)
        ⟦V⟧   :                 [ 𝒱 σ               →̇ 𝒞 σ          ]
        ⟦A⟧   :                 [ 𝒞 (σ ⇒ τ) →̇ 𝒞 σ   →̇ 𝒞 τ          ]
        ⟦L⟧   : (σ : Type) →    [ □ (𝒱 σ →̇ 𝒞 τ)     →̇ 𝒞 (σ ⇒ τ)    ]
```

Fig. 5. Semantics for Lam

Agda allows us to package, together with the fields of the record Sem, the generic traversal function sem, which is brought into scope for any instance of Sem. We thus realise the promise made earlier, namely that any given Sem $\mathcal{V}$ $\mathcal{C}$ induces a function which, given a value in $\mathcal{V}$ for each variable in scope, transforms a Lam term into a computation $\mathcal{C}$.

```
sem : (Γ ─Env) 𝒱 Δ → (Lam σ Γ → 𝒞 σ Δ)
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)
```

Fig. 6. Fundamental Lemma of Semantics for Lam, relative to a given Sem $\mathcal{V}$ $\mathcal{C}$

Coming back to renaming and substitution, we see that they both fit in the Sem framework. We notice that the definition of substitution depends on the definition of renaming: to be able to push terms under binder, we need to have already proven that they are thinnable.

In both cases we use (pack s) (where pack is the constructor for environments and s, defined in Section 2.1, is the function lifting an existing de Bruijn variable into an extended scope) as the definition of the thinning embedding Γ into $\sigma :: \Gamma$.

We also include the definition of a basic printer relying on a name supply to highlight the fact that computations can very well be effectful. The Printing semantics is defined by using Strings as values and State ℕ String as computations. We use a Wrapper with a type and a context as

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

Fig. 7. Renaming and Substitution as Instances of Sem

```
record Wrap (A : Set) (σ : Type) (Γ : List Type) : Set where
   constructor MkW; field getW : A

fresh : ∀ σ → State ℕ (Wrap String σ (σ :: Γ))
fresh σ = get >>= λ x → put (suc x) >> return (MkW (show x))
```

Fig. 8. Wrapper and fresh name generation

phantom types in order to help Agda's inference propagate the appropriate constraints. We define a function fresh that generates new concrete names using a State monad.

The wrapper Wrap does not depend on the scope Γ so it is automatically a Thinnable functor. We jump straight to the definition of the printer. To print an application, we produce a string representation of the term in function position, then of its argument and combine them by putting the argument between parentheses. To print a λ-abstraction, we start by generating a fresh name for the newly-bound variable, use that name to generate a string representing the body of the function to which we prepend a "λ" binding the fresh name.

```
Printing : Sem (Wrap String) (Wrap (State ℕ String))
Printing = record
   { th^𝒱   =   th^Wrap
   ; ⟦V⟧    =  map^Wrap return
   ; ⟦A⟧    =  λ mf mt → MkW $ getW mf >>= λ f → getW mt >>= λ t →
               return $ f ++ "(" ++ t ++ ")"
   ; ⟦L⟧    =  λ σ mb → MkW $ fresh σ >>= λ x →
               getW (mb extend x) >>= λ b →
               return $ "λ" ++ getW x ++ "." ++ b }
```

Fig. 9. Printing as an instance of Sem

Both printing and renaming highlight the importance of distinguishing values and computations: the type of values in their respective environments are distinct from their type of computations.

All of these examples are already described at length by ACMM (2017) 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 sem. We will come back to this in Section 8.1.

One important observation to make is the tight connection between the constraints described in Sem and the definition of Lam: the semantical counterparts of the Lam constructors are obtained by replacing the recursive occurences of the inductive family with either a computation or a Kripke function space depending on whether an extra variable was bound. This suggests that it ought to be possible to compute the definition of Sem from the syntax description. Before doing this in Section 5, we need to look at a generic descriptions of datatypes.

4 A PRIMER ON THE UNIVERSE OF DATA TYPES

Chapman, Dagand, McBride and Morris (CDMM) (2010) defined a universe of data types inspired by Dybjer and Setzer's finite axiomatisation of Inductive-Recursive definitions (1999) and Benke, Dybjer and Jansson's universes for generic programs and proofs (2003). 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 CDMM's universe's definition is an inductive type of Descriptions of strictly positive functors from $\mathbf{Set}^J$ to $\mathbf{Set}^I$. It has three constructors: ‘σ to store data (the rest of the description can depend upon this stored value), ‘X to attach a recursive substructure indexed by J and ‘■ to stop with a particular index value.

The recursive function ⟦_⟧ makes the interpretation of the descriptions formal. Interpretation of descriptions give rise right-nested tuples terminated by equality constraints.

```
data Desc (I J : Set) : Set₁ where                     ⟦_⟧ : Desc I J → (J → Set) → (I → Set)
  ‘σ : (A : Set) → (A → Desc I J)  →  Desc I J         ⟦ ‘σ A d  ⟧ X i = Σ[ a ∈ A ] (⟦ d a ⟧ X i)
  ‘X : J → Desc I J                →  Desc I J         ⟦ ‘X j d  ⟧ X i = X j × ⟦ d ⟧ X i
  ‘■ : I                           →  Desc I J         ⟦ ‘■ i′   ⟧ X i = i ≡ i′
```

Fig. 10. Datatype Descriptions and their Meaning as Functors

These constructors give the programmer the ability to build up the data types they are used to. For instance, the functor corresponding to lists of elements in A stores a Boolean which stands for whether the current node is the empty list or not. Depending on that value, 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. The List type is unindexed, we represent the lack of an index with the unit type ⊤.

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

Fig. 11. The Description of the base functor for List A

Indexes can be used to enforce invariants. For example, the type Vec A n of length-indexed lists. It has the same structure as the definition of listD. We start with a Boolean distinguishing the two constructors: either the empty list (in which case the branch's index is enforced to be 0) or a non-empty one in which case we store a natural number n, the head of type A and a tail of size n (and the branch's index is enforced to be suc n).

```
vecD : Set → Desc ℕ ℕ
vecD A =  ‘σ Bool $ λ isNil →
          if isNil then ‘■ 0
          else ‘σ ℕ (λ n → ‘σ A (λ _ → ‘X n (‘■ (suc n))))
```

Fig. 12. The Description of the base functor for Vec *A n*

The payoff for encoding our datatypes as descriptions is that we can define generic programs for whole classes of data types. The decoding function ⟦_⟧ acted on the objects of $\mathbf{Set}^J$, and we will now define the function fmap by recursion over a code *d*. It describes the action of the functor corresponding to *d* over morphisms in $\mathbf{Set}^J$. This is the first example of generic programming over all the functors one can obtain as the meaning of a description.

```
fmap : (d : Desc I J) → [ X ⟶̇ Y ] → [ ⟦ d ⟧ X ⟶̇ ⟦ d ⟧ Y ]
fmap (‘σ A d)  f (a , v)   = (a , fmap (d a) f v)
fmap (‘X j d)  f (r , v)   = (f r , fmap d f v)
fmap (‘■ i)    f t         = t
```

Fig. 13. Action on Morphisms of the Functor corresponding to a Description

All the functors obtained as meanings of Descriptions are strictly positive. So we can build the least fixpoint of the ones that are endofunctors (i.e. the ones for which *I* equals *J*). This fixpoint is called μ and its iterator is given by the definition of fold *d*[1] .

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

fold : (d : Desc I I) → [ ⟦ d ⟧ X ⟶̇ X ] → [ μ d s ⟶̇ X ]
fold d alg (‘con t) = alg (fmap d (fold d alg) t)
```

Fig. 14. Least Fixpoint of an Endofunctor and Corresponding Generic Fold

The CDMM approach therefore allows us to generically define iteration principles for all data types that can be described. These are exactly the features we desire for a universe of data types with binding, so in the next section we will see how to extend CDMM's approach to include binding.

The functor underlying any well scoped and sorted syntax can be coded as some Desc (I × List I) (I × List I), with the free monad construction from CDMM uniformly adding the variable case. Whilst a good start, Desc treats its index types as unstructured, so this construction is blind to what makes the List I index a *scope*. The resulting 'bind' operator demands a function which maps variables in *any* sort and scope to terms in the *same* sort and scope. However, the behaviour we need is to preserve sort while mapping between specific source and target scopes which may differ. We need to account for the fact that scopes change only by extension, and hence that our specifically scoped operations can be pushed under binders by weakening.

[1] **NB** In Figure 14 the Size [Abel 2010] 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.

5 A UNIVERSE OF SCOPE SAFE AND WELL KINDED SYNTAXES

Our universe of scope safe and well kinded syntaxes follows the same principle as CDMM's universe of datatypes, except that we are not building endofunctors on Set anymore but rather on I –Scoped. We now think of the index type I as the sorts used to distinguish terms in our embedded language. The ‘σ and ‘■ constructors are as in the CDMM Desc type, and are used to represent data and index constraints respectively. What distinguishes this new universe Desc from that of Section 4 is that the ‘X constructor is now augmented with an additional List I argument that describes the new binders that are brought into scope at this recursive position. This list of the kinds of the newly-bound variables will play a crucial role when defining the description's semantics as a binding structure in Figures 16, 17 and 18.

```
data Desc (I : Set) : Set₁ where
  ‘σ : (A : Set) → (A → Desc I)   → Desc I
  ‘X : List I → I → Desc I        → Desc I
  ‘■ : I                          → Desc I
```

Fig. 15. Syntax Descriptions

The meaning function ⟦_⟧ we associate to a description follows closely its CDMM equivalent. It only departs from it in the ‘X case and the fact it is not an endofunctor on I –Scoped; it is more general than that. The function takes an X of type List $I \longrightarrow I$ –Scoped to interpret ‘X Δ j (i.e. substructures of sort j with newly-bound variables in Δ) in an ambient scope Γ as $X\,\Delta\,j\,\Gamma$.

```
⟦_⟧ : Desc I → (List I → I ─Scoped) → I ─Scoped
⟦ ‘σ A d    ⟧ X i Γ = Σ[ a ∈ A ] (⟦ d a ⟧ X i Γ)
⟦ ‘X Δ j d  ⟧ X i Γ = X Δ j Γ × ⟦ d ⟧ X i Γ
⟦ ‘■ i′     ⟧ X i Γ = i ≡ i′
```

Fig. 16. Descriptions' Meanings

The astute reader may have noticed that ⟦_⟧ is uniform in X and Γ; however refactoring ⟦_⟧ to use the partially applied X _ _ Γ following this observation would lead to a definition harder to use with the combinators for indexed sets described in Section 2.1 which make our types much more readable.

If we pre-compose the meaning function ⟦_⟧ with a notion of ‘de Bruijn scopes’ (denoted Scope here) which turns any I –Scoped family into a function of type List $I \longrightarrow I$ –Scoped by appending the two List indices, we recover a meaning function producing an endofunctor on I –Scoped. So far we have only shown the action of the functor on objects; its action on morphisms is given by a function fmap defined by induction over the description just like in Section 4.

```
Scope : I ─Scoped → List I → I ─Scoped
Scope T Δ i = (Δ ++_) ⊢ T i
```

Fig. 17. De Bruijn Scopes

The endofunctors thus defined are strictly positive and we can take their fixpoints. As we want to define the terms of a language with variables, instead of considering the initial algebra, this time we opt for the free relative monad [Altenkirch et al. 2014] (with respect to the functor Var): the ‘var constructor corresponds to return, and we will define bind (also known as the parallel substitution sub) in the next section.

```
data Tm (d : Desc I) : Size ⟶ I –Scoped where
  ‘var : [ Var i                       →̇ Tm d (↑ s) i ]
  ‘con : [ ⟦ d ⟧ (Scope (Tm d s)) i     →̇ Tm d (↑ s) i ]
```

Fig. 18. Term Trees: The Free Var-Relative Monads on Descriptions

Coming back to our original examples, we now have the ability to give codes for the well scoped untyped λ-calculus and, just as well, the intrinsically typed simply typed λ-calculus. The variable case will be added by the free monad construction so we only have to describe two constructors: application where we have two substructures which do not bind any extra argument and λ-abstraction which has exactly one substructure with precisely one extra bound variable. In the untyped case a single Boolean is enough to distinguish the two constructors whilst in the typed case, we need our tags to carry extra information about the types involved so we use the ad-hoc ‘STLC type, and its decoding STLC defined by a pattern-matching λ-expression in Agda.

```
UTLC : Desc ⊤
UTLC =   ‘σ Bool $ λ isApp ⟶ if isApp
         then ‘X [] tt (‘X [] tt (‘■ tt))
         else ‘X (tt :: []) tt (‘■ tt)
```

```
data ‘STLC : Set where
  App Lam : Type ⟶ Type ⟶ ‘STLC

STLC : Desc Type
STLC =   ‘σ ‘STLC $ λ where
  (App σ τ) ⟶ ‘X [] (σ ⇒ τ) (‘X [] σ (‘■ τ))
  (Lam σ τ) ⟶ ‘X (σ :: []) τ (‘■ (σ ⇒ τ))
```

Fig. 19. Examples: The Untyped and Simply Typed Lambda Calculi

For convenience we use Agda’s pattern synonyms corresponding to the original constructors in Section 2.1: ‘V for V the variable constructor, ‘A for A the application one and ‘L for L the λ-abstraction. These synonyms can be used when pattern-matching on a term and Agda resugars them when displaying a goal. This means that the end user can seamlessly work with encoded terms without dealing with the gnarly details of the encoding. These pattern definitions can omit some arguments by using “_”, in which case they will be filled in by unification just like any other implicit argument: there is no extra cost to using an encoding! The only downside is that the language currently does not allow the user to specify type annotations for pattern synonyms.

```
pattern ‘V x     = ‘var x                       pattern ‘V x     = ‘var x
pattern ‘A f t   = ‘con (true , f , t , refl)   pattern ‘A f t   = ‘con (App _ _ , f , t , refl)
pattern ‘L b     = ‘con (false , b , refl)      pattern ‘L b     = ‘con (Lam _ _ , b , refl)
```

Fig. 20. Respective Pattern Synonyms for the Untyped and Simply Typed Lambda Calculus

It is the third time (the first and second times being the definition of listD and vecD in Figure 11 and 12) 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 finite sums and finite products of recursive positions.

5.1 Common Combinators and Their Properties

As seen previously, we can use a dependent pair whose first component is a Boolean to take the coproduct of two descriptions: depending on the value of the first component, we will return one or the other. We can abstract this common pattern as a combinator _‘+_ together with an appropriate eliminator case which, given two continuations, picks the one corresponding to the chosen branch.

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

```
case : (⟦ d      ⟧ X i Γ → A) →
       (⟦ e      ⟧ X i Γ → A) →
       (⟦ d ‘+ e ⟧ X i Γ → A)
```

Fig. 21. Descriptions are closed under Sum

Closure under product does not hold in general. Indeed, the equality constraints introduced by the two end tokens of two descriptions may be incompatible. So far, a limited form of closure (closure under finite product of recursive positions) has been sufficient for all of our use cases. As with coproducts, the appropriate eliminator unXs takes a value in the encoding and extracts its constituents (All P xs is defined in Agda's standard library and makes sure that the predicate P holds true of all the elements in the list xs).

```
‘Xs : List I → Desc I → Desc I
‘Xs js d = foldr (‘X []) d js
```

```
unXs : (Δ : List I) → ⟦ ‘Xs Δ d ⟧ X i Γ →
       All (λ i → X [] i Γ) Δ × ⟦ d ⟧ X i Γ
```

Fig. 22. Descriptions are closed under Finite Products of Recursive Positions

A concrete use case for both of these combinators will be given in section 7.1 where we explain how to seamlessly enrich an existing syntax with let-bindings and how to use the Sem framework to elaborate them away.

6 GENERIC SCOPE SAFE AND WELL KINDED PROGRAMS FOR SYNTAXES

Based on the Sem type we defined for the specific example of the simply typed λ-calculus in Section 3, we can define a generic notion of semantics for all syntax descriptions. It is once more parametrised by two I–Scoped 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:

- th$^{\mathcal{V}}$ Values should be thinnable so that we can push the evaluation environment under binders;
- var Values should embed into computations for us to be able to return the value associated to a variable as the result of its evaluation;
- alg We should have an algebra turning a term whose substructures have been replaced with computations (possibly under some binders, represented semantically by the Kripke type-valued function defined below) into computations

```
record Sem (d : Desc I) (𝒱 𝒞 : I ─Scoped) : Set where
field  th^𝒱  : Thinnable (𝒱 i)
       var   : [ 𝒱 i                      ⟶̇ 𝒞 i ]
       alg   : [ ⟦ d ⟧ (Kripke 𝒱 𝒞) i     ⟶̇ 𝒞 i ]
```

Fig. 23. A Generic Notion of Semantics

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 List *I* ⟶ *I*–Scoped, i.e. that gets access to the current scope but also the exact list of the newly bound variables' kinds. We define a function Kripke by case analysis on the number of newly bound variables. It is essentially a subcomputation waiting for a value associated to each one of the fresh variables.

- If it's 0 we expect the substructure to be a computation corresponding to the result of the evaluation function's recursive call;
- But if there are newly bound variables then we expect to have a function space. In any context extension, it will take an environment of values for the newly-bound variables and produce a computation corresponding to the evaluation of the body of the binder.

```
Kripke :   (𝒱 𝒞 : I ─Scoped) ⟶ (List I ⟶ I ─Scoped)
Kripke 𝒱 𝒞 []   i = 𝒞 i
Kripke 𝒱 𝒞 Γ    i = □ ((Γ ─Env) 𝒱 ⟶̇ 𝒞 i)
```

Fig. 24. Substructures as either Computations or Kripke Function Spaces

It is once more the case that the abstract notion of Semantics comes with a fundamental lemma: all *I* ─Scoped families 𝒱 and 𝒞 satisfying the three criteria we have put forward give rise to an evaluation function. We introduce a notion of computation _─Comp analogous to that of environments: instead of associating values to variables, it associates computations to terms.

```
_─Comp : List I ⟶ I ─Scoped ⟶ List I ⟶ Set
(Γ ─Comp) 𝒞 Δ = Tm d s i Γ ⟶ 𝒞 i Δ
```

Fig. 25. _─Comp: Associating Computations to Terms

6.1 Fundamental Lemma of Semantics

We can now define the type of the fundamental lemma (called sem) which takes a semantics and returns a function from environments to computations. It is defined mutually with a function body turning syntactic binders into semantics binders: to each de Bruijn Scope (i.e. a substructure in a potentially extended context) it associates a Kripke (i.e. a subcomputation expecting a value for each newly bound variable).

The proof of sem is straightforward now that we have clearly identified the problem structure and the constraints we need to enforce. We use postfix projections (of the form .name) to make use of the semantic combinators packaged in the Sem parameter 𝒮. If the term considered is a variable, we lookup the associated value in the evaluation environment and turn it into a computation using

```
sem   :  Sem d 𝒱 𝒞 → (Γ ─Env) 𝒱 Δ → (Γ ─Comp) 𝒞 Δ
body  :  Sem d 𝒱 𝒞 → (Γ ─Env) 𝒱 Δ → ∀ Θ i → Scope (Tm d s) Θ i Γ → Kripke 𝒱 𝒞 Θ i Δ
```

Fig. 26. Statement of the Fundamental Lemma of Semantics

var. If it is a non variable constructor then we call fmap to evaluate the substructures using body and then call the algebra to combine these results.

```
sem 𝒮 ρ (‘var k) = (𝒮 .var) (lookup ρ k)
sem 𝒮 ρ (‘con t) = (𝒮 .alg) (fmap d (body 𝒮 ρ) t)
```

Fig. 27. Proof of the Fundamental Lemma of Semantics – sem

The auxiliary lemma body distinguishes two cases. If no new variable has been bound in the recursive substructure, it is a matter of calling sem recursively. Otherwise we are provided with a Thinning, some additional values and evaluate the substructure in the thinned and extended evaluation environment (thanks to a auxiliary function _>>_ which given two environments (Γ ─Env) 𝒱 Θ and (Δ ─Env) 𝒱 Θ produces an environment ((Γ ++ Δ) ─Env) 𝒱 Θ).

```
body 𝒮 ρ []        i t = sem 𝒮 ρ t
body 𝒮 ρ (_ :: _)  i t = λ σ vs → sem 𝒮 (vs >> th^Env (𝒮 .th^𝒱) ρ σ) t
```

Fig. 28. Proof of the Fundamental Lemma of Semantics – body

Given that fmap introduces one level of indirection between the recursive calls and the subterms they are acting upon, the fact that our terms are indexed by a Size is once more crucial in getting the termination checker to see that our proof is indeed well founded.

6.2 Our First Generic Programs: Renaming and Substitution

Similarly to ACMM (2017) renaming can be defined generically for all syntax descriptions as a semantics with Var as values and Tm as computations. The first two constraints on Var described earlier are trivially satisfied. Observing that renaming strictly respects the structure of the term it goes through, it makes sense for the algebra to be implemented using fmap. When dealing with the body of a binder, we ‘reify’ the Kripke function by evaluating it in an extended context and feeding it placeholder 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 (2002)).

Substitution can be defined in a similar manner with Tm as both values and computations. 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.

The reification process mentioned in the definition of renaming and substitution can be implemented generically for Semantics families which have VarLike values ($\mathrm{vl}^{\mathrm{Var}}$ and $\mathrm{vl}^{\mathrm{Tm}}$ are proofs of VarLike for Var and Tm respectively) i.e. values which are thinnable and such that we can craft placeholder values in non-empty contexts.

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

ren :    (Γ ─Env) Var Δ →
         (Γ ─Comp) (Tm d ∞) Δ
ren = Sem.sem Renaming
```

```
Substitution : Sem d (Tm d ∞) (Tm d ∞)
Substitution = record
   { th^𝒱   = λ t ρ → ren ρ t
   ; var    = id
   ; alg    = ‘con ∘ fmap d (reify vl^Tm) }

sub :    (Γ ─Env) (Tm d ∞) Δ →
         (Γ ─Comp) (Tm d ∞) Δ
sub = Sem.sem Substitution
```

Fig. 29. Generic Renaming and Substitution for All Scope Safe Syntaxes with Binding

```
record VarLike (𝒱 : I ─Scoped) : Set where
   field  new    : [ (i ::_) ⊢ 𝒱 i ]
          th^𝒱   : Thinnable (𝒱 i)
```

Fig. 30. VarLike: Thinnable and with placeholder values

For any VarLike $\mathcal{V}$, we can define fresh$^{\mathsf{r}}$ of type (Γ −Env) $\mathcal{V}$ (Δ ++ Γ) and fresh$^{\mathsf{l}}$ of type (Γ −Env) $\mathcal{V}$ (Γ ++ Δ) by combining the use of placeholder values and thinnings, and it is almost immediate that variables are VarLike. Hence, we can then write reify like so:

```
reify : VarLike 𝒱 → ∀ Δ i → Kripke 𝒱 C Δ i Γ → Scope C Δ i Γ
reify vl^𝒱 []           i b = b
reify vl^𝒱 Δ@(_ :: _)   i b = b (fresh^r vl^Var Δ) (fresh^l vl^𝒱 _)
```

Fig. 31. Generic Reification thanks to VarLike Values

7 A CATALOGUE OF GENERIC PROGRAMS FOR SYNTAX WITH BINDING

One of the advantages of having a universe of programming language descriptions is the ability to concisely define an *extension* of an existing language by using Description transformers grafting extra constructors à la Swiestra (2008). This is made extremely simple by the disjoint sum combinator _‘+_ which we defined in Section 5.1. An example of such an extension is the addition of let-bindings to an existing language.

7.1 Sugar and Desugaring as a Semantics

Let bindings allow the user to avoid repeating themselves by naming sub-expressions and then using these names to refer to the associated terms. Preprocessors adding these types of mechanisms to existing languages (from C to CSS) are rather popular. We introduce a description of Let-bindings which can be used to extend any language description d to d ‘+ Let (where ‘+ is the disjoint of sum of two descriptions defined in Figure 21):

This description states that a let-binding node stores a pair of types σ and τ and two subterms. First comes the let-bound expression of type σ and second comes the body of the let which has type τ in a context extended with a fresh variable of type σ. This defines a term of type τ.

```
Let : Desc I
Let =  ‘σ (I × I) $ uncurry λ σ τ →
       ‘X [] σ (‘X (σ :: []) τ (‘■ τ))
```

Fig. 32. Description of a Single Let Binding

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 they used a variable referring to a let-bound expression or the expression itself. Various typechecking strategies can accomodate this expectation: in Coq [The Coq Development Team 2017] let bindings 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 typechecking can be implemented generically as a semantics over (d ‘+ Let). For this semantics values in the environment and computations are both let-free terms. The algebra of the semantics can be defined by parts thanks to case defined in Section 5.1: the old constructors are kept the same by interpreting them using the generic Substitution algebra; whilst the let-binder precisely provides the extra value to be added to the environment. The process of removing let binders is kickstarted with a placeholder environment associating each variable to itself.

```
UnLet : Sem (d ‘+ Let) (Tm d ∞) (Tm d ∞)
Sem.th^𝒱   UnLet = th^Tm
Sem.var    UnLet = id                        unlet : [ Tm (d ‘+ Let) ∞ i ∸→ Tm d ∞ i ]
Sem.alg    UnLet =                           unlet = Sem.sem UnLet (pack ‘var)
    case (Sem.alg Substitution) λ where
    (_ , e , t , refl) → extract t (ε ∙ e)
```

Fig. 33. Inlining Let Binding

In less than 10 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 construct. In their own setting working on STLC, ACMM (2017) have shown that it is similarly possible to implement a Continuation Passing Style transformation as a semantics.

We have demonstrated how easily one can define extensions and combine them on top of a base language without having to reimplement common traversals for each one of the intermediate representations. Moreover, it is possible to define *generic* transformations elaborating these added features in terms of lower-level ones. This suggests that this setup could be a good candidate to implement generic compilation passes and could deal with a framework using a wealth of slightly different intermediate languages à la Nanopass [Keep and Dybvig 2013].

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 one of our running examples: 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 an unsafe 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 go wrong.

```
{-# NO_POSITIVITY_CHECK #-}
data Dm (d : Desc I) : Size → I −Scoped where
  V : [ Var i                                  →̇  Dm d s       i  ]
  C : [ ⟦ d ⟧ (Kripke (Dm d s) (Dm d s)) i  →̇  Dm d (↑ s)   i  ]
  ⊥ : [                                           Dm d (↑ s)   i  ]
```

Fig. 34. Generic Reflexive Domain

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-indexed approach.

But this domain does make it possible to define a generic nbe semantics which, given a term, produces a value in the reflexive domain. Thanks to the fact we have picked a universe of finitary syntaxes, we can *traverse* [McBride and Paterson 2008] the functor to define a (potentially failing) 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 for λ-abstractions, Σ-types, fixpoints, or anything else.

```
reify^Dm   : [ Dm d s i →̇ Maybe ∘ Tm d ∞ i ]
nbe        : Alg d (Dm d ∞) (Dm d ∞) → Sem d (Dm d ∞) (Dm d ∞)

norm       : Alg d (Dm d ∞) (Dm d ∞) → [ Tm d ∞ i →̇ Maybe ∘ Tm d ∞ i ]
norm alg   = reify^Dm ∘ Sem.sem (nbe alg) (base vl^Dm)
```

Fig. 35. Generic Normalisation by Evaluation Framework

Using this setup, we can write a normaliser for the untyped λ-calculus: we use case from section 5.1 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.

We have not used the ⊥ constructor so *if* the evaluation terminates (by disabling totality checking we have lost all guarantees with respect to termination) we know we will get a term in normal form.

```
norm^LC : [ Tm UTLC ∞ tt ⟶̇ Maybe ∘ Tm UTLC ∞ tt ]
norm^LC = norm $ case app (C ∘ (false ,_)) where

  Model = Dm UTLC ∞

  app : [ ⟦ ‘X [] tt (‘X [] tt (‘■ tt)) ⟧ (Kripke Model Model) tt ⟶̇ Model tt ]
  app (C (false , f , _)    , t   , _) = f (base vl^Var) (ε • t)   -- redex
  app (f                    , t   , _) = C (true , f , t , refl)   -- stuck application
```

Fig. 36. Normalisation by Evaluation for the Untyped λ-Calculus

7.3 An Algebraic Approach to Typechecking

Following Atkey (2015), we can consider type checking and type inference as a possible semantics for a bi-directional [Pierce and Turner 2000] language. We represent the raw syntax of a simply typed bi-directional calculus as a bi-sorted language using a notion of Mode to distinguish between terms for which we will be able to Infer the type and the ones for which we will have to Check a type candidate.

Following traditional presentations, eliminators give rise to Inferrable terms under the condition that the term they are eliminating is also Inferrable and the other arguments are Checkable whilst constructors are always Checkable. Two extra constructors allow changes of direction: Cut annotates a Checkable term with its type thus making it Inferrable whilst Emb embeds Inferrables into Checkables.

```
data LangC : Set where
  App Lam Emb : LangC
  Cut : Type → LangC
```

```
Lang : Desc Mode
Lang    =    ‘σ LangC $ λ where
  App        → ‘X [] Infer (‘X [] Check (‘■ Infer))
  Lam        → ‘X (Infer :: []) Check (‘■ Check)
  (Cut σ)    → ‘X [] Check (‘■ Infer)
  Emb        → ‘X [] Infer (‘■ Check)
```

Fig. 37. A Bidirectional Simply Typed Language

The values stored in the environment will be Type information for bound variables no matter what their Mode is. In constrast, the generated computations will, depending on the mode, either take a type candidate and Check it is valid or Infer a type for their argument. These computations are always potentially failing so we use the Maybe monad.

```
Var- : Mode → Set
Var- _ = Type
```

```
Type- : Mode → Set
Type- Check    = Type →    Maybe ⊤
Type- Infer    =           Maybe Type
```

Fig. 38. Var- and Type- Relations indexed by the Mode

We can now define typechecking as a Semantics. The algebra describes the algorithm:

- when facing an application: infer the type of the function, make sure it is an arrow type, check the argument at the domain's type and return the codomain
- for a λ-abstraction: check the input type is an arrow type and check the body at the codomain type in the extended environment where the newly-bound variable as the domain's type
- a cut always comes with a type candidate against which to check the term and to be returned in case of success
- finally, the change of direction from Inferrable to Checkable is successful when the inferred term is equal to the expected one.

```
Typecheck : Sem Lang (const ∘ Var-) (const ∘ Type-)
Typecheck = record { th𝒱 = λ v ρ → v; var = var _; alg = alg } where

  var : (i : Mode) → Var- i → Type- i
  var Infer     = just
  var Check     = _==_

  alg : ⟦ Lang ⟧ (Kripke (κ ∘ Var-) (κ ∘ Type-)) i Γ → Type- i
  alg (App , f , t , refl)   =  f              >>= λ σ⇒τ →
                                isArrow σ⇒τ    >>= uncurry λ σ τ →
                                τ <$ t σ
  alg (Lam , b , refl)       =  λ σ⇒τ → isArrow σ⇒τ >>= uncurry λ σ τ →
                                b (extend {σ = Infer}) (ε • σ) τ
  alg (Cut σ , t , refl)     =  σ <$ t σ
  alg (Emb , t , refl)       =  λ σ → t >>= λ τ → σ == τ
```

Fig. 39. Type- Inference / Checking as a Semantics

We have defined a bidirectional typechecker for this simple language by leveraging the Semantics framework. The code attached to this paper also contains a variant with more informative types: instead of simply generating a type or checking that a candidate will do, we can use our Descriptions to describe a language of evidence and generate not only an expression's type but also a well scoped and well typed term of that type.

7.4 Binding as Self-Reference: Representing Cyclic Structures

Ghani, Hamana, Uustalu and Vene (2006) have demonstrated how Altenkirch and Reus' type-level de Bruijn indices (1999) can be used to represent potentially cyclic structures by a finite object. In their representation each bound variable is a pointer to the node that introduced it. Given that we are, at the top-level, only interested in structures with no "dangling pointers", we introduce the notation TM d to mean closed terms (i.e. terms of type Tm d ∞ []).

A basic example of such a structure is a potentially cyclic list which offers a choice of two constructors: [] which ends the list and _:_ which combines a head and a tail but also acts as a binder for a self-reference; these pointers can be used by using the var constructor which we have renamed ↶ (pronounced "backpointer") to match the domain-specific meaning. We can see this approach in action in the examples [0, 1] and 01↻ (pronounced "0-1-cycle") which describe respectively a finite list containing 0 followed by 1 and a cyclic list starting with 0, then 1, and then repeating the whole list again by referring to the first cons cell represented here by the de Bruijn variable 1 (i.e. s z).

```
CListD : Set ⟶ Desc ⊤
CListD A   =    ‘■ tt                                      [0,1]   : TM (CListD ℕ) tt
               ‘+   ‘σ A (λ _ ⟶ ‘X (tt :: []) tt (‘■ tt))     01↻     : TM (CListD ℕ) tt

pattern []          = ‘con (true , refl)                 [0,1]   =    0 :: 1 :: []
pattern _::_ x xs   = ‘con (false , x , xs , refl)       01↻     =    0 :: 1 :: ↶ s z
pattern ↶_ k        = ‘var k
```

Fig. 40. Potentially Cyclic Lists: Description, Pattern Synonyms and Examples

These finite representations are interesting in their own right and we can use the generic semantics framework defined earlier to manipulate them. A basic building block is the unroll function which takes a closed tree, exposes its top node and unrolls any cycle which has it as its starting point. We can decompose it using the plug function which, given a closed and an open term, closes the latter by plugging the former at each free ‘var leaf. Noticing that plug's fundamental nature is that of substituting a term for each leaf, it makes sense to implement it by re-using the Substitution semantics we already have.

```
plug : TM d tt ⟶ ∀ Δ i ⟶ Scope (Tm d ∞) Δ i [] ⟶ TM d i
plug t Δ i = Sem.sem Substitution (pack (λ _ ⟶ t))

    unroll : TM d tt ⟶ ⟦ d ⟧ (λ _ i _ ⟶ TM d i) tt []
    unroll t′@(‘con t) = fmap d (plug t′) t
```

Fig. 41. Plug and Unroll: Exposing a Cyclic Tree's Top Layer

However, one thing still out of our reach with our current tools is the underlying co-finite trees these finite objects are meant to represent. We can start by defining the coinductive type corresponding to them as the greatest fixpoint of a notion of layer. One layer of a co-finite tree is precisely given by the meaning of its description where we completely ignore the binding structure. We show with 01··· the infinite list that to correspond to the cyclic example 01↻ given above. The definition proceeds by copattern-matching as introduced in [Abel et al. 2013] and showcased in [Thibodeau et al. 2016].

```
record ∞Tm (d : Desc I) (s : Size) (i : I) : Set where     01··· : ∞Tm (CListD ℕ) ∞ tt
    coinductive; constructor ‘con                          10··· : ∞Tm (CListD ℕ) ∞ tt
    field force :   {s′ : Size< s} ⟶                         01··· .force = false , 0 , 10··· , refl
                    ⟦ d ⟧ (λ _ i _ ⟶ ∞Tm d s′ i) i []         10··· .force = false , 1 , 01··· , refl
```

Fig. 42. Co-finite Trees: Definition and Example

We can then make the connection between potentially cyclic structures and the co-finite trees formal by giving an unfold function which, given a closed term, produces its unfolding. The definition proceeds by unrolling the term's top layer and co-recursively unfolding all the subterms.

```
unfold : TM d tt → ∞Tm d s tt
unfold t .force = fmap d (λ _ _ → unfold) (unroll t)
```

Fig. 43. Generic Unfold of Potentially Cyclic Structures

Even if the powerful notion of semantics described in Section 6 cannot encompass all the traversals we may be interested in, it provides us with reusable building blocks: the definition of unfold was made very simple by reusing the generic program fmap and the Substitution semantics whilst the definition of ∞Tm was made easy by reusing ⟦_⟧.

8 BUILDING GENERIC PROOFS ABOUT GENERIC PROGRAMS

ACMM (2017) have already shown that, for the simply typed λ-calculus, 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 inherits a lot of its relational arguments' properties: whenever R is reflexive (respectively symmetric or transitive) so is Zip d R.

It is defined by induction on the description and case analysis on the two layers which are meant to be equal:

- In the stop token case ‘■ i, the two layers are considered to be trivially equal (i.e. the constraint generated is the unit type)
- When facing a recursive position ‘X Δ j d, we demand that the two substructures are related by R Δ j and that the rest of the layers are related by Zip d R
- Two nodes of type ‘σ A d will be related if they both carry the same payload a of type A and if the rest of the layers are related by Zip (d a) R.

```
Zip :   (d : Desc I) (R : (δ : List I) (i : I) → [ X δ i ⟶̇ Y δ i ⟶̇ κ Set ]) →
        [ ⟦ d ⟧ X i ⟶̇ ⟦ d ⟧ Y i ⟶̇ κ Set ]
Zip (‘■ i′)     R x        y         = ⊤
Zip (‘X δ j d)  R (r , x)  (r’ , y)  = R δ j 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 ⟧ _ _ _)
```

Fig. 44. Zip: Characterising Structurally Equal Values with Related Substructures

If we were to take a fixpoint of Zip, we could obtain a structural notion of equality for terms which we could prove equivalent to propositional equality. Although interesting in its own right, this section will focus on more advanced use-cases.

8.1 Simulation Lemma

A Zip constraint appears naturally when we want to say that a semantics can simulate another one. Given a relation $\mathcal{R}^{\mathcal{V}}$ connecting values in $\mathcal{V}_1$ and $\mathcal{V}_2$, and a relation $\mathcal{R}^{C}$ connecting computations in C_1 and C_2, we can define Kripke$^{\mathsf{R}}$ relating values Kripke $\mathcal{V}_1$ C_1 and Kripke $\mathcal{V}_2$ 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 in a pointwise manner.

```
Kripkeᴿ : (Δ : List I) (τ : I) → [ Kripke 𝓥₁ 𝓒₁ Δ τ ⟶ Kripke 𝓥₂ 𝓒₂ Δ τ ⟶ κ Set ]
Kripkeᴿ []           σ k₁ k₂ = 𝓡ᶜ k₁ k₂
Kripkeᴿ Δ@(_ ∷ _)    σ k₁ k₂ = ∀ th → ∀[ 𝓡ᵛ ] ρ₁ ρ₂ → 𝓡ᶜ (k₁ th ρ₁) (k₂ th ρ₂)
```

Fig. 45. Relational Kripke Function Spaces: From Related Inputs to Related Outputs

We can then combine Zip and 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 the algebras for two Semantics 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 I) (𝓢₁ : Sem d 𝓥₁ 𝓒₁) (𝓢₂ : Sem d 𝓥₂ 𝓒₂) : Set where
  field  thᴿ   : (σ : Thinning Γ Δ) → 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  :   → (b : ⟦ d ⟧ (Scope (Tm d s)) i Γ) → ∀[ 𝓡ᵛ ] ρ₁ ρ₂ →
                   let  v₁ = fmap d (Sem.body 𝓢₁ ρ₁) b
                        v₂ = fmap d (Sem.body 𝓢₂ ρ₂) b
                   in Zip d (Kripkeᴿ 𝓡ᵛ 𝓡ᶜ) v₁ v₂ → 𝓡ᶜ (Sem.alg 𝓢₁ v₁) (Sem.alg 𝓢₂ v₂)
```

Fig. 46. A Generic Notion of Simulation

The fundamental lemma of simulations is 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    :  ∀[ 𝓡ᵛ ] ρ₁ ρ₂ → (t : Tm d s i Γ) → 𝓡ᶜ (Sem.sem 𝓢₁ ρ₁ t) (Sem.sem 𝓢₂ ρ₂ t)
body   :  ∀[ 𝓡ᵛ ] ρ₁ ρ₂ → ∀ Δ j → (t : Scope (Tm d s) Δ j Γ) →
          Kripkeᴿ 𝓡ᵛ 𝓡ᶜ Δ j (Sem.body 𝓢₁ ρ₁ Δ j t) (Sem.body 𝓢₂ ρ₂ Δ j t)
```

Fig. 47. Fundamental Lemma of Simulations

Instantiating this generic simulation lemma, we can for instance get that renaming and substitution are extensional (given extensionally equal environments they produce syntactically equal terms), or that renaming is a special case of substitution. Of course these results are not new but having them generically over all syntaxes with binding is convenient; which we have experienced first hand when tackling the POPLMark Reloaded challenge where rensub was actually needed.

When studying specific languages, new opportunities to deploy the fundamental lemma of simulations arise. Our solution to the POPLMark Reloaded challenge for instance describes the fact that sub ρ t reduces to sub ρ' t whenever for all v, $\rho(v)$ reduces to ρ'(v) as a Simulation. The main

```
rensub :   (ρ : Thinning Γ Δ) (t : Tm d ∞ i Γ) → ren ρ t ≡ sub (‘var <$> ρ) t
rensub ρ = Sim.sim RenSub (packᴿ (λ _ → refl))
```

Fig. 48. Renaming as a Substitution via Simulation

theorem (strong normalisation of STLC via a logical relation) is itself an instance of (the unary version of) the simulation lemma.

The Simulation proof framework is the simplest examples of the abstract proof frameworks ACMM (2017) introduce. 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. Now that we are familiarised with the techniques at hand, we can tackle this more complex example for all syntaxes definable in our framework.

8.2 Fusion Lemma

Results which can be reformulated as the ability to fuse two traversals obtained as Semantics into one abound. When claiming that Tm is a Functor, we have to prove that two successive renamings can be fused into a single renaming where the Thinnings have been composed. Similarly, demonstrating that Tm is a relative Monad [Altenkirch et al. 2014] implies proving that two consecutive substitutions can be merged into a single one whose environment is the first one, where the second one has been applied in a pointwise manner. The *Substitution Lemma* central to most model constructions (see for instance [Mitchell and Moggi 1991]) states that a syntactic substitution followed by the evaluation of the resulting term into the model is equivalent to the evaluation of the original term with an environment corresponding to the evaluated substitution.

A direct application of these results is our (to be published) entry to the POPLMark Reloaded challenge (2017). By using a Desc-based representation of intrisincally well typed and well scoped terms we directly inherit not only renaming and substitution but also all four fusion lemmas as corollaries of our generic results. This allows us to remove the usual boilerplate and go straight to the point. As all of these statements have precisely the same structure, we can once more devise a framework which will, provided that its constraints are satisfied, prove a generic fusion lemma.

Fusion is quite a bit more involved than simulation so we will step through each one of the constraints individually, trying to give the reader an intuition for why they are shaped the way they are.

8.2.1 The Fusion constraints. The notion of fusion is defined for a triple of Semantics; each $\mathcal{S}_i$ being defined for values in $\mathcal{V}_i$ and computations in $\mathcal{C}_i$. The fundamental lemma associated to such a set of constraints will state that running $\mathcal{S}_2$ after $\mathcal{S}_1$ is equivalent to running $\mathcal{S}_3$ only.

The definition of fusion is parametrised by three relations: $\mathcal{R}^E$ relates triples of environments of values in (Γ –Env) $\mathcal{V}_\infty$ Δ, (Δ –Env) $\mathcal{V}_\in$ Θ and (Γ –Env) $\mathcal{V}_\ni$ Θ respectively; $\mathcal{R}^\mathcal{V}$ relates pairs of values$\mathcal{V}_\in$ and $\mathcal{V}_\ni$; and $\mathcal{R}^C$, our notion of equivalence for evaluation results, relates pairs of computation in $C_\in$ and $C_\ni$.

The first obstacle we face is the formal definition of “running $\mathcal{S}_2$ after $\mathcal{S}_1$”: for this statement to make sense, the result of running $\mathcal{S}_1$ ought to be a term. Or rather, we ought to be able to extract a term from a C_∞. Hence the first constraint: the existence of a quote_1 function, which we supply as a field of the record Fusion. When dealing with syntactic semantics such as renaming or substitution this function will be the identity. However nothing prevents to try to prove for instance that normalisation by evaluation is idempotent in which case a bona fide reification function extracting terms from model values will be used.

$$\mathsf{quote}_1 \quad : \quad (i : I) \to [\, C_1\ i \mathrel{\dot{\to}} \mathsf{Tm}\ d\ \infty\ i \,]$$

Then, we have to think about what happens when going under a binder: $\mathcal{S}_1$ will produce a Kripke function space where a syntactic value is required. Provided that $\mathcal{V}_1$ is VarLike, we can make use of reify to get a Scope back. Hence the second constraint.

$$\mathsf{vl}^{\mathcal{V}_1} \quad : \quad \mathsf{VarLike}\ \mathcal{V}_1$$

Still thinking about going under binders: if three evaluation environments ρ_1 in $(\Gamma\ \mathsf{-Env})\ \mathcal{V}_\infty\ \Delta$, ρ_2 in $(\Delta\ \mathsf{-Env})\ \mathcal{V}_\in\ \Theta$, and ρ_3 in $(\Gamma\ \mathsf{-Env})\ \mathcal{V}_\ni\ \Theta$ are related by $\mathcal{R}^E$ and we are given a thinning σ from Θ to Ω then ρ_1, the thinned ρ_2 and the thinned ρ_3 should still be related.

$$\begin{aligned}\mathsf{th}^{\mathsf{R}} \quad : \quad & (\sigma : \mathsf{Thinning}\ \Theta\ \Xi) \to \mathcal{R}^E\ \rho_1\ \rho_2\ \rho_3 \to \\ & \mathcal{R}^E\ \rho_1\ (\mathsf{th}^{\mathsf{Env}}\ (\mathsf{Sem.th}^{\mathcal{V}}\ \mathcal{S}_2)\ \rho_2\ \sigma)\ (\mathsf{th}^{\mathsf{Env}}\ (\mathsf{Sem.th}^{\mathcal{V}}\ \mathcal{S}_3)\ \rho_3\ \sigma)\end{aligned}$$

Remembering that $\_\gg\_$ is used in the definition of body (Figure 28) to combine two disjoint environments $(\Gamma\ \mathsf{-Env})\ \mathcal{V}\ \Theta$ and $(\Delta\ \mathsf{-Env})\ \mathcal{V}\ \Theta$ into one of type $((\Gamma +\!\!+ \Delta)\ \mathsf{-Env})\ \mathcal{V}\ \Theta)$, we mechanically need a constraint stating that $\_\gg\_$ is compatible with $\mathcal{R}^E$. We demand as an extra precondition that the values ρ_2 and ρ_3 are extended with are related according to $\mathcal{R}^{\mathcal{V}}$. Lastly, for all the types to match up, ρ_1 has to be extended with placeholder variables.

$$\begin{aligned}\gg^{\mathsf{R}} \quad : \quad & \{\rho_1 : (\Gamma\ \mathsf{-Env})\ \mathcal{V}_1\ \Delta\}\ \mathcal{R}^E\ \rho_1\ \rho_2\ \rho_3 \to \forall[\, \mathcal{R}^{\mathcal{V}}\,]\ \rho_4\ \rho_5 \to \\ & \mathcal{R}^E\ (\mathsf{fresh}^{\mathsf{l}}\ \mathsf{vl}^{\mathcal{V}_1}\ \Delta \gg \mathsf{th}^{\mathsf{Env}}\ (\mathsf{Sem.th}^{\mathcal{V}}\ \mathcal{S}_1)\ \rho_1\ (\mathsf{fresh}^{\mathsf{r}}\ \mathsf{vl}^{\mathsf{Var}}\ \Xi))\ (\rho_4 \gg \rho_2)\ (\rho_5 \gg \rho_3)\end{aligned}$$

We finally arrive at the constraints focusing on the semantical counterparts of the terms' constructors. When evaluating a variable, on the one hand $\mathcal{S}_1$ will look up its meaning in the evaluation environment, turn the resulting value into a computation which will get quoted and then the result will be evaluated with $\mathcal{S}_2$. Provided that all three evaluation environments are related by $\mathcal{R}^E$ this should be equivalent to looking up the value in $\mathcal{S}_3$'s environment and turning it into a computation. Hence the constraint $\mathsf{var}^{\mathsf{R}}$:

$$\begin{aligned}\mathsf{var}^{\mathsf{R}} \quad : \quad & \mathcal{R}^E\ \rho_1\ \rho_2\ \rho_3 \to (v : \mathsf{Var}\ i\ \Gamma) \to \\ & \mathcal{R}^C \quad (\mathsf{Sem.sem}\ \mathcal{S}_2\ \rho_2\ (\mathsf{quote}_1\ i\ (\mathsf{Sem.var}\ \mathcal{S}_1\ (\mathsf{lookup}\ \rho_1\ v)))) \\ & \qquad (\mathsf{Sem.var}\ \mathcal{S}_3\ (\mathsf{lookup}\ \rho_3\ v))\end{aligned}$$

The case of the algebra follows a similar idea albeit being more complex: a term gets evaluated using $\mathcal{S}_1$ and to be able to run $\mathcal{S}_2$ afterwards we need to recover a piece of syntax. This is possible if the Kripke functional spaces are reified by being fed placeholder $\mathcal{V}_\infty$ arguments (which can be manufactured thanks to the $\mathsf{vl}^{\mathcal{V}_\infty}$ we mentioned before) and then quoted. Provided that the result of running $\mathcal{S}_2$ on that term is related via $\mathsf{Zip}\ d\ (\mathsf{Kripke}^{\mathsf{R}}\ \mathcal{R}^{\mathcal{V}}\ \mathcal{R}^C$ to the result of running $\mathcal{S}_3$ on the original term, the $\mathsf{alg}^{\mathsf{R}}$ constraint states that the two evalutions yield related computations.

$$\begin{aligned}\mathsf{alg}^{\mathsf{R}} \quad : \quad & (b : [\![\, d \,]\!]\ (\mathsf{Scope}\ (\mathsf{Tm}\ d\ s))\ i\ \Gamma) \to \mathcal{R}^E\ \rho_1\ \rho_2\ \rho_3 \to \\ & \mathsf{let}\quad v_1 = \mathsf{fmap}\ d\ (\mathsf{Sem.body}\ \mathcal{S}_1\ \rho_1)\ b \\ & \qquad\ \ v_3 = \mathsf{fmap}\ d\ (\mathsf{Sem.body}\ \mathcal{S}_3\ \rho_3)\ b \\ & \mathsf{in}\ \mathsf{Zip}\ d\ (\mathsf{Kripke}^{\mathsf{R}}\ \mathcal{R}^{\mathcal{V}}\ \mathcal{R}^C) \\ & \qquad (\mathsf{fmap}\ d\ (\lambda\ \Delta\ i \to \mathsf{Sem.body}\ \mathcal{S}_2\ \rho_2\ \Delta\ i \circ \mathsf{quote}_1\ i \circ \mathsf{reify}\ \mathsf{vl}^{\mathcal{V}_1}\ \Delta\ i)\ v_1) \\ & \qquad v_3 \to \\ & \mathcal{R}^C\ (\mathsf{Sem.sem}\ \mathcal{S}_2\ \rho_2\ (\mathsf{quote}_1\ i\ (\mathsf{Sem.alg}\ \mathcal{S}_1\ v_1)))\ (\mathsf{Sem.alg}\ \mathcal{S}_3\ v_3)\end{aligned}$$

8.2.2 The Fundamental Lemma of Fusion. This set of constraint is enough to prove a fundamental lemma of Fusion stating that from a triple of related environments, one gets a pair of related computations: the composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ on one hand and $\mathcal{S}_3$ on the other. This lemma is once again proven mutually with its counterpart for Sem's body's action on Scopes.

```
fus   :  ℛᴱ ρ₁ ρ₂ ρ₃ → (t : Tm d s i Γ) → ℛᶜ   (Sem.sem 𝒮₂ ρ₂ (quote₁ i (Sem.sem 𝒮₁ ρ₁ t)))
                                             (Sem.sem 𝒮₃ ρ₃ t)
body :  ℛᴱ ρ₁ ρ₂ ρ₃ → (Δ : List I) (i : I) (b : Scope (Tm d s) Δ i Γ) →
        Kripkeᴿ ℛ𝒱 ℛᶜ Δ i   (Sem.body 𝒮₂ ρ₂ Δ i (quote₁ i (reify vl^𝒱₁ Δ i (Sem.body 𝒮₁ ρ₁ Δ i b))))
                             (Sem.body 𝒮₃ ρ₃ Δ i b)
```

Fig. 49. Fundamental Lemma of Fusion

8.2.3 Instances of Fusion. A direct consequence of this result is the four lemmas collectively stating that any pair of renamings and / or substitutions can be fused together to produce either a renaming (in the renaming-renaming interaction case) or a substitution (in all the other cases). One such example is the fusion of substitution followed by renaming into a single substitution where the renaming has been applied to the environment.

```
subren :  ∀ (t : Tm d s i Γ) (ρ₁ : (Γ ─Env) (Tm d ∞) Δ) (ρ₂ : Thinning Δ Θ) →
          ren ρ₂ (sub ρ₁ t) ≡ sub (ren ρ₂ <$> ρ₁) t
subren t ρ₁ ρ₂ = Fus.fus SubRen (packᴿ (λ k → refl)) t
```

Fig. 50. A Corollary: Substitution-Renaming Fusion

Another corollary of the fundamental lemma of fusion is the observation that Kaiser, Schäfer, and Stark (2018) make: *assuming functional extensionality*, all the ACMM (2017) traversals are compatible with variable renaming. We can reproduce this result generically for all syntaxes (see accompanying code) but refrain from using it in practice when an axiom-free alternative is provable.

8.3 Definition of Bisimilarity for co-finite objects

Although we were able to use propositional equality when studying syntactic traversals working on terms, it is not the appropriate notion of equality for co-finite trees. What we want is a generic coinductive notion of bisimilarity for all co-finite tree types obtained as the unfolding of a description. Two trees are bisimilar if their top layers have the same shape and their substructures are themselves bisimilar. This is precisely the type of relation Zip was defined to express. Hence the following coinductive relation.

```
record ≈^∞Tm (d : Desc I) (s : Size) (i : I) (t u : ∞Tm d s i) : Set where
  coinductive
  field force : {s′ : Size< s} → Zip d (λ _ i → ≈^∞Tm d s′ i) (t .force) (u .force)
```

Fig. 51. Generic Notion of Bisimilarity for Co-finite Trees

We can then prove by coinduction that this generic definition always gives rise to an equivalence relation by using Zip's stability properties (if *R* is reflexive / symmetric / transitive then so is Zip *d* *R*) mentioned in Section 8.

```
refl  : ≈∞Tm d s i t t
sym   : ≈∞Tm d s i t u → ≈∞Tm d s i u t
trans : ≈∞Tm d s i t u → ≈∞Tm d s i u v → ≈∞Tm d s i t v
```

This definition can be readily deployed to prove e.g. that the unfolding of 01⥁ defined in Section 7.4 is indeed bisimilar to 01··· which was defined in direct style. The proof is straightforward due to the simplicity of this example: the first refl witnesses the fact that both definitions pick the same constructor (a cons cell), the second that they carry the same natural number, and we can conclude by an appeal to the coinduction hypothesis.

```
eq-01 : ≈∞Tm (CListD ℕ) i tt 01··· (unfold 01⥁)
eq-10 : ≈∞Tm (CListD ℕ) i tt 10··· (unfold (1 :: 0 :: 1 :: ↶ s z))

eq-01 .force = refl , refl , eq-10 , tt
eq-10 .force = refl , refl , eq-01 , tt
```

9 RELATED WORK

9.1 Variable Binding

The representation of variable binding in formal systems has been a hot topic for decades. Part of the purpose of the first POPLMark challenge (2005) was to explore and compare various methods.

Having based our work on a de Bruijn encoding of variables, and thus a canonical treatment of α-equivalence classes, our work has no direct comparison with permutation-based treatments such as those of Pitts' and Gabbay's nominal syntax [Gabbay and Pitts 2001].

Our generic universe of syntax is based on scope-and-typed de Bruijn indices [de Bruijn 1972] but it is not a necessity. It is for instance possible to give an interpretation of Descriptions corresponding to Chlipala's Parametric Higher-Order Abstract Syntax (2008) and we would be interested to see what the appropriate notion of Semantics is for this representation.

9.2 Alternative Binding Structures

The binding structure we present here is based on a flat, lexical scoping strategy. There are other strategies and it would be interesting to see whether our approach could be reused in these cases.

Bach Poulsen, Rouvoet, Tolmach, Krebbers and Visser (2018) introduce notions of scope graphs and frames to scale the techniques typical of well scoped and typed deep embeddings to imperative languages. They can already handle a large subset of Middleweight Java.

We have demonstrated how to write generic programs over the potentially cyclic structures of Ghani, Hamana, Uustalu and Vene (2006). Further work by Hamana (2009) yielded a different presentation of cyclic structures which preserves sharing: pointers can not only refer to nodes above them but also across from them in the cyclic tree. Capturing this class of inductive types as a set of syntaxes with binding and writing generic programs over them is still an open problem.

9.3 Semantics of Syntaxes with Binding

An early foundational study of a general *semantic* framework for signatures with binding, algebras for such signatures, and initiality of the term algebra, giving rise to a categorical 'program' for substitution and proofs of its properties, was given by Fiore, Plotkin and Turi [Fiore et al. 1999], working in the category of presheaves over renamings, (a skeleton of) the category of finite sets. The presheaf condition corresponds to our notion of being Thinnable. Exhibiting algebras based on both de Bruijn *level* and *index* encodings, their approach isolates the usual (abstract) arithmetic required of such encodings.

By contrast, working in an *implemented* type theory, where the encoding can be understood as its own foundation, without appeal to an external mathematical semantics, we are able to go further in developing machine-checked such implementations and proofs, themselves generic with respect to an abstract syntax Desc of syntaxes-with-binding. Moreover, the usual source of implementation anxiety, namely concrete arithmetic on de Bruijn indices, has been successfully encapsulated via the □ coalgebra structure. It is perhaps noteworthy that our type-theoretic constructions, by contrast with their categorical ones, appear to make fewer commitments as to functoriality, thinnability, etc. in our specification of semantics, with such properties typically being *provable* as a further instance of our framework.

9.4 Meta-Theory Automation via Tactics and Code Generation

The tediousness of repeatedly proving similar statements has unsurprisingly led to various attempts at automating the pain away via either code generation or the definition of tactics. These solutions can be seen as untrusted oracles driving the interactive theorem prover.

Polonowski's DBGen (2013) takes as input a raw syntax with comments annotating binding sites. It generates a module defining lifting, substitution as well as a raw syntax using names and a validation function transforming named terms into de Bruijn ones; we refrain from calling it a scopechecker as terms are not statically proven to be well scoped.

Kaiser, Schäfer, and Stark (2018) build on our previous paper to draft possible theoretical foundations for Autosubst, a so-far untrusted set of tactics. The paper is based on a specific syntax: well-scoped call-by-value System F. In contrast, our effort has been here to carve out a precise universe of syntaxes with binding and give a systematic account of their semantics and proofs.

Keuchel, Weirich, and Schrijvers' Needle (2016) is a code generator written in Haskell producing syntax-specific Coq modules implementing common traversals and lemmas about them.

9.5 Universes of Syntaxes with Binding

Keeping in mind Altenkirch and McBride's observation that generic programming is everyday programming in dependently-typed languages (2003), we can naturally expect generic, provably sound, treatments of these notions in tools such as Agda or Coq.

Keuchel, Weirich, and Schrijvers' Knot (2016) implements as a set of generic programs the traversals and lemmas generated in specialised forms by their Needle program. They see Needle as a pragmatic choice: working directly with the free monadic terms over finitary containers would be too cumbersome. In our experience solving the POPLMark Reloaded challenge, Agda's pattern synonyms [Pickering et al. 2016] make working with an encoded definition almost seamless.

The GMeta generic framework (2012) provides a universe of syntaxes and offers various binding conventions (locally nameless [Charguéraud 2012] or de Bruijn indices). It also generically implements common traversals (e.g. computing the sets of free variables, shifting de Bruijn indices or substituting terms for parameters) as well as common predicates (e.g. being a closed term) and provides generic lemmas proving that they are well behaved. It does not offer a generic framework for defining new well scoped-and-typed semantics and proving their properties.

Érdi (2018) defines a universe inspired by a first draft of this paper and gives three different interpretations (raw, scoped and typed syntax) related via erasure. He provides scope-and-type preserving renaming and substitution as well as various generic proofs that they are well behaved but offers neither a generic notion of semantics, nor generic proof frameworks.

Copello (2017) works with *named* binders and defines nominal techniques (e.g. name swapping) and ultimately α-equivalence over a universe of regular trees with binders inspired by Morris' (2006).

10 CONCLUSION AND FUTURE WORK

Recalling Allais, Chapman, McBride and McKinna's earlier work (2017) we have started from an example of a scope-and-type safe language (the simply typed λ-calculus), have studied common invariant preserving traversals and noticed their similarity. After introducing a notion of semantics and refactoring these traversals as 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 given a generic presentation of syntaxes with binding as well as a large class of scope-and-type safe generic programs acting on all of them: from renaming and substitution, to normalisation by evaluation, and the desugaring of new constructors added by a language transformer. The code accompanying the paper also demonstrates how to generically write a printer or a scope-checker elaborating values of a raw syntax using strings as variable names into scope-safe ones.

We have seen how to construct generic proofs about these generic programs. We first introduced a Simulation relation showing what it means for two semantics to yield related outputs whenever they are fed related input environments. We then built on our experience to tackle a more involved case: identifying a set of constraints guaranteeing that two semantics run consecutively can be subsumed by a single pass of a third one.

We have put all of these results into practice by using them to solve the (to be published) POPLMark Reloaded challenge which consists of formalising strong normalisation for the simply typed λ-calculus via a logical-relation argument. This also gave us the opportunity to try our framework on larger languages by tackling the challenge's extensions to sum types and Gödel's System T.

Finally, we have demonstrated that this formalisation can be re-used in other domains by seeing our syntaxes with binding as potentially cyclic terms. Their unfolding is a non-standard semantics and we provide the user with a generic notion of bisimilarity to reason about them.

The diverse influences leading to this work suggest many opportunities for future research.

Our example of the elaboration of an enriched language to a core one, and ACMM's implementation of a Continuation Passing Style conversion function raises the question of how many such common compilation passes can be implemented generically.

An extension of McBride's theory of ornaments (2017) could provide an appropriate framework to highlight the connection between various languages, some being seen as refinements of others. This is particularly evident when considering the informative typechecker (see the accompanying code) which given a scoped term produces a scoped-and-typed term by type-checking or type-inference.

Our work on the POPLMark Reloaded challenge highlights a need for generic notions of congruence closure which would come with guarantees (if the original relation is stable under renaming and substitution so should the closure). Similarly, the "evaluation contexts" corresponding to a syntax could be derived automatically by building on the work of Huet (1997) and Abbott, Altenkirch, McBride and Ghani (2005).

Finally, now knowing how to generically describe syntaxes and their well behaved semantics, we can start asking what it means to define well behaved judgments. Why stop at helping the user write their specific language's meta-theory when we could study meta-meta-theory?

REFERENCES

Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. 2005. ∂ for data: Differentiating data structures. *Fundamenta Informaticae* 65, 1-2 (2005), 1–28.

Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010. (EPTCS)*, Ana Bove, Ekaterina Komendantskaya, and Milad Niqui (Eds.), Vol. 43. 14–28. DOI: http://dx.doi.org/10.4204/EPTCS.43.2

Andreas Abel, Alberto Momigliano, and Brigitte Pientka. 2017. POPLMark Reloaded. *Proceedings of the Logical Frameworks and Meta-Languages: Theory and Practice Workshop* (2017).

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 27–38.

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, New York, NY, USA, 195–207. DOI: http://dx.doi.org/10.1145/3018610.3018613

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

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2014. Relative Monads Formalised. *Journal of Formalized Reasoning* 7, 1 (2014), 1–43.

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

Thorsten Altenkirch and Conor McBride. 2003. Generic Programming Within Dependently Typed Programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1–20. http://dl.acm.org/citation.cfm?id=647100.717294

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

Robert Atkey. 2015. An Algebraic Approach to Typechecking and Elaboration. (2015). http://bentnib.org/posts/2015-04-19-algebraic-approach-typechecking-and-elaboration.html

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65.

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (Jan. 2018), 34 pages. DOI: http://dx.doi.org/10.1145/3158104

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

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic J. of Computing* 10, 4 (Dec. 2003), 265–289. http://dl.acm.org/citation.cfm?id=985799.985801

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

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

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 3–14. DOI: http://dx.doi.org/10.1145/1863543.1863547

James Maitland Chapman. 2009. *Type checking and normalisation.* Ph.D. Dissertation. University of Nottingham (United Kingdom).

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408. DOI: http://dx.doi.org/10.1007/s10817-011-9225-2

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, Vol. 43. ACM, 143–156.

Ernesto Copello. 2017. *On the Formalisation of the Metatheory of the Lambda Calculus and Languages with Binders.* Ph.D. Dissertation. Universidad de la República (Uruguay).

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

Nicolaas Govert de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies. In *Indagationes Mathematicae*, Vol. 75. Elsevier, 381–392.

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. DOI: http://dx.doi.org/10.1007/978-3-319-21401-6_26

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

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

Gergő Érdi. 2018. Generic description of well-scoped, well-typed syntaxes. (2018). https://github.com/gergoerdi/universe-of-syntax Unpublished draft, privately communicated.

Marcelo Fiore, Gordon Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding (Extended Abstract). In *Proc. 14th LICS Conf.* IEEE, Computer Society Press, 193–202.

Murdoch J. Gabbay and Andrew M. Pitts. 2001. A New Approach to Abstract Syntax with Variable Binding. 13, 3–5 (July 2001), 341–363. DOI: http://dx.doi.org/10.1007/s001650200016

Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. 2006. Representing cyclic structures as nested datatypes. In *Proc. of 7th Symp. on Trends in Functional Programming, TFP*, Vol. 2006.

Makoto Hamana. 2009. *Initial Algebra Semantics for Cyclic Sharing Structures.* Springer Berlin Heidelberg, Berlin, Heidelberg, 127–141. DOI: http://dx.doi.org/10.1007/978-3-642-02273-9_11

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

Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554.

Alan Jeffrey. 2011. Associativity for free! http://thread.gmane.org/gmane.comp.lang.agda/3259. (2011).

Jonas Kaiser, Steven Schäfer, and Kathrin Stark. 2018. Binder Aware Recursion over Well-scoped De Bruijn Syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018).* ACM, New York, NY, USA, 293–306. DOI: http://dx.doi.org/10.1145/3167098

Andrew W. Keep and R. Kent Dybvig. 2013. A Nanopass Framework for Commercial Compiler Development. *SIGPLAN Not.* 48, 9 (Sept. 2013), 343–350. DOI: http://dx.doi.org/10.1145/2544174.2500618

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Springer-Verlag New York, Inc., New York, NY, USA, 419–445. DOI: http://dx.doi.org/10.1007/978-3-662-49498-1_17

Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. 2012. GMeta: A Generic Formal Metatheory Framework for First-Order Representations. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 436–455.

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

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

Conor McBride. 2017. Ornamental algebras, algebraic ornaments. (2017). https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/Ornament.pdf

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. DOI: http://dx.doi.org/10.1017/S0956796807006326

John C Mitchell and Eugenio Moggi. 1991. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic* 51, 1-2 (1991), 99–124.

Peter Morris, Thorsten Altenkirch, and Conor McBride. 2006. Exploring the Regular Tree Types. In *Types for Proofs and Programs*, Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 252–267.

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

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016).* ACM, New York, NY, USA, 80–91. DOI: http://dx.doi.org/10.1145/2976002.2976013

Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

Emmanuel Polonowski. 2013. Automatically Generated Infrastructure for De Bruijn Syntaxes. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 402–417.

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

David Thibodeau, Alberto Momigliano, and Brigitte Pientka. 2016. *A case-study in programming coinductive proofs: Howe's method.* Technical Report. Technical report, McGill University.

Received July 2017