doi:10.1017/xxxxx

A type- and scope-safe universe of syntaxes with binding: their semantics and proofs

GUILLAUME ALLAIS

University of St Andrews, St Andrews KY16 9AJ, UK
e-mail: guillaume.allais@ens-lyon.org

ROBERT ATKEY

University of Strathclyde, Glasgow G1 1XQ, UK
e-mail: robert.atkey@strath.ac.uk

JAMES CHAPMAN

Input Output HK Ltd., Edinburgh EH8 9BT, UK
e-mail: james.chapman@iohk.io

CONOR MCBRIDE

University of Strathclyde, Glasgow G1 1XQ, UK
e-mail: conor.mcbride@strath.ac.uk

JAMES MCKINNA

Heriot-Watt University, Edinburgh EH14 4AS, UK
e-mail: J.McKinna@hw.ac.uk

Abstract

The syntax of almost every programming language 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), 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 available online at `https://github.com/gallais/generic-syntax`.

1 Introduction

In modern typed programming languages, programmers writing embedded Domain-Specific Languages (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)) to represent syntax, programmers can *statically* enforce some of the invariants in their languages. For example, managing variable scope is a popular use case in LEGO, Idris, Coq, Agda and Haskell (Altenkirch and Reus (1999); Brady and Hammond (2006); Hirschowitz and Maggesi (2012); Keuchel and Jeuring (2012); Bach Poulsen et al. (2018); Wadler and Kokke (2018); Eisenberg (2020)) as directly manipulating raw de Bruijn indices is notoriously error-prone. Solutions have been proposed that range from enforcing well-scopedness of variables to ensuring full type correctness. In short, these techniques use the host languages' types to ensure that "illegal states are unrepresentable", where illegal states correspond to ill scoped or ill typed terms in the object language.

Despite the large body of knowledge in how to use types to define well formed syntax (see the related work in Section 10), 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 and boilerplate, in this paper we apply the methodology of data type genericity to programming and proving in the domain of syntaxes with binding.

To motivate our approach, let us look at the formalisation of an apparently straightforward program transformation: the inlining of let-bound variables by substitution together with a soundness lemma proving that reductions in the source languages can be simulated by reductions in the target one. There are 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, an implementer needs 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, they will discover that there is actually a large amount of work:

1. To define the operational semantics, one needs to define substitution, and hence renaming. This needs to be done separately for both the source and target languages, even though they are very similar;
2. In the course of proving the correctness lemma, one needs 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 observed by Benton, Hur, Kennedy and McBride (2012)).

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

The main contribution of this paper is that by 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 To introduce the basic ideas that this paper builds on, 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 7.5), and normalisation by evaluation (Section 7.7). Further, we show how to construct generic proofs by formally describing what it means for one semantics to simulate another (Section 9.2), or for two semantics to be fusible (Section 9.3). This allows us to prove the lemmas required above for renaming, substitution, and desugaring of let-binders 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.

Changes with respect to the ICFP 2018 version This paper is a revised and expanded version of a paper of the same title that appeared at ICFP 2018. This extended version of the paper includes many more examples of the use of our universe of syntax with binding for writing generic programs in Section 7: pretty printing with human readable names (Section 7.1), scope checking (Section 7.2), type checking (Section 7.3), elaboration (Section 7.4), inlining of single use let-bound expressions (shrinking reductions) (Section 7.6), and normalisation by evaluation (Section 7.7). We have also included a discussion of how to define generic programs for deciding equality of terms. Additionally, we have elaborated our descriptions and examples throughout, and expanded our discussion of related work in Section 10.

2 A primer on scope- and sort-safe terms

From inductive types to inductive families for abstract syntax. A reasonable way to represent the abstract syntax of the untyped λ-calculus in a typed functional programming language is to use an inductive type:

```
data Lam : Set where
  'var  :  ℕ → Lam
  'lam  :  Lam → Lam
  'app  :  Lam → Lam → Lam
```

We have used de Bruijn (1972) indices to represent variables by the number of 'lam binders one has to pass up through to reach the binding occurrence. The de Bruijn representation has the advantage that terms are automatically represented up to α-equivalence. If the index goes beyond the number of binders enclosing it, then we assume that it is referring to some context, left implicit in this representation.

This representation works well enough for writing programs, but the programmer must constantly be vigilant to guard against the accidental construction of ill scoped terms. The implicit context that accompanies each represented term is prone to being forgotten or muddled with another, leading to confusing behaviour when variables either have dangling pointers or point to the wrong thing.

To improve on this situation, previous authors have proposed to use the host language's type system to make the implicit context explicit, and to enforce well-scopedness of variables. 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 Paterson (1999), and Altenkirch and Reus (1999) introduced the classic presentation of scope safety using inductive *families* (Dybjer (1994)) instead of plain 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 type constructor $1 + (\_)$ extends the running scope with an extra variable.

```
data Lam : Set → Set where
  ‘var  :  X → Lam X
  ‘lam  :  Lam (1+X) → Lam X
  ‘app  :  Lam X → Lam X → Lam X
```

Implicit generalisation of variables in Agda. The careful reader may have noticed that we use a seemingly out-of-scope variable X of type Set. The latest version of Agda allows us to declare variables that the system should implicitly quantify over if it happens to find them used in types. This allows us to lighten the presentation by omitting a large number of prenex quantifiers. The reader will hopefully be familiar enough with prenex polymorphic types in the style of Standard ML (Milner et al. (1997)) that this will seem natural to them.

The Lam type is now a family of types, indexed by the set of variables in scope. Thus, the context for each represented term has been made visible to the type system, and the types enforce that only variables that have been explicitly declared can be referenced in the ‘var constructor. We have made illegal terms unrepresentable.

Since Lam is defined to be a function Set → Set, it makes sense to ask whether it is also a functor and a monad. Indeed it is, as Altenkirch and Reus showed. The functorial action corresponds to renaming, the monadic "return" corresponds to the use of variables (the ‘var constructor), and the monadic "bind" corresponds to substitution. The functor and monad laws correspond to well known properties from the equational theories of renaming and substitution. We will revisit these properties, for our whole universe of syntax with binding, in Section 9.3.

A Typed Variant of Altenkirch and Reus' Calculus. There is no reason to restrict this technique to inductive families indexed by **Set**. The more general case of inductive families in $\mathbf{Set}^J$ can be endowed with similar functorial and monadic operations by using Altenkirch, Chapman and Uustalu's relative monads (2015; 2014).

We pick as our index type J the category whose objects are inhabitants of List I (I is a parameter of the construction) and whose morphisms are *thinnings* (permutations that may forget elements, we give the definition in Section 3.1). Values of type List I are intended to represent the list of the sorts (or kinds, or types, depending on the application) of the de Bruijn variables in scope. We can recover an unsorted approach by picking I to be the unit type. Given this sorted setting, our functors take an extra I argument corresponding to the sort of the expression being built. This is captured by the large type I –Scoped:

```
_–Scoped : Set → Set₁
I –Scoped = I → List I → Set
```

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. We conform to the convention (see e.g. Martin-Löf (1982)) of mentioning only context *extensions* when presenting judgements. That is to say that we aim to write sequents with an *implicit* ambient context. Concretely: in the simply typed λ-calculus (STLC) we would rather use the rule app_i than app_e as the inference rule for application.

$$\frac{f : \sigma \to \tau \qquad t : \sigma}{f\ t : \tau}\, app_i \qquad\qquad \frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash f\ t : \tau}\, app_e$$

In this discipline, the turnstile is used in rules which are binding fresh variables. It separates the *extension* applied to the ambient context on its left and the judgment that lives in the thus extended context on its right. Concretely: we would rather use the rule lam_i than lam_e as the inference rule for λ-abstraction in STLC.

$$\frac{x : \sigma \vdash b : \tau}{\lambda x.t : \sigma \to \tau}\, lam_i \qquad\qquad \frac{\Gamma, x : \sigma \vdash b : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}\, lam_e$$

This observation that an ambient context is either passed around as is or extended for subterms is critical to our whole approach to syntax with binding, and will arise again in our generic formulation of syntax traversals in Section 6. To facilitate this, we make use of the following combinators for building indexed sets:

```
_⇒_ : (P Q : A → Set) → (A → Set)        _⊢_ : (A → B) → (B → Set) → (A → Set)
(P ⇒ Q) x = P x → Q x                     (f ⊢ P) x = P (f x)

const : Set → (A → Set)                   ∀[_] : (A → Set) → Set
const P x = P                             ∀[_] P = ∀ {x} → P x
```

We lift the function space pointwise with _⇒_, silently threading the underlying scope. The _⊢_ makes explicit the *adjustment* made to the index by a function, a generalisation of the idea of *extension*. We write $f \vdash T$ where f is the adjustment 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, const is the constant combinator, which ignores the index.

We make _⇒_ 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 ∀[(const P ⇒ s ⊢ Q) ⇒ R] desugars to the more verbose type ∀ {i} → (P → Q (s i)) → R i.

As the context argument 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, as for example in this data type for scope- and sort-aware de Bruijn indices:

```
data Var : I –Scoped where
  z : ∀[ (σ ::_) ⊢ Var σ ]
  s : ∀[ Var σ ⇒ (τ ::_) ⊢ Var σ ]
```

The inductive family Var represents well scoped and well sorted de Bruijn 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 : ∀ {σ Γ} → Var σ (σ :: Γ)        s : ∀ {σ τ Γ} → Var σ Γ → Var σ (τ :: Γ)
```

We will reuse the Var family to represent variables in all the syntaxes defined in this paper. We start with the simply typed λ-calculus (STLC):

```
data Type : Set where                data Lam : Type –Scoped where
  α       : Type                       'var  : ∀[ Var σ ⇒ Lam σ ]
  _'→_ : Type → Type → Type            'app : ∀[ Lam (σ '→ τ) ⇒ Lam σ ⇒ Lam τ ]
                                       'lam : ∀[ (σ ::_) ⊢ Lam τ ⇒ Lam (σ '→ τ) ]
```

The Type –Scoped family Lam is Altenkirch and Reus' intrinsically typed representation of the simply typed λ-calculus, where Type is the Agda type of simple types. We can readily write well scoped-and-typed terms such as application, a closed term of type ((σ '→ τ) '→ (σ '→ τ)) ({- and -} delimit comments meant to help the reader see to which binders each de Bruijn index refers):

```
apply : Lam ((σ '→ τ) '→ (σ '→ τ)) []
apply = 'lam {- f -} ('lam {- x -}
          ('app ('var (s z) {- f -}) ('var z {- x -})))
```

3 A primer on type- and scope-safe programs

The type- and scope-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 want to write programs manipulating terms. Renaming and substitution are the two typical examples that are required for almost all syntaxes. 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
ren ρ (‘var k)   = var_r (lookup ρ k)
ren ρ (‘app f t) = ‘app (ren ρ f) (ren ρ t)
ren ρ (‘lam b)   = ‘lam (ren (extend_r ρ) b)
```

```
sub : (Γ –Env) Lam Δ →
      Lam σ Γ → Lam σ Δ
sub ρ (‘var k)   = var_s (lookup ρ k)
sub ρ (‘app f t) = ‘app (sub ρ f) (sub ρ t)
sub ρ (‘lam b)   = ‘lam (sub (extend_s ρ) b)
```

We have intentionally hidden technical details behind some auxiliary definitions left abstract here: var and extend. Their implementations are distinct for ren and sub but they serve the same purpose: var 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 Section 3.2 where we define an abstract notion of semantics and the corresponding generic traversal.

3.1 A generic notion of environments

Both renaming and substitution are defined in terms of *environments*. We typically call an environment that associates values to each variable in Γ a Γ-environment. This informs our notation choice: we write ((Γ –Env) $\mathcal{V}$ Δ) for an environment that associates a value $\mathcal{V}$ (variables for renaming, terms for substitution) well scoped and well typed in Δ to every entry in Γ. Formally, we have the following record structure (using a record helps Agda's type inference reconstruct the type family $\mathcal{V}$ of values for us):

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

Environments as records in Agda. As with (all) other record structures defined in this paper, we are able to profit from Agda's *copattern* syntax, as introduced in (Abel et al. (2013)) and showcased in (Thibodeau et al. (2016)). That is, when defining an environment ρ, we may either use the constructor pack, packaging a function r as an environment ρ = pack r, or else define ρ in terms of the underlying function obtained from it by projecting out the (in this case, unique) lookup field, as lookup $\rho = r$. A value of a record type with more than one field requires each of its fields to be given, either by a named constructor (or else Agda's default record syntax), or in copattern style. By analogy with record/object syntax in other languages, Agda further supports "dot" notation, so that an equivalent definition here could be expressed as ρ .lookup = r.

We can readily define some basic building blocks for environments:

```
ε : ([] –Env) 𝒱 Δ
lookup ε ()
```

```
_•_ : (Γ –Env) 𝒱 Δ → 𝒱 σ Δ → ((σ :: Γ) –Env) 𝒱 Δ
lookup (ρ • v) z     = v
lookup (ρ • v) (s k) = lookup ρ k
```

```
_<$>_ : (∀ {i} → 𝒱 i Δ → 𝒲 i Θ) → (Γ –Env) 𝒱 Δ → (Γ –Env) 𝒲 Θ
lookup (f <$> ρ) k = f (lookup ρ k)
```

The empty environment (ε) is implemented by remarking that there can be no variable of type (Var σ []) and to correspondingly dismiss the case with the impossible pattern (). The function _•_ extends an existing Γ-environment with a new value of type σ thus returning a (σ :: Γ)-environment. We also include the definition of _<$>_, which lifts in a pointwise manner a function acting on values into a function acting on environment of such values.

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 (Benton et al. (2012); Allais et al. (2017)) have achieved this goal and refactored renaming and substitution, but also normalisation by evaluation, printing with names or continuation-passing style (CPS) conversion as various instances of a more general traversal. As we will show in Section 7.3, type checking 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 below: Thinning, a generalisation of renaming; the □ functor, which freely adds the ability to absorb Thinnings to any indexed type; and Thinnables, which are □-coalgebras, i.e., types that permit thinning. 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.

The special case of thinnings

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

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). We define the following identity, weaken and (generalised) transitivity combinators for Thinnings:

```
identity : Thinning Γ Γ
lookup identity k = k

weaken : Thinning Γ (σ :: Γ)
lookup weaken v = s v

select : Thinning Γ Δ → (Δ ─Env) 𝒱 Θ → (Γ ─Env) 𝒱 Θ
lookup (select ren ρ) k = lookup ρ (lookup ren k)
```

Next, the □ combinator turns any (List I)-indexed Set into one that can absorb thinnings.

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

Thinnable : (List I → Set) → Set
Thinnable T = ∀[ T ⇒ □ T ]

extract : ∀[ □ T ⇒ T ]
extract t = t identity

duplicate : ∀[ □ T ⇒ □ (□ T) ]
duplicate t ρ σ = t (select ρ σ)

th^□ : Thinnable (□ T)
th^□ = duplicate
```

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 mentioned 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.

3.2 A Generic Notion of Semantics

As we showed in Allais, Chapman, McBride and McKinna (2017), which we will refer to mnemonically as ACMM, once equipped with these new notions we can define an abstract concept of semantics for our type- and scope-safe language. Provided that a set of constraints on two (Type −Scoped) families $\mathcal{V}$ and $\mathcal{C}$ is satisfied, we will obtain a traversal of the following type:

```
semantics : (Γ −Env) 𝒱 Δ → (Lam σ Γ → 𝒞 σ Δ)
```

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 type- and scope-safe traversals by using different "host languages" for this shallow embedding.

Semantics, specified by a record Semantics, are defined in terms of a choice of values $\mathcal{V}$ and computations $\mathcal{C}$. A semantics must satisfy constraints on the notions of values $\mathcal{V}$ and computations $\mathcal{C}$ at hand.

In the following paragraphs, we interleave the definition of the record of constraints Semantics with explanations of our choices. It is important to understand that all of the indented Agda snippets are part of the record's definition. Some correspond to record fields (highlighted in pink) while others are mere auxiliary definitions (highlighted in blue) as permitted by Agda.

```
record Semantics (𝒱 𝒞 : Type −Scoped) : Set where
```

First of all, values $\mathcal{V}$ should be Thinnable so that semantics may push the environment under binders. We call this constraint th^$\mathcal{V}$, using a caret to generate a mnemonic name: th refers to *th*innable and $\mathcal{V}$ clarifies the family which is proven to be thinnable[1].

```
  th^𝒱 : Thinnable (𝒱 σ)
```

This constraint allows us to define extend, the generalisation of the two auxiliary definitions we used when defining ren and sub at the start of Section 3, in terms of the building blocks introduced in Section 3.1. It takes a context extension from Δ to Θ in the form of a thinning, an existing evaluation environment mapping Γ variables to Δ values and a value living in the extended context Θ and returns an evaluation environment mapping (σ :: Γ) variables to Θ values.

[1] We use this convention consistently throughout the paper, using names such as vl^Tm for the proof that terms are VarLike in Section 6

```
extend : Thinning Δ Θ → (Γ −Env) 𝒱 Δ → 𝒱 σ Θ → ((σ :: Γ) −Env) 𝒱 Θ
extend σ ρ v = ((λ t → th^𝒱 t σ) <$> ρ) • v
```

Second, the set of computations needs to be closed under various combinators which are the semantical counterparts of the language's constructors. For instance in the variable case we obtain a value from the evaluation environment but we need to return a computation. This means that values should embed into computations.

```
var : ∀[ 𝒱 σ ⇒ C σ ]
```

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.

```
app : ∀[ C (σ ‘→ τ) ⇒ C σ ⇒ C τ ]
```

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 (Berger and Schwichtenberg (1991); Berger (1993); Coquand and Dybjer (1997); Coquand (2002)). 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 constraint's type:

```
lam : ∀[ □ (𝒱 σ ⇒ C τ) ⇒ C (σ ‘→ τ) ]
```

Agda allows us to package the definition of the generic traversal function semantics together with the fields of the record Semantics. This causes the definition to be specialised and brought into scope for any instance of Semantics the user will define. We thus realise the promise made earlier, namely that any given Semantics $\mathcal{V}$ C induces a function which, given a value in $\mathcal{V}$ for each variable in scope, transforms a Lam term into a computation C. This function is the proof of the Fundamental Lemma of Semantics for Lam, relative to a given Semantics $\mathcal{V}$ C:

```
semantics : (Γ −Env) 𝒱 Δ → (Lam σ Γ → C σ Δ)
semantics ρ (‘var k)   = var (lookup ρ k)
semantics ρ (‘app f t) = app (semantics ρ f) (semantics ρ t)
semantics ρ (‘lam b)   = lam (λ σ v → semantics (extend σ ρ v) b)
```

3.3 Instances of Semantics

Recall that each Semantics is parametrised by two families: $\mathcal{V}$ and C. During the evaluation of a term, variables are replaced by values of type $\mathcal{V}$ and the overall result is a computation of type C. Coming back to renaming and substitution:

```
ren : (Γ −Env) Var Δ → Lam σ Γ → Lam σ Δ
ren = Semantics.semantics Renaming

sub : (Γ −Env) Lam Δ → Lam σ Γ → Lam σ Δ
sub = Semantics.semantics Substitution
```

we see that they both fit in the Semantics framework:

```
Renaming : Semantics Var Lam              Substitution : Semantics Lam Lam
Renaming = record                         Substitution = record
  { th^𝒱 = th^Var                           { th^𝒱 = λ t ρ → ren ρ t
  ; var   = 'var                            ; var   = id
  ; app   = 'app                            ; app   = 'app
  ; lam   = λ b → 'lam (b weaken z) }       ; lam   = λ b → 'lam (b weaken ('var z)) }
```

The family $\mathcal{V}$ of values is respectively the family of variables for renaming, and the family of λ-terms for substitution. In both cases C is the family of λ-terms because the result of the operation will be a term. We notice that the definition of substitution depends on the definition of renaming: to be able to push terms under a binder, we need to have already proven that they are thinnable. In both cases we use weaken defined in Section 3.1 as the definition of the thinning which embeds 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 ability to generate fresh names is given to us by a monad that here we decide to call Fresh. Concretely, Fresh is implemented as an instance of the State monad where the state is a stream of distinct strings:

```
Fresh : Set → Set
Fresh = State (Stream String _)
```

The Printing semantics is defined by using Names (i.e. Strings) as values and Printers (i.e. monadic actions in Fresh returning a String) as computations. We use a Wrapper with a type and a context as phantom types in order to help Agda's inference propagate the appropriate constraints. We define a function fresh that fetches a name from the name supply and makes sure it is not available anymore.

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

```
Name : I –Scoped                    fresh : ∀ σ → Fresh (Name σ (σ :: Γ))
Name = Wrap String                  fresh σ = do
                                      names ← get
Printer : I –Scoped                   put (tail names)
Printer = Wrap (Fresh String)         pure (MkW (head names))
```

The wrapper Wrap does not depend on the scope Γ so it is automatically a thinnable functor, that is to say that we have the (used but not shown here) definitions map^Wrap witnessing the functoriality of Wrap and th^Wrap witnessing its thinnability. We jump straight to the definition of the printer.

To print a variable, we are handed the Name associated to it by the environment and return it immediately.

```
var : ∀[ Name σ ⇒ Printer σ ]
var = map^Wrap return
```

To print an application, we produce a string representation, *f*, of the term in function position, then one, *t*, of its argument and combine them by putting the argument between parentheses.

```
app : ∀[ Printer (σ ‘→ τ) ⇒ Printer σ ⇒ Printer τ ]
app mf mt = MkW do
  f ← getW mf
  t ← getW mt
  return (f ++ " (" ++ t ++ ")")
```

To print a λ-abstraction, we start by generating a fresh name, *x*, for the newly bound variable, use that name to generate a string *b* representing the body of the function to which we prepend a "λ" binding the name *x*.

```
lam : ∀[ □ (Name σ ⇒ Printer τ) ⇒ Printer (σ ‘→ τ) ]
lam {σ} mb = MkW do
  x ← fresh σ
  b ← getW (mb weaken x)
  return ("λ" ++ getW x ++ ". " ++ b)
```

Putting all of these pieces together, we get the Printing semantics:

```
Printing : Semantics Name Printer
Printing = record { th^𝒱 = th^Wrap; var = var; app = app; lam = lam }
```

We show how one can use this newly defined semantics to implement print, a printer for closed terms assuming that we have already defined names, a stream of distinct strings used as our name supply. We show the result of running print on the term apply.

```
print : Lam σ [] → String
print t = proj₁ (getW printer names) where

  empty : ([] –Env) Name []
  empty = ε

  printer = semantics Printing empty t

apply : Lam ((σ ‘→ τ) ‘→ (σ ‘→ τ)) []
apply = ‘lam (‘lam (‘app (‘var (s z)) (‘var z)))

_ : print apply ≡ "λa. λb. a (b)"
_ = refl
```

Both printing and renaming highlight the importance of distinguishing values and computations: the type of values in their respective environments is 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. In ACMM we 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 semantics. We will come back to this in Section 9.2.

One important observation to make is the tight connection between the constraints described in Semantics and the definition of Lam: the semantical counterparts of the Lam constructors are obtained by replacing the recursive occurrences 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 Semantics from the syntax description. Before doing this in Section 5, we need to look at a generic descriptions of data types.

4 A primer on universes 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 the definition of CDMM's universe (defined below) is an inductive type of Descriptions of strictly positive functors from $\mathbf{Set}^J$ to $\mathbf{Set}^I$. These functors correspond to I-indexed containers of J-indexed payloads. Keeping these index types distinct prevents mistaking one for the other when constructing the interpretation of descriptions. Later of course we can use these containers as the nodes of recursive datastructures by interpreting some payloads sorts as requests for subnodes (Altenkirch et al. (2015)).

The inductive type of descriptions 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 to right-nested tuples terminated by equality constraints.

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

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

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 its value, the rest of the description is either the “stop” token or a pair of an element in A and a recursive substructure, that is, the tail of the list. The List type is unindexed, and we represent the lack of an index with the unit type ⊤ whose unique inhabitant is tt.

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

Indices 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))))
```

The pay-off for encoding our data types 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
```

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[2] .

```
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)
```

This least fixpoint allows us to recover the data types we would otherwise declare recursively and generatively. Pattern synonyms let us hide away the encoding: programmers can use them to pattern-match on lists and Agda conveniently resugars them when displaying a goal. Finally, we can get our hands on the types' eliminators by instantiating the generic fold:

```
List : Set → Set
List A = μ (listD A) ∞ tt

pattern []' = (true , refl)
pattern []  = ‘con []'

pattern _::'_ x xs = (false , x , xs , refl)
pattern _::_ x xs  = ‘con (x ::' xs)
```

```
foldr : (A → B → B) → B → List A → B
foldr c n = fold (listD _) $ λ where
  []'            → n
  (hd ::' rec) → c hd rec
```

2 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.

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. While 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.

5 A universe of scope-safe and well sorted syntaxes

Our universe of scope-safe and well sorted syntaxes follows the same principle as CDMM's universe of data types, except that we are not building endofunctors on $\mathbf{Set}^I$ any more 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 sorts of the newly bound variables will play a crucial role when defining the description's semantics as a binding structure below.

```
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
```

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 → 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 Δ j Γ.

```
⟦_⟧ : 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
⟦ ‘■ j     ⟧ X i Γ = i ≡ j
```

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 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 $I \to I$ –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 as in Section 4.

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

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 ]
```

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 STLC. We add a third example to showcase the whole spectrum of syntaxes: a well scoped and well sorted but not well typed bidirectional language. In all examples, the variable case will be added by the free monad construction so we only have to describe the other constructors.

Un(i)typed λ-calculus (UTLC). For the untyped case, the lack of type translates to picking the unit type (⊤) as our notion of sort. We have two possible 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. A single Boolean is enough to distinguish the two constructors.

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

Bidirectional STLC. Our second example is a bidirectional (Pierce and Turner (2000)) language hence the introduction of a notion of Mode: each term is either part of the Infer or Check fraction of the language. This language has four constructors which we list in the ad hoc 'Bidi type of constructor tags, its decoding Bidi is defined by a pattern-matching λ-expression in Agda. Application and λ-abstraction behave as expected, with the important observation that λ-abstraction binds an Inferrable term. The two remaining constructors correspond to changes of direction: one can freely Embbed inferrable terms as checkable ones whereas we require a type annotation when forming a Cut (we reuse the notion of Type introduced in the STLC example at the end of Section 2).

```
data Mode : Set where              Bidi : Desc Mode
  Check Infer : Mode               Bidi = ‘σ ‘Bidi $ λ where
                                     App     → ‘X [] Infer (‘X [] Check (‘■ Infer))
data ‘Bidi : Set where               Lam     → ‘X (Infer :: []) Check (‘■ Check)
  App Lam Emb : ‘Bidi                (Cut σ) → ‘X [] Check (‘■ Infer)
  Cut : Type → ‘Bidi                 Emb     → ‘X [] Infer (‘■ Check)
```

Intrinsically typed STLC. In the typed case (for the same notion of Type), we are back to two constructors: the terms are fully annotated and therefore it is not necessary to distinguish between Modes anymore. We need our tags to carry extra information about the types involved so we use once more an ad hoc data type ‘STLC, and define its decoding STLC by a pattern-matching λ-expression.

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

For convenience we use Agda's pattern synonyms corresponding to the original constructors in Section 2. 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 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. We only include examples of pattern synonyms for the two extreme examples, the definition for Bidi are similar.

```
pattern ‘app f t = ‘con (true , f , t , refl)     pattern ‘app f t = ‘con (App _ _ , f , t , refl)
pattern ‘lam b   = ‘con (false , b , refl)         pattern ‘lam b   = ‘con (Lam _ _ , b , refl)
```

As a usage example of these pattern synonyms, we define the identity function in all three languages, using the same caret-based naming convention we introduced earlier. The code is virtually the same except for Bidi which explicitly records the change of direction from Check to Infer.

```
id^U : Tm UTLC ∞ tt []     id^B : Tm Bidi ∞ Check []     id^S : Tm STLC ∞ (σ ‘→ σ) []
id^U = ‘lam (‘var z)       id^B = ‘lam (‘emb (‘var z))   id^S = ‘lam (‘var z)
```

A sum combinator for syntaxes. The definition of UTLC is the third time (the first and second times being the definition of listD and vecD in Section 4) that we use a Bool to distinguish between two constructors. We can abstract this common pattern as a combinator _‘+_ together with an appropriate eliminator case which, given two methods, picks the one corresponding to the chosen branch.

```
_‘+_ : Desc I → Desc I → Desc I     case : (⟦ d ⟧ X i Γ → A) → (⟦ e ⟧ X i Γ → A) →
d ‘+ e = ‘σ Bool $ λ isLeft →              (⟦ d ‘+ e ⟧ X i Γ → A)
         if isLeft then d else e     case l r (true  , t) = l t
                                     case l r (false , t) = r t
```

A concrete use case for this combinator will be given in Section 7.5 where we explain how to seamlessly enrich an existing syntax with let-bindings and how to use the Semantics framework to elaborate them away.

6 Generic scope-safe and well sorted programs for syntaxes

Based on the Semantics 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 Semantics (d : Desc I) (𝒱 C : I –Scoped) : Set where
  field th^𝒱 : Thinnable (𝒱 σ)
        var   : ∀[ 𝒱 σ ⇒ C σ ]
        alg   : ∀[ ⟦ d ⟧ (Kripke 𝒱 C) σ ⇒ C σ ]
```

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 $I \rightarrow I$–Scoped, that is, that gets access to the current scope but also the exact list of the sorts of the newly bound variables. 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 is 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 [] j = C j
Kripke 𝒱 C Δ j = □ ((Δ –Env) 𝒱 ⇒ C j)
```

It is once more the case that the abstract notion of Semantics comes with a fundamental lemma: all I –Scoped families $\mathcal{V}$ 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 = ∀ {s σ} → Tm d s σ Γ → C σ Δ
```

6.1 Fundamental lemma of semantics

We can now define the type of the fundamental lemma (called semantics) which takes a semantics and returns a function from environments to computations. It is defined mutually with a function body turning syntactic binders into semantic 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).

```
semantics : (Γ –Env) 𝒱 Δ → (Γ –Comp) C Δ
body : (Γ –Env) 𝒱 Δ → ∀ Θ σ →
       Scope (Tm d s) Θ σ Γ → Kripke 𝒱 C Θ σ Δ
```

The semantics proof is straightforward now that we have clearly identified the problem structure and the constraints we need to enforce. If the term considered is a variable, we look up the associated value in the evaluation environment and turn it into a computation using 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.

```
semantics ρ (‘var k) = var (lookup ρ k)
semantics ρ (‘con t) = alg (fmap d (body ρ) t)
```

The auxiliary lemma body distinguishes two cases. If no new variable has been bound in the recursive substructure, it is a matter of calling semantics 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 ρ []       i t = semantics ρ t
body ρ (_ :: _) i t = λ σ vs → semantics (vs >> th^Env th^𝒱 ρ σ) t
```

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.

We immediately introduce closed, a corollary of the fundamental lemma of semantics for the special cases of closed terms.

```
closed : TM d σ → C σ []
closed = semantics ε
```

Given a Semantics with value type 𝒱 and computation type C, we can evaluate a closed term of type σ and obtain a computation of type (C σ []) by kickstarting the evaluation with an empty environment.

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. Catarina Coquand's formal development (2002)).

Substitution is 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 to renaming and the second one is trivial. The algebra is once more defined by using fmap and reifying the bodies of binders.

```
Ren : Semantics d Var (Tm d ∞)
Ren .th^𝒱 = th^Var
Ren .var   = ‘var
Ren .alg   = ‘con ∘ fmap d (reify vl^Var)

ren : (Γ –Env) Var Δ →
      Tm d ∞ σ Γ → Tm d ∞ σ Δ
ren ρ t = Semantics.semantics Ren ρ t
```

```
Sub : Semantics d (Tm d ∞) (Tm d ∞)
Sub .th^𝒱 = th^Tm
Sub .var   = id
Sub .alg   = ‘con ∘ fmap d (reify vl^Tm)

sub : (Γ –Env) (Tm d ∞) Δ →
      Tm d ∞ σ Γ → Tm d ∞ σ Δ
sub ρ t = Semantics.semantics Sub ρ t
```

The reification process mentioned in the definition of renaming and substitution can be implemented generically for Semantics families which have VarLike values, that is, values which are Thinnable and such that we can craft placeholder values in non-empty contexts. It is almost immediate that both Var and Tm are VarLike (with proofs vl^Var and vl^Tm, respectively).

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

Given a proof that $\mathcal{V}$ is VarLike, we can manufacture several useful environments of values $\mathcal{V}$. We provide users with base of type (Γ –Env) $\mathcal{V}$ Γ, fresh^r of type (Γ –Env) $\mathcal{V}$ (Δ ++ Γ) and fresh^l of type (Γ –Env) $\mathcal{V}$ (Γ ++ Δ) by combining the use of placeholder values and thinnings. In the Var case these very general definitions respectively specialise to the identity renaming for a context Γ and the injection of Γ fresh variables to the right or the left of an ambient context Δ. Similarly, in the Tm case, we can show base vl^Tm extensionally equal to the identity environment id^Tm given by lookup id^Tm = ‘var, which associates each variable to itself (seen as a term). Using these definitions, we can then implement reify as follows:

```
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 (freshʳ vl^Var Δ) (freshˡ vl^𝒱 _)
```

7 A catalogue of generic programs for syntax with binding

In this section we explore a large part of the spectrum of traversals a compiler writer may need when implementing their own language. In Section 7.1 we look at the production of human-readable representations of internal syntax; in Section 7.2 we write a generic scope checker thus bridging the gap between raw data fresh out of a parser to well scoped syntax; we then demonstrate how to write a type checker in Section 7.3 and even an elaboration function turning well scoped into well scoped and typed syntax in Section 7.4. We then study type and scope respecting transformations on internal syntax: desugaring in Section 7.5 and size preserving inlining in Section 7.6. We conclude with an unsafe but generic evaluator defined using normalisation by evaluation in Section 7.7.

7.1 Printing with names

We have seen in Section 3.3 that printing with names is an instance of ACMM's notion of Semantics. We will now show that this observation can be generalised to arbitrary syntaxes with binding. Unlike renaming or substitution, this generic program will require user guidance: there is no way for us to guess how an encoded term should be printed. We can however take care of the name generation (using the Fresh monad from Page 11), deal with variable binding, and implement the traversal generically. We want our printer to have type:

```
print : Display d → Tm d i σ Γ → String
```

where Display explains how to print one 'layer' of term provided that we are handed the Pieces corresponding to the printed subterm and names for the bound variables:

```
Display : Desc I → Set
Display d = ∀ {i Γ} → ⟦ d ⟧ Pieces i Γ → String
```

Reusing the notion of Name introduced in Section 3.3, we can make Pieces formal. A subterm has already been printed if we have a string representation of it together with an environment of Names we have attached to the newly bound variables this structure contains. That is to say:

```
Pieces : List I → I −Scoped
Pieces [] i Γ = String
Pieces Δ i Γ = (Δ −Env) Name (Δ ++ Γ) × String
```

The key observation that will help us define a generic printer is that Fresh composed with Name is VarLike. Indeed, as the composition of a functor and a trivially thinnable Wrapper, Fresh is Thinnable, and fresh (defined on Page 11) is the proof that we can generate placeholder values thanks to the name supply.

```
vl^FreshName : VarLike (λ (σ : I) → Fresh ∘ (Name σ))
vl^FreshName = record
  { th^𝒱 = th^Functor functor^M th^Wrap
  ; new   = fresh _
  }
```

This VarLike instance empowers us to reify in an effectful manner a Kripke function space taking Names and returning a Printer to a set of Pieces.

```
reify^Pieces : ∀ Δ i → Kripke Name Printer Δ i Γ → Fresh (Pieces Δ i Γ)
```

In case there are no newly bound variables, the Kripke function space collapses to a mere Printer which is precisely the wrapped version of the type we expect.

```
reify^Pieces [] i p = getW p
```

Otherwise we proceed in a manner reminiscent of the pure reification function defined at the end of Section 6.2. We start by generating an environment of names for the newly bound variables by using the fact that Fresh composed with Name is VarLike together with the fact that environments are Traversable (McBride and Paterson (2008)), and thus admit the standard Haskell-like mapA and sequenceA traversals. We then run the Kripke function on these names to obtain the string representation of the subterm. We finally return the names we used together with this string.

```
reify^Pieces Δ@(_ :: _) i f = do
  ρ ← sequenceA (fresh^l vl^FreshName _)
  b ← getW (f (fresh^r vl^Var Δ) ρ)
  return (ρ , b)
```

We can put all of these pieces together to obtain the Printing semantics. The first two constraints can be trivially discharged. When defining the algebra we start by reifying the subterms, then use the fact that one "layer" of term of our syntaxes with binding is always traversable to combine all of these results into a value we can apply our display function to.

```
Printing : Display d → Semantics d Name Printer
Printing dis .th^𝒱 = th^Wrap
Printing dis .var   = map^Wrap return
Printing dis .alg   = λ v → MkW $ dis <$> mapA d reify^Pieces v
```

This allows us to write a printer for open terms:

```
print : Display d → Tm d i σ Γ → String
print dis t = proj₁ (printer names) where
  printer : Fresh String
  printer = do
    init ← sequenceA (base vl^FreshName)
    getW (Semantics.semantics (Printing dis) init t)
```

We start by using base (defined in Section 6.2) to generate an environment of Names for the free variables, then use our semantics to get a printer which we can run using a stream names of distinct strings as our name supply.

Untyped λ-calculus. Defining a printer for the untyped λ-calculus is now very easy: we define a Display by case analysis. In the application case, we combine the string representation of the function, wrap its argument's representation between parentheses and concatenate the two together. In the lambda abstraction case, we are handed the name the bound variable was assigned together with the body's representation; it is once more a matter of putting the Pieces together.

```
printUTLC : Display UTLC
printUTLC = λ where
  (‘app’ f t)      → f ++ "  (" ++ t ++ ")"
  (‘lam’ (x , b)) → "λ" ++ getW (lookup x z) ++ ".  " ++ b
```

As always, these functions are readily executable and we can check their behaviour by writing tests. First, we print the identity function defined in Section 5 in an empty context and verify that we do obtain the string "λa. a". Next, we print an open term in a context of size two and can immediately observe that names are generated for the free variables first, and then the expression itself is printed.

```
_ : print printUTLC id^U ≡ "λa. a"
_ = refl

_ : let tm : Tm UTLC _ _ (_ :: _ :: [])
        tm = ‘app (‘var z) (‘lam (‘var (s (s z))))
    in print printUTLC tm ≡ "b (λc. a)"
_ = refl
```

7.2 Writing a generic scope checker

Converting terms in the internal syntax to strings which can in turn be displayed in a terminal or an editor window is only part of a compiler's interaction loop. The other direction takes strings as inputs and attempts to produce terms in the internal syntax. The first step is to parse the input strings into structured data, the second is to perform scope checking, and the third step consists of type checking.

Parsing is currently out of scope for our library; users can write safe ad-hoc parsers for their object language by either using a library of total parser combinators (Danielsson (2010); Allais (2018)) or invoking a parser generator oracle whose target is a total language (Stump (2016)). As we will see shortly, we can write a generic scope checker transforming terms in a raw syntax where variables are represented as strings into a well scoped syntax. We will come back to type checking with a concrete example in section 7.3 and then discuss related future work in the conclusion.

Our scope checker will be a function taking two explicit arguments: a name for each variable in scope Γ and a raw term for a syntax description *d*. It will either fail (the Monad Fail granting us the ability to fail is defined below) or return a well scoped and sorted term for that description.

```
toTm : Names Γ → Raw d i σ → Fail (Tm d i σ Γ)
```

Scope. We can obtain Names, the datastructure associating to each variable in scope its raw name as a string by reusing the standard library's All. The inductive family All is a predicate transformer making sure a predicate holds of all the element of a list. It is defined in a style common in Agda: because All's constructors are in one to one correspondence with that of its index type (List A), the same name are reused: [] is the name of the proof that P trivially holds of all the elements in the empty list []; similarly _::_ is the proof that provided that P holds of the element a on the one hand and of the elements of the list as on the other then it holds of all the elements of the list (a :: as).

```
data All (P : A → Set) : List A → Set where
  []   : All P []
  _::_ : P a → All P as → All P (a :: as)
```

```
Names : List I → Set
Names = All (const String)
```

Raw terms. The definition of WithNames is analogous to Pieces in the previous section: we expect Names for the newly bound variables. Terms in the raw syntax then leverage these definitions. They are either a variables or another "layer" of raw terms. Variables 'var carry a String and potentially some extra information E (typically a position in a file). The other constructor 'con carries a layer of raw terms where subterms are raw terms equiped with names for any newly bound variables.

```
WithNames : (I → Set) → List I → I −Scoped
WithNames T [] j Γ = T j
WithNames T Δ j Γ = Names Δ × T j
```

```
data Raw (d : Desc I) : Size → I → Set where
  'var : E → String → Raw d (↑ i) σ
  'con : ⟦ d ⟧ (WithNames (Raw d i)) σ [] → Raw d (↑ i) σ
```

Error handling. Various things can go wrong during scope checking: evidently a name can be out of scope but it is also possible that it may be associated to a variable of the wrong sort. We define an enumerating type covering these two cases. The scope checker will return a computation in the Monad Fail thus allowing us to fail and return an error, the string that caused the failure and the extra data of type E that accompanied it.

```
data Error : Set where
  OutOfScope : Error
  WrongSort  : (σ τ : I) → σ ≢ τ → Error
```

```
Fail : Set → Set
Fail A = (Error × E × String) ⊎ A

fail : Error → E → String → Fail A
fail err e str = inj₁ (err , e , str)
```

Equipped with these notions, we can write down the type of toVar which tackles the core of the problem: variable resolution. The function takes a string and a sort as well the names and sorts of the variables in the ambient scope. Provided that we have a function _≟I_ to decide equality on sorts, we can check whether the string corresponds to an existing variable and whether that binding is of the right sort. Thus we either fail or return a well scoped and well sorted Var.

If the ambient scope is empty then we can only fail with an OutOfScope error. Alternatively, if the variable's name corresponds to that of the first one in scope we check that the sorts match up and either return z or fail with a WrongSort error. Otherwise we look for the variable further down the scope and use s to lift the result to the full scope.

```
toVar : E → String → ∀ σ Γ → Names Γ → Fail (Var σ Γ)
toVar e x σ [] [] = fail OutOfScope e x
toVar e x σ (τ :: Γ) (y :: scp) with x ≟ y | σ ≟I τ
... | yes _ | yes refl = pure z
... | yes _ | no ¬eq  = fail (WrongSort σ τ ¬eq) e x
... | no ¬p | _        = s <$> toVar e x σ Γ scp
```

Scope checking an entire term then amounts to lifting this action on variables to an action on terms. The error Monad Fail is by definition an Applicative and by design our terms are Traversable (Bird and Paterson (1999); Gibbons and d. S. Oliveira (2009)). The action on term is defined mutually with the action on scopes. As we can see in the second equation for toScope, thanks to the definition of WithNames, concrete names arrive just in time to check the subterm with newly bound variables.

```
toTm    : Names Γ → Raw d i σ → Fail (Tm d i σ Γ)
toScope : Names Γ → ∀ Δ σ → WithNames (Raw d i) Δ σ [] →
          Fail (Scope (Tm d i) Δ σ Γ)

toTm scp (‘var e v) = ‘var <$> toVar e v _ _ scp
toTm scp (‘con b)   = ‘con <$> mapA d (toScope scp) b

toScope scp []              σ b          = toTm scp b
toScope scp Δ@(_ :: _) σ (bnd , b) = toTm (bnd ++ scp) b
```

7.3 An algebraic approach to type checking

Following Atkey (2015), we can consider type checking and type inference as a possible semantics for a bidirectional (Pierce and Turner (2000)) language. We reuse the syntax introduced in Section 5 and the types introduced for the STLC at the end of Section 2; it gives us a simply typed bidirectional calculus as a bisorted 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.

The values stored in the environment of the type checking function attach Type information to bound variables whose Mode is Infer, guaranteeing no variable ever uses the Check mode. In contrast, 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. In an actual compiler pipeline we would naturally use a different error monad and generate helpful error messages pointing out where the type error occured. The interested reader can see a fine-grained analysis of type errors in the extended example of a type checker in McBride and McKinna (2004).

```
data Var- : Mode → Set where
  ‘var : Type → Var- Infer
```

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

A change of direction from inferring to checking will require being able to check that two types agree so we introduce the function _=?_. Similarly we will sometimes expect a function type but may be handed anything so we will have to check with isArrow that our candidate's head constructor is indeed an arrow, and collect the domain and codomain.

```
_=?_ : (σ τ : Type) → Maybe ⊤
α          =? α         = just tt
(σ ‘→ τ) =? (ϕ ‘→ ψ) = (σ =? ϕ) >> (τ =? ψ)
_          =? _         = nothing

isArrow : Type → Maybe (Type × Type)
isArrow (σ ‘→ τ) = just (σ , τ)
isArrow _          = nothing
```

We can now define type checking as a Semantics. We describe the algorithm constructor by constructor; in the Semantics definition (omitted here) the algebra will simply perform a dispatch and pick the relevant auxiliary lemma. Note that in the following code, _<$_ is, following classic Haskell notations, the function which takes an A and a Maybe B and returns a Maybe A which has the same structure as its second argument.

Application. 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.

```
app : Type- Infer → Type- Check → Type- Infer
app f t = do
  arr     ← f
  (σ , τ) ← isArrow arr
  τ <$ t σ
```

λ-abstraction. For a λ-abstraction: check that the input type *arr* is an arrow type and check the body b at the codomain type in the extended environment (using bind) where the newly bound variable is of mode Infer and has the domain's type.

```
lam : Kripke (const ∘ Var-) (const ∘ Type-) (Infer :: []) Check Γ → Type- Check
lam b arr = do
  (σ , τ) ← isArrow arr
  b (bind Infer) (ε • ‘var σ) τ
```

Embedding of Infer into Check. The change of direction from Inferrable to Checkable is successful when the inferred type is equal to the expected one.

```
emb : Type- Infer → Type- Check
emb t σ = do
  τ ← t
  σ =? τ
```

Cut: A Check in an Infer position. So far, our bidirectional syntax only permits the construction of STLC terms in *canonical form* (Pfenning (2004); Dunfield and Pfenning (2004)). In order to construct non-normal (redex) terms, whose semantics is given logically by the 'cut' rule, we need to reverse direction. Our final semantic operation, cut, always comes with a type candidate against which to check the term and to be returned in case of success.

```
cut : Type → Type- Check → Type- Infer
cut σ t = σ <$ t σ
```

We have defined a bidirectional type checker for this simple language by leveraging the Semantics framework. We can readily run it on closed terms using the closed corollary defined in Section 6.1 and (defining β to be (α ‘→ α)) infer the type of the expression (λx. x : $\beta \to \beta$) (λx. x).

```
type- : ∀ p → TM Bidi p → Type- p
type- p = Semantics.closed Typecheck

_ : type- Infer (‘app (‘cut (β ‘→ β) id^B) id^B) ≡ just β
_ = refl
```

The output of this function is not very informative. As we will see shortly, there is nothing stopping us from moving away from a simple computation returning a (Maybe Type) to an evidence-producing function elaborating a term in Bidi to a well scoped and typed term in STLC.

7.4 An algebraic approach to elaboration

Instead of generating a type or checking that a candidate will do, we can use our language of Descriptions to define not only an untyped source language but also an intrinsically typed internal language. During type checking we simultaneously generate an expression's type and a well scoped and well typed term of that type. We use STLC (defined in Section 5) as our internal language.

Before we can jump right in, we need to set the stage: a Semantics for a Bidi term will involve (Mode −Scoped) notions of values and computations but an STLC term is (Type −Scoped). We first introduce a Typing associating types to each of the modes in scope, together with an erasure function ⌞_⌟ extracting the context of types implicitly defined by such a Typing. We will systematically distinguish contexts of modes (typically named *ms*) and their associated typings (typically named Γ).

```
Typing : List Mode → Set
Typing = All (const Type)

⌞_⌟ : Typing ms → List Type
⌞ []      ⌟ = []
⌞ σ :: Γ ⌟ = σ :: ⌞ Γ ⌟
```

We can then explain what it means for an elaboration process of type σ in a context of modes *ms* to produce a term of the (Type −Scoped) family T: for any typing Γ of this context of modes, we should get a value of type ($T\ \sigma$ ⌞ Γ ⌟).

```
Elab : Type −Scoped → Type → (ms : List Mode) → Typing ms → Set
Elab T σ _ Γ = T σ ⌞ Γ ⌟
```

Our first example of an elaboration process is our notion of environment values. To each variable in scope of mode Infer we associate an elaboration function targeting Var. In other words: our values are all in scope i.e. provided any typing of the scope of modes, we can assuredly return a type together with a variable of that type.

```
data Var- : Mode −Scoped where
  ‘var : (infer : ∀ Γ → Σ[ σ ∈ Type ] Elab Var σ ms Γ) → Var- Infer ms
```

We can for instance prove that we have such an inference function for a newly bound variable of mode Infer: given that the context has been extended with a variable of mode Infer, the Typing must also have been extended with a type σ. We can return that type paired with the variable z.

```
var0 : Var- Infer (Infer :: ms)
var0 = ‘var λ where (σ :: _) → (σ , z)
```

The computations are a bit more tricky. On the one hand, if we are in checking mode then we expect that for any typing of the scope of modes and any type candidate we can Maybe return a term at that type in the induced context. On the other hand, in the inference mode we expect that given any typing of the scope, we can Maybe return a type together with a term at that type in the induced context.

```
Elab- : Mode −Scoped
Elab- Check ms = ∀ Γ → (σ : Type) → Maybe (Elab (Tm STLC ∞) σ ms Γ)
Elab- Infer    ms = ∀ Γ → Maybe (Σ[ σ ∈ Type ] Elab (Tm STLC ∞) σ ms Γ)
```

Because we are now writing a type checker which returns evidence of its claims, we need more informative variants of the equality and isArrow checks. In the equality checking case we want to get a proof of propositional equality but we only care about the successful path and will happily return nothing when failing. Agda's support for (dependent!) do-notation makes writing the check really easy. For the arrow type, we introduce a family Arrow constraining the shape of its index to be an arrow type and redefine isArrow as a *view* targeting this inductive family (Wadler (1987); McBride and McKinna (2004)). We deliberately overload the constructor of the isArrow family by calling it _‘→_. This means that the proof that a given type has the shape (σ ‘→ τ) is literally written (σ ‘→ τ). This allows us to specify *in the type* whether we want to work with the full set of values in Type or only the subset corresponding to function types and to then proceed to write the same programs a Haskell programmers would, with the added confidence that ours are guaranteed to be total.

```
_=?_ : (σ τ : Type) → Maybe (σ ≡ τ)
α          =? α          = just refl
(σ ‘→ τ) =? (φ ‘→ ψ) = do
  refl ← σ =? φ
  refl ← τ =? ψ
  return refl
_ =? _ = nothing
```

```
data Arrow : Type → Set where
  _‘→_ : ∀ σ τ → Arrow (σ ‘→ τ)

isArrow : ∀ σ → Maybe (Arrow σ)
isArrow (σ ‘→ τ) = just (σ ‘→ τ)
isArrow _           = nothing
```

We now have all the basic pieces and can start writing elaboration code. We will use lowercase letter for terms in Bidi and uppercase ones for their elaborated counterparts in STLC. We once more start by dealing with each constructor in isolation before putting everything together to get a Semantics. These steps are very similar to the ones in the previous section.

Application. In the application case, we start by elaborating the function and we get its type together with its internal representation. We then check that the inferred type is indeed an Arrow and elaborate the argument using the corresponding domain. We conclude by returning the codomain together with the internal function applied to the internal argument.

```
app : ∀[ Elab- Infer ⇒ Elab- Check ⇒ Elab- Infer ]
app f t Γ = do
  (arr , F)  ← f Γ
  (σ ‘→ τ) ← isArrow arr
  T          ← t Γ σ
  return (τ , ‘app F T)
```

λ-abstraction. For the λ-abstraction case, we start by checking that the type candidate *arr* is an Arrow. We can then elaborate the body *b* of the lambda in a context of modes extended with one Infer variable, and the corresponding Typing extended with the function's domain. From this we get an internal term *B* corresponding to the body of the λ-abstraction and conclude by returning it wrapped in a ‘lam constructor.

```
lam : ∀[ Kripke Var- Elab- (Infer :: []) Check ⇒ Elab- Check ]
lam b Γ arr = do
  (σ ‘→ τ) ← isArrow arr
  B          ← b (bind Infer) (ε • var₀) (σ :: Γ) τ
  return (‘lam B)
```

Cut: A Check in an Infer position. For cut, we start by elaborating the term with the type annotation provided and return them paired together.

```
cut : Type → ∀[ Elab- Check ⇒ Elab- Infer ]
cut σ t Γ = (σ ,_) <$> t Γ σ
```

Embedding of Infer into Check. For the change of direction Emb we not only want to check that the inferred type and the type candidate are equal: we need to cast the internal term labelled with the inferred type to match the type candidate. Luckily, Agda's dependent do-notation make our job easy once again: when we make the pattern refl explicit, the equality holds in the rest of the block.

```
emb : ∀[ Elab- Infer ⇒ Elab- Check ]
emb t Γ σ = do
  (τ , T) ← t Γ
  refl     ← σ =? τ
  return T
```

We have almost everything we need to define elaboration as a semantics. Discharging the th^𝒱 constraint is a bit laborious and the proof doesn't yield any additional insight so we leave it out here. The semantical counterpart of variables (var) is fairly straightforward: provided a Typing, we run the inference and touch it up to return a term rather than a mere variable. Finally we define the algebra (alg) by pattern-matching on the constructor and using our previous combinators.

```
Elaborate : Semantics Bidi Var- Elab-
Elaborate .th^𝒱 = th^Var-
Elaborate .var    = λ where (‘var infer) Γ → just (map₂ ‘var (infer Γ))
Elaborate .alg    = λ where
  (‘app’ f t) → app f t
  (‘lam’ b)   → lam b
  (‘emb’ t)   → emb t
  (‘cut’ σ t) → cut σ t
```

We can once more define a specialised version of the traversal induced by this Semantics for closed terms: not only can we give a (trivial) initial environment (using the closed corollary defined in Section 6.1) but we can also give a (trivial) initial Typing. This leads to these definitions:

```
Type- : Mode → Set                          type- : ∀ p → TM Bidi p → Type- p
Type- Check = ∀ σ → Maybe (TM STLC σ)       type- Check t = closed Elaborate t []
Type- Infer  = Maybe (∃ λ σ → TM STLC σ)    type- Infer   t = closed Elaborate t []
```

Revisiting the example introduced in Section 7.3, we can check that elaborating the expression $(\lambda x.\ x : \beta \to \beta)\ (\lambda x.\ x)$ yields the type β together with the term $(\lambda x.\ x)\ (\lambda x.\ x)$ in internal syntax. Type annotations have disappeared in the internal syntax as all the type invariants are enforced intrinsically.

```
_ : type- Infer ( B.‘app (B.‘cut (β ‘→ β) id^B) id^B)
  ≡ just (β      , S.‘app                 id^S  id^S)
_ = refl
```

7.5 Sugar and desugaring as a semantics

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. An example of such an extension is the addition of let-bindings to an existing language.

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 Let which can be used to extend any language description *d* to a language with let-bindings (*d* ‘+ Let).

```
Let : Desc I                                  pattern 'let'_'in'_ e t = (_ , e , t , refl)
Let = 'σ (I × I) $ uncurry $ λ σ τ →          pattern 'let_'in_  e t = 'con ('let' e 'in' t)
      'X [] σ ('X (σ :: []) τ ('■ τ))
```

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.

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 type checking strategies can accommodate 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 type checking 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, the eliminator for _'+_ defined in Section 5: the old constructors are kept the same by interpreting them using the generic substitution algebra (Sub); whilst the let-binder precisely provides the extra value to be added to the environment.

```
UnLet : Semantics (d '+ Let) (Tm d ∞) (Tm d ∞)
Semantics.th^𝒱 UnLet = th^Tm
Semantics.var   UnLet = id
Semantics.alg   UnLet = case (Semantics.alg Sub) $ λ where
  ('let' e 'in' t) → extract t (ε • e)
```

The process of removing let-binders is then kickstarted with the placeholder environment id^Tm = pack 'var of type (Γ −Env) (Tm d ∞) Γ.

```
unlet : ∀[ Tm (d '+ Let) ∞ σ ⇒ Tm d ∞ σ ]
unlet = Semantics.semantics UnLet id^Tm
```

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 ACMM (2017), we focused on STLC only and showed that it is similarly possible to implement a Continuation Passing Style transformation as the composition of two semantics à la Hatcliff and Danvy (1994). The first semantics embeds STLC into Moggi's Meta-Language (1991) and thus fixes an evaluation order. The second one translates Moggi's ML back into STLC in terms of explicit continuations with a fixed return type.

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)).

7.6 Reference counting and inlining as a semantics

Although useful in its own right, desugaring all let-bindings can lead to an exponential blow-up in code size. Compiler passes typically try to maintain sharing by only inlining let-bound expressions which appear at most one time. Unused expressions are eliminated as dead code whilst expressions used exactly one time can be inlined: this transformation is size preserving and opens up opportunities for additional optimisations.

As we will see shortly, we can implement reference counting and size respecting let-inlining as a generic transformation over all syntaxes with binding equipped with let-binders. This two-pass simple transformation takes linear time which may seem surprising given the results due to Appel and Jim (1997). Our optimisation only inlines let-bound variables whereas theirs also encompasses the reduction of static β-redexes of (potentially) recursive function. While we can easily count how often a variable is used in the body of a let-binder, the interaction between inlining and β-reduction in theirs creates cascading simplification opportunities thus making the problem much harder.

But first, we need to look at an example demonstrating that this is a slightly subtle matter. Assuming that *expensive* takes a long time to evaluate, inlining all of the lets in the first expression is a really good idea whilst we only want to inline the one binding *y* in the second one to avoid duplicating work. That is to say that the contribution of the expression bound to *y* in the overall count depends directly on whether *y* itself appears free in the body of the let which binds it.

```
_ = let x = expensive in          _ = let x = expensive in
    let y = (x , x)   in              let y = (x , x)   in
    x                                 y
```

Our transformation will consist of two passes: the first one will annotate the tree with accurate count information precisely recording whether let-bound variables are used zero, one, or many times. The second one will inline precisely the let-binders whose variable is used at most once.

During the counting phase we need to be particularly careful not to overestimate the contribution of a let-bound expression. If the let-bound variable is not used then we can naturally safely ignore the associated count. But if it used many times then we know we will not inline this let-binding and the count should therefore only contribute once to the running total. We define the control combinator below precisely to explicitly handle this subtle case.

The first step is to introduce the Counter additive monoid. Addition will allow us to combine counts coming from different subterms: if any of the two counters is zero then we return the other, otherwise we know we have many occurences.

```
data Counter : Set where             _+_ : Counter → Counter → Counter
  zero : Counter                     zero + n    = n
  one  : Counter                     m    + zero = m
  many : Counter                     _    + _    = many
```

The syntax extension CLet defined as follows is a variation on the Let syntax extension of Section 7.5, attaching a Counter to each Let node. The annotation process can then be described as a function computing a (*d* '+ CLet) term from a (*d* '+ Let) one.

```
CLet : Desc I
CLet = 'σ Counter $ λ _ → Let
```

We keep a tally of the usage information for the variables in scope. This allows us to know which Counter to attach to each Let node. Following the same strategy as in Section 7.2, we use the standard library's All to represent this mapping. We say that a scoped value has been Counted if it is paired with a Count.

```
Count : List I → Set
Count = All (const Counter)
```

```
Counted : I –Scoped → I –Scoped
Counted T i Γ = T i Γ × Count Γ
```

The two most basic counts are zeros and fromVar: the empty one is zero everywhere and the one corresponding to a single use of a single variable *v* which is zero everywhere except for *v* where it is one.

```
zeros : ∀[ Count ]
zeros {[]}       = []
zeros {σ :: Γ} = zero :: zeros
```

```
fromVar : ∀[ Var σ ⇒ Count ]
fromVar z     = one :: zeros
fromVar (s v) = zero :: fromVar v
```

When we collect usage information from different subterms, we need to put the various counts together. The combinators we now define allow us to easily do so: merge adds up two counts in a pointwise manner while control uses one Counter to decide whether to erase an existing Count. This is particularly convenient when computing the contribution of a let-bound expression to the total tally: the contribution of the let-bound expression will only matter if the corresponding variable is actually used.

```
merge : ∀[ Count ⇒ Count ⇒ Count ]
merge []         []         = []
merge (m :: cs) (n :: ds) =
  (m + n) :: merge cs ds
```

```
control : Counter → ∀[ Count ⇒ Count ]
control zero   cs = zeros
control one    cs = cs -- inlined
control many cs = cs -- not inlined
```

We can now focus on the core of the annotation phase, defining a Semantics whose values are variables themselves and whose computations are the pairing of a term in (*d* '+ CLet) together with a Count. The variable case is trivial: provided a variable *v*, we return ('var *v*) together with the count (fromVar *v*).

The non-let case is purely structural: we reify the Kripke function space and obtain a scope together with the corresponding Count. We unceremoniously drop the Counters associated to the variables bound in this subterm and return the scope together with the tally for the ambient context.

```
reify^Count : ∀ Δ σ → Kripke Var (Counted (Tm (d '+ CLet) ∞)) Δ σ Γ →
                              Counted (Scope (Tm (d '+ CLet) ∞) Δ) σ Γ
reify^Count Δ σ kr = let (scp , c) = reify vl^Var Δ σ kr in scp , drop Δ c
```

The Let-to-CLet case is the most interesting one. We start by reifying the *body* of the let-binder which gives us a tally *cx* for the bound variable and *ct* for the body's contribution to the ambient environment's Count. We annotate the node with *cx* and use it as a control to decide whether we are going to merge any of the let-bound's expression contribution *ce* to form the overall tally.

```
clet : ⟦ Let ⟧ (Kripke Var (Counted (Tm (d '+ CLet) ∞))) σ Γ →
      Counted (⟦ CLet ⟧ (Scope (Tm (d '+ CLet) ∞))) σ Γ
clet (στ , (e , ce) , body , eq) = case body weaken (ε • z) of λ where
  (t , cx :: ct) → (cx , στ , e , t , eq) , merge (control cx ce) ct
```

Putting all of these things together we obtain the Semantics Annotate. We promptly specialise it using an environment of placeholder values to obtain the traversal annotate elaborating raw let-binders into counted ones.

```
annotate : ∀[ Tm (d '+ Let) ∞ σ ⇒ Tm (d '+ CLet) ∞ σ ]
annotate t = let (t' , _) = Semantics.semantics Annotate identity t in t'
```

Using techniques similar to the ones described in Section 7.5, we can write an Inline semantics working on (*d* '+ CLet) terms and producing (*d* '+ Let) ones. We make sure to preserve all the let-binders annotated with many and to inline all the other ones. By composing Annotate with Inline we obtain a size-preserving generic optimisation pass.

7.7 (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 σ ⇒ Dm d s σ ]
  C : ∀[ ⟦ d ⟧ (Kripke (Dm d s) (Dm d s)) σ ⇒ Dm d (↑ s) σ ]
  ⊥ : ∀[ Dm d (↑ s) σ ]
```

This data type definition is utterly unsafe. The more conservative user will happily restrict themselves to particular syntaxes where the typed settings allows for a domain to 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); Gibbons and d. S. Oliveira (2009)) 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 σ ⇒ Maybe ∘ Tm d ∞ σ ]
nbe      : Alg d (Dm d ∞) (Dm d ∞) → Semantics d (Dm d ∞) (Dm d ∞)

norm     : Alg d (Dm d ∞) (Dm d ∞) → ∀[ Tm d ∞ σ ⇒ Maybe ∘ Tm d ∞ σ ]
norm alg = reify^Dm ∘ Semantics.semantics (nbe alg) (base vl^Dm)
```

Using this setup, we can write a normaliser for the untyped λ-calculus by providing an algebra. The key observation that allows us to implement this algebra is that we can turn a Kripke function, f, mapping values of type σ to computations of type τ into an Agda function from values of type σ to computations of type τ. This is witnessed by the application function (_$$_): we first use extract, defined in Section 3.1, to obtain a function taking environments of values to computations. We then use the environment building combinators defined there to manufacture the singleton environment ($\varepsilon \bullet t$) containing the value t of type σ.

```
_$$_ : ∀[ Kripke 𝒱 C (σ :: []) τ ⇒ (𝒱 σ ⇒ C τ) ]
f $$ t = extract f (ε ∙ t)
```

We now define two patterns for semantical values: one for application and the other for lambda abstraction. This should make the case of interest of our algebra (a function applied to an argument) fairly readable.

```
pattern LAM f = C (false , f , refl)
pattern APP' f t = (true , f , t , refl)
```

We finally define the algebra by case analysis: if the node at hand is an application and its first component evaluates to a lambda, we can apply the function to its argument using _$$_. Otherwise we have either a stuck application or a lambda, in other words we already have a value and can simply return it using C.

```
norm^LC : ∀[ Tm UTLC ∞ tt ⇒ Maybe ∘ Tm UTLC ∞ tt ]
norm^LC = norm $ λ where
  (APP' (LAM f) t) → f $$ t -- redex
  t                → C t    -- value
```

We have not used the ⊥ constructor so *if* the evaluation terminates (by disabling totality checking we have lost all guarantees of the sort) we know we will get a term in normal form. For instance, we can evaluate an untyped yet normalising term (λx. x) ((λx. x) (λx. x)) that normalises to (λx. x):

```
_ : norm^LC (‘app id^U (‘app id^U id^U)) ≡ just id^U
_ = refl
```

8 Other opportunities for generic programming

Some generic programs of interest do not fit in the Semantics framework. They can still be implemented once and for all, and even benefit from the Semantics-based definitions.

We will first explore existing work on representing cyclic structures using a syntax with binding: a binder is a tree node declaring a pointer giving subtrees the ability to point back to it, thus forming a cycle. Substitution will naturally play a central role in giving these finite terms a semantics as their potentially infinite unfolding.

We will then see that many of the standard traversals produced by the "deriving" machinery familiar to Haskell programmers can be implemented on syntaxes too, sometimes with more informative types.

8.1 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
         ‘+ ‘σ A (λ _ → ‘X (tt :: []) tt (‘■ tt))

pattern []         = ‘con (true , refl)
pattern _::_ x xs = ‘con (false , x , xs , refl)
pattern ↶_ k       = ‘var k
```

```
[0,1] : TM (CListD ℕ) tt
01↻ : TM (CListD ℕ) tt

[0,1] = 0 :: 1 :: []
01↻ = 0 :: 1 :: ↶ s z
```

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 = Semantics.semantics Sub (pack (λ _ → t))
```

```
unroll : TM d tt → ⟦ d ⟧ (Const (TM d)) tt []
unroll t@(‘con b) = fmap d (plug t) b
```

However, one thing still out of our reach with our current tools is the underlying cofinite trees these finite objects are meant to represent. We start by defining the coinductive type corresponding to them as the greatest fixpoint of a notion of layer. One layer of a cofinite tree is precisely given by the meaning of its description where we completely ignore the binding structure. We show with 01··· (mutually defined with 10···) the infinite list that corresponds to the unfolding of the example 01↺ given above.

```
record ∞Tm (d : Desc I) (s : Size) (i : I) : Set where
  coinductive; constructor ‘con
  field force : {s’ : Size< s} →
               ⟦ d ⟧ (Const (∞Tm d s’)) i []
```

```
01··· : ∞Tm (CListD ℕ) i tt                10··· : ∞Tm (CListD ℕ) i tt
01··· .force = false , 0 , 10··· , refl    10··· .force = false , 1 , 01··· , refl
```

We can then make the connection between potentially cyclic structures and the cofinite 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)
```

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 ⟦_⟧.

8.2 Generic decidable equality for terms

Haskell programmers are used to receiving help from the “deriving” mechanism (Hinze and Peyton Jones (2000); Magalhães et al. (2010)) to automatically generate common traversals for every inductive type they define. Recalling that generic programming is normal programming over a universe in a dependently typed language (Altenkirch and McBride (2002)), we ought to be able to deliver similar functionalities for syntaxes with binding.

We will focus in this section on the definition of an equality test. The techniques used in this concrete example are general enough that they also apply to the definition of an ordering test, a `Show` instance, etc. In type theory we can do better than an uninformative boolean function claiming that two terms are equal: we can implement a decision procedure for propositional equality (Löh and Magalhães (2011)) which either returns a proof that its two inputs are equal or a proof that they cannot possibly be.

The notion of decidability can be neatly formalised by an inductive family with two constructors: a Set P is decidable if we can either say yes and return a proof of P or no and provide a proof of the negation of P (here, a proof that P implies the empty type ⊥).

```
data ⊥ : Set where                data Dec (P : Set) : Set where
                                    yes : P           → Dec P
                                    no  : (P → ⊥) → Dec P
```

To get acquainted with these new notions we can start by proving variable equality decidable.

8.2.1 Deciding variable equality

The type of the decision procedure for equality of variables is as follows: given any two variables (of the same type, in the same context), the set of equality proofs between them is Decidable.

```
eq^Var : (v w : Var σ Γ) → Dec (v ≡ w)
```

We can easily dismiss two trivial cases: if the two variables have distinct head constructors then they cannot possibly be equal. Agda allows us to dismiss the impossible premise of the function stored in the no contructor by using an absurd pattern ().

```
eq^Var z      (s w) = no (λ ())
eq^Var (s v) z      = no (λ ())
```

Otherwise if the two head constructors agree we can be in one of two situations. If they are both z then we can conclude that the two variables are indeed equal to each other.

```
eq^Var z z = yes refl
```

Finally if the two variables are (s v) and (s w) respectively then we need to check recursively whether v is equal to w. If it is the case we can conclude by invoking the congruence rule for s. If v and w are not equal then a proof that (s v) and (s w) are will lead to a direct contradiction by injectivity of the constructor s.

```
eq^Var (s v) (s w) with eq^Var v w
... | yes p = yes (cong s p)
... | no ¬p = no λ where refl → ¬p refl
```

8.2.2 Deciding term equality

The constructor ‘σ for descriptions gives us the ability to store values of any Set in terms. For some of these Sets (e.g. (ℕ → ℕ)), equality is not decidable. As a consequence our decision procedure will be conditioned to the satisfaction of a certain set of Constraints which we can compute from the Desc itself. We demand that we are able to decide equality for all of the Sets mentioned in a description.

```
Constraints : Desc I → Set
Constraints (‘σ A d)    = ((a b : A) → Dec (a ≡ b)) × (∀ a → Constraints (d a))
Constraints (‘X _ _ d) = Constraints d
Constraints (‘■ _)      = ⊤
```

Remembering that our descriptions are given a semantics as a big right-nested product terminated by an equality constraint, we realise that proving decidable equality will entail proving equality between proofs of equality. We are happy to assume Streicher's axiom K (Hofmann and Streicher (1994)) to easily dismiss this case. A more conservative approach would be to demand that equality is decidable on the index type I and to then use the classic Hedberg construction (Hedberg (1998)) to recover uniqueness of identity proofs for I.

Assuming that the constraints computed by (Constraints d) are satisfied, we define the decision procedure for equality of terms together with its equivalent for bodies. The function eq^Tm is a straightforward case analysis dismissing trivially impossible cases where terms have distinct head constructors ('var vs. 'con) and using either eq^Var or eq^⟦⟧ otherwise. The latter is defined by induction over e. The somewhat verbose definitions are not enlightening so we leave them out here.

```
eq^Tm : (t u : Tm d i σ Γ) → Dec (t ≡ u)
eq^⟦⟧ : ∀ e → Constraints e → (b c : ⟦ e ⟧ (Scope (Tm d i)) σ Γ) → Dec (b ≡ c)
```

We now have an informative decision procedure for equality between terms provided that the syntax they belong to satisfies a set of constraints. Other generic functions and decision procedures can be defined following the same approach: implement a similar function for variables first, compute a set of constraints, and demonstrate that they are sufficient to handle any input term.

9 Building generic proofs about generic programs

In ACMM (2017) we 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 us to give abstract proof frameworks for simulation or fusion lemmas. This idea naturally extends to our generic presentation of semantics for all syntaxes.

9.1 Relations and relation transformers

In our exploration of generic proofs about the behaviour of various Semantics, we are going to need to manipulate relations between distinct notions of values or computations. In this section, we introduce the notion of relation we are going to use as well as these two key relation transformers.

In Section 3.1 we introduced a generic notion of well typed and scoped environment as a function from variables to values. Its formal definition is given as a record type. This record wrapper helps Agda's type inference reconstruct the type family of values whenever it is passed an environment.

For the same reason, we will use a record wrapper for the concrete implementation of our notion of relation over (I –Scoped) families. A Relation between two such families T and U is a function which to any σ and Γ associates a relation between ($T\ \sigma\ \Gamma$) and ($U\ \sigma\ \Gamma$). Our first example of such a relation is Eq^R the equality relation between an (I–Scoped) family T and itself.

```
record Rel (T U : I –Scoped) : Set₁ where        Eq^R : Rel T T
  constructor mkRel                               rel Eq^R i = _≡_
  field rel : ∀ σ → ∀[ T σ ⇒ U σ ⇒ const Set ]
```

Once we know what relations are, we are going to have to lift relations on values and computations to relations on environments, Kripke function spaces or on d-shaped terms whose subterms have been evaluated already. This is what the rest of this section focuses on.

Environment relator. Provided a relation $\mathcal{V}^R$ for notions of values $\mathcal{V}^A$ and $\mathcal{V}^B$, by pointwise lifting we can define a relation (All $\mathcal{V}^R$ Γ) on Γ-environments of values $\mathcal{V}^A$ and $\mathcal{V}^B$ respectively. We once more use a record wrapper simply to facilitate Agda's job when reconstructing implicit arguments.

```
record All (𝒱^R : Rel 𝒱^A 𝒱^B) (Γ : List I)
           (ρ^A : (Γ –Env) 𝒱^A Δ) (ρ^B : (Γ –Env) 𝒱^B Δ) : Set where
  constructor pack^R
  field lookup^R : ∀ k → rel 𝒱^R σ (lookup ρ^A k) (lookup ρ^B k)
```

The first example of two environment being related is refl^R that, to any environment ρ associates a trivial proof of the statement (All Eq^R $\Gamma\ \rho\ \rho$). The combinators we introduced in Section 3.1 to build environments (ε, _•_, etc.) have natural relational counterparts. We reuse the same names for them, simply appending an ^{R} suffix.

Kripke relator. We assume that we have two types of values $\mathcal{V}^A$ and $\mathcal{V}^B$ as well as a relation $\mathcal{V}^R$ for pairs of such values, and two types of computations C^A and C^B whose notion of relatedness is given by C^R. We can define Kripke^R relating Kripke functions of type (Kripke $\mathcal{V}^A$ C^A) and (Kripke $\mathcal{V}^B$ C^B) respectively by stating that they send related inputs to related outputs. We use the relation transformer All defined in the previous paragraph.

```
Kripke^R : ∀ Δ i → ∀[ Kripke 𝒱^A C^A Δ i ⇒ Kripke 𝒱^B C^B Δ i ⇒ const Set ]
Kripke^R []           σ k^A k^B = rel C^R σ k^A k^B
Kripke^R Δ@(_ :: _) σ k^A k^B = ∀ {Θ} (ρ : Thinning _ Θ) {vs^A vs^B} →
                                 All 𝒱^R Δ vs^A vs^B → rel C^R σ (k^A ρ vs^A) (k^B ρ vs^B)
```

Desc relator. The relator ($[\![\ d\ ]\!]^R$) is 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 ($[\![\ d\ ]\!]^R\ R$).

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 $\Delta\ j\ d$, we demand that the two substructures are related by $R\ \Delta\ j$ and that the rest of the layers are related by ($[\![\ d\ ]\!]^R\ R$)
- 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 ($[\![\ d\ a\ ]\!]^R\ R$)

```
⟦_⟧R : (d : Desc I) → (∀ Δ σ → ∀[ X Δ σ ⇒ Y Δ σ ⇒ const Set ])
                    → ∀[ ⟦ d ⟧ X σ ⇒ ⟦ d ⟧ Y σ ⇒ const Set ]
⟦ '■ j      ⟧R R x       y       = ⊤
⟦ 'X Δ j d ⟧R R (r , x) (r' , y) = R Δ j r r' × ⟦ d ⟧R R x y
⟦ 'σ A d   ⟧R R (a , x) (a' , y) = Σ (a' ≡ a) (λ where refl → ⟦ d a ⟧R R x y)
```

If we were to take a fixpoint of ⟦_⟧^{R}, 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.

9.2 Simulation lemma

A constraint mentioning all three relation transformers appears naturally when we want to say that a semantics can simulate another one. For instance, renaming is simulated by substitution: we simply have to restrict ourselves to environments mapping variables to terms which happen to be variables. More generally, given a semantics $\mathcal{S}^A$ with values $\mathcal{V}^A$ and computations C^A and a semantics $\mathcal{S}^B$ with values $\mathcal{V}^B$ and computations C^B, we want to establish the constraints under which these two semantics yield related computations provided they were called with environments of related values.

These constraints are packaged in a record type called Simulation and parametrised over the semantics as well as the notion of relatedness used for values (given by a relation $\mathcal{V}^R$) and computations (given by a relation C^R).

```
record Simulation (d : Desc I)
  (𝒮A : Semantics d 𝒱A CA) (𝒮B : Semantics d 𝒱B CB)
  (𝒱R : Rel 𝒱A 𝒱B) (CR : Rel CA CB) : Set where
```

The two first constraints are self-explanatory: the operations th^𝒱 and var defined by each semantics should be compatible with the notions of relatedness used for values and computations.

```
  thR : (ρ : Thinning Γ Δ) → rel 𝒱R σ vA vB → rel 𝒱R σ (𝒮A.th^𝒱 vA ρ) (𝒮B.th^𝒱 vB ρ)

  varR : rel 𝒱R σ vA vB → rel CR σ (𝒮A.var vA) (𝒮B.var vB)
```

The third constraint is similarly simple: the algebras (alg) should take related recursively evaluated subterms of respective types ⟦ d ⟧ (Kripke $\mathcal{V}^A$ C^A) and ⟦ d ⟧ (Kripke $\mathcal{V}^B$ C^B) to related computations. The difficuly is in defining an appropriate notion of relatedness body^R for these recursively evaluated subterms.

```
  algR : (b : ⟦ d ⟧ (Scope (Tm d s)) σ Γ) → All 𝒱R Γ ρA ρB →
         let vA = fmap d (𝒮A.body ρA) b
             vB = fmap d (𝒮B.body ρB) b
         in bodyR vA vB → rel CR σ (𝒮A.alg vA) (𝒮B.alg vB)
```

We can combine ⟦_⟧^{R} and Kripke^R to express the idea that two recursively evaluated subterms are related whenever they have an equal shape (which means their Kripke functions can be grouped in pairs) and that all the pairs of Kripke function spaces take related inputs to related outputs.

```
bodyᴿ : ⟦ d ⟧ (Kripke 𝒱ᴬ Cᴬ) σ Δ → ⟦ d ⟧ (Kripke 𝒱ᴮ Cᴮ) σ Δ → Set
bodyᴿ vᴬ vᴮ = ⟦ d ⟧ᴿ (Kripkeᴿ 𝒱ᴿ Cᴿ) vᴬ vᴮ
```

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   : All 𝒱ᴿ Γ ρᴬ ρᴮ → (t : Tm d s σ Γ) →
        rel Cᴿ σ (𝒮ᴬ.semantics ρᴬ t) (𝒮ᴮ.semantics ρᴮ t)
body : All 𝒱ᴿ Γ ρᴬ ρᴮ → ∀ Δ j → (t : Scope (Tm d s) Δ j Γ) →
        Kripkeᴿ 𝒱ᴿ Cᴿ Δ j (𝒮ᴬ.body ρᴬ Δ j t) (𝒮ᴮ.body ρᴮ Δ j t)

sim ρᴿ (‘var k) = varᴿ (lookupᴿ ρᴿ k)
sim ρᴿ (‘con t) = algᴿ t ρᴿ (liftᴿ d (body ρᴿ) t)

body ρᴿ []        i t = sim ρᴿ t
body ρᴿ (_ :: _) i t = λ σ vsᴿ → sim (vsᴿ >>ᴿ (thᴿ σ <$>ᴿ ρᴿ)) t
```

Instantiating this generic simulation lemma, we can for instance prove that renaming is a special case of substitution, or that renaming and substitution are extensional, that is, that given environments equal in a pointwise manner they produce syntactically equal terms. Of course these results are not new but having them generically over all syntaxes with binding is convenient. The first author experienced this first hand when tackling the POPLMark Reloaded challenge (2017) where rensub was actually needed.

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

RenSub : Simulation d Ren Sub VarTmᴿ Eqᴿ
```

When studying specific languages, new opportunities to deploy the fundamental lemma of simulations arise. The first author's solution to the POPLMark Reloaded challenge (2019) 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 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 example of the abstract proof frameworks introduced in ACMM (2017). We 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.

9.3 Fusion lemma

Results that 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 (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 the first author's entry (2019) to the POPLMark Reloaded challenge (2017). Using a Desc-based representation of intrinsically 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 more involved than simulation; we will once more step through each one of the constraints individually, trying to give the reader an intuition for why they are shaped the way they are.

9.3.1 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 C^i. The fundamental lemma associated to such a set of constraints will state that running $\mathcal{S}^B$ after $\mathcal{S}^A$ is equivalent to running $\mathcal{S}^{AB}$ only.

The definition of fusion is parametrised by three relations: $\mathcal{E}^R$ relates triples of environments of values in $(\Gamma\ \mathsf{-Env})\ \mathcal{V}^A\ \Delta$, $(\Delta\ \mathsf{-Env})\ \mathcal{V}^B\ \Theta$ and $(\Gamma\ \mathsf{-Env})\ \mathcal{V}^{AB}\ \Theta$ respectively; $\mathcal{V}^R$ relates pairs of values $\mathcal{V}^B$ and $\mathcal{V}^{AB}$; and C^R, our notion of equivalence for evaluation results, relates pairs of computation in C^B and C^{AB}.

$$
\begin{array}{l}
\mathsf{record}\ \mathsf{Fusion}\ (d : \mathsf{Desc}\ I)\ (\mathcal{S}^A : \mathsf{Semantics}\ d\ \mathcal{V}^A\ C^A)\ (\mathcal{S}^B : \mathsf{Semantics}\ d\ \mathcal{V}^B\ C^B) \\
\quad (\mathcal{S}^{AB} : \mathsf{Semantics}\ d\ \mathcal{V}^{AB}\ C^{AB}) \\
\quad (\mathcal{E}^R : \forall\ \Gamma\ \Delta\ \{\Theta\} \to (\Gamma\ \mathsf{-Env})\ \mathcal{V}^A\ \Delta \to (\Delta\ \mathsf{-Env})\ \mathcal{V}^B\ \Theta \to (\Gamma\ \mathsf{-Env})\ \mathcal{V}^{AB}\ \Theta \to \mathsf{Set}) \\
\quad (\mathcal{V}^R : \mathsf{Rel}\ \mathcal{V}^B\ \mathcal{V}^{AB})\ (C^R : \mathsf{Rel}\ C^B\ C^{AB}) : \mathsf{Set}\ \mathsf{where}
\end{array}
$$

The first obstacle we face is the formal definition of "running $\mathcal{S}^B$ after $\mathcal{S}^A$": for this statement to make sense, the result of running $\mathcal{S}^A$ ought to be a term. Or rather, we ought to be able to extract a term from a C^A. Hence the first constraint: the existence of a reify^A 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. Nothing prevents proofs, such as the idempotence of NbE, which use a bona fide reification function that extracts terms from model values.

$$
\mathsf{reify}^A : \forall\ \sigma \to \forall[\ C^A\ \sigma \Rightarrow \mathsf{Tm}\ d\ \infty\ \sigma\ ]
$$

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

```
vl^𝒱ᴬ : VarLike 𝒱ᴬ
```

Still thinking about going under binders: if three evaluation environments ρ^A in (Γ –Env) $\mathcal{V}^A$ Δ, ρ^B in (Δ –Env) $\mathcal{V}^B$ Θ, and ρ^{AB} in (Γ –Env) $\mathcal{V}^{AB}$ Θ are related by $\mathcal{E}^R$ and we are given a thinning σ from Θ to Ω then ρ^A, the thinned ρ^B and the thinned ρ^{AB} should still be related.

```
th^ℰᴿ : ℰᴿ Γ Δ ρᴬ ρᴮ ρᴬᴮ → (ρ : Thinning Θ Ω) →
        ℰᴿ Γ Δ ρᴬ (th^Env 𝒮ᴮ.th^𝒱 ρᴮ ρ) (th^Env 𝒮ᴬᴮ.th^𝒱 ρᴬᴮ ρ)
```

Remembering that _>>_ is used in the definition of body (Section 6.1) to combine two disjoint environments (Γ –Env) $\mathcal{V}$ Θ and (Δ –Env) $\mathcal{V}$ Θ into one of type ((Γ ++ Δ) –Env) $\mathcal{V}$ Θ), we mechanically need a constraint stating that _>>_ is compatible with $\mathcal{E}^R$. We demand as an extra precondition that the values ρ^B and ρ^{AB} are extended with are related according to $\mathcal{V}^R$. Lastly, for all the types to match up, ρ^A has to be extended with placeholder variables which is possible because we have already insisted on $\mathcal{V}^A$ being VarLike.

```
_>>ᴿ_ : ℰᴿ Γ Δ ρᴬ ρᴮ ρᴬᴮ → All 𝒱ᴿ Θ vsᴮ vsᴬᴮ →
        let id>>ρᴬ = freshˡ vl^𝒱ᴬ Δ >> th^Env 𝒮ᴬ.th^𝒱 ρᴬ (freshʳ vl^Var Θ)
        in ℰᴿ (Θ ++ Γ) (Θ ++ Δ) id>>ρᴬ (vsᴮ >> ρᴮ) (vsᴬᴮ >> ρᴬᴮ)
```

We finally arrive at the constraints focusing on the semantical counterparts of the terms' constructors. Each constraint essentially states that evaluating a term with $\mathcal{S}^A$, reifying the result and running $\mathcal{S}^B$ is equivalent to using $\mathcal{S}^{AB}$ straight away. This can be made formal by defining the following relation $\mathcal{R}$.

```
ℛ : ∀ σ → (Γ –Env) 𝒱ᴬ Δ → (Δ –Env) 𝒱ᴮ Θ → (Γ –Env) 𝒱ᴬᴮ Θ →
    Tm d s σ Γ → Set
ℛ σ ρᴬ ρᴮ ρᴬᴮ t = rel 𝒞ᴿ σ (evalᴮ ρᴮ (reifyᴬ σ (evalᴬ ρᴬ t))) (evalᴬᴮ ρᴬᴮ t)
```

When evaluating a variable, on the one hand $\mathcal{S}^A$ will look up its meaning in the evaluation environment, turn the resulting value into a computation which will get reified and then the result will be evaluated with $\mathcal{S}^B$. Provided that all three evaluation environments are related by $\mathcal{E}^R$ this should be equivalent to looking up the value in $\mathcal{S}^{AB}$'s environment and turning it into a computation. Hence the constraint var^R:

```
varᴿ : ℰᴿ Γ Δ ρᴬ ρᴮ ρᴬᴮ → ∀ v → ℛ σ ρᴬ ρᴮ ρᴬᴮ (`var v)
```

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

$\textsf{alg}^R : \mathcal{E}^R\ \Gamma\ \Delta\ \rho^A\ \rho^B\ \rho^{AB} \to (b : [\![\ d\ ]\!]\ (\textsf{Scope}\ (\textsf{Tm}\ d\ s))\ \sigma\ \Gamma) \to$
$\quad \textsf{let}\ b^A : [\![\ d\ ]\!]\ (\textsf{Kripke}\ \mathcal{V}^A\ C^A)\ \_\ \_$
$\qquad b^A \ = \textsf{fmap}\ d\ (\mathcal{S}^A.\textsf{body}\ \rho^A)\ b$
$\qquad b^B \ = \textsf{fmap}\ d\ (\lambda\ \Delta\ i \to \mathcal{S}^B.\textsf{body}\ \rho^B\ \Delta\ i \circ \textsf{quote}^A\ \Delta\ i)\ b^A$
$\qquad b^{AB} = \textsf{fmap}\ d\ (\mathcal{S}^{AB}.\textsf{body}\ \rho^{AB})\ b$
$\quad \textsf{in}\ [\![\ d\ ]\!]^R\ (\textsf{Kripke}^R\ \mathcal{V}^R\ C^R)\ b^B\ b^{AB} \to \mathcal{R}\ \sigma\ \rho^A\ \rho^B\ \rho^{AB}\ (\text{`con}\ b)$

9.3.2 The fundamental lemma of fusion

This set of constraints 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 $\mathcal{S}^A$ and $\mathcal{S}^B$ on one hand and $\mathcal{S}^{AB}$ on the other. This lemma is once again proven mutually with its counterpart for Semantics's body's action on Scopes.

$\textsf{fusion} : \mathcal{E}^R\ \Gamma\ \Delta\ \rho^A\ \rho^B\ \rho^{AB} \to (t : \textsf{Tm}\ d\ s\ \sigma\ \Gamma) \to \mathcal{R}\ \sigma\ \rho^A\ \rho^B\ \rho^{AB}\ t$

9.3.3 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.

$\textsf{subren} : (t : \textsf{Tm}\ d\ i\ \sigma\ \Gamma)\ (\rho_1 : (\Gamma\ {-}\textsf{Env})\ (\textsf{Tm}\ d\ \infty)\ \Delta)\ (\rho_2 : \textsf{Thinning}\ \Delta\ \Theta) \to$
$\quad \textsf{ren}\ \rho_2\ (\textsf{sub}\ \rho_1\ t) \equiv \textsf{sub}\ (\textsf{ren}\ \rho_2\ \texttt{<\$>}\ \rho_1)\ t$

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 reproduced this result generically for all syntaxes (see accompanying code). The need for functional extensionality arises in the proof when dealing with subterms which have extra bound variables. These terms are interpreted as Kripke functional spaces in the host language and we can only prove that they take equal inputs to equal outputs. An intensional notion of equality will simply not do here. As a consequence, we refrain from using the generic result in practice when an axiom-free alternative is provable. Kaiser, Schäfer and Stark's observation naturally raises the question of whether the same semantics are also stable under substitution. Our semantics implementing printing with names is a clear counterexample.

9.4 Definition of bisimilarity for cofinite 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 cofinite trees. What we want is a generic coinductive notion of bisimilarity for all cofinite 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 $[\![\_]\!]^R$ 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} → ⟦ d ⟧^R (λ _ i → ≈^∞Tm d s′ i) (t .force) (u .force)
```

We can then prove by coinduction that this generic definition always gives rise to an equivalence relation using the relator's stability properties (if R is reflexive / symmetric / transitive then so is ($⟦ d ⟧^R$ R) mentioned in Section 9.1.

```
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, for example, that the unfolding of 01↺ defined in Section 8.1 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 : ∀ {i} → ≈^∞Tm (CListD ℕ) i tt 01··· (unfold 01↺)
eq-10 : ∀ {i} → ≈^∞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
```

10 Related work

10.1 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 (2002).

Our generic universe of syntax is based on scoped 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.

10.2 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.

Weirich, Yorgey, and Sheard's work (2011) encompassing a large array of patterns (nested, recursive, telescopic, and n-ary) can inform our design. They do not enforce scoping invariants internally which forces them to introduce separate constructors for a simple binder, a recursive one, or a telescopic pattern. They recover guarantees by giving their syntaxes a nominal semantics thus bolting down the precise meaning of each combinator and then proving that users may only generate well formed terms.

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 showcase the core ideas of their work using STLC extended with references and then demonstrate that they can already handle a large subset of Middleweight Java. We have demonstrated that our framework could be used to define effectful semantics by choosing an appropriate monad stack (Moggi (1991)). This suggests we should be able to model STLC+Ref. It is however clear that the scoping structures handled by scope graphs and frames are, in their full generality, out of reach for our framework. In constrast, our work shines by its generality: we define an entire universe of syntaxes and provide users with traversals and lemmas implemented *once and for all*.

Many other opportunities to enrich the notion of binder in our library are highlighted by Cheney (2005). As we have demonstrated in Sections 7.5 and 7.6 we can already handle let-bindings generically for all syntaxes. We are currently considering the modification of our system to handle deeply nested patterns by removing the constraint that the binders' and variables' sorts are identical. A notion of binding corresponding to hierarchical namespaces would be an exciting addition.

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.

10.3 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)). They worked 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, we are 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.

10.4 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 scope checker 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 these syntaxes' 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.

10.5 Universes of syntaxes with binding

Keeping in mind Altenkirch and McBride's observation that generic programming is everyday programming in dependently typed languages (2002), we can naturally expect generic, provably sound, treatments of these notions in tools such as Agda or Coq.

Keuchel (2011) together with Jeuring (2012) define a universe of syntaxes with binding with a rich notion of binding patterns closed under products but also sums as long as the disjoint patterns bind the same variables. They give their universe two distinct semantics: a first one based on well scoped de Bruijn indices and a second one based on Parametric Higher-Order Abstract Syntax (PHOAS) (Chlipala (2008)) together with a generic conversion function from the de Bruijn syntax to the PHOAS one. Following McBride's unpublished 2005 manuscript, which emerged as (Benton et al. (2012)), they implement both renaming and substitution in one fell swoop. They leave other opportunities for generic programming and proving to future work.

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 the first author's experience solving the POPLMark Reloaded challenge, Agda's pattern synonyms 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 type- and scope-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).

10.6 Fusion of successive traversals

The careful characterisation of the successive recursive traversals which can be fused together into a single pass in a semantics-preserving way is not new. This transformation is a much needed optimisation principle in a high-level functional language.

Through the careful study of the recursion operator associated to each strictly positive data type, Malcolm (1990) defined optimising fusion proof principles. Other optimisations such as deforestation (Wadler (1990)) or the compilation of a recursive definition into an equivalent abstract machine-based tail-recursive program (Cortiñas and Swierstra (2018)) rely on similar generic proofs that these transformations are meaning-preserving.

11 Conclusion and future work

Recalling our earlier work (2017) we have started from an example of a type- and scope-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 data type descriptions to support a notion of binding, we have given a generic presentation of syntaxes with binding. We then described a large class of type- and scope-safe generic programs acting on all of them. We started with syntactic traversals such as renaming and substitution. We then demonstrated how to write a small compiler pipeline: scope checking, type checking and elaboration to a core language, desugaring of new constructors added by a language transformer, dead code elimination and inlining, partial evaluation, and printing with names.

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 using them to solve the POPLMark Reloaded challenge (2019) 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 reused 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.

11.1 Limitations of the current framework

Although quite versatile already our current framework has some limitations which suggest avenues for future work. We list these limitations from easiest to hardest to resolve. Remember that each modification to the universe of syntaxes needs to be given an appropriate semantics.

Closure under products. Our current universe of descriptions is closed under sums as demonstrated in Section 5. It is however not closed under products: two arbitrary right-nested products conforming to a description may disagree on the sort of the term they are constructing. An approach where the sort is an input from which the description of allowed constructors is computed (à la Dagand (2013) where, for instance, the ‘lam constructor is only offered if the input sort is a function type) would not suffer from this limitation.

Unrestricted variables. Our current notion of variable can be used to form a term of any sort. We remarked in Sections 7.3 and 7.4 that in some languages we want to restrict this ability to one sort in particular. In that case, we wanted users to only be able to use variables at the sort Infer of our bidirectional language. For the time-being we made do by restricting the environment values our Semantics use to a subset of the sorts: terms with variables of the wrong sort will not be given a semantics.

Flat binding structure. Our current set-up limits us to flat binding structures: variables and binders share the same sorts. This prevents us from representing languages with binding patterns, for instance pattern-matching let-binders which can have arbitrarily nested patterns taking pairs apart.

Closure under derivation. One-hole contexts play a major role in the theory of programming languages. Just like the one-hole context of a data type is a data type (Abbott et al. (2005)), we would like our universe to be closed under derivatives so that the formalisation of, for example, evaluation contexts could benefit directly from the existing machinery.

Closure under closures. Jander's work on formalising and certifying continuation-passing style transformations (Jander (2019)) highlighted the need for a notion of syntaxes with closures. Recalling that our notion of Semantics is always compatible with precomposition with a renaming (Kaiser et al. (2018)) but not necessarily precomposition with a substitution (printing is, for instance, not stable under substitution), accommodating terms with suspended substitutions is a real challenge. Preliminary experiments show that a drastic modification of the type of the fundamental lemma of Semantics makes dealing with such closures possible. Whether the resulting traversal has good properties that can be proven generically is still an open problem.

11.2 Future work

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

- Our example of elaborating an enriched language to a core one, ACMM's implementation of a continuation-passing style conversion function, and Jander's work (2019) on the certification of a intrinsically typed CPS transformation raises the question of how many such common compilation passes can be implemented generically.
- Our universe only includes syntaxes that allow unrestricted variable use. Variables may be used multiple times or never, with no restriction. We are interested in representing syntaxes that only allow single use of variables, such as term calculi for linear logic (Benton et al. (1993); Barber (1996)), or that annotate variables with usage information (Brunel et al. (2014); Ghica and Smith (2014); Petricek et al. (2014); Atkey and Wood (2018)), or arrange variables into non-list-like structures such as bunches (O'Hearn (2003)), or arbitrary algebraic structures (Licata et al. (2017)), and in investigating what form a generic semantics for these syntaxes takes.
- An extension of Dagand and McBride's theory of ornaments (2014) could provide an appropriate framework to formalise and mechanise the connection between various languages, some being seen as refinements of others. This is particularly evident when considering the informative type checker (see the accompanying code) which given a scoped term produces a scoped-and-typed term by type checking or type inference.
- The first author's 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), allowing us to revisit previous work based on concrete instances of ACMM such as McLaughlin, McKinna and Stark (2018).

We now know how to generically describe syntaxes and their well behaved semantics. We can now 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?

Acknowledgements

Allais acknowledges the support of EPSRC grant no. EP/M016951/1 *Homotopy Type Theory: Programming and Verification*. Part of the research leading to these results has also received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nr. 320571.

Atkey acknowledges the support of EPSRC grant no. EP/T026960/1 *AISEC: AI Secure and Explainable by Construction*.

Chapman thanks IOHK for their support.

McKinna gratefully acknowledges the support of LFCS, School of Informatics, University of Edinburgh.

The authors would like to thank the ICFP and JFP reviewers for their useful comments, and the JFP editors and staff for their patience.

Conflicts of Interest

None.

References

Abbott, M. G., Altenkirch, T., McBride, C. & Ghani, N. (2005) δ for data: Differentiating data structures. *Fundamenta Informaticae*. **65**(1-2), 1–28.

Abel, A. (2010) MiniAgda: Integrating Sized and Dependent Types. Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010. pp. 14–28.

Abel, A., Allais, G., Hameer, A., Pientka, B., Momigliano, A., Schäfer, S. & Stark, K. (2019) POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*. **29**, e19.

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

Abel, A., Pientka, B., Thibodeau, D. & Setzer, A. (2013) Copatterns: programming infinite structures by observations. ACM SIGPLAN Notices. ACM. pp. 27–38.

Allais, G. (2018) agdarsec – Total parser combinators. JFLA 2018 Journées Francophones des Langages Applicatifs. Banyuls-sur-Mer, France. publié par les auteurs. pp. 45–59.

Allais, G., Chapman, J., McBride, C. & McKinna, J. (2017) Type-and-scope safe programs and their proofs. Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs. ACM. pp. 195–207.

Altenkirch, T., Chapman, J. & Uustalu, T. (2014) Relative monads formalised. *Journal of Formalized Reasoning*. **7**(1), 1–43.

Altenkirch, T., Chapman, J. & Uustalu, T. (2015) Monads need not be endofunctors. *Logical Methods in Computer Science*. **11**(1).

Altenkirch, T., Ghani, N., Hancock, P., McBride, C. & Morris, P. (2015) Indexed containers. *Journal of Functional Programming*. **25**, e5.

Altenkirch, T., Hofmann, M. & Streicher, T. (1995) Categorical reconstruction of a reduction free normalization proof. LNCS. Springer. pp. 182–199.

Altenkirch, T. & McBride, C. (2002) Generic programming within dependently typed programming. Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany. Kluwer. pp. 1–20.

Altenkirch, T. & Reus, B. (1999) Monadic presentations of lambda terms using generalized inductive types. CSL. Springer. pp. 453–468.

Appel, A. W. & Jim, T. (1997) Shrinking lambda expressions in linear time. *Journal of Functional Programming*. **7**(5), 515–540.

Atkey, R. (2015) An algebraic approach to typechecking and elaboration. `http://bentnib.org/posts/2015-04-19-algebraic-approach-typechecking-and-elaboration.html`.

Atkey, R. & Wood, J. (2018) Context constrained computation. 3rd Workshop on Type-Driven

Development (TyDe '18), Extended Abstract.
Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005) Mechanized Metatheory for the Masses: The POPLMark Challenge. Theorem Proving in Higher Order Logics. Springer. pp. 50–65.
Bach Poulsen, C., Rouvoet, A., Tolmach, A., Krebbers, R. & Visser, E. (2018) Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* **2**(POPL), 16:1–16:34.
Barber, A. (1996) Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347. LFCS, University of Edinburgh.
Bellegarde, F. & Hook, J. (1994) Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*. **23**(2), 287 – 311.
Benke, M., Dybjer, P. & Jansson, P. (2003) Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*. **10**(4), 265–289.
Benton, N., Hur, C.-K., Kennedy, A. J. & McBride, C. (2012) Strongly typed term representations in Coq. *Journal of Automated Reasoning*. **49**(2), 141–159.
Benton, P. N., Bierman, G. M., de Paiva, V. & Hyland, M. (1993) A term calculus for intuitionistic linear logic. Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings. Springer. pp. 75–90.
Berger, U. (1993) Program extraction from normalization proofs. In *TLCA*. Springer. pp. 91–106.
Berger, U. & Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed λ-calculus. LICS. IEEE. pp. 203–211.
Bird, R. S. & Paterson, R. (1999) De Bruijn notation as a nested datatype. *Journal of Functional Programming*. **9**(1), 77–91.
Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*. **23**(5), 552–593.
Brady, E. & Hammond, K. (2006) A verified staged interpreter is a verified compiler. Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM. pp. 111–120.
Brunel, A., Gaboardi, M., Mazza, D. & Zdancewic, S. (2014) A Core Quantitative Coeffect Calculus. Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014. pp. 351–370.
Chapman, J., Dagand, P.-E., McBride, C. & Morris, P. (2010) The gentle art of levitation. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ACM. pp. 3–14.
Chapman, J. M. (2009) *Type checking and normalisation*. Ph.D. thesis. University of Nottingham (UK).
Charguéraud, A. (2012) The locally nameless representation. *Journal of Automated Reasoning*. **49**(3), 363–408.
Cheney, J. (2005) Toward a general theory of names: binding and scope. ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2005, Tallinn, Estonia, September 30, 2005. ACM. pp. 33–40.
Chlipala, A. (2008) Parametric higher-order abstract syntax for mechanized semantics. Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008. ACM. pp. 143–156.
Copello, E. (2017) *On the Formalisation of the Metatheory of the Lambda Calculus and Languages with Binders*. Ph.D. thesis. Universidad de la República (Uruguay).
Coquand, C. (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), 57–90.
Coquand, T. & Dybjer, P. (1997) Intuitionistic model constructions and normalization proofs. *MSCS*. **7**(01), 75–94.
Cortiñas, C. T. & Swierstra, W. (2018) From algebra to abstract machine: a verified generic construction. Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018. ACM. pp. 78–90.

Dagand, P. (2013) *A cosmology of datatypes : reusability and dependent types*. Ph.D. thesis. University of Strathclyde, Glasgow, UK.

Dagand, P. & McBride, C. (2014) Transporting functions across ornaments. *Journal of Functional Programming*. **24**(2-3), 316–383.

Danielsson, N. A. (2010) Total parser combinators. Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. ACM. pp. 285–296.

de Bruijn, N. G. (1972) Lambda Calculus notation with nameless dummies. Indagationes Mathematicae. Elsevier. pp. 381–392.

de Moura, L. M., Kong, S., Avigad, J., van Doorn, F. & von Raumer, J. (2015) The Lean theorem prover (system description). Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Springer. pp. 378–388.

Dunfield, J. & Pfenning, F. (2004) Tridirectional typechecking. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM. pp. 281–292.

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

Dybjer, P. & Setzer, A. (1999) A finite axiomatization of inductive-recursive definitions. Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings. Springer. pp. 129–146.

Eisenberg, R. A. (2020) Stitch: the sound type-indexed type checker (Functional Pearl). Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020. ACM. pp. 39–53.

Érdi, G. (2018) Generic description of well-scoped, well-typed syntaxes. *CoRR*. **abs/1804.00119**.

Fiore, M. P., Plotkin, G. D. & Turi, D. (1999) Abstract syntax and variable binding. 14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999. IEEE Computer Society. pp. 193–202.

Gabbay, M. & Pitts, A. M. (2002) A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*. **13**(3-5), 341–363.

Ghani, N., Hamana, M., Uustalu, T. & Vene, V. (2006) Representing cyclic structures as nested datatypes. Proceedings of 7th Trends in Functional Programming, 2006. Intellect. pp. 173–188.

Ghica, D. R. & Smith, A. I. (2014) Bounded linear types in a resource semiring. Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014. pp. 331–350.

Gibbons, J. & d. S. Oliveira, B. C. (2009) The essence of the Iterator pattern. *Journal of Functional Programming*. **19**(3-4), 377–402.

Hamana, M. (2009) Initial algebra semantics for cyclic sharing structures. Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings. Springer. pp. 127–141.

Hatcliff, J. & Danvy, O. (1994) A generic account of continuation-passing styles. Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM. pp. 458–471.

Hedberg, M. (1998) A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*. **8**(4), 413–436.

Hinze, R. & Peyton Jones, S. L. (2000) Derivable type classes. *Electronic Notes in Theoretical Computer Science*. **41**(1), 5–35.

Hirschowitz, A. & Maggesi, M. (2012) Nested abstract syntax in Coq. *Journal of Automated Reasoning*. **49**(3), 409–426.

Hofmann, M. & Streicher, T. (1994) The groupoid model refutes uniqueness of identity proofs. Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994. IEEE Computer Society. pp. 208–212.

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

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

Jander, P. (2019) *Verifying Type-and-Scope Safe Program Transformations*. Master's thesis. University of Edinburgh.

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

Kaiser, J., Schäfer, S. & Stark, K. (2018) Binder aware recursion over well-scoped de Bruijn syntax. Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM. pp. 293–306.

Keep, A. W. & Dybvig, R. K. (2013) A nanopass framework for commercial compiler development. *SIGPLAN Not.* **48**(9), 343–350.

Keuchel, S. (2011) *Generic Programming With Binders and Scope*. Master's thesis. Utrecht University.

Keuchel, S. & Jeuring, J. (2012) Generic conversions of abstract syntax representations. Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2012, Copenhagen, Denmark, September 9-15, 2012. ACM. pp. 57–68.

Keuchel, S., Weirich, S. & Schrijvers, T. (2016) Needle & Knot: Binder boilerplate tied up. Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632. Springer-Verlag New York, Inc. pp. 419–445.

Lee, G., Oliveira, B. C. D. S., Cho, S. & Yi, K. (2012) GMeta: A generic formal metatheory framework for first-order representations. Programming Languages and Systems. Springer. pp. 436–455.

Licata, D. R., Shulman, M. & Riley, M. (2017) A fibrational framework for substructural and modal logics. 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. pp. 25:1–25:22.

Löh, A. & Magalhães, J. P. (2011) Generic programming with indexed functors. Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011. ACM. pp. 1–12.

Magalhães, J. P., Dijkstra, A., Jeuring, J. & Löh, A. (2010) A generic deriving mechanism for haskell. Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010. ACM. pp. 37–48.

Malcolm, G. (1990) Data structures and program transformation. *Science of Computer Programming*. **14**(2-3), 255–279.

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

The Coq Development Team. (2017) *The Coq proof assistant reference manual*. πr^2 Team. Version 8.6.

McBride, C. & McKinna, J. (2004) The view from the left. *Journal of Functional Programming*. **14**(1), 69–111.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *Journal of Functional Programming*. **18**(1), 1–13.

McLaughlin, C., McKinna, J. & Stark, I. (2018) Triangulating context lemmas. Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs. ACM. pp. 102–114.

Milner, R., Tofte, M. & Macqueen, D. (1997) *The Definition of Standard ML*. MIT Press. Cambridge, MA, USA.

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

Moggi, E. (1991) Notions of computation and monads. *Information and Computation*. **93**(1), 55–92.

Morris, P., Altenkirch, T. & McBride, C. (2006) Exploring the regular tree types. Types for Proofs and Programs. Springer. pp. 252–267.

Norell, U. (2009) Dependently typed programming in Agda. In *AFP Summer School*. Springer. pp. 230–266.

O'Hearn, P. W. (2003) On bunched typing. *Journal of Functional Programming*. **13**(4), 747–796.

Petricek, T., Orchard, D. A. & Mycroft, A. (2014) Coeffects: a calculus of context-dependent computation. Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. ACM. pp. 123–135.

Pfenning, F. (2004) Lecture 17: Bidirectional type checking. 15-312: Foundations of Programming Languages.

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

Polonowski, E. (2013) Automatically generated infrastructure for de Bruijn syntaxes. Interactive Theorem Proving. Springer. pp. 402–417.

Stump, A. (2016) *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool. New York, NY, USA.

Swiestra, W. (2008) Data types à la carte. *Journal of Functional Programming*. **18**(4), 423–436.

Thibodeau, D., Momigliano, A. & Pientka, B. (2016) A case-study in programming coinductive proofs: Howe's method. Technical report. Technical report, McGill University.

Wadler, P. (1987) Views: A way for pattern matching to cohabit with data abstraction. Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987. ACM Press. pp. 307–313.

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*. **73**(2), 231–248.

Wadler, P. & Kokke, W. (2018) *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk.

Weirich, S., Yorgey, B. A. & Sheard, T. (2011) Binders unbound. Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. ACM. pp. 333–345.