Type and Scope Preserving Semantics

Guillaume Allais, James Chapman, and Conor McBride

University of Strathclyde

Abstract. We introduce a notion of type and scope preserving semantics generalising Goguen and McKinna's "Candidates for Substitution" approach to defining one traversal generic enough to be instantiated to renaming first and then substitution. Its careful distinction of environment and model values as well as its variation on a structure typical of a Kripke semantics make it capable of expressing renaming and substitution but also various forms of Normalisation by Evaluation and, perhaps more surprisingly, monadic computations such as a printing function.
We then demonstrate that expressing these algorithms in a common framework yields immediate benefits: we can deploy some logical relations generically over these instances and obtain for instance the fusion lemmas for renaming, substitution and normalisation by evaluation as simple corollaries of the appropriate fundamental lemma. All of this work has been formalised in Agda.

Introduction

In order to implement an embedded Domain Specific Language (eDSL) [18], a developer can opt for either a shallow or a deep embedding [30,16]. In the shallow approach, she will use the host language's own types and term constructs to model the domain specific language's building blocks. This will allow her to rely on any and all of the host's libraries when writing programs in the eDSL. Should she decide to use a deep embedding, representing expressions directly as their abstract syntax tree will allow her to inspect, optimise, and compile terms as she sees fit. This ability to inspect the tree comes at the cost of having to reimplement basic notions such as renaming or substitution with the risk of introducing bugs. Trying to get the compiler to detect these bugs leads to a further distinction between different kinds of deep embeddings: she may either prove type and scope safety on paper and use an inductive *type* to describe an *untyped* syntax, follow Carette, Kiselyov, and Shan [6] and rely on parametric polymorphism to guarantee the existence of an underlying type and scope safe term, or use an inductive *family* to represent the term itself whilst enforcing these invariants in its indices.

Goguen and McKinna's Candidates for Substitution [17] begot work by McBride [23] in Epigram [24] and Benton, Hur, Kennedy and McBride [3] in Coq [22] showing how to alleviate the programmer's burden when she opts for the strongly-typed approach based on inductive families. They both define a traversal generic enough to be instantiated to renaming first and then substitution. In Benton et al., the bulk of the work has to be repeated when defining Normalisation by Evaluation. Reasoning about these definitions is still mostly done in an ad-hoc manner: Coq's tactics do help them to discharge the four fusion lemmas involving renaming and substitution, but the same work has to be repeated when studying the evaluation function. They choose to prove the evaluation

function correct by using propositional equality and assuming function extensionality rather than resorting to the traditional Partial Equivalence Relation approach we use.

We build on these insights and define an abstract notion of Semantics encompassing these two important operations as well as others Carette et al. could represent (e.g. measuring the size of a term) and even Normalisation by Evaluation [5]. By highlighting the common structure of all of these algorithms, we get the opportunity to not only implement them but also prove their properties generically.

Outline We shall start by defining the simple calculus we will use as a running example. We will then introduce a notion of environments as well as one well-known instance: the preorder of renamings. This will lead us to defining a generic notion of type and scope-preserving Semantics together with a generic evaluation function. We will then showcase the ground covered by these Semantics: from the syntactic ones corresponding to renaming and substitution to printing with names or some variations on Normalisation by Evaluation. Finally, we will demonstrate how, the definition of Semantics being generic enough, we can prove fundamental lemmas about these evaluation functions: we characterise the semantics which are synchronisable and give an abstract treatment of composition yielding compaction and reuse of proofs compared to Benton et al. [3]

Notations This article is a literate Agda file typeset using the LaTeX backend with as little post-processing as possible: we simply hide telescopes of implicit arguments as well as Set levels and properly display (super / sub)-scripts as well as special operators such as $\gg\!=$ or $+\!+$. As such, a lot of the notations have a meaning in Agda: green identifiers are data constructors, pink names refer to record fields, and blue is characteristic of defined symbols. Underscores have a special status: when defining mixfix identifiers [10], they mark positions where arguments may be inserted; our using the development version of Agda means that we have access to Haskell-style sections i.e. one may write $\_+\ 5$ for the partial application of $\_+\_$ corresponding to $\lambda\ x \to x + 5$ or, to mention something that we will use later on, $\mathsf{Renaming} \vDash [\![\_]\!]\ \_$ for the partial application of $\_\vDash [\![\_]\!]\ \_$ to Renaming.

Formalisation This whole development has been checked by Agda [28] which guarantees that all constructions are indeed well-typed, and all functions are total. Nonetheless, it should be noted that the generic model constructions and the various examples of Semantics given here can be fully replicated in Haskell using type families, higher rank polymorphism and generalised algebraic data types to build singletons [14] providing the user with the runtime descriptions of their types or their contexts' shapes. This yields, to the best of our knowledge, the first tagless and typeful implementation of Normalisation by Evaluation in Haskell. The subtleties of working with dependent types in Haskell [21] are outside the scope of this paper but we do provide a (commented) Haskell module containing all the translated definitions. It should be noted that Danvy, Keller and Puech have achieved a similar goal in OCaml [11] but their formalisation uses parametric higher order abstract syntax [7] which frees them from having to deal with variable binding, contexts and use models à la Kripke. However we consider these to be primordial: they can still guide the implementation of more complex type theories where, until now, being typeful is still out of reach. Type-level guarantees

about scope preservation can help root out bugs related to fresh name generation, name capture or arithmetic on de Bruijn levels to recover de Bruijn indices.

1 The Calculus

We are going to define and study various semantics for a simply-typed λ-calculus with `'Bool` and `'Unit` as base types. This serves as a minimal example of a system with a sum type and a record type equipped with an η-rule.

```
data ty : Set where
  'Unit  : ty
  'Bool  : ty
  _'→_   : (σ τ : ty) → ty
```

In order to be able to talk about the types of the variables in scope, we need a notion of contexts. We choose to represent them as snoc lists of types; ε denotes the empty context and $\Gamma \bullet \sigma$ the context Γ extended with a fresh variable of type σ. Variables are then positions in such a context represented as typed de Bruijn indices [12].

```
data Con : Set where
  ε    : Con
  _•_  : Con → ty → Con
```

```
data _∈_ (σ : ty) : Con → Set where
  zero  : σ ∈ (Γ • σ)
  1+_   : σ ∈ Γ → σ ∈ (Γ • τ)
```

The syntax for this calculus is designed to guarantee that terms are well-scoped and well-typed by construction. This presentation due to Altenkirch and Reus [2] relies heavily on Dybjer's inductive families [13]. Rather than having untyped pre-terms and a typing relation assigning a type to them, the typing rules are here enforced in the syntax: we can see for example that the `'var` constructor takes a typed de Bruijn index; that application (`_'$_`) ensures that the domain of the function coincides with the type of its argument; that the body of a λ-abstraction (`'λ`) is defined in a context extended with a fresh variable whose type corresponds to the domain of the function; or that the two branches of a conditional (`'ifte`) need to have the same type.

```
data _⊢_ (Γ : Con) : (σ : ty) → Set where
  'var    : (υ : σ ∈ Γ) → Γ ⊢ σ
  _'$_    : (t : Γ ⊢ (σ '→ τ)) (u : Γ ⊢ σ) → Γ ⊢ τ
  'λ      : (t : Γ • σ ⊢ τ) → Γ ⊢ (σ '→ τ)
  '⟨⟩     : Γ ⊢ 'Unit
  'tt 'ff : Γ ⊢ 'Bool
  'ifte   : (b : Γ ⊢ 'Bool) (l r : Γ ⊢ σ) → Γ ⊢ σ
```

2 A Generic Notion of Environment

All the semantics we are interested in defining associate to a term t of type $\Gamma \vdash \sigma$, a value of type $\mathscr{M}\ \Gamma\ \sigma$ given an interpretation $\mathscr{E}\ \Delta\ \tau$ for each one of its free variables τ

in Γ. We call the collection of these interpretations an $\mathscr{E}$-(evaluation) environment. We leave out $\mathscr{E}$ when it can easily be inferred from the context.

The content of environments may vary wildly between different semantics: when defining renaming, the environments will carry variables whilst the ones used for normalisation by evaluation contain elements of the model. But their structure stays the same which prompts us to define the notion generically. Formally, this translates to $\mathscr{E}$-environments being the pointwise lifting of the relation $\mathscr{E}$ between contexts and types to a relation between two contexts. Rather than using a datatype to represent such a lifting, we choose to use a function space. This decision is based on Jeffrey's observation that one can obtain associativity of append for free by using difference lists [20]. In our case the interplay between various combinators (e.g. refl and trans) defined later on is vastly simplified by this rather simple decision.

```
record _[_]_ (Δ : Con) (ℰ : Con → ty → Set) (Γ : Con) : Set where
  constructor pack
  field lookup : (v : σ ∈ Γ) → ℰ Δ σ
open _[_]_ public
```

For a fixed context Δ and relation $\mathscr{E}$, these environments can be built step by step by noticing that the environment corresponding to an empty context is trivial and that one may extend an already existing environment provided a proof of the right type. In concrete cases, there will be no sensible way to infer $\mathscr{E}$ when using the second combinator hence our decision to make it possible to tell Agda which relation we are working with.

```
`ε : Δ [ ℰ ] ε
`ε = pack $ λ ()

_`•_ : Δ [ ℰ ] Γ → ℰ Δ σ → Δ [ ℰ ] (Γ • σ)
lookup (ρ `• s) zero    = s
lookup (ρ `• s) (1+ n)  = lookup ρ n
```

The Preorder of Renamings A key instance of environments playing a predominant role in this paper is the notion of renaming. The reader may be accustomed to the more restrictive notion of context inclusions as described by Order Preserving Embeddings [1]. Writing non-injective or non-order preserving renamings would take perverse effort given that we only implement generic interpretations. In practice, the only combinators we use do guarantee that all the renamings we generate are context inclusions. As a consequence, we will use the two expressions interchangeably from now on.

A context inclusion $\Gamma \subseteq \Delta$ is an environment pairing each variable of type σ in Γ to one of the same type in Δ.

```
_⊆_ : (Γ Δ : Con) → Set
Γ ⊆ Δ = Δ [ flip _∈_ ] Γ
```

Context inclusions allow for the formulation of weakening principles explaining how to transport properties along inclusions. By a "weakening principle", we mean that if P holds of Γ and $\Gamma \subseteq \Delta$ then P holds for Δ too. In the case of variables, weakening merely corresponds to applying the renaming function in order to obtain a new variable. The

environments' case is also quite simple: being a pointwise lifting of a relation $\mathscr{E}$ between contexts and types, they enjoy weakening if $\mathscr{E}$ does.

```
wk∈ : Γ ⊆ Δ → σ ∈ Γ → σ ∈ Δ
wk∈ inc v = lookup inc v

wk[_] : (wk : (inc : Δ ⊆ Θ) → 𝓔 Δ σ → 𝓔 Θ σ)
        Δ ⊆ Θ → Δ [ 𝓔 ] Γ → Θ [ 𝓔 ] Γ
wk[ wk ] inc ρ = pack $ wk inc ∘ lookup ρ
```

These simple observations allow us to prove that context inclusions form a preorder which, in turn, lets us provide the user with the constructors Altenkirch, Hofmann and Streicher's "Category of Weakenings" [1] is based on.

```
refl : Γ ⊆ Γ
refl = pack id

trans : ty → Set} (inc₁ : Γ ⊆ Δ) (inc₂ : Θ [ 𝓔 ] Δ) → Θ [ 𝓔 ] Γ
lookup (trans inc₁ inc₂) = lookup inc₂ ∘ lookup inc₁
```

```
step : (inc : Γ ⊆ Δ) → Γ ⊆ (Δ • σ)
step inc = trans inc $ pack 1+_

pop! : (inc : Γ ⊆ Δ) → (Γ • σ) ⊆ (Δ • σ)
pop! inc = step inc `• zero
```

Now that we are equipped with the notion of inclusion, we have all the pieces necessary to describe the Kripke structure of our models of the simply-typed λ-calculus.

3 Semantics and Generic Evaluation Functions

The upcoming sections are dedicated to demonstrating that renaming, substitution, printing with names, and normalisation by evaluation all share the same structure. We start by abstracting away a notion of Semantics encompassing all these constructions. This approach will make it possible for us to implement a generic traversal parametrised by such a Semantics once and for all and to focus on the interesting model constructions instead of repeating the same pattern over and over again.

A Semantics is indexed by two relations $\mathscr{E}$ and $\mathscr{M}$ describing respectively the values in the environment and the ones in the model. In cases such as substitution or normalisation by evaluation, $\mathscr{E}$ and $\mathscr{M}$ will happen to coincide but keeping these two relations distinct is precisely what makes it possible to go beyond these and also model renaming or printing with names. The record packs the properties of these relations necessary to define the evaluation function.

```
record Semantics (𝓔 : Con → ty → Set) (𝓜 : Con → ty → Set) : Set where
```

The first two methods of a Semantics are dealing with environment values. These values need to come with a notion of weakening (wk) so that the traversal may introduce fresh variables when going under a binder and keep the environment well-scoped. We also need to be able to manufacture environment values given a variable in scope (embed) in order to be able to craft a diagonal environment to evaluate an open term.

```
  wk    : (inc : Γ ⊆ Δ) (r : 𝓔 Γ σ) → 𝓔 Δ σ
```

```
  embed : (v : σ ∈ Γ) → 𝓔 Γ σ
```

The structure of the model is quite constrained: each constructor in the language needs a semantic counterpart. We start with the two most interesting cases: ⟦var⟧ and ⟦λ⟧. The variable case corresponds to the intuition that the environment attaches interpretations to the variables in scope: it guarantees that one can turn a value from the environment into a model one. The traversal will therefore be able to, when hitting a variable, lookup the corresponding value in the environment and return it.

```
  ⟦var⟧ : (v : 𝓔 Γ σ) → 𝓜 Γ σ
```

The semantic λ-abstraction is notable for two reasons: first, following Mitchell and Moggi [26], its structure is typical of models à la Kripke allowing arbitrary extensions of the context; and second, instead of being a function in the host language taking values in the model as arguments, it is a function that takes *environment* values. Indeed, the body of a λ-abstraction exposes one extra free variable thus prompting us to extend the evaluation environment with an additional value. This slight variation in the type of semantic λ-abstraction guarantees that such an argument will be provided to us.

```
  ⟦λ⟧ : (t : (pr : Γ ⊆ Δ) (u : 𝓔 Δ σ) → 𝓜 Δ τ) → 𝓜 Γ (σ ' → τ)
```

The remaining fields' types are a direct translation of the types of the constructor they correspond to where the type constructor characterising typing derivations (_⊢_) has been replaced with the one corresponding to model values (𝓜).

```
  _⟦$⟧_ : 𝓜 Γ (σ ' → τ) → 𝓜 Γ σ → 𝓜 Γ τ
  ⟦⟨⟩⟧  : 𝓜 Γ 'Unit
  ⟦tt⟧  : 𝓜 Γ 'Bool
  ⟦ff⟧  : 𝓜 Γ 'Bool
  ⟦ifte⟧ : (b : 𝓜 Γ 'Bool) (l r : 𝓜 Γ σ) → 𝓜 Γ σ
```

The fundamental lemma of semantics is then proven in a module indexed by a Semantics, which would correspond to using a Section in Coq. It is defined by structural recursion on the term. Each constructor is replaced by its semantic counterpart in order to combine the induction hypotheses for its subterms. In the λ-abstraction case, the type of ⟦λ⟧ guarantees, in a fashion reminiscent of Normalisation by Evaluation, that the semantic argument can be stored in the environment which will have been weakened beforehand.

```
module Eval (𝓢 : Semantics 𝓔 𝓜) where
  open Semantics 𝓢
  lemma : (t : Γ ⊢ σ) (ρ : Δ [ 𝓔 ] Γ) → 𝓜 Δ σ
  lemma ('var v)     ρ = ⟦var⟧ $ lookup ρ v
  lemma (t '$ u)     ρ = lemma t ρ ⟦$⟧ lemma u ρ
  lemma ('λ t)       ρ = ⟦λ⟧ λ inc u → lemma t $ wk[ wk ] inc ρ '• u
  lemma '⟨⟩          ρ = ⟦⟨⟩⟧
```

```
lemma 'tt          ρ = ⟦tt⟧
lemma 'ff          ρ = ⟦ff⟧
lemma ('ifte b l r) ρ = ⟦ifte⟧ (lemma b ρ) (lemma l ρ) (lemma r ρ)
```

We introduce `_⊨⟦_⟧_` as an alternative name for the fundamental lemma and `_⊨eval_` for the special case where we use `embed` to generate a diagonal environment of type `Γ [ 𝒞 ] Γ`. We open the module `Eval` unapplied thus discharging (λ-lifting) its members over the `Semantics` parameter. This means that a partial application of `_⊨⟦_⟧_` will correspond to the specialisation of the fundamental lemma to a given semantics. `𝒮 ⊨⟦ t ⟧ ρ` is meant to convey the idea that the semantics `𝒮` is used to evaluate the term `t` in the environment `ρ`. Similarly, `𝒮 ⊨eval t` is meant to denote the evaluation of the term `t` in the semantics `𝒮` (using a diagonal environment).

```
_⊨⟦_⟧_ : Γ ⊢ σ → Δ [ 𝒞 ] Γ → 𝓜 Δ σ        _⊨eval_ : Γ ⊢ σ → 𝓜 Γ σ
_⊨⟦_⟧_ = lemma                            _⊨eval_ t = _⊨⟦_⟧_ t (pack embed)
```

The diagonal environment generated using `embed` when defining the `_⊨eval_` function lets us kickstart the evaluation of arbitrary *open* terms. In the case of printing with names, this corresponds to picking a naming scheme for free variables whilst in the usual model construction used to perform normalisation by evaluation, it corresponds to η-expanding the variables.

4 Syntax is the Identity Semantics

As we have explained earlier, this work has been directly influenced by McBride's manuscript [23]. It seems appropriate to start our exploration of `Semantics` with the two operations he implements as a single traversal. We call these operations syntactic because the values in the model are actual terms and almost all term constructors are kept as their own semantic counterpart. As observed by McBride, it is enough to provide three operations describing the properties of the values in the environment to get a full-blown `Semantics`. This fact is witnessed by our simple `Syntactic` record type together with the `syntactic` function turning its inhabitants into associated `Semantics`.

```
record Syntactic (𝒞 : (Γ : Con) (σ : ty) → Set) : Set where
  field embed : σ ∈ Γ → 𝒞 Γ σ
        wk    : Γ ⊆ Δ → 𝒞 Γ σ → 𝒞 Δ σ
        ⟦var⟧ : 𝒞 Γ σ → Γ ⊢ σ

syntactic : (syn : Syntactic 𝒞) → Semantics 𝒞 _⊢_
syntactic syn = let open Syntactic syn in record
  { wk     = wk; embed = embed; ⟦var⟧ = ⟦var⟧
  ; ⟦λ⟧    = λ t → 'λ $ t (step refl) $ embed zero
  ; _⟦$⟧_  = _'$_; ⟦⟨⟩⟧ = '⟨⟩; ⟦tt⟧ = 'tt; ⟦ff⟧ = 'ff; ⟦ifte⟧ = 'ifte }
```

The shape of `⟦λ⟧` or `⟦⟨⟩⟧` should not trick the reader into thinking that this definition performs some sort of η-expansion: `lemma` indeed only ever uses one of these when the

evaluated term's head constructor is already respectively a `λ or a `⟨⟩. It is therefore absolutely possible to define renaming or substitution using this approach. We can now port McBride's definitions to our framework.

Functoriality, also known as Renaming Our first example of a Syntactic operation works with variables as environment values. As a consequence, embedding is trivial; we have already defined weakening earlier (see Section 2) and we can turn a variable into a term by using the `var constructor.

```
syntacticRenaming : Syntactic (flip _∈_)
syntacticRenaming = record { embed = id; wk = wk∈; ⟦var⟧ = `var }

Renaming : Semantics (flip _∈_) _⊢_; Renaming = syntactic syntacticRenaming
```

We obtain a rather involved definition of the identity of type $\Gamma \vdash \sigma \to \Gamma \vdash \sigma$ as Renaming ⊨eval_. But this construction is not at all useless: indeed, the more general Renaming ⊨⟦_⟧_ function has type $\Gamma \vdash \sigma \to \Gamma \subseteq \Delta \to \Delta \vdash \sigma$ which turns out to be precisely the notion of weakening for terms we need once its arguments have been flipped.

```
wk⊢ : (inc : Γ ⊆ Δ) (t : Γ ⊢ σ) → Δ ⊢ σ
wk⊢ inc t = Renaming ⊨⟦ t ⟧ inc
```

Simultaneous Substitution Our second example of a semantics is another spin on the syntactic model: the environment values are now terms. We can embed variables into environment values by using the `var constructor and we inherit weakening for terms from the previous example.

```
syntacticSubstitution : Syntactic _⊢_
syntacticSubstitution = record { embed = `var; wk = wk⊢; ⟦var⟧ = id }

Substitution : Semantics _⊢_ _⊢_; Substitution = syntactic syntacticSubstitution
```

Because the diagonal environment used by Substitution ⊨eval_ is obtained by embedding membership proofs into terms using the `var constructor, we get yet another definition of the identity function on terms. The semantic function Substitution ⊨⟦_⟧_ is once again more interesting: it is an implementation of simultaneous substitution.

```
subst : (t : Γ ⊢ σ) (ρ : Δ [ _⊢_ ] Γ) → Δ ⊢ σ
subst t ρ = Substitution ⊨⟦ t ⟧ ρ
```

5 Printing with Names

Before considering the various model constructions involved in defining normalisation functions deciding different equational theories, let us make a detour to a perhaps slightly

more surprising example of a Semantics: printing with names. A user-facing project would naturally avoid directly building a String and rather construct an inhabitant of a more sophisticated datatype in order to generate a prettier output [19,31]. But we stick to the simpler setup as pretty printing is not our focus here.

This example is quite interesting for two reasons. Firstly, the distinction between the type of values in the environment and the ones in the model is once more instrumental in giving the procedure a precise type guiding our implementation. Indeed, the environment carries *names* for the variables currently in scope whilst the inhabitants of the model are *computations* threading a stream to be used as a source of fresh names every time a new variable is introduced by a λ-abstraction. If the values in the environment were allowed to be computations too, we would not root out all faulty implementations: the typechecker would for instance quite happily accept a program picking a new name every time a variable appears in the term.

```
record Name (Γ : Con) (σ : ty) : Set where
  constructor mkName
  field runName : String

record Printer (Γ : Con) (σ : ty) : Set where
  constructor mkPrinter
  field runPrinter : State (Stream String) String
```

Secondly, the fact that values in the model are computations and that this poses no problem whatsoever in this framework means it is appropriate for handling languages with effects [27], or effectful semantics e.g. logging the various function calls. Here is the full definition of the printer assuming the existence of formatλ, format$, and formatIf picking a way to display these constructors.

```
Printing : Semantics Name Printer
Printing = record
  { embed  = mkName ∘ show ∘ deBruijn
  ; wk     = λ _ → mkName ∘ runName
  ; ⟦var⟧  = mkPrinter ∘ return ∘ runName
  ; _⟦$⟧_  = λ mf mt → mkPrinter $ format$ <$> runPrinter mf ⊛ runPrinter mt
  ; ⟦λ⟧    = λ mb →
             mkPrinter $ get >>= λ names → let `x` = head names in
             put (tail names)                              >>= λ _ →
             runPrinter (mb (step refl) (mkName `x`))      >>= λ `b` →
             return $ formatλ `x` `b`
  ; ⟦⟨⟩⟧   = mkPrinter $ return "⟨⟩"
  ; ⟦tt⟧   = mkPrinter $ return "tt"
  ; ⟦ff⟧   = mkPrinter $ return "ff"
  ; ⟦ifte⟧ = λ mb ml mr → mkPrinter $
             formatIf <$> runPrinter mb ⊛ runPrinter ml ⊛ runPrinter mr }
```

Our definition of embed erases the membership proofs to recover the corresponding de Bruijn indices which are then turned into strings using show, defined in Agda's standard library. This means that, using Printing ⊨eval_, the free variables will be displayed as numbers whilst the bound ones will be given names taken from the name supply. This is quite clearly a rather crude name generation strategy and our approach to naming would naturally be more sophisticated in a user-facing language. We can for instance imagine that the binders arising from a user input would carry naming hints based on the name the user picked and that binders manufactured by the machine would be following a type-based scheme: functions would be fs or gs, natural numbers ms, ns, etc.

```
  where
    deBruijn : σ ∈ Γ → ℕ
    deBruijn zero    = 0
    deBruijn (1+ n)  = 1 + deBruijn n
```

We still need to provide a Stream of fresh names to this computation in order to run it. Given that embed erases free variables to numbers, we'd rather avoid using numbers if we want to avoid capture. We define names (not shown here) as the stream cycling through the letters of the alphabet and keeping the identifiers unique by appending a natural number incremented by 1 each time we are back to the beginning of the cycle.

Before defining print, we introduce nameContext (implementation omitted here) which is a function delivering a stateful computation using the provided stream of fresh names to generate an environment of names for a given context. This means that we are now able to define a printing function using names rather than numbers for the variables appearing free in a term.

```
nameContext : (Δ : Con) (Γ : Con) → State (Stream String) (Δ [ Name ] Γ)

print : (t : Γ ⊢ σ) → String
print t = proj₁ $ (nameContext Γ Γ >>= runPrinter ∘ λ ρ → Printing ⊨⟦ t ⟧ ρ) names
```

We can observe print's behaviour by writing a test. If we state this test as a propositional equality and prove it using refl, the typechecker will have to check that both expressions indeed compute to the same value. Here we display a term corresponding to the η-expansion of the first free variable in the context $\varepsilon \bullet (\sigma \text{ `→ } \tau)$. As we can see, it receives the name "a" whilst the binder introduced by the η-expansion is called "b".

```
pretty$ : print (`λ $ `var (1+ zero) `$ `var zero) ≡ "λb.a(b)"
pretty$ = PEq.refl
```

6 Normalisation by Evaluation

Normalisation by Evaluation is a technique exploiting the computational power of a host language in order to normalise expressions of a deeply embedded one. The process is based on a model construction describing a family of types $\mathscr{M}$ indexed by a context Γ

and a type σ. Two procedures are then defined: the first one (eval) constructs an element of $\mathscr{M}\ \Gamma\ \sigma$ provided a well-typed term of the corresponding $\Gamma \vdash \sigma$ type whilst the second one (reify) extracts, in a type-directed manner, normal forms $\Gamma \vdash^{\mathsf{nf}} \sigma$ from elements of the model $\mathscr{M}\ \Gamma\ \sigma$. Normalisation is achieved by composing the two procedures. The definition of this eval function is a natural candidate for our Semantics framework. Normalisation is always defined for a given equational theory so we are going to start by recalling the various rules a theory may satisfy.

Thanks to Renaming and Substitution respectively, we can formally define η-expansion and β-reduction. The η-rules are saying that for some types, terms have a canonical form: functions will all be λ-headed whilst record will be a collection of fields which translates here to all the elements of the `'Unit` type being equal to `'⟨⟩`.

```
eta : (t : Γ ⊢ σ '→ τ) → Γ ⊢ σ '→ τ
eta t = 'λ $ wk⊢ (step refl) t '$ 'var zero
```

$$\frac{}{t \rightsquigarrow \mathsf{eta}\ t}\,\eta_1 \qquad \frac{t : \Gamma \vdash \text{'Unit}}{t \rightsquigarrow \text{'}\langle\rangle}\,\eta_2$$

```
_⟨_/var₀⟩ : (t : Γ • σ ⊢ τ) (u : Γ ⊢ σ) → Γ ⊢ τ
t ⟨ u /var₀⟩ = subst t $ pack 'var '• u
```

$$\frac{}{(\text{'}\lambda\ t)\ \text{'\$}\ u \rightsquigarrow t\ \langle\ u\ /\mathsf{var}_0\rangle}\,\beta$$

The β-rule is the main driving force when it comes to actually computing but the presence of an inductive data type (`'Bool`) and its eliminator (`'ifte`) means we have an extra opportunity for redexes: whenever the boolean the eliminator is branching over is in canonical form, we may apply a ι-rule. Finally, the ξ-rule is the one making it possible to reduce under λ-abstractions which is the distinction between weak-head normalisation and strong normalisation.

$$\frac{}{\text{'ifte 'tt}\ l\ r \rightsquigarrow l}\,\iota_1 \qquad \frac{}{\text{'ifte 'ff}\ l\ r \rightsquigarrow r}\,\iota_2 \qquad \frac{t \rightsquigarrow u}{\text{'}\lambda\ t \rightsquigarrow \text{'}\lambda\ u}\,\xi$$

Now that we have recalled all these rules, we can talk precisely about the sort of equational theory decided by the model construction we choose to perform. We start with the usual definition of Normalisation by Evaluation which goes under λs and produces η-long βι-short normal forms.

6.1 Normalisation by Evaluation for βιξη

In the case of Normalisation by Evaluation, the elements of the model and the ones carried by the environment will both have the same type: $\_\vDash^{\beta\iota\xi\eta}\_$, defined by induction on its second argument. In order to formally describe this construction, we need to have a precise notion of normal forms. Indeed if the η-rules guarantee that we can represent functions (respectively inhabitants of `'Unit`) in the source language as function spaces (respectively `⊤`) in Agda, there are no such rules for `'Boolean` values which will be represented as normal forms of the right type i.e. as either `'tt`, `'ff` or a neutral expression.

These normal forms can be formally described by two mutually defined inductive families: $\_\vdash[\_]^{\mathsf{ne}}\_$ is the type of stuck terms made up of a variable to which a spine of eliminators in normal forms is applied; and $\_\vdash[\_]^{\mathsf{nf}}\_$ describes the normal forms.

These families are parametrised by a predicate R characterising the types at which the user is allowed to turn a neutral expression into a normal form as demonstrated by the constructor `'embed`'s first argument.

```
data _⊢[_]ne_ (Γ : Con) (R : ty → Set) (σ : ty) : Set where
  'var   : (v : σ ∈ Γ) → Γ ⊢[ R ]ne σ
  _'$_   : (t : Γ ⊢[ R ]ne τ '→ σ) (u : Γ ⊢[ R ]nf τ) → Γ ⊢[ R ]ne σ
  'ifte  : (b : Γ ⊢[ R ]ne 'Bool) (l r : Γ ⊢[ R ]nf σ) → Γ ⊢[ R ]ne σ

data _⊢[_]nf_ (Γ : Con) (R : ty → Set) : (σ : ty) → Set where
  'embed : (pr : R σ) (t : Γ ⊢[ R ]ne σ) → Γ ⊢[ R ]nf σ
  '⟨⟩    : Γ ⊢[ R ]nf 'Unit
  'tt    : Γ ⊢[ R ]nf 'Bool
  'ff    : Γ ⊢[ R ]nf 'Bool
  'λ     : (b : Γ • σ ⊢[ R ]nf τ) → Γ ⊢[ R ]nf (σ '→ τ)
```

Once more, context inclusions induce the expected notions of weakening wk^{ne} and wk^{nf}. We omit their purely structural implementation here and would thoroughly enjoy doing so in the source file too: our constructions so far have been syntax-directed and could hopefully be leveraged by a generic account of syntaxes with binding.

We now come to the definition of the model. We introduce the predicate $\mathsf{R}^{\beta\iota\xi\eta}$ characterising the types for which we may turn a neutral expression into a normal form. It is equivalent to the unit type $\top$ for `'Bool` and to the empty type $\bot$ otherwise. This effectively guarantees that we use the η-rules eagerly: all inhabitants of $\Gamma \vdash[\ \mathsf{R}^{\beta\iota\xi\eta}\ ]^{nf}$ `'Unit` and $\Gamma \vdash[\ \mathsf{R}^{\beta\iota\xi\eta}\ ]^{nf}$ $(\sigma$ `'→` $\tau)$ are equal to `'⟨⟩` and a `'λ`-headed term respectively.

The model construction then follows the usual pattern pioneered by Berger [4] and formally analysed and thoroughly explained by Catarina Coquand [8] in the case of a simply-typed lambda calculus with explicit substitutions. We proceed by induction on the type and make sure that η-expansion is applied eagerly: all inhabitants of $\Gamma \vDash^{\beta\iota\xi\eta}$ `'Unit` are indeed equal and all elements of $\Gamma \vDash^{\beta\iota\xi\eta}$ $(\sigma$ `'→` $\tau)$ are functions in Agda.

```
R^βιξη : ty → Set
R^βιξη 'Bool = ⊤
R^βιξη _     = ⊥

_⊨^βιξη_ : (Γ : Con) (σ : ty) → Set
Γ ⊨^βιξη 'Unit      = ⊤
Γ ⊨^βιξη 'Bool      = Γ ⊢[ R^βιξη ]nf 'Bool
Γ ⊨^βιξη (σ '→ τ)   = Γ ⊆ Δ → Δ ⊨^βιξη σ → Δ ⊨^βιξη τ
```

Normal forms may be weakened, and context inclusions may be composed hence the rather simple definition of weakening for inhabitants of the model.

```
wk^βιξη : (σ : ty) (inc : Γ ⊆ Δ) (T : Γ ⊨^βιξη σ) → Δ ⊨^βιξη σ
wk^βιξη 'Unit     inc T = T
wk^βιξη 'Bool     inc T = wk^nf inc T
wk^βιξη (σ '→ τ)  inc T = λ inc' → T $' trans inc inc'
```

The semantic counterpart of application combines two elements of the model: a functional part of type $\Gamma \vDash^{\beta\iota\xi\eta}$ $(\sigma$ `'→` $\tau)$ and its argument of type $\Gamma \vDash^{\beta\iota\xi\eta}$ σ which can

be fed to the functional given a proof that $\Gamma \subseteq \Gamma$. But we already have proven that _⊆_ is a preorder (see Section 2) so this is not at all an issue.

```
_$^βιξη_ : (t : Γ ⊨^βιξη σ `→ τ) (u : Γ ⊨^βιξη σ) → Γ ⊨^βιξη τ
t $^βιξη u = t refl u
```

Conditional Branching on the other hand is a bit more subtle: because the boolean value `ifte is branching over may be a neutral term, we are forced to define the reflection and reification mechanisms first. These functions, also known as unquote and quote respectively, are showing the interplay between neutral terms, model values and normal forms. reflect$^{\beta\iota\xi\eta}$ performs a form of semantical η-expansion: all stuck `Unit terms have the same image and all stuck functions are turned into functions in the host language.

```
var₀^βιξη : Γ • σ ⊨^βιξη σ
var₀^βιξη = reflect^βιξη _ $′ `var zero

reflect^βιξη : (σ : ty) (t : Γ ⊢[ R^βιξη ]^ne σ) → Γ ⊨^βιξη σ
reflect^βιξη `Unit      t = ⟨⟩
reflect^βιξη `Bool      t = `embed _ t
reflect^βιξη (σ `→ τ)  t = λ inc u → reflect^βιξη τ $′ wk^ne inc t `$ reify^βιξη σ u

reify^βιξη : (σ : ty) (T : Γ ⊨^βιξη σ) → Γ ⊢[ R^βιξη ]^nf σ
reify^βιξη `Unit      T = `⟨⟩
reify^βιξη `Bool      T = T
reify^βιξη (σ `→ τ)  T = `λ $′ reify^βιξη τ $′ T (step refl) var₀^βιξη
```

The semantic counterpart of `ifte can then be defined: if the boolean is a value, the appropriate branch is picked; if it is stuck the whole expression is reflected in the model.

```
ifte^βιξη : (b : Γ ⊨^βιξη `Bool) (l r : Γ ⊨^βιξη σ) → Γ ⊨^βιξη σ
ifte^βιξη `tt          l r = l
ifte^βιξη `ff          l r = r
ifte^βιξη (`embed _ T) l r = reflect^βιξη _ $′ `ifte T (reify^βιξη _ l) (reify^βιξη _ r)
```

The Semantics corresponding to Normalisation by Evaluation for βιξη-rules uses _⊨^βιξη_ for values in the environment as well as the ones in the model. The semantic counterpart of a λ-abstraction is simply the identity: the structure of the functional case in the definition of the model matches precisely the shape expected in a Semantics. Because the environment carries model values, the variable case is trivial.

```
Normalise^βιξη : Semantics _⊨^βιξη_ _⊨^βιξη_
Normalise^βιξη = record
  { embed = reflect^βιξη _ ∘ `var; wk = wk^βιξη _; ⟦var⟧ = id
  ; _⟦$⟧_ = _$^βιξη_; ⟦λ⟧ = id
  ; ⟦⟨⟩⟧ = ⟨⟩; ⟦tt⟧ = `tt; ⟦ff⟧ = `ff; ⟦ifte⟧ = ifte^βιξη }
```

The diagonal environment built up in $\mathsf{Normalise}^{\beta\iota\xi\eta}\ \vDash\mathsf{eval\_}$ consists of η-expanded variables. Normalisation is obtained by reifying the result of evaluation.

$$\mathsf{norm}^{\beta\iota\xi\eta} : (\sigma : \mathsf{ty})\ (t : \Gamma \vdash \sigma) \to \Gamma \vdash[\ \mathsf{R}^{\beta\iota\xi\eta}\ ]^{\mathsf{nf}}\ \sigma$$
$$\mathsf{norm}^{\beta\iota\xi\eta}\ \sigma\ t = \mathsf{reify}^{\beta\iota\xi\eta}\ \sigma\ \$'\ \mathsf{Normalise}^{\beta\iota\xi\eta}\ \vDash\mathsf{eval}\ t$$

6.2 Normalisation by Evaluation for βιξ

As we have just seen, the traditional typed model construction leads to a normalisation procedure outputting βι-normal η-long terms. However evaluation strategies implemented in actual proof systems tend to avoid applying η-rules as much as possible: unsurprisingly, it is a rather bad idea to η-expand proof terms which are already large when typechecking complex developments. Garillot and colleagues [15] report that common mathematical structures packaged in records can lead to terms of such a size that theorem proving becomes impractical.

In these systems, normal forms are neither η-long nor η-short: the η-rule is actually never considered except when comparing two terms for equality, one of which is neutral whilst the other is constructor-headed. Instead of declaring them distinct, the algorithm will perform one step of η-expansion on the neutral term and compare their subterms structurally. The conversion test will only fail when confronted with two neutral terms with distinct head variables or two normal forms with different head constructors.

To reproduce this behaviour, the normalisation procedure needs to be amended. It is possible to alter the model definition described earlier so that it avoids unnecessary η-expansions. We proceed by enriching the traditional model with extra syntactical artefacts in a manner reminiscent of Coquand and Dybjer's approach to defining a Normalisation by Evaluation procedure for the SK combinator calculus [9]. Their resorting to glueing terms to elements of the model was dictated by the sheer impossibily to write a sensible reification procedure but, in hindsight, it provides us with a powerful technique to build models internalizing alternative equational theories. This leads us to mutually defining the model ($\_\vDash^{\beta\iota\xi}\_$) together with the *acting* model ($\_\vDash^{\beta\iota\xi\star}\_$):

$$\begin{aligned}
&_\vDash^{\beta\iota\xi}_ &&: (\Gamma : \mathsf{Con})\ (\sigma : \mathsf{ty}) \to \mathsf{Set}\\
&\Gamma \vDash^{\beta\iota\xi} \sigma &&= \Gamma \vdash[\ \mathsf{R}^{\beta\iota\xi}\ ]^{\mathsf{ne}}\ \sigma \uplus \Gamma \vDash^{\beta\iota\xi\star} \sigma
\end{aligned}$$

$$\begin{aligned}
&_\vDash^{\beta\iota\xi\star}_ &&: (\Gamma : \mathsf{Con})\ (\sigma : \mathsf{ty}) \to \mathsf{Set}\\
&\Gamma \vDash^{\beta\iota\xi\star} \text{`}\mathsf{Unit} &&= \top\\
&\Gamma \vDash^{\beta\iota\xi\star} \text{`}\mathsf{Bool} &&= \mathsf{Bool}\\
&\Gamma \vDash^{\beta\iota\xi\star} (\sigma \text{`}\to \tau) &&= \Gamma \subseteq \Delta \to \Delta \vDash^{\beta\iota\xi} \sigma \to \Delta \vDash^{\beta\iota\xi} \tau
\end{aligned}$$

These mutual definitions allow us to make a careful distinction between values arising from (non expanded) stuck terms and the ones wich are constructor headed and have a computational behaviour associated to them. The values in the acting model are storing these behaviours be it either actual proofs of $\top$, actual Booleans or actual Agda functions depending on the type of the term. It is important to note that the functions in the acting

model have the model as both domain and codomain: there is no reason to exclude the fact that both the argument or the body may or may not be stuck.

Weakening for these structures is rather straightforward albeit slightly more complex than for the usual definition of Normalisation by Evaluation seen in Section 6.1.

```
wk^βιξ⋆ : (inc : Γ ⊆ Δ) (T : Γ ⊨^βιξ⋆ σ) → Δ ⊨^βιξ⋆ σ
wk^βιξ⋆ inc {`Unit    } T = T
wk^βιξ⋆ inc {`Bool    } T = T
wk^βιξ⋆ inc {σ `→ τ   } T = λ inc′ → T $′ trans inc inc′

wk^βιξ : (inc : Γ ⊆ Δ) (T : Γ ⊨^βιξ σ) → Δ ⊨^βιξ σ
wk^βιξ inc (inj₁ ne)  = inj₁ $ wk^ne inc ne
wk^βιξ inc (inj₂ T)   = inj₂ $ wk^βιξ⋆ inc T
```

What used to be called reflection in the previous model is now trivial: stuck terms are indeed perfectly valid model values. Reification becomes quite straightforward too because no η-expansion is needed. When facing a stuck term, we simply embed it in the set of normal forms. Even though reify^βιξ⋆ may look like it is performing some η-expansions, it is not the case: all the values in the acting model are notionally obtained from constructor-headed terms.

```
reflect^βιξ : (σ : ty) (t : Γ ⊢[ R^βιξ ]^ne σ) → Γ ⊨^βιξ σ
reflect^βιξ σ = inj₁

  reify^βιξ⋆ : (σ : ty) (T : Γ ⊨^βιξ⋆ σ) → Γ ⊢[ R^βιξ ]^nf σ
  reify^βιξ⋆ `Unit      T = `⟨⟩
  reify^βιξ⋆ `Bool      T = if T then `tt else `ff
  reify^βιξ⋆ (σ `→ τ)   T = `λ $′ reify^βιξ τ $′ T (step refl) var₀^βιξ
    where var₀^βιξ = inj₁ $ `var zero

  reify^βιξ : (σ : ty) (T : Γ ⊨^βιξ σ) → Γ ⊢[ R^βιξ ]^nf σ
  reify^βιξ σ (inj₁ ne)  = `embed _ ne
  reify^βιξ σ (inj₂ T)   = reify^βιξ⋆ σ T
```

Semantic application is slightly more interesting: we have to dispatch depending on whether the function is a stuck term or not. In case it is, we can reify its argument and grow the spine of the stuck term. Otherwise we have an Agda function ready to be applied. We proceed similarly for the definition of the semantical "if then else".

```
_$^βιξ_ : (t : Γ ⊨^βιξ (σ `→ τ)) (u : Γ ⊨^βιξ σ) → Γ ⊨^βιξ τ
(inj₁ ne)  $^βιξ u = inj₁ $ ne `$ reify^βιξ _ u
(inj₂ F)   $^βιξ u = F refl u

ifte^βιξ : (b : Γ ⊨^βιξ `Bool) (l r : Γ ⊨^βιξ σ) → Γ ⊨^βιξ σ
ifte^βιξ (inj₁ ne) l r = inj₁ $ `ifte ne (reify^βιξ _ l) (reify^βιξ _ r)
```

```
ifte^βιξ (inj₂ T) l r = if T then l else r
```

Finally, we have all the necessary components to show that evaluating the term whilst not η-expanding all stuck terms is a perfectly valid Semantics. As usual, normalisation is defined by composing reification and evaluation on the diagonal environment.

```
Normalise^βιξ : Semantics _⊢^βιξ_ _⊨^βιξ_
Normalise^βιξ = record
  { embed = reflect^βιξ _ ∘ 'var; wk = wk^βιξ; [[var]]  = id
  ; _[[$]]_ = _$^βιξ_; [[λ]] = inj₂
  ; [[⟨⟩]] = inj₂ ⟨⟩; [[tt]] = inj₂ true; [[ff]] = inj₂ false; [[ifte]]  = ifte^βιξ }

norm^βιξ : (σ : ty) (t : Γ ⊢ σ) → Γ ⊢[ R^βιξ ]^nf σ
norm^βιξ σ t = reify^βιξ σ $′ Normalise^βιξ ⊨eval t
```

6.3 Normalisation by Evaluation for βι

The decision to lazily apply the η-rule can be pushed even further: one may forgo using the ξ-rule too and simply perform weak-head normalisation. This leads to pursuing the computation only when absolutely necessary e.g. when two terms compared for equality have matching head constructors and one needs to inspect these constructors' arguments to conclude. For that purpose, we introduce an inductive family describing terms in weak-head normal forms. Naturally, it is possible to define the corresponding weakenings wk^whne and wk^whnf as well as erasure functions erase^whnf and erase^whne with codomain _⊢_ (we omit their simple definitions here).

```
data _⊢^whne_ (Γ : Con) (σ : ty) : Set where
  'var  : (v : σ ∈ Γ) → Γ ⊢^whne σ
  _'$_  : (t : Γ ⊢^whne (τ '→ σ)) (u : Γ ⊢ τ) → Γ ⊢^whne σ
  'ifte : (b : Γ ⊢^whne 'Bool) (l r : Γ ⊢ σ) → Γ ⊢^whne σ

data _⊢^whnf_ (Γ : Con) : (σ : ty) → Set where
  'embed : (t : Γ ⊢^whne σ) → Γ ⊢^whnf σ
  '⟨⟩    : Γ ⊢^whnf 'Unit
  'tt 'ff : Γ ⊢^whnf 'Bool
  'λ     : (b : Γ • σ ⊢ τ) → Γ ⊢^whnf (σ '→ τ)
```

The model construction is quite similar to the previous one except that source terms are now stored in the model too. This means that from an element of the model, one can pick either the reduced version of the original term (i.e. a stuck term or the term's computational content) or the original term itself. We exploit this ability most notably at reification time where once we have obtained either a head constructor (respectively a head variable), none of the subterms need to be evaluated.

```
_⊨^βι_ : (Γ : Con) (σ : ty) → Set
```

```
Γ ⊨^βι σ  = Γ ⊢ σ  × (Γ ⊢^whne σ ⊎ Γ ⊨^βι⋆ σ)

_⊨^βι⋆_ : (Γ : Con) (σ : ty) → Set
Γ ⊨^βι⋆ 'Unit      = ⊤
Γ ⊨^βι⋆ 'Bool      = Bool
Γ ⊨^βι⋆ (σ '→ τ)  = Γ ⊆ Δ → Δ ⊨^βι σ → Δ ⊨^βι τ
```

Weakening, reflection, and reification can all be defined rather straightforwardly based on the template provided by the previous section. The application and conditional branching rules are more interesting: one important difference with respect to the previous subsection is that we do not grow the spine of a stuck term using reified versions of its arguments but rather the corresponding *source* term thus staying true to the idea that we only head reduce enough to expose either a constructor or a variable.

```
_$^βι_ : (t : Γ ⊨^βι σ '→ τ) (u : Γ ⊨^βι σ) → Γ ⊨^βι τ
(t , inj₁ ne)  $^βι (u , U) = t '$ u , inj₁ (ne '$ u)
(t , inj₂ T)   $^βι (u , U) = t '$ u , proj₂ (T refl (u , U))

ifte^βι : (b : Γ ⊨^βι 'Bool) (l r : Γ ⊨^βι σ) → Γ ⊨^βι σ
ifte^βι (b , inj₁ ne)  (l , L) (r , R) = 'ifte b l r , inj₁ ('ifte ne l r)
ifte^βι (b , inj₂ B)   (l , L) (r , R) = 'ifte b l r , (if B then L else R)
```

We can finally put together all of these semantic counterpart to obtain a Semantics corresponding to weak-head normalisation. We omit the now self-evident definition of $\mathsf{norm}^{\beta\iota}$ as the composition of evaluation and reification.

```
Normalise^βι : Semantics _⊨^βι_ _⊨^βι_
Normalise^βι = record
  { embed = reflect^βι _ ∘ 'var; wk = wk^βι; ⟦var⟧ = id
  ; _⟦$⟧_ = _$^βι_; ⟦λ⟧ = λ t → 'λ (proj₁ $ t (step refl) (reflect^βι _ $ 'var zero)) , inj₂ t
  ; ⟦⟨⟩⟧ = '⟨⟩ , inj₂ ⟨⟩; ⟦tt⟧ = 'tt  , inj₂ true; ⟦ff⟧ = 'ff  , inj₂ false; ⟦ifte⟧  = ifte^βι }
```

7 Proving Properties of Semantics

Thanks to the introduction of Semantics, we have already saved quite a bit of work by not reimplementing the same traversals over and over again. But this disciplined approach to building models and defining the associated evaluation functions can also help us refactor the process of proving some properties of these semantics.

Instead of using proof scripts as Benton et al. [3] do, we describe abstractly the constraints the logical relations [29] defined on model (and environment) values have to respect for us to be able to conclude that the evaluation of a term in related environments produces related outputs. This gives us a generic proof framework to state and prove, in one go, properties about all of these semantics.

Our first example of such a framework will stay simple on purpose. However this does not entail that it is a meaningless exercise: the result proven here will actually be useful in the following subsections when considering more complex properties.

7.1 The Synchronisation Relation

This first example is basically describing the relational interpretation of the terms. It should give the reader a good idea of the structure of this type of setup before we move on to a more complex one. The types involved might look a bit scary because of the level of generality that we adopt but the idea is rather simple: two Semantics are said to be *synchronisable* if, when evaluating a term in related environments, they output related values. The bulk of the work is to make this intuition formal.

The evidence that two Semantics are Synchronisable is packaged in a record. The record is indexed by the two semantics as well as two relations. The first relation ($\mathscr{E}^R$) characterises the elements of the (respective) environment types which are to be considered synchronised, and the second one ($\mathscr{M}^R$) describes what synchronisation means in the model. We can lift $\mathscr{E}^R$ in a pointwise manner to talk about entire environments using the `'∀[_, _]` predicate transformer omitted here.

```
record Synchronisable (𝒮ᴬ : Semantics ℰᴬ ℳᴬ) (𝒮ᴮ : Semantics ℰᴮ ℳᴮ)
  (ℰᴿ  : ℰᴬ Γ σ → ℰᴮ Γ σ → Set)
  (ℳᴿ  : ℳᴬ Γ σ → ℳᴮ Γ σ → Set) : Set where
```

The record's fields are describing the structure these relations need to have. $\mathscr{E}^R_{wk}$ states that two synchronised environments can be weakened whilst staying synchronised.

```
  ℰᴿwk : (inc : Δ ⊆ Θ) (ρᴿ : '∀[ ℰᴿ ] ρᴬ ρᴮ) →
         '∀[ ℰᴿ ] (wk[ 𝒮ᴬ.wk ] inc ρᴬ) (wk[ 𝒮ᴮ.wk ] inc ρᴮ)
```

We then have the relational counterparts of the term constructors. To lighten the presentation, we will focus on the most interesting ones and give only one example quite characteristic of the remaining ones. Our first interesting case is the relational counterpart of `'var`: it states that given two synchronised environments, we indeed get synchronised values in the model by applying ⟦var⟧ to the looked up values.

```
  R⟦var⟧ : (v : σ ∈ Γ) (ρᴿ : '∀[ ℰᴿ ] ρᴬ ρᴮ) →
           ℳᴿ (𝒮ᴬ.⟦var⟧ (lookup ρᴬ v)) (𝒮ᴮ.⟦var⟧ (lookup ρᴮ v))
```

The second, and probably most interesting case, is the relational counterpart to the ⟦λ⟧ combinator. The ability to evaluate the body of a `'λ` in weakened environments, each extended by related values, and deliver synchronised values is enough to guarantee that evaluating the lambdas in the original environments will produce synchronised values.

```
  R⟦λ⟧ : (fʳ : (inc : Γ ⊆ Δ) (uᴿ : ℰᴿ uᴬ uᴮ) → ℳᴿ (fᴬ inc uᴬ) (fᴮ inc uᴮ)) →
         ℳᴿ (𝒮ᴬ.⟦λ⟧ fᴬ) (𝒮ᴮ.⟦λ⟧ fᴮ)
```

All the remaining cases are similar. We show here the relational counterpart of the application constructor: it states that given two induction hypotheses (and the knowledge that the two environment used are synchronised), one can combine them to obtain a proof about the evaluation of an application-headed term.

```
R⟦$⟧ : 𝓜ᴿ fᴬ fᴮ → 𝓜ᴿ uᴬ uᴮ → 𝓜ᴿ (fᴬ 𝓢ᴬ.⟦$⟧ uᴬ) (fᴮ 𝓢ᴮ.⟦$⟧ uᴮ)
```

For this specification to be useful, we need to verify that we can indeed benefit from its introduction. This is witnessed by two facts. First, our ability to prove a fundamental lemma stating that given relations satisfying this specification, the evaluation of a term in related environments yields related values; second, our ability to find with various instances of such synchronised semantics. Let us start with the fundamental lemma.

Fundamental Lemma of Synchronisable Semantics The fundamental lemma is indeed provable. We introduce a Synchronised module parametrised by a record packing the evidence that two semantics are Synchronisable. This allows us to bring all of the corresponding relational counterpart of term constructors into scope by opening the record. The traversal then uses them to combine the induction hypotheses arising structurally. We use `[_,_,_]_•ᴿ_` as a way to circumvent Agda's inhability to infer $\mathscr{E}^A$, $\mathscr{E}^B$ and $\mathscr{E}^R$.

```
_•ᴿ_ : '∀[ 𝓔ᴿ ] ρᴬ ρᴮ → 𝓔ᴿ uᴬ uᴮ → '∀[ 𝓔ᴿ ] (ρᴬ '• uᴬ) (ρᴮ '• uᴮ)
lookupᴿ (ρᴿ •ᴿ uᴿ) zero    = uᴿ
lookupᴿ (ρᴿ •ᴿ uᴿ) (1+ v)  = lookupᴿ ρᴿ v

module Synchronised (ℛ : Synchronisable 𝓢ᴬ 𝓢ᴮ 𝓔ᴿ 𝓜ᴿ) where
  open Synchronisable ℛ
  lemma : (t : Γ ⊢ σ) (ρᴿ : '∀[ 𝓔ᴿ ] ρᴬ ρᴮ) →
          𝓜ᴿ (𝓢ᴬ ⊨⟦ t ⟧ ρᴬ) (𝓢ᴮ ⊨⟦ t ⟧ ρᴮ)
  lemma ('var v)      ρᴿ = R⟦var⟧ v ρᴿ
  lemma (f '$ t)      ρᴿ = R⟦$⟧ (lemma f ρᴿ) (lemma t ρᴿ)
  lemma ('λ t)        ρᴿ = R⟦λ⟧ (λ inc uᴿ → lemma t $ 𝓔ᴿwk inc ρᴿ •ᴿ uᴿ)
  lemma '⟨⟩           ρᴿ = R⟦⟨⟩⟧
  lemma 'tt           ρᴿ = R⟦tt⟧
  lemma 'ff           ρᴿ = R⟦ff⟧
  lemma ('ifte b l r) ρᴿ = R⟦ifte⟧ (lemma b ρᴿ) (lemma l ρᴿ) (lemma r ρᴿ)
```

Examples of Synchronisable Semantics Our first example of two synchronisable semantics is proving the fact that Renaming and Substitution have precisely the same behaviour whenever the environment we use for Substitution is only made up of variables. The (mundane) proofs which mostly consist of using the congruence of propositional equality are left out.

```
SynchronisableRenamingSubstitution : Synchronisable Renaming Substitution
                                     (λ v t → 'var v ≡ t) _≡_
```

We show with the lemma RenamingIsASubstitution how the result we meant to prove is derived directly from the fundamental lemma of Synchronisable semantics:

```
RenamingIsASubstitution : (t : Γ ⊢ σ) (ρ : Γ ⊆ Δ) →
  Renaming ⊨⟦ t ⟧ ρ ≡ Substitution ⊨⟦ t ⟧ trans ρ (pack 'var)
RenamingIsASubstitution t ρ = lemma t (pack^R $ λ _ → PEq.refl)
  where open Synchronised SynchronisableRenamingSubstitution
```

Another example of a synchronisable semantics is Normalisation by Evaluation which can be synchronised with itself. This may appear like mindless symbol pushing but it is actually crucial to prove such a theorem: we can only define a Partial Equivalence Relation [25] (PER) on the model used to implement Normalisation by Evaluation. The proofs of the more complex properties of the procedure will rely heavily on the fact that the exotic elements that may exist in the host language are actually never produced by the evaluation function run on a term as long as all the elements of the environment used were, themselves, not exotic i.e. equal to themselves according to the PER.

We start with the definition of the PER for the model. It is constructed by induction on the type and ensures that terms which behave the same extensionally are declared equal. Two values of type 'Unit are always trivially equal; values of type 'Bool are normal forms and are declared equal when they are effectively syntactically the same; finally functions are equal whenever given equal inputs they yield equal outputs.

```
EQREL : (Γ : Con) (σ : ty) (T U : Γ ⊨^βιξη σ) → Set
EQREL Γ 'Unit      T U = ⊤
EQREL Γ 'Bool      T U = T ≡ U
EQREL Γ (σ '→ τ)  T U = (inc : Γ ⊆ Δ) (eqV W : EQREL Δ σ V W) →
                        EQREL Δ τ (T inc V) (U inc W)
```

It is indeed a PER as witnessed by the (omitted here) symEQREL and transEQREL functions and it respects weakening as wk^EQREL shows.

```
symEQREL : EQREL Γ σ S T → EQREL Γ σ T S
transEQREL : EQREL Γ σ S T → EQREL Γ σ T U → EQREL Γ σ S U
wk^EQREL : EQREL Γ σ T U → EQREL Δ σ (wk^βιξη σ inc T) (wk^βιξη σ inc U)
```

The interplay of reflect and reify with this notion of equality has to be described in one go because of their being mutually defined. It confirms our claim that EQREL is indeed an appropriate notion of semantic equality: values related by EQREL are reified to propositionally equal normal forms whilst propositionally equal neutral terms are reflected to values related by EQREL.

```
reify^EQREL   : EQREL Γ σ T U → reify^βιξη σ T ≡ reify^βιξη σ U
reflect^EQREL : t ≡ u → EQREL Γ σ (reflect^βιξη σ t) (reflect^βιξη σ u)
```

And that's enough to prove that evaluating a term in two environments related in a pointwise manner by EQREL yields two semantic objects themselves related by EQREL.

```
SynchronisableNormalise : Synchronisable Normalise^βιξη Normalise^βιξη
                          (EQREL _ _) (EQREL _ _)
```

We omit the details of the easy proof but still recall the type of the corollary of the fundamental lemma one obtains in this case:

```
reflˡ^βιξη : (t : Γ ⊢ σ) (ρ^R : `∀[ EQREL _ _ ] ρ^A ρ^B) →
             EQREL Δ σ (Normalise^βιξη ⊨⟦ t ⟧ ρ^A) (Normalise^βιξη ⊨⟦ t ⟧ ρ^B)
reflˡ^βιξη t ρ^R = lemma t ρ^R where open Synchronised SynchronisableNormalise
```

We can now move on to the more complex example of a proof framework built generically over our notion of Semantics

7.2 Fusions of Evaluations

When studying the meta-theory of a calculus, one systematically needs to prove fusion lemmas for various semantics. For instance, Benton et al. [3] prove six such lemmas relating renaming, substitution and a typeful semantics embedding their calculus into Coq. This observation naturally led us to defining a fusion framework describing how to relate three semantics: the pair we want to run sequentially and the third one they correspond to. The fundamental lemma we prove can then be instantiated six times to derive the corresponding corollaries.

The evidence that $\mathcal{S}^A$, $\mathcal{S}^B$ and $\mathcal{S}^C$ are such that $\mathcal{S}^A$ followed by $\mathcal{S}^B$ can be said to be equivalent to $\mathcal{S}^C$ (e.g. think Substitution followed by Renaming can be reduced to Substitution) is packed in a record Fusable indexed by the three semantics but also three relations. The first one ($\mathcal{E}^R_{BC}$) states what it means for two environment values of $\mathcal{S}^B$ and $\mathcal{S}^C$ respectively to be related. The second one ($\mathcal{E}^R$) characterises the triples of environments (one for each one of the semantics) which are compatible. Finally, the last one ($\mathcal{M}^R$) relates values in $\mathcal{S}^B$ and $\mathcal{S}^C$'s respective models.

```
record Fusable
  (𝒮^A : Semantics ℰ^A ℳ^A) (𝒮^B : Semantics ℰ^B ℳ^B) (𝒮^C : Semantics ℰ^C ℳ^C)
  (ℰ^R_BC : (e^B : ℰ^B Γ σ) (e^C : ℰ^C Γ σ) → Set)
  (ℰ^R : (ρ^A : Δ [ ℰ^A ] Γ) (ρ^B : Θ [ ℰ^B ] Δ) (ρ^C : Θ [ ℰ^C ] Γ) → Set)
  (ℳ^R : (m^B : ℳ^B Γ σ) (m^C : ℳ^C Γ σ) → Set)
  : Set where
```

Similarly to the previous section, most of the fields of this record describe what structure these relations need to have. However, we start with something slightly different: given that we are planing to run the Semantics $\mathcal{S}^B$ *after* having run $\mathcal{S}^A$, we need a way to extract a term from an element of $\mathcal{S}^A$'s model. Our first field is therefore reify^A:

```
  reify^A : (m : ℳ^A Γ σ) → Γ ⊢ σ
```

Then come two constraints dealing with the relations talking about evaluation environments. $\mathscr{E}^{\mathsf{R}}_{\cdot}$ tells us how to extend related environments: one should be able to push related values onto the environments for $\mathcal{S}^B$ and $\mathcal{S}^C$ whilst merely extending the one for $\mathcal{S}^A$ with a token value generated using embed.

$\mathscr{E}^{\mathsf{R}}_{\mathsf{wk}}$ guarantees that it is always possible to weaken the environments for $\mathcal{S}^B$ and $\mathcal{S}^C$ in a $\mathscr{E}^R$ preserving manner.

$$
\begin{aligned}
\mathscr{E}^{\mathsf{R}}_{\cdot} \;:\; & (\rho^{\mathsf{R}} : \mathscr{E}^{\mathsf{R}}\ \rho^{\mathsf{A}}\ \rho^{\mathsf{B}}\ \rho^{\mathsf{C}})\ (u^{\mathsf{R}} : \mathscr{E}^{\mathsf{R}}_{\mathsf{BC}}\ u^{\mathsf{B}}\ u^{\mathsf{C}}) \to \\
& \mathscr{E}^{\mathsf{R}}\ (\mathsf{wk}[\ \mathcal{S}^{\mathsf{A}}.\mathsf{wk}\ ]\ (\mathsf{step}\ \mathsf{refl})\ \rho^{\mathsf{A}}\ \text{`}\bullet\ \mathcal{S}^{\mathsf{A}}.\mathsf{embed}\ \mathsf{zero}) \\
& \quad (\rho^{\mathsf{B}}\ \text{`}\bullet\ u^{\mathsf{B}})\ (\rho^{\mathsf{C}}\ \text{`}\bullet\ u^{\mathsf{C}})
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{E}^{\mathsf{R}}_{\mathsf{wk}} \;:\; & (inc : \Theta \subseteq E)\ (\rho^{\mathsf{R}} : \mathscr{E}^{\mathsf{R}}\ \rho^{\mathsf{A}}\ \rho^{\mathsf{B}}\ \rho^{\mathsf{C}}) \to \\
& \mathscr{E}^{\mathsf{R}}\ \rho^{\mathsf{A}}\ (\mathsf{wk}[\ \mathcal{S}^{\mathsf{B}}.\mathsf{wk}\ ]\ inc\ \rho^{\mathsf{B}})\ (\mathsf{wk}[\ \mathcal{S}^{\mathsf{C}}.\mathsf{wk}\ ]\ inc\ \rho^{\mathsf{C}})
\end{aligned}
$$

Then we have the relational counterpart of the various term constructors. As with the previous section, only a handful of them are out of the ordinary. We will start with the `var case. It states that fusion indeed happens when evaluating a variable using related environments.

$$
\begin{aligned}
\mathsf{R}[\![\mathsf{var}]\!] \;:\; & (v : \sigma \in \Gamma)\ (\rho^{\mathsf{R}} : \mathscr{E}^{\mathsf{R}}\ \rho^{\mathsf{A}}\ \rho^{\mathsf{B}}\ \rho^{\mathsf{C}}) \to \\
& \mathscr{M}^{\mathsf{R}}\ (\mathcal{S}^{\mathsf{B}} \vDash [\![\ reify^{\mathsf{A}}\ (\mathcal{S}^{\mathsf{A}}.[\![\mathsf{var}]\!]\ (\mathsf{lookup}\ \rho^{\mathsf{A}}\ v))\ ]\!]\ \rho^{\mathsf{B}})\ (\mathcal{S}^{\mathsf{C}}.[\![\mathsf{var}]\!]\ (\mathsf{lookup}\ \rho^{\mathsf{C}}\ v))
\end{aligned}
$$

The `λ-case puts some rather strong restrictions on the way the λ-abstraction's body may be used by $\mathcal{S}^A$: we assume it is evaluated in an environment weakened by one variable and extended using $\mathcal{S}^A$'s embed. But it is quite natural to have these restrictions: given that $\mathsf{reify}^{\mathsf{A}}$ quotes the result back, we are expecting this type of evaluation in an extended context (i.e. under one lambda). And it turns out that this is indeed enough for all of our examples. The evaluation environments used by the semantics $\mathcal{S}^B$ and $\mathcal{S}^C$ on the other hand can be arbitrarily weakened before being extended with related values to be substituted for the variable bound by the `λ.

$$
\begin{aligned}
& \mathsf{R}[\![\lambda]\!] \;: \\
& \quad (t : \Gamma \bullet \sigma \vdash \tau)\ (\rho^{\mathsf{R}} : \mathscr{E}^{\mathsf{R}}\ \rho^{\mathsf{A}}\ \rho^{\mathsf{B}}\ \rho^{\mathsf{C}}) \to \\
& \quad (r : (inc : \Theta \subseteq E)\ (u^{\mathsf{R}} : \mathscr{E}^{\mathsf{R}}_{\mathsf{BC}}\ u^{\mathsf{B}}\ u^{\mathsf{C}}) \to \\
& \qquad \mathsf{let}\ \ \rho^{\mathsf{A}\prime} = \mathsf{wk}[\ \mathcal{S}^{\mathsf{A}}.\mathsf{wk}\ ]\ (\mathsf{step}\ \mathsf{refl})\ \rho^{\mathsf{A}}\ \text{`}\bullet\ \mathcal{S}^{\mathsf{A}}.\mathsf{embed}\ \mathsf{zero} \\
& \qquad\quad\ \ \rho^{\mathsf{B}\prime} = \mathsf{wk}[\ \mathcal{S}^{\mathsf{B}}.\mathsf{wk}\ ]\ inc\ \rho^{\mathsf{B}}\ \text{`}\bullet\ u^{\mathsf{B}} \\
& \qquad\quad\ \ \rho^{\mathsf{C}\prime} = \mathsf{wk}[\ \mathcal{S}^{\mathsf{C}}.\mathsf{wk}\ ]\ inc\ \rho^{\mathsf{C}}\ \text{`}\bullet\ u^{\mathsf{C}} \\
& \qquad \mathsf{in}\ \mathscr{M}^{\mathsf{R}}\ (\mathcal{S}^{\mathsf{B}} \vDash [\![\ reify^{\mathsf{A}}\ (\mathcal{S}^{\mathsf{A}} \vDash [\![\ t\ ]\!]\ \rho^{\mathsf{A}\prime})\ ]\!]\ \rho^{\mathsf{B}\prime})\ (\mathcal{S}^{\mathsf{C}} \vDash [\![\ t\ ]\!]\ \rho^{\mathsf{C}\prime})) \to \\
& \mathscr{M}^{\mathsf{R}}\ (\mathcal{S}^{\mathsf{B}} \vDash [\![\ reify^{\mathsf{A}}\ (\mathcal{S}^{\mathsf{A}} \vDash [\![\ \text{`}\lambda\ t\ ]\!]\ \rho^{\mathsf{A}})\ ]\!]\ \rho^{\mathsf{B}})\ (\mathcal{S}^{\mathsf{C}} \vDash [\![\ \text{`}\lambda\ t\ ]\!]\ \rho^{\mathsf{C}})
\end{aligned}
$$

The other cases are just a matter of stating that, given the expected induction hypotheses, one can deliver a proof that fusion can happen on the compound expression.

Fundamental Lemma of Fusable Semantics As with synchronisation, we measure the usefulness of this framework by the fact that we can prove its fundamental lemma first

and that we get useful theorems out of it second. Once again, having carefully identified what the constraints should be, proving the fundamental lemma turns out to amount to a simple traversal we choose to omit here.

```
module Fusion (fusable : Fusable 𝒮ᴬ 𝒮ᴮ 𝒮ᶜ ℰᴿBC ℰᴿ ℳᴿ) where
  open Fusable fusable

  lemma : (t : Γ ⊢ σ) (ρᴿ : ℰᴿ ρᴬ ρᴮ ρᶜ) →
          ℳᴿ (𝒮ᴮ ⊨⟦ reifyᴬ (𝒮ᴬ ⊨⟦ t ⟧ ρᴬ) ⟧ ρᴮ) (𝒮ᶜ ⊨⟦ t ⟧ ρᶜ)
```

The Special Case of Syntactic Semantics Given that Syntactic semantics use a lot of constructors as their own semantic counterpart, it is possible to generate evidence of them being fusable with much fewer assumptions. We isolate them and prove the result generically in order to avoid repeating ourselves. A SyntacticFusable record packs the evidence necessary to prove that the Syntactic semantics syn^A and syn^B can be fused using the Syntactic semantics syn^C. It is indexed by these three Syntactics as well as two relations corresponding to the $\mathscr{E}^R_{BC}$ and $\mathscr{E}^R$ ones of the Fusable framework.

It contains the same $\mathscr{E}^R_{\cdot}$, $\mathscr{E}^R_{wk}$ and R⟦var⟧ fields as a Fusable as well as a fourth one (embed^{BC}) saying that syn^B and syn^C's respective embeds are producing related values.

```
  embedᴮᶜ : ℰᴿBC (Synᴮ.embed zero) (Synᶜ.embed zero)
```

The important result is that given a SyntacticFusable relating three Syntactic semantics, one can deliver a Fusable relating the corresponding Semantics where $\mathscr{M}^R$ is the propositional equality.

```
syntacticFusable : (synᴿ : SyntacticFusable synᴬ synᴮ synᶜ ℰᴿBC ℰᴿ) →
  Fusable (syntactic synᴬ) (syntactic synᴮ) (syntactic synᶜ) ℰᴿBC ℰᴿ _≡_
```

It is then trivial to prove that Renaming can be fused with itself to give rise to another renaming (obtained by composing the two context inclusions): $\mathscr{E}^R_{\cdot}$ uses [_, _], a case-analysis combinator for $\sigma \in (\Gamma \bullet \tau)$ distinguishing the case where $\sigma \in \Gamma$ and the one where σ equals τ, whilst the other connectives are either simply combining induction hypotheses using the congruence of propositional equality or even simply its reflexivity (the two embeds we use are identical: they are both the one of syntacticRenaming hence why embed^{BC} is so simple).

```
RenamingFusable :
  SyntacticFusable syntacticRenaming syntacticRenaming syntacticRenaming
                   _≡_ (λ ρᴬ ρᴮ ρᶜ → ∀ σ pr → lookup (trans ρᴬ ρᴮ) pr ≡ lookup ρᶜ pr)
RenamingFusable = record
  { ℰᴿ·     = λ ρᴿ eq → [ eq , ρᴿ ]
  ; ℰᴿwk    = λ inc ρᴿ σ pr → PEq.cong (lookup inc) (ρᴿ σ pr)
  ; R⟦var⟧  = λ v ρᴿ → PEq.cong `var (ρᴿ _ v)
```

```
  ; embed^BC  = PEq.refl }
```

Similarly, a Substitution following a Renaming is equivalent to a Substitution where the evaluation environment is the composition of the two previous ones.

```
RenamingSubstitutionFusable :
  SyntacticFusable syntacticRenaming syntacticSubstitution syntacticSubstitution
  _≡_ (λ ρ^A ρ^B ρ^C → ∀ σ pr → lookup ρ^B (lookup ρ^A pr) ≡ lookup ρ^C pr)
```

Using the newly established fact about fusing two Renamings together, we can establish that a Substitution followed by a Renaming is equivalent to a Substitution where the elements in the evaluation environment have been renamed.

```
SubstitutionRenamingFusable :
  SyntacticFusable syntacticSubstitution syntacticRenaming syntacticSubstitution
  (λ υ t → 'var υ ≡ t) (λ ρ^A ρ^B ρ^C → ∀ σ pr → Renaming ⊨[[ lookup ρ^A pr ]] ρ^B ≡ lookup ρ^C pr)
```

Finally, using the fact that we now know how to fuse a Substitution and a Renaming together no matter in which order they are performed, we can prove that two Substitutions can be fused together to give rise to another Substitution.

```
SubstitutionFusable :
  SyntacticFusable syntacticSubstitution syntacticSubstitution syntacticSubstitution
  _≡_ (λ ρ^A ρ^B ρ^C → ∀ σ pr → Substitution ⊨[[ lookup ρ^A pr ]] ρ^B ≡ lookup ρ^C pr)
```

These four lemmas are usually painfully proven one after the other. Here we managed to discharge them by simply instantiating our framework four times in a row, using the former instances to discharge the constraints arising in the later ones. But we are not at all limited to proving statements about Syntactics only.

Examples of Fusable Semantics The most simple example of Fusable Semantics involving a non Syntactic one is probably the proof that Renaming followed by Normalise$^{\beta\iota\xi\eta}$ is equivalent to Normalisation by Evaluation where the environment has been tweaked.

```
RenamingNormaliseFusable : Fusable Renaming Normalise^βιξη Normalise^βιξη (EQREL _ _)
  (λ ρ^A ρ^B ρ^C → ∀ σ pr → EQREL _ σ (lookup ρ^B (lookup ρ^A pr)) (lookup ρ^C pr)) (EQREL _ _)
```

Then, we use the framework to prove that to Normalise$^{\beta\iota\xi\eta}$ by Evaluation after a Substitution amounts to normalising the original term where the substitution has been evaluated first. The constraints imposed on the environments might seem quite restrictive but they are actually similar to the Uniformity condition described by C. Coquand [8] in her detailed account of Normalisation by Evaluation for a simply-typed λ-calculus with explicit substitution.

```
SubstitutionNormaliseFusable : Fusable  Substitution Normalise^βιξη Normalise^βιξη
  (EQREL _ _)
```

```
(λ ρᴬ ρᴮ ρᶜ → '∀[ EQREL _ _ ] ρᴮ ρᴮ
  × ((σ : ty) (pr : σ ∈ _) (inc : _ ⊆ Θ) →
    EQREL Θ σ (Normaliseᵝᶦᵡᵑ ⊨⟦ lookup ρᴬ pr ⟧ (pack $ λ pr → wkᵝᶦᵡᵑ _ inc $ lookup ρᴮ pr))
      (wkᵝᶦᵡᵑ σ inc $ lookup ρᶜ pr))
  × ((σ : ty) (pr : σ ∈ _) → EQREL _ σ (Normaliseᵝᶦᵡᵑ ⊨⟦ lookup ρᴬ pr ⟧ ρᴮ) (lookup ρᶜ pr)))
(EQREL _ _)
```

Finally, we may use the notion of Fusable to prove that our definition of pretty-printing ignores Renamings. In other words, as long as the names provided for the free variables are compatible after the renaming and as long as the name supplies are equal then the string produced, as well as the state of the name supply at the end of the process, are equal.

```
RenamingPrettyPrintingFusable : Fusable Renaming Printing Printing _≡_
  (λ ρᴬ ρᴮ → '∀[ _≡_ ] (trans ρᴬ ρᴮ))
  (λ p q → ∀ names₁ ≡ names₂ → runPrinter p names₁ ≡ runPrinter q names₂)
```

A direct corollary is that pretty printing a weakened closed term amounts to pretty printing the term itself in a dummy environment.

```
PrettyRenaming : (t : ε ⊢ σ) (inc : ε ⊆ Γ) →
  print (wkᵀ inc t) ≡ proj₁ (runPrinter (Printing ⊨⟦ t ⟧ 'ε) $ Stream.drop (size Γ) names)
PrettyRenaming t inc = PEq.cong proj₁ $ lemma t (packᴿ $ λ ()) $ proof Γ Γ
  where open Fusion RenamingPrettyPrintingFusable
```

8 Conclusion

We have explained how to make using an inductive family to only represent the terms of an eDSL which are well-scoped and well-typed by construction more tractable. We proceeded by factoring out a common notion of Semantics encompassing a wide range of type and scope preserving traversals such as renaming and substitution, which were already handled by the state of the art [23,3], but also pretty printing, or various variations on normalisation by evaluation. Our approach crucially relied on the careful distinction we made between values in the environment and values in the model, as well as the slight variation on the structure typical of Kripke-style models. Indeed, in our formulation, the domain of a binder's interpretation is an environment value rather than a model one.

We have then demonstrated that, having this shared structure, one could further alleviate the implementer's pain by tackling the properties of these Semantics in a similarly abstract approach. We characterised, using a first logical relation, the traversals which were producing related outputs provided they were fed related inputs. A more involved second logical relation gave us a general description of triples of Fusable semantics such that composing the two first ones would yield an instance of the third one.

References

1. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, pages 182–199. Springer, 1995.
2. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, pages 453–468. Springer, 1999.
3. Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of automated reasoning*, 49(2):141–159, 2012.
4. Ulrich Berger. Program extraction from normalization proofs. In *Typed Lambda Calculi and Applications*, pages 91–106. Springer, 1993.
5. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 203–211. IEEE, 1991.
6. Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated. *Journal of Functional Programming*, 2009.
7. Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, volume 43, pages 143–156. ACM, 2008.
8. Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15(1):57–90, 2002.
9. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(01):75–94, 1997.
10. Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages*, pages 80–99. Springer, 2011.
11. Olivier Danvy, Chantal Keller, and Matthias Puech. Tagless and typeful normalization by evaluation using generalized algebraic data types.
12. Nicolaas Govert de Bruijn. Lambda Calculus notation with nameless dummies. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.
13. Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical frameworks*, 2:6, 1991.
14. Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.
15. François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.
16. Andy Gill. Domain-specific languages and code synthesis using Haskell. *Queue*, 12(4):30, 2014.
17. Healfdene Goguen and James McKinna. Candidates for substitution. *LFCS REPORT SERIES-LABORATORY FOR FOUNDATIONS OF COMPUTER SCIENCE ECS LFCS*, 1997.
18. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.
19. John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, pages 53–96. Springer, 1995.
20. Alan Jeffrey. Associativity for free! `http://thread.gmane.org/gmane.comp.lang.agda/3259`, 2011.
21. Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. *ACM SIGPLAN Notices*, 48(12):81–92, 2014.

22. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
23. Conor McBride. Type-preserving renaming and substitution. 2005.
24. Conor McBride and James McKinna. The view from the left. *Journal of functional programming*, 14(01):69–111, 2004.
25. John C Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, 1996.
26. John C Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1):99–124, 1991.
27. Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.
28. Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.
29. John C Reynolds. Types, abstraction and parametric polymorphism. 1983.
30. Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*, pages 21–36. Springer, 2013.
31. Philip Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.