Type-and-Scope Safe Programs and Their Proofs

Guillaume Allais
gallais@cs.ru.nl
Radboud University

James Chapman Conor McBride
{james.chapman,conor.mcbride}@strath.ac.uk
University of Strathclyde

James McKinna
james.mckinna@ed.ac.uk
University of Edinburgh

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness Proofs; D.3.2 [*Language Classifications*]: Applicative (functional) languages; F.3.2 [*Semantics of Programming Languages*]: Denotational semantics, Partial evaluation

Keywords Lambda-calculus, Mechanized Meta-Theory, Normalisation by Evaluation, Semantics, Generic Programming, Agda

Abstract

We abstract the common type-and-scope safe structure from computations on λ-terms that deliver, e.g., renaming, substitution, evaluation, CPS-transformation, and printing with a name supply. By exposing this structure, we can prove generic simulation and fusion lemmas relating operations built this way. This work has been fully formalised in Agda.

1. Introduction

A programmer implementing an embedded language with bindings has a wealth of possibilities. However, should she want to be able to inspect the terms produced by her users in order to optimise or even compile them, she will have to work with a deep embedding. Which means that she will have to (re)implement a great number of traversals doing such mundane things as renaming, substitution, or partial evaluation. Should she want to get help from the typechecker in order to fend off common bugs, she can opt for inductive families (Dybjer 1991) to enforce precise invariants. But the traversals now have to be invariant preserving too!

In an unpublished manuscript, McBride (2005) observes the similarity between the types and implementations of renaming and substitution for simply typed λ-calculus (STλC) in a dependently typed language as shown in fig. 1. There are three differences between the implemenations of renaming and substitution: (1) in the variable case, after renaming a variable we must wrap it in a `'var` constructor whereas a substitution directly produces a term; (2) when weakening a renaming to push it under a λ we need only post-compose the remaning with the De Bruijn variable successor constructor `su` (which is essentially weakening for variables) whereas for a substitution we need a weakening operation for terms which can be given by renaming via the successor constructor `ren su`. (3) also in the λ case when pushing a renaming or substitution under a binder we must extend it to ensure that the variable bound by the λ mapped to itself. For renaming this involves extended by the zeroth variable `ze` whereas for subsitutions we must extend by the zeroth variable seen as a term `'var ze`. He defines a notion of "Kit" abstracting these differences. The uses of `Kit.—` operations in the generalising

```
ren : (∀σ. Var σ Γ → Var σ Δ) → (∀σ. Tm σ Γ → Tm σ Δ)
ren ρ ('var υ)  = 'var (ρ υ)
ren ρ (f '$ t)  = ren ρ f '$ ren ρ t
ren ρ ('λ b)    = 'λ (ren ((su ∘ ρ) –, ze) b)

sub : (∀σ. Var σ Γ → Tm σ Δ) → (∀σ. Tm σ Γ → Tm σ Δ)
sub ρ ('var υ)  = ρ υ
sub ρ (f '$ t)  = sub ρ f '$ sub ρ t
sub ρ ('λ b)    = 'λ (sub ((ren su ∘ ρ) –, 'var ze) b)
```

Figure 1. Renaming and Substitution for the STλC

```
kit : (∀σ. Var σ Γ → ♦ σ Δ) → (∀σ. Tm σ Γ → Tm σ Δ)
kit ρ ('var υ)  = Kit.var κ (ρ υ)
kit ρ (f '$ t)  = kit ρ f '$ kit ρ t
kit ρ ('λ b)    = 'λ (kit ((Kit.wkn κ ∘ ρ) –, Kit.zro κ) b)
```

Figure 2. Kit traversal for the STλC, for κ of type `Kit ♦`

```
nbe : (∀σ. Var σ Γ → Val σ Δ) → (∀σ. Tm σ Γ → Val σ Δ)
nbe ρ ('var υ)  = ρ υ
nbe ρ (f '$ t)  = APP (nbe ρ f) (nbe ρ t)
nbe ρ ('λ t)    = LAM (λ re υ → nbe ((wk re ∘ ρ) –, υ) t)
```

Figure 3. Normalisation by Evaluation for the STλC

the traversal function kit are shown (in pink) in fig. 2.

The contributions of the present paper are twofold:

- We generalise the "Kit" approach from syntax to semantics bringing operations like normalisation (cf. fig. 3) and printing with a name supply into our framework.
- We prove generic results about simulations between and fusions of semantics given by, and enabled by, Kit.

Outline We start by defining the simple calculus we will use as a running example. We then introduce a notion of environments and one well known instance: the category of renamings. This leads us to defining a generic notion of type and scope-preserving Semantics together with a generic evaluation function. We then showcase the ground covered by these Semantics: from the syntactic ones corresponding to renaming and substitution to printing with names, variations of Normalisation by Evaluation or CPS transformations. Finally, given the generic definition of Semantics, we can prove fundamental lemmas about these evaluation functions: we characterise the semantics which can simulate one another and give an abstract treatment of composition yielding compaction and reuse of proofs compared to Benton et al. (2012).

Notation This article is a literate Agda file. We hide telescopes of implicit arguments and Set levels, and properly display (super / sub)-scripts as well as special operators such as $\gg\!=$ or $+\!\!+$. Colours help: green identifiers are data constructors, pink names refer to record fields, blue is characteristic of defined symbols, and comments are `red` typewrite font. Underscores have a special status: when defining mixfix identifiers (Danielsson and Norell 2011), they mark positions where arguments may be inserted.

Formalisation This whole development[1] has been checked by Agda (Norell 2009) 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, although not the proofs, can and have been fully replicated in Haskell using type families, higher rank polymorphism and GADTs to build singletons (Eisenberg and Weirich 2013) 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 a Kripke-style Normalisation by Evaluation in Haskell.

2. The Calculus and its Embedding

$$\begin{array}{rcl} \sigma,\tau & ::= & 1 \mid 2 \mid \sigma \to \tau \\ b,t,u & ::= & x \mid t\,u \mid \lambda x.\,b \mid \langle\rangle \\ & & \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ u \end{array}$$

We work with a deeply embedded simply typed λ-calculus. It has `1` and `2` as base types and serves as a minimal example of a system with a record type equipped with an η-rule and a sum type. This grammar is represented in Agda as follows:

```
data Ty : Set where
  '1 '2  : Ty
  _'→_   : Ty → Ty → Ty

data Cx (ty : Set) : Set where
  ε      : Cx ty
  _•_    : Cx ty → ty → Cx ty
```

To talk about the types of the variables in scope, we need *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.

To make type signatures more readabale, we introduce combinators acting on context-indexed types. The most straightforward ones are pointwise lifting of existing operators on types, and we denote them as dotted versions of their counterparts: the definition of the pointwise function space `_→̇_` is shown here and the reader will infer the corresponding one for pointwise disjoint sums (`_⊎̇_`) and products (`_×̇_`). The "universally" operator `[_]` turn a context-indexed type into a type using an (implicit) universal quantification. Last but not least, the operator `_⊢_` mechanizes the mathematical convention of only mentioning context *extensions* when presenting judgements (Martin-Löf 1982).

```
_→̇_ : (Cx ty → Set) → (Cx ty → Set) → (Cx ty → Set)
(S →̇ T) Γ = S Γ → T Γ

[_] : (Cx ty → Set) → Set
[ T ] = ∀ {Γ} → T Γ

_⊢_ : ty → (Cx ty → Set) → (Cx ty → Set)
(σ ⊢ S) Γ = S (Γ • σ)
```

Variables are then positions in such a context represented as typed de Bruijn (1972) indices. As shown in the comments, this amounts to an inductive definition of context membership. We use the combinators defined above to show only local changes to the context.

```
data Var (τ : ty) : Cx ty → Set where
  ze : - ∀ Γ. Var τ (Γ • τ)
       [          τ ⊢ Var τ ]
  su : - ∀ Γ σ. Var τ Γ → Var τ (Γ • σ)
       [ Var τ →̇ (σ ⊢ Var τ) ]
```

The syntax for this calculus guarantees that terms are well scoped-and-typed by construction. This presentation due to Altenkirch and Reus (1999) relies heavily on Dybjer's (1991) inductive families. Rather than having untyped pre-terms and a typing relation assigning a type to them, the typing rules are here enforced in the syntax. Notice that the only use of `_⊢_` to extend the context is for the body of a λ.

```
data Tm : Ty → Cx Ty → Set where
  'var    : [ Var σ →̇                  Tm σ          ]
  _'$_    : [ Tm (σ '→ τ) →̇ Tm σ →̇     Tm τ          ]
  'λ      : [ σ ⊢ Tm τ →̇                Tm (σ '→ τ)   ]
  '⟨⟩     : [                           Tm '1         ]
  'tt 'ff : [                           Tm '2         ]
  'if     : [ Tm '2 →̇ Tm σ →̇ Tm σ →̇     Tm σ          ]
```

[1] https://github.com/gallais/type-scope-semantics

3. A Generic Notion of Environment

All the semantics we are interested in defining associate to a term t of type Tm σ Γ, a value of type $\mathcal{C}$ σ Δ given an interpretation $\mathcal{E}$ Δ τ for each one of its free variables τ in Γ. We call the collection of these interpretations an $\mathcal{E}$-(evaluation) environment. We leave out $\mathcal{E}$ 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 for a notion of Model.

```
Model : Set
Model = ty → Cx ty → Set
```

Formally, this translates to $\mathcal{E}$-environments being the pointwise lifting of the relation $\mathcal{E}$ 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 (2011) that one can obtain associativity of append for free by using difference lists. In our case the interplay between various combinators (e.g. refl and select) defined later on is vastly simplified by this rather simple decision.

```
record _–Env (Γ : Cx ty) (𝒱 : Model) (Δ : Cx ty) : Set
  where constructor pack
        field lookup : Var σ Γ → 𝒱 σ Δ
```

Just as an environment interprets variables in a model, a computation gives a meaning to terms into a model.

```
_–Comp : Cx Ty → (𝒞 : Model) → Cx Ty → Set
(Γ –Comp) 𝒞 Δ = Tm σ Γ → 𝒞 σ Δ
```

An appropriate notion of semantics for the calculus is one that will map environments to computations. In other words, a set of constraints on $\mathcal{V}$ and $\mathcal{C}$ guaranteeing the existence of a function of type: [$(\Gamma$ –Env$)$ $\mathcal{V}$ $\dot{\to}$ $(\Gamma$ –Comp$)$ $\mathcal{C}$]

These environments naturally behave like the contexts they are indexed by: there is a trivial environment for the empty context and one can easily extend an existing one by providing an appropriate value. The packaging of the function representing to the environment in a record allows for two things: it helps the typechecker by stating explicitly which Model the values correspond to and it empowers us to define environments by copattern-matching (Abel et al. 2013) thus defining environments by their use cases.

```
`ε : [ (ε –Env) 𝒱 ]
_`•_ : [ (Γ –Env) 𝒱 ⇾ 𝒱 σ ⇾ (Γ • σ –Env) 𝒱 ]

lookup `ε       ()
lookup (ρ `• s) ze      = s
lookup (ρ `• s) (su n)  = lookup ρ n
```

The Category 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 renamings as described variously as Order Preserving Embeddings (Chapman 2009), thinnings (which we use) or context inclusions, or just weakenings (Altenkirch et al. 1995). Writing non-injective or non-order preserving renamings would take perverse effort given that we only implement generic interpretations. In practice, although the type of renamings is more generous, we only introduce weakenings (skipping variables at the beginning of the context) that become thinnings (skipping variables at arbitrary points in the context) when we push them under binders.

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

```
_⊆_ : (Γ Δ : Cx ty) → Set
Γ ⊆ Δ = (Γ –Env) Var Δ
```

We formulate a thinning principle using $\subseteq$. By a "thinning principle", we mean that if P holds of Γ and $\Gamma \subseteq \Delta$ then P holds for Δ too. In the case of variables, thinning 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 $\mathcal{V}$ between contexts and types, they enjoy thinning if $\mathcal{V}$ does.

```
Thinnable : (Cx ty → Set) → Set
Thinnable S = Γ ⊆ Δ → (S Γ → S Δ)

thVar : (σ : ty) → Thinnable (Var σ)
thVar σ inc v = lookup inc v

th[_] : ((σ : ty) → Thinnable (𝒱 σ)) →
        Thinnable ((Γ –Env) 𝒱)
lookup (th[ th ] inc ρ) = th _ inc ∘ lookup ρ
```

These simple observations allow us to prove that thinnings form a category which, in turn, lets us provide the user with the constructors Altenkirch, Hofmann and Streicher's "Category of Weakening" (1995) is based on.

```
refl : Γ ⊆ Γ
refl = pack id

select : Γ ⊆ Δ → (Δ –Env) 𝒱 Θ → (Γ –Env) 𝒱 Θ
lookup (select inc ρ) = lookup ρ ∘ lookup inc

step : Γ ⊆ Δ → Γ ⊆ (Δ • σ)
step inc = select inc (pack su)

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

The modal operator $\Box$ states that a given predicate holds for all thinnings of a context. It is a closure operator for Thinnable.

```
□ : (Cx ty → Set) → (Cx ty → Set)
(□ S) Γ = {Δ : Cx _} → Γ ⊆ Δ → S Δ

th□ : Thinnable (□ S)
th□ inc s = s ∘ select inc
```

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.

4. Semantics and their Generic Evaluators

The upcoming sections demonstrate 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 $\mathcal{V}$ and $\mathcal{C}$ describing respectively the values in the environment and the ones in the model. In cases such as substitution or normalisation by evaluation, $\mathcal{V}$ and $\mathcal{C}$ 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 (𝒱 : Model) (𝒞 : Model) : Set where
```

The first method of a Semantics deals with environment values. They need to be thinnable (th) so that the traversal may introduce fresh variables when going under a binder whilst keeping the environment well-scoped.

```
  th : (σ : Ty) → Thinnable (𝒱 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 bridges the gap between the fact that the environment translates variables into values $\mathcal{V}$ but the evaluation function returns computations $\mathcal{C}$.

```
  ⟦var⟧ : [ 𝒱 σ ·→ 𝒞 σ ]
```

The semantic λ-abstraction is notable for two reasons: first, following Mitchell and Moggi (1991), 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 computations to computations, it takes *values* to computations. It matches precisely the fact that the body of a λ-abstraction exposes one extra free variable, prompting us to extend the environment with a value for it. In the special case where $\mathcal{V} = \mathcal{C}$ (normalisation by evaluation for instance), we recover the usual Kripke structure.

```
  ⟦λ⟧ : [ □ (𝒱 σ ·→ 𝒞 τ) ·→ 𝒞 (σ ↣ τ) ]
```

The remaining fields' types are a direct translation of the types of the constructor they correspond to: substructures have simply been replaced with computations thus making these operators ideal to combine induction hypotheses.

```
  _⟦$⟧_ : [ 𝒞 (σ ↣ τ) ·→ 𝒞 σ ·→   𝒞 τ  ]
  ⟦⟨⟩⟧  : [                      𝒞 'one ]
  ⟦tt⟧  : [                      𝒞 'two ]
  ⟦ff⟧  : [                      𝒞 'two ]
  ⟦if⟧  : [ 𝒞 'two ·→ 𝒞 σ ·→ 𝒞 σ ·→ 𝒞 σ ]
```

The type we chose for ⟦λ⟧ makes the Semantics notion powerful enough that even logical predicates are instances of it. And we indeed exploit this power when defining normalisation by evaluation as a semantics: the model construction is, after all, nothing but a logical predicate. As a consequence it seems rather natural to call sem, the fundamental lemma of semantics. We prove it in a module parameterised 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 which combines the induction hypotheses for its subterms.

```
module Eval (𝒮 : Semantics 𝒱 𝒞) where
open Semantics 𝒮
sem : [ (Γ –Env) 𝒱 ·→ (Γ –Comp) 𝒞 ]
sem ρ ('var v)   = ⟦var⟧ (lookup ρ v)
sem ρ (t '$ u)   = sem ρ t ⟦$⟧ sem ρ u
sem ρ ('λ b)     = ⟦λ⟧ (λ σ v →
                     sem (th[ th ] σ ρ '• v) b)
sem ρ '⟨⟩        = ⟦⟨⟩⟧
sem ρ 'tt        = ⟦tt⟧
sem ρ 'ff        = ⟦ff⟧
sem ρ ('if b l r) = ⟦if⟧ (sem ρ b) (sem ρ l) (sem ρ r)
```

5. Syntax is the Identity Semantics

As we have explained earlier, this work has been directly influenced by McBride's (2005) manuscript. 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 computations 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 (𝒱 : Model) : Set where
  field th     : (σ : Ty) → Thinnable (𝒱 σ)
        var₀   : [ σ ⊢ 𝒱 σ      ]
        ⟦var⟧  : [ 𝒱 σ ·→ Tm σ ]
```

```
syntactic : Syntactic 𝒱 → Semantics 𝒱 Tm
syntactic syn = let open Syntactic syn in record
  { th    = th; ⟦var⟧  = ⟦var⟧
  ; ⟦λ⟧   = λ t → `λ (t (step refl) var₀) ; _⟦$⟧_ = _`$_
  ; ⟦⟨⟩⟧  = `⟨⟩; ⟦tt⟧ = `tt; ⟦ff⟧ = `ff; ⟦if⟧  = `if }
```

The shape of ⟦λ⟧ or ⟦⟨⟩⟧ should not trick the reader into thinking that this definition performs some sort of η-expansion: sem 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. We have already defined thinning earlier (see Section 3) and we can turn a variable into a term by using the `var constructor. The type of sem specialised to this semantics is then precisely the proof that terms are thinnable.

```
thTm : (σ : Ty) → Thinnable (Tm σ)
thTm σ ρ t = let open Eval Renaming in sem ρ t
```

Simultaneous Substitution Our second example of a semantics is another spin on the syntactic model: environment values are now terms. We get thinning for terms from the previous example. Again, specialising the type of sem reveals that it delivers precisely the simultaneous substitution.

```
subst : (Γ –Env) Tm Δ → Tm σ Γ → Tm σ Δ
subst ρ t = let open Eval Substitution in sem ρ t
```

6. 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 (Hughes 1995; Wadler 2003). But we stick to the simpler setup as *pretty* printing is not our focus here.

This example is interesting for two reasons. Firstly, the distinction between values and computations is once more instrumental: we get to give the procedure a precise type guiding our implementation. The environment carries *names* for the variables currently in scope whilst the computations thread a name-supply (a stream of strings) to be used to generate fresh names for bound variables. If the values in the environment had to be computations too, we would not root out some faulty implementations e.g a program picking a new name each time a variable is mentioned.

```
record Name (σ : Ty) (Γ : Cx Ty) : Set where
  constructor mkN; field getN : String
record Printer (σ : Ty) (Γ : Cx Ty) : Set where
  constructor mkP
  field runP : State (Stream String) String
```

Secondly, the fact that the model's computation type is a monad and that this poses no problem whatsoever in this framework means it is appropriate for handling languages with effects (Moggi 1991), or effectful semantics e.g. logging the various function calls. Here is the full definition of the printer assuming the existence of various format primitives picking a way to display `λ, `$ and `if.

```
Printing : Semantics Name Printer
Printing = record
  { th    = λ _ _ → mkN ∘ getN
  ; ⟦var⟧  = mkP ∘ return ∘ getN
  ; _⟦$⟧_  = λ mf mt → mkP (
               format$ <$> runP mf ⊛ runP mt)
  ; ⟦λ⟧    = λ mb → mkP (
    get >>= λ ns → let x′ = head ns in
    put (tail ns) >>= λ _ →
    runP (mb (step refl) (mkN x′)) >>= λ b′ →
    return (formatλ x′ b′))
  ; ⟦⟨⟩⟧   = mkP (return "⟨⟩")
  ; ⟦tt⟧   = mkP (return "tt")
  ; ⟦ff⟧   = mkP (return "ff")
  ; ⟦if⟧   = λ mb ml mr → mkP (
    formatIf <$> runP mb ⊛ runP ml ⊛ runP mr) }
```

The evaluation function sem will deliver a printer which needs to be run on a Stream of distinct Strings. Our definition of names (not shown here) simply cycles through the letters of the alphabet and guarantess uniqueness by appending a natural number incremented each time we are back at the beginning of the cycle. This crude name generation strategy would naturally be replaced with a more sophisticated one in a user-facing language: we could e.g. use naming hints for user-introduced binders and type-based schemes otherwise (f or g for function, is or js for integers, etc.).

In order to kickstart the evaluation, we still need to provide Names for each one of the free variables in scope. We deliver that environment by a simple stateful computation init chopping off an initial segment of the name supply of the appropriate length. The definition of print follows:

```
init : State (Stream String) ((Γ –Env) Name Γ)

print : Tm σ Γ → String
print t = let open Eval Printing in
  proj₁ ((init >>= λ ρ → runP (sem ρ t)) names)
```

We can observe print's behaviour by writing a test; we state it as a propositional equality and prove it using refl, forcing the typechecker to check that both expressions indeed compute to the same normal form. Here we display the iden-

tity function defined in a context of size 2. As we can see, the binder receives the name `"c"` because `"a"` and `"b"` have already been assigned to the free variables in scope.

```
prettyId : (`λ (`var ze)) ≡ "λc. c"
prettyId = PEq.refl
```

7. Normalisation by Evaluation

Normalisation by Evaluation (NBE) is a technique leveraging 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 by induction on its Ty index. Two procedures are then defined: the first (eval) constructs an element of $\mathcal{C}\ \sigma\ \Gamma$ provided a well typed term of the corresponding Tm $\sigma\ \Gamma$ type whilst the second (reify) extracts, in a type-directed manner, normal forms $\Gamma \vdash \mathsf{nf}\ \sigma$ from elements of the model $\mathcal{C}\ \sigma\ \Gamma$. NBE composes the two procedures. The definition of this eval function is a natural candidate for our Semantics framework. NBE is always defined *for* a given equational theory; we 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 say that for some types, terms have a canonical form: functions will all be λ-headed whilst records will collect their fields — here this makes all elements of `⟨⟩ equal to `⟨⟩.

```
eta : [ Tm (σ `→ τ) ⇾ Tm (σ `→ τ) ]
eta t = `λ (thTm _ (step refl) t `$ `var ze)
```

```
_⟨_/var₀⟩ : [ σ ⊢ Tm τ ⇾ Tm σ ⇾ Tm τ ]
t ⟨ u /var₀⟩ = subst (pack `var `• u) t
```

$$\frac{t : \mathsf{Tm}\ (\sigma \mathbin{`\!\to} \tau)\ \Gamma}{t \rightsquigarrow \mathsf{eta}\ t}\eta_1 \qquad \frac{t : \mathsf{Tm}\ `1\ \Gamma}{t \rightsquigarrow `\langle\rangle}\eta_2$$

$$\frac{}{(`\lambda\ t)\ `\$\ u \rightsquigarrow t \langle\, u \,/\mathsf{var}_0\rangle}\beta$$

The β-rule is the main driver for actual computation, but the presence of an inductive data type (`2) and its eliminator (`if) means we have further redexes: whenever the boolean the eliminator branches on is in canonical form, we may apply a ι-rule. Finally, the ξ-rule lets us reduce under λ-abstractions — the distinction between weak-head normalisation and strong normalisation.

$$\frac{}{`\mathsf{if}\ `\mathsf{tt}\ l\ r \rightsquigarrow l}\iota_1 \qquad \frac{}{`\mathsf{if}\ `\mathsf{ff}\ l\ r \rightsquigarrow r}\iota_2 \qquad \frac{t \rightsquigarrow u}{`\lambda\ t \rightsquigarrow `\lambda\ u}\xi$$

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 NBE which goes under λs and produces η-long βι-short normal forms.

7.1 Normalisation by Evaluation for βιξη

In the case of NBE, the environment values and the computations in the model will both have the same type Kr (standing for "Kripke"), defined by induction on the Ty argument. The η-rules allow us to represent functions (resp. inhabitants of `1) in the source language as function spaces (resp. T). In Agda, there are no such rules for boolean values. We thus need a notion of syntactic normal forms. We parametrise the mutually defined inductive families Ne and Nf by a predicate R constraining the types at which one may embed a neutral as a normal form. This make it possible to control the way NBE η-expands all terms at certain types.

```
data Ne : Model  where
  `var  : [ Var σ ⇾                    Ne σ ]
  _`$_  : [ Ne (σ `→ τ) ⇾ Nf σ ⇾       Ne τ ]
  `if   : [ Ne `2 ⇾ Nf σ ⇾ Nf σ ⇾      Ne σ ]

data Nf : Model  where
  `ne   : R σ → [ Ne σ ⇾       Nf σ      ]
  `⟨⟩   :       [              Nf `1     ]
  `tt `ff :     [              Nf `2     ]
  `λ    :       [ σ ⊢ Nf τ ⇾   Nf (σ `→ τ) ]
```

Once more, the expected notions of thinning thne and thnf are induced as Nf and Ne are syntaxes. We omit their purely structural implementation here and wish we could do so in source code, too: our constructions so far have been syntax-directed and could surely be leveraged by a generic account of syntaxes with binding. We now define the model. The R predicate characterising the types for which neutral terms may be considered normal is here equivalent to the unit type for `2 and the empty type otherwise. This makes us use η-rules eagerly: all inhabitants of Nf $\Gamma$ `1 and Nf Γ $(\sigma \mathbin{`\!\to} \tau)$ are equal to `⟨⟩ and `λ-headed respectively.

The model construction then follows the usual pattern pioneered by Berger (1993) and formally analysed and thoroughly explained by Catarina Coquand (2002). We work by induction on the type and describe η-expanded values: all inhabitants of Kr `1 $\Gamma$ are equal and all elements of Kr $(\sigma \mathbin{`\!\to} \tau)$ Γ are functions in Agda.

```
Kr : Model
Kr `1  = const T
Kr `2  = Nf `2
Kr (σ `→ τ)  = □ (Kr σ ⇾ Kr τ)
```

This model is defined by induction on the type in terms either of syntactic objects (Nf) or using the □-operator which is a closure operator for Thinnings. As such, it is trivial to prove that for all type σ, Kr σ is Thinnable. Application's semantic counterpart is easy to define: given that $\mathcal{V}$ and $\mathcal{C}$ are equal in this instance definition, we just feed the argument directly to the function, with the identity renaming: f $\$\$\ t = f$ refl t. Conditional branching however is more subtle: the boolean value `if branches on may be a neutral term in which case the whole elimination form is stuck. This forces us to define reify and reflect first. These functions, also known as quote and unquote respectively, give the inter-

play between neutral terms, model values and normal forms. reflect performs a form of semantic η-expansion: all stuck `1 terms are equated and all functions are λ-headed. It allows us to define $\mathsf{var}_0$, the semantic counterpart of `var ze.

```
reify   : (σ : Ty) → [ Kr σ ⇾ Nf σ ]
reflect : (σ : Ty) → [ Ne σ ⇾ Kr σ ]

reflect `1        t = ⟨⟩
reflect `2        t = `ne _ t
reflect (σ `→ τ)  t = λ ρ u → let b = th^ne (σ `→ τ) ρ t
                              in reflect τ (b `$ reify σ u)

reify `1        T = `⟨⟩
reify `2        T = T
reify (σ `→ τ)  T = `λ (reify τ (T (step refl) (var₀ σ)))
```

We can then give the semantics of `if: if the boolean is a value, the appropriate branch is picked; if it is stuck the whole expression is reflected in the model.

```
if : [ Kr `2 ⇾ Kr σ ⇾ Kr σ ⇾ Kr σ ]
if `tt       l r = l
if `ff       l r = r
if (`ne _ T) l r = reflect σ (`if T (reify σ l) (reify σ r))
```

We can then combine these components. The semantics of a λ-abstraction is simply the identity function: 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. We obtain a normaliser by kickstarting the evaluation with a dummy environment of reflected variables.

```
Normalise : Semantics Kr Kr
Normalise = record
  { th^Kr = th^Kr; ⟦var⟧ = id; _⟦$⟧_ = _$$_ ; ⟦λ⟧ = id
  ; ⟦⟨⟩⟧ = ⟨⟩; ⟦tt⟧ = `tt; ⟦ff⟧ = `ff; ⟦if⟧ = if }

nbe : [ (Γ –Env) Kr ⇾ (Γ –Comp) Kr ]
nbe ρ t = let open Eval Normalise in sem ρ t

norm : (σ : Ty) → [ Tm σ ⇾ Nf σ ]
norm σ t = reify σ (nbe (pack (reflect _ ∘ `var)) t)
```

7.2 Normalisation by Evaluation for βιξ

As seen above, the traditional typed model construction leads to an NBE procedure outputting βι-normal η-long terms. However actual proof systems rely on evaluation strategies that 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.

In these systems, normal forms are neither η-long nor η-short: the η-rule is never deployed except when comparing a neutral and a constructor-headed term for equality. Instead of declaring them distinct, the algorithm does one step of η-expansion on the neutral term and compares their subterms structurally. The conversion test fails only when confronted with neutral terms with distinct head variables or normal forms with different head constructors.

To reproduce this behaviour, NBE must 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 (1997) approach to defining an NBE procedure for the SK combinator calculus. 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 using a predicate R allowing embedding of neutrals into normal forms at all types and mutually defining the model (Kr) together with the *acting* model (Go):

```
Kr : Model
Kr σ = Ne σ ⊎ Go σ

Go : Model
Go `1        = const ⊤
Go `2        = const Bool
Go (σ `→ τ)  = □ (Kr σ ⇾ Kr τ)
```

Most combinators acting on this model follow a pattern similar to their counterpart's in the previous section. Semantic application is more interesting: in case the function is a stuck term, we grow its spine by reifying its argument; otherwise we have an Agda function ready to be applied. We proceed similarly for the definition of the semantical "if" (omitted here). Altogether, we get another normaliser which is, this time, *not* producing η-long normal forms.

```
_$$_ : [ Kr (σ `→ τ) ⇾ Kr σ ⇾ Kr τ ]
(inj₁ ne) $$ u = inj₁ (ne `$ reify _ u)
(inj₂ F)  $$ u = F refl u
```

7.3 Normalisation by Evaluation for βι

The decision to apply the η-rule lazily can be pushed even further: one may forgo using the ξ-rule too and simply perform weak-head normalisation. This drives 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.

The model construction is much like 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 input term (i.e. a stuck term or the term's computational content) or the original. We exploit this ability most notably in reification where once we have obtained either a head constructor or a head variable, no subterms need be evaluated.

```
Kr : Model
Kr σ = Tm σ ẋ
  (Whne σ ⊎̇ Go σ)

Go : Model
Go `1        = const ⊤
Go `2        = const Bool
Go (σ `→ τ)  = □ (Kr σ ⇾ Kr τ)
```

8. CPS Transformation

In their generic account of continuation passing styles, Hatcliff and Danvy (1994) decompose both call by name and call by value CPS transformations in two phases. The first one, an embedding of the source language into Moggi's Meta Language (1991), picks an evaluation strategy whilst the second one is a generic erasure from Moggi's ML back to the original language. Looking closely at the structure of the first pass, we can see that it is an instance of our Semantics framework. Let us start with the definition of Moggi's Meta Language. Its types are fairly straightforward, we simply have an extra constructor #_ for computations and the arrow has been turned into a *computational* arrow meaning that its codomain is considered to be a computational type:

```
data CTy : Set where
  '1 '2   : CTy
  _'↪#_   : CTy → CTy → CTy
  #_      : CTy → CTy
```

Then comes the Meta-Language itself. It incorporates Tm constructors and eliminators with slightly different types: *value* constructors are associated to *value* types whilst eliminators (and their branches) have *computational* types. Two new term constructors have been added: 'ret and _'≫=_ make #_ a monad. They can be used to explicitly schedule the evaluation order of various subterms.

```
data Ml : CTy → Cx CTy → Set where
  'var    : [ Var σ                          ⇾ Ml σ      ]
  _'$_    : [ Ml (σ '↪# τ) ⇾ Ml σ            ⇾ Ml (# τ)  ]
  '⟨⟩      : [                                  Ml '1     ]
  'tt 'ff : [                                  Ml '2     ]
  'ret    : [ Ml σ                           ⇾ Ml (# σ)  ]
  _'≫=_   : [ Ml (# σ) ⇾ Ml (σ '↪# τ)  ⇾ Ml (# τ)  ]
  'λ      :   [ σ ⊢ Ml (# τ) ⇾ Ml (σ '↪# τ)          ]
  'if     : [ Ml '2 ⇾ Ml (# σ) ⇾ Ml (# σ) ⇾ Ml (# σ) ]
```

As explained in Hatcliff and Danvy's paper, the translation from Ty to CTy fixes the calling convention the CPS translation will have. Both call by name (CBV) and call by value (CBV) can be encoded. They behave the same way on base types (and we group the corresponding equations under the CBX name) but differ in case of the function space. In CBN the argument of a function is a computation whilst it is expected to have been fully evaluated in CBV.

```
CBX : Ty → CTy
CBX '1         = '1
CBX '2         = '2
CBN (σ '↪ τ) = (# CBN σ) '↪# CBN τ
CBV (σ '↪ τ) =    CBV σ  '↪# CBV τ
```

From these translations, we can described the respective interpretations of variables and terms for the two CPS transformations. In both cases the return type of the compiled term is a computational type: the source term is a simple Tm and as such can contain redexes. Variables then play different roles: in the by name strategy, they are all computations whereas in the by value one they are expected to be evaluated already. This leads to the following definitions:

```
VarN σ Γ = Var (# CBN σ) (mapCx (#_ ∘ CBN) Γ)
MlN σ Γ = Ml (# CBN σ) (mapCx (#_ ∘ CBN) Γ)
Varv σ Γ = Var (CBV σ)    (mapCx CBV Γ)
Mlv σ Γ = Ml (# CBV σ) (mapCx CBV Γ)
```

Finally, the corresponding Semantics can be defined (code omitted here) and we get the two CPS transformations by creating dummy environments to kickstart the evaluation:

```
CPSN : Semantics VarN MlN
CPSv : Semantics Varv Mlv

cpsN : [ Tm σ ⇾ MlN σ ]
cpsN = let open Eval CPSN in sem dummy
  where dummy = pack (mapVar (#_ ∘ CBN))
cpsv : [ Tm σ ⇾ Mlv σ ]
cpsv = let open Eval CPSv in sem dummy
  where dummy = pack (mapVar CBV)
```

9. Proving Properties of Semantics

Thanks to Semantics, we have already saved work by not reiterating the same traversals. Moreover, this disciplined approach to building models and defining the associated evaluation functions can help us refactor the proofs of some properties of these semantics.

Instead of using proof scripts as Benton et al. (2012) do, we describe abstractly the constraints the logical relations (Reynolds 1983) defined on computations (and environment values) have to respect to ensure that evaluating a term in related environments produces related outputs. This gives us a generic 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 it is no mere bureaucracy: the result proven here will actually be useful in the next section when considering more complex properties.

9.1 The Simulation Relation

This first example is describing the relational interpretation of the terms. It should give the reader a good introduction to the setup before we take on more complexity. The types involved might look a bit scarily abstract but the idea is rather simple: we have a Simulation between two Semantics when evaluating a term in related environments yields related values. The bulk of the work is to make this intuition formal.

The evidence that we have a Simulation between two Semantics is packaged in a record indexed by the semantics as well as two relations. We call RModel (for *R*elational *Model*) the type of these relations; the first one ($\mathcal{V}R$) relates values in the respective environments and the second one ($\mathcal{C}R$) describes simulation for computations.

```
record Simulation
  (𝒮A : Semantics 𝒱A 𝒞A) (𝒮B : Semantics 𝒱B 𝒞B)
  (𝒱R : RModel 𝒱A 𝒱B) (𝒞R : RModel 𝒞A 𝒞B) : Set where
```

The record's fields say what structure these relations need to have. $\mathcal{V}R_{th}$ states that two similar environments can be thinned whilst staying in simulation. It is stated using the `'∀[_]` predicate transformer (omitted here) which lifts $\mathcal{V}R$ to contexts in a pointwise manner.

```
  𝒱R_th : '∀[ 𝒱R ] ρA ρB →
          '∀[ 𝒱R ] (th[ 𝒮A.th ] inc ρA) (th[ 𝒮B.th ] inc ρB)
```

We then have the relational counterparts of the term constructors. To lighten the presentation we introduce $\mathscr{R}$, which states that the evaluation of a term in distinct contexts yields related computations. And we focus on the most interesting combinators, giving only one characteristic example of the remaining ones.

```
ℛ : Tm σ Γ → (Γ –Env) 𝒱A Δ → (Γ –Env) 𝒱B Δ → Set
ℛ t ρA ρB = rmodel 𝒞R (semA ρA t) (semB ρB t)
```

Our first interesting case is the relational counterpart of `'var`: a variable evaluated in two related environments yields related computations. In other words [[var]] turns related values in related computations.

```
R[[var]] : '∀[ 𝒱R ] ρA ρB → ℛ ('var v) ρA ρB
```

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 thinned environments, each extended by related values, and deliver similar values is enough to guarantee that evaluating the λs in the original environments will produce similar values.

```
R[[λ]] : (r : ∀ inc → rmodel 𝒱R uA uB →
  let ρA′ = th[ 𝒮A.th ] inc ρA '• uA
      ρB′ = th[ 𝒮B.th ] inc ρB '• uB
  in ℛ b ρA′ ρB′) →
       '∀[ 𝒱R ] ρA ρB → ℛ ('λ b) ρA ρB
```

All the remaining cases follow suit: assuming that the evaluation of subterms produces related computations and that the current environments are related, we conclude that the evaluation of the whole term should yield related computations. We show here the relational counterpart of the application constructor and omit the remaining ones:

```
R[[$]] : ℛ f ρA ρB → ℛ t ρA ρB →
         '∀[ 𝒱R ] ρA ρB → ℛ (f '$ t) ρA ρB
```

This specification is only useful if some semantics satisfy it and if given that these constraints are satisfied we can prove the fundamental lemma of simulations stating that the evaluation of a term on related inputs yields related output.

Theorem 1 (Fundamental Lemma of Simulations)**.** *Given two Semantics $\mathcal{S}A$ and $\mathcal{S}B$ in simulation with respect to relations $\mathcal{V}R$ for values and $\mathcal{C}R$ for computations, we have:*

For any term t and environments ρA and ρB, if the two environments are $\mathcal{V}R$-related in a pointwise manner then the semantics associated to t by $\mathcal{S}A$ using ρA is $\mathcal{C}R$-related to the one associated to t by $\mathcal{S}B$ using ρB.

Proof. The proof is a structural induction on t like the one used to define sem. It uses the combinators provided by the constraint that $\mathcal{S}A$ and $\mathcal{S}B$ are in simulation to make use of the induction hypotheses. □

Corollary 1 (Renaming is a Substitution)**.** *Applying a renaming ρ to a term t amounts to applying the substitution* mapEnv `'var` ρ *to that same term t.*

Proof. This is shown by instantiating the fundamental lemma of simulations for the special case where: $\mathcal{S}A$ is Renaming, $\mathcal{S}B$ is Substitution, $\mathcal{V}R\ v\ t$ is `'var` $v \equiv t$ (in other words: the terms in the substitution are precisely the variables in the renaming), and $\mathcal{C}R$ is propositional equality.

The constraints corresponding to the various combinators are mundane: propositional equality is a congruence. □

Another corollary of the simulation lemma relates NBE to itself. This may seem bureaucratic but it is crucial: the model definition Kr uses the host language's function space which contains more functions than simply the ones obtained by evaluating a λ-term. These exotic functions have undesirable behaviours and need to be ruled out to ensure that normalisation has good properties. This is done by defining a Partial Equivalence Relation (Mitchell 1996) (PER) on the model: the elements equal to themselves will be guaranteed to be well behaved. We show that given an environment of values PER-related to themselves, the evaluation of a λ-term produces a computation equal to itself too.

We start by defining 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 `'1` are always trivially equal; values of type `'2` are normal forms and are declared equal when they are effectively syntactically the same; finally functions are equal whenever equal inputs map to equal outputs.

```
PER : (σ : Ty) → [ Kr σ ⇾̇ Kr σ ⇾̇ const Set ]
PER '1       T U = ⊤
PER '2       T U = T ≡ U
PER (σ '→ τ) T U = ∀ inc → PER σ V W →
                   PER τ (T inc V) (U inc W)
```

It is indeed a PER as witnessed by the (omitted here) proofs that PER σ is symmetric and transitive. It also respects the notion of thinning defined for Kr.

```
symPER : PER σ S T → PER σ T S
```

```
transPER : PER σ S T → PER σ T U → PER σ S U
thPER  : PER σ T U → PER σ (thKr σ inc T) (thKr σ inc U)
```

The interplay of reflect and reify with this notion of equality has to be described in one go because of their mutual definition. It confirms that PER is an appropriate notion of semantic equality: PER-related values are reified to propositionally equal normal forms whilst propositionally equal neutral terms are reflected to PER-related values.

```
reifyPER    : PER σ T U → reify σ T ≡ reify σ U
reflectPER  : t ≡ u → PER σ (reflect σ t) (reflect σ u)
```

That suffices to show that evaluating a term in two environments related pointwise by PER yields two semantic objects themselves related by PER.

Corollary 2 (No exotic values)**.** *The evaluation of a term t in an environment of values equal to themselves according to* PER *yields a value equal to itself according to* PER

Proof. By the fundamental lemma of simulations with $\mathcal{S}A$ and $\mathcal{S}B$ equal to Normalise, $\mathcal{V}R$ and $\mathcal{C}R$ to PER. □

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

9.2 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. (2012) 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 sequence and their sequential composition. 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 $\mathcal{S}B$ is equivalent to $\mathcal{S}C$ (e.g. 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 ($\mathcal{V}R_{BC}$) states what it means for two environment values of $\mathcal{S}B$ and $\mathcal{S}C$ respectively to be related. The second one ($\mathcal{V}R$) characterises the triples of environments (one for each one of the semantics) which are compatible. The last one ($\mathcal{C}R$) relates values in $\mathcal{S}B$ and $\mathcal{S}C$'s models.

```
record Fusable (SA : Semantics VA CA)
(SB : Semantics VB CB) (SC : Semantics VC CC)
(VRBC : RModel VB VC)
(VR : (Γ –Env) VA Δ → (Δ –Env) VB Θ →
      (Γ –Env) VC Θ → Set)
(CR : RModel CB CC) : Set where
```

As before, 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 two components: a way to extract a term from an $\mathcal{S}A$ and a way to manufacture a dummy $\mathcal{S}A$ value when going under a binder. Our first two fields are therefore:

```
reifyA  : [ CA σ ·→ Tm σ ]
var0A   : [ σ ⊢ VA σ     ]
```

Then come two constraints dealing with the relations talking about evaluation environments. $\mathcal{V}R_{\bullet}$ 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 the token value var_{0A}.

$\mathcal{V}R_{th}$ guarantees that it is always possible to thin the environments for $\mathcal{S}B$ and $\mathcal{S}C$ in a $\mathcal{V}R$ preserving manner.

```
VR•  : VR ρA ρB ρC → rmodel VRBC uB uC →
       let ρA' = th[ SA.th ] (step refl) ρA ‵• var0A
       in VR ρA' (ρB ‵• uB) (ρC ‵• uC)

VRth : VR ρA ρB ρC →
       VR ρA (th[ SB.th ] inc ρB) (th[ SC.th ] inc ρC)
```

Then we have the relational counterpart of the various term constructors. We can once more introduce an extra definition $\mathscr{R}$ which will make the type of the combinators defined later on clearer. $\mathscr{R}$ relates a term and three environments by stating that the computation one gets by sequentially evaluating the term in the first and then the second environment is related to the one obtained by directly evaluating the term in the third environment.

```
ℛ t ρA ρB ρC = rmodel CR (semB ρB (reifyA (semA ρA t)))
                         (semC ρC t)
```

As with the previous section, only a handful of these combinators 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.

```
R⟦var⟧ : ∀ v → VR ρA ρB ρC → ℛ (‵var v) ρA ρB ρC
```

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 thinned by one variable and extended using var_{0A}. But it is quite natural to have these restrictions: given that reifyA 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 thinned before being extended with related values to be substituted for the variable bound by the ‵λ.

```
R⟦λ⟧ : (t : Tm τ (Γ • σ))
       ( ∀ inc → rmodel VRBC uB uC →
         let ρA' = th[ SA.th ] (step refl) ρA ‵• var0A
```

```
    ρB′ = th[ SB.th ] inc ρB `• uB
    ρC′ = th[ SC.th ] inc ρC `• uC
  in R t ρA′ ρB′ ρC′) →
V_R ρA ρB ρC → R (`λ t) ρA ρB ρC
```

The other cases (omitted here) are just stating that, given the expected induction hypotheses, and the assumption that the three environments are $\mathcal{V}R$-related we can deliver a proof that fusion can happen on the compound expression.

As with simulation, we measure the utility of this framework by the way we can prove its fundamental lemma and then obtain useful corollaries. Once again, having carefully identified what the constraints should be, proving the fundamental lemma is not a problem:

Theorem 2 (Fundamental Lemma of Fusable Semantics)**.** *Given three Semantics $\mathcal{S}A$, $\mathcal{S}B$ and $\mathcal{S}C$ which are fusable with respect to the relations $\mathcal{V}R_{\mathsf{BC}}$ for values of $\mathcal{S}B$ and $\mathcal{S}C$, $\mathcal{V}R$ for environemnts and $\mathcal{C}R$ for computations, we have that:*

For any term t and environments ρA, ρB, and ρC, if the three environments are $\mathcal{V}R$-related then the semantics associated to t by $\mathcal{S}A$ using ρA followed by $\mathcal{S}B$ using ρB is $\mathcal{C}R$-related to the one associated to t by $\mathcal{S}C$ using ρC.

Proof. The proof is by structural induction on t using the combinators to assemble the induction hypotheses. □

The Special Case of Syntactic Semantics The translation from Syntactic to Semantics uses a lot of constructors as their own semantic counterpart, it is hence possible to generate evidence of Syntactic triplets being fusable with much fewer assumptions. We isolate them and prove the result generically to avoid repetition. A SyntacticFusable record packs the evidence for Syntactic semantics $synA$, $synB$ and $synC$. It is indexed by these three Syntactics as well as two relations corresponding to the $\mathcal{V}R_{BC}$ and $\mathcal{V}R$ ones of the Fusable framework. It contains the same $\mathcal{V}\mathsf{R}_{\bullet}$, $\mathcal{V}\mathsf{R}_{\mathsf{th}}$ and $\mathsf{R}[\![\mathsf{var}]\!]$ fields as a Fusable as well as a fourth one ($\mathsf{var}_{\mathsf{0BC}}$) saying that $synB$ and $synC$'s respective var_0s are producing related values.

```
varBC0 : rmodel V_RBC SynB.var0 SynC.var0
```

Theorem 3 (Fundamental Lemma of Fusable Syntactics)**.** *Given a SyntacticFusable relating three Syntactic semantics, we get a Fusable relating the corresponding Semantics where $\mathcal{C}R$ is the propositional equality.*

Proof. The proof relies on the way the translation from Syntactic to Semantics is formulated in section 5. □

Corollary 3 (Renaming-Renaming fusion)**.** *Given two renamings ρ from Γ to Δ and ρ' from Δ to Θ and a term t of type σ with free variables in Γ, we have that:*

```
thTm σ ρ′ (thTm σ ρ t) ≡ thTm σ (select ρ ρ′) t
```

Corollary 4 (Renaming-Substitution fusion)**.** *Given a renaming ρ from Γ to Δ, a substitution ρ' from Δ to Θ and a term t of type σ with free variables in Γ, we have that:*

```
subst ρ′ (thTm σ ρ t) ≡ subst (select ρ ρ′) t
```

Corollary 5 (Substitution-Renaming fusion)**.** *Given a substitution ρ from Γ to Δ, a renaming ρ' from Δ to Θ and a term t of type σ with free variables in Γ, we have that:*

```
thTm σ ρ′ (subst ρ t) ≡ subst (mapEnv (thTm _ ρ′) ρ) t
```

Corollary 6 (Substitution-Substitution fusion)**.** *Given two substitutitons, ρ from Γ to Δ and ρ' from Δ to Θ, and a term t of type σ with free variables in Γ, we have that:*

```
subst ρ′ (subst ρ t) ≡ subst (mapEnv (subst ρ′) ρ) t
```

These four lemmas are usually proven in painful separation. Here we discharged them by rapid successive instantiation of our framework, using the earlier results to satisfy the later constraints. We are not limited to Syntactic statements:

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 NBE with an adjusted environment.

Corollary 7 (Renaming-Normalise fusion)**.** *Given a renaming ρ from Γ to Δ, an environment of values ρ' from Δ to Θ such that they are all equal to themselves in the PER and a term t of type σ with free variables in Γ, we have that:*

```
PER σ (nbe ρ′ (thTm σ ρ t)) (nbe (select ρ ρ′) t)
```

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 (2002) in her detailed account of NBE for a STλC with explicit substitution.

Corollary 8 (Substitution-Normalise fusion)**.** *Given a substitution ρ from Γ to Δ, an environment of values ρ' from Δ to Θ such that all these values are equal to themselves and thinning and evaluation in ρ' commute, and a term t of type σ with free variables in Γ, we have that:*

```
PER σ (nbe ρ′ (subst ρ t)) (nbe (mapEnv (nbe ρ′) ρ) t)
```

10. Future and Related Work

The programming part of this work can be replicated in Haskell and a translation of the definitions is available in the paper's repository[2]. The subtleties of working with dependent types in Haskell (Lindley and McBride 2014) are outside the scope of this paper.

[2] https://github.com/gallais/type-scope-semantics

If the Tagless and Typeful NbE procedure derived in Haskell from our Semantics framework is to the best of our knowledge the first of its kind, Danvy, Keller and Puech have achieved a similar goal in OCaml (2013). But their formalisation uses parametric higher order abstract syntax (Chlipala 2008) freeing them from having to deal with variable binding, contexts and use models à la Kripke at the cost of using a large encoding. However we find scope safety enforced at the type level to be a helpful guide when formalising complex type theories. It helps us root out bugs related to fresh name generation, name capture or conversion from de Bruijn levels to de Bruijns indices.

This paper's method really shines in a simply typed setting but it is not limited to it: we have successfully used an analogue of our Semantics framework to enforce scope safety when implementing the expected traversals (renaming, substitution, untyped normalisation by evaluation and printing with names) for the untyped λ-calculus (for which the notion of type safety does not make sense) or Martin-Löf type theory. Apart from NbE (which relies on a non strictly-positive datatype), all of these traversals are total. Simulation and Fusion fundamental theorems akin to the ones proven in this paper also hold true. The common structure across all these variations suggests a possible generic scope safe treatment of syntaxes with binding.

This work is at the intersection of two traditions: the formal treatment of programming languages and the implementation of embedded Domain Specific Languages (eDSL) (Hudak 1996) both require the designer to deal with name binding and the associated notions of renaming and substitution but also partial evaluation (Danvy 1999), or even printing when emitting code or displaying information back to the user (Wiedijk 2012). The mechanisation of a calculus in a *meta language* can use either a shallow or a deep embedding (Svenningsson and Axelsson 2013; Gill 2014).

The well-scoped and well typed final encoding described by Carette, Kiselyov, and Shan (2009) allows the mechanisation of a calculus in Haskell or OCaml by representing terms as expressions built up from the combinators provided by a "Symantics". The correctness of the encoding relies on parametricity (Reynolds 1983) and although there exists an ongoing effort to internalise parametricity (Bernardy and Moulin 2013) in Type Theory, this puts a formalisation effort out of the reach of all the current interactive theorem provers.

Because of the strong restrictions on the structure our Models may have, we cannot represent all the interesting traversals imaginable. Chapman and Abel's work on normalisation by evaluation (2009; 2014) which decouples the description of the big-step algorithm and its termination proof is for instance out of reach for our system. Indeed, in their development the application combinator may *restart* the computation by calling the evaluator recursively whereas the Applicative constraint we impose means that we may only combine induction hypotheses.

McBride's original unpublished work (2005) implemented in Epigram (McBride and McKinna 2004) was inspired by Goguen and McKinna's Candidates for Substitution (1997). It focuses on renaming and substitution for the simply typed λ-calculus and was later extended to a formalisation of System F (Girard 1972) in Coq (The Coq development team 2004) by Benton, Hur, Kennedy and McBride (2012). Benton et al. both implement a denotational semantics for their language and prove the properties of their traversals. However both of these things are done in an ad-hoc manner: the meaning function associated to their denotational semantics is not defined in terms of the generic traversal and the proofs are manually discharged one by one. They also 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.

11. 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, 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

A. Abel and J. Chapman. Normalization by evaluation in the delay monad. *MSFP 2014*, 2014.

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

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

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

N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride. Strongly typed term representations in Coq. *JAR*, 49(2):141–159, 2012.

U. Berger. Program extraction from normalization proofs. In *TLCA*, pages 91–106. Springer, 1993.

J.-P. Bernardy and G. Moulin. Type-theory in color. *SIGPLAN Notices*, 48(9):61–72, 2013.

J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated. *JFP*, 2009.

J. M. Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

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

C. 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.

T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *MSCS*, 7(01):75–94, 1997.

N. A. Danielsson and U. Norell. Parsing mixfix operators. In *IFL*, pages 80–99. Springer, 2011.

O. Danvy. Type-directed partial evaluation. In *Partial Evaluation*, pages 367–411. Springer, 1999.

O. Danvy, C. Keller, and M. Puech. Tagless and typeful normalization by evaluation using generalized algebraic data types. 2013.

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

P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical Frameworks*, 2:6, 1991.

R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. *SIGPLAN Notices*, 47(12):117–130, 2013.

A. Gill. Domain-specific languages and code synthesis using Haskell. *Queue*, 12(4):30, 2014.

J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

H. Goguen and J. McKinna. Candidates for substitution. *LFCS, Edinburgh Techreport*, 1997.

J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471. ACM, 1994.

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

J. Hughes. The design of a pretty-printing library. In *AFP Summer School*, pages 53–96. Springer, 1995.

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

S. Lindley and C. McBride. Hasochism. *SIGPLAN Notices*, 48(12): 81–92, 2014.

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

The Coq development team. *The Coq proof assistant reference manual*, 2004. URL `http://coq.inria.fr`. Version 8.0.

C. McBride. Type-preserving renaming and substitution. 2005.

C. McBride and J. McKinna. The view from the left. *JFP*, 14(01): 69–111, 2004.

J. C. Mitchell. *Foundations for programming languages*, volume 1. MIT press, 1996.

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

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

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

J. C. Reynolds. Types, abstraction and parametric polymorphism. 1983.

J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, pages 21–36. Springer, 2013.

P. Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

F. Wiedijk. Pollack-inconsistency. *ENTCS*, 285:85–100, 2012.