Syntaxes with Binding, Their Programs, and Proofs

$$\frac{\Box\,(\mathcal{V}\,\sigma \Rightarrow \mathcal{C}\,\tau)}{\mathcal{C}\,(\sigma \to \tau)}$$

Allais Guillaume

Syntaxes with Binding, Their Programs, and Proofs

Guillaume ALLAIS

January 17, 2019

Contents

Chapter 1

Introduction

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

The definition of scope-and-type safe representations is naturally only a start: once the programmer has access to a good representation of the language they are interested in, they will then want to (re)implement standard traversals manipulating terms. Renaming and substitution are the two most typical examples of such traversals. Other common examples include an evaluator, a printer for debugging purposes and eventually various compilation passes such as a continuation passing style transformation or some form of inlining. Now that well-typedness and well-scopedness are enforced statically, all of these traversals have to be implemented in a type and scope safe manner.

The third hurdle is only faced by those that want really high assurance or whose work is to study a language's meta-theory: they need to prove theorems about these traversals. The most common statements are simulation lemmas stating that two semantics transport related inputs to related outputs, or fusion lemmas demonstrating that the sequential execution of two semantics is equivalent to a single traversal by a third one. These proofs often involve a wealth of auxiliary lemmas which are known to be true for all syntaxes but have to be re-proven for every new language e.g. identity and extensionality lemmas for renaming and substitution.

Despite the large body of knowledge in how to use types to define well formed syntax (see the Related Work in chapter 7), it is still necessary for the working DSL designer or formaliser to redefine essential functions like renaming and substitution for each new syntax, and then to reprove essential lemmas about those functions.

1.1 Our Contributions

In this thesis we address all three challenges by applying the methodology of datatype-genericity to programming and proving with syntaxes with binding. We ultimately give a binding-aware syntax for well typed and scoped DSLs; we spell out a set of sufficient constraints entailing the existence of a fold-like type-and-scope preserving traversal; and we provide proof frameworks to either demonstrate that two traversals are in simulation or that they can be fused together into a third one.

The first part of this work focuses on the simply-typed lambda calculus. We identify a large catalogue of traversals which can be refactored into a common scope aware fold-like function. Once this shared structure has been made visible, we can design proof frameworks allowing us to show that some traversals are related.

The second part builds on the observation that the shape of the constraints we see both in the definition of the generic traversal and the proof frameworks are in direct correspondence with the language's constructors. This pushes us to define a description language for syntaxes with binding and to *compute* the constraints from such a description. We can then write generic programs for *all* syntaxes with binding and state and prove generic theorems characterising these programs.

1.2 Source Material

This thesis is based on the content of two fully-formalised published papers which correspond roughly to part one and two: "Type-and-scope Safe Programs and Their Proofs" (Allais et al. [2017a,b]) and "A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs" (Allais et al. [2018a,b]).

The study of variations on normalisation by evaluation in chapter 5 originated from the work on a third paper "New equations for neutral terms" (Allais et al. [2013]) which, combined with McBride's Kit for renaming and substitution (2005), led to the identification of a shared structure between renaming, substitution and the model construction in normalisation by evaluation.

The first consequent application of this work is our solution to the POPLMark Reloaded challenge currently under review for publication (Abel et al. [2017, 2018]) for which we formalised a proof of strong normalisation for the simply-typed lambda-calculus (and its extension with sum types).

Chapter 2

Introduction to Agda

The techniques and abstractions defined in this thesis are language-independent: all the results can be replicated in any Martin Löf Type Theory (1982) equipped with inductive families (Dybjer [1994]). In practice, all of the content of this thesis has been formalised in Agda (Norell [2009]) so we provide a (brutal) introduction to dependently-typed programming in Agda. It is a dependently-typed programming language based on Martin-Löf Type Theory with inductive families, induction-recursion (Dybjer and Setzer [1999]), copattern-matching (Abel et al. [2013b]) and sized types (Abel [2010]).

Mixfix Identifiers We use Agda's mixfix operator notation (Danielsson and Norell [2011]) where underscores denote argument positions. See e.g. the notation _×_ for the type of pairs defined in section 2.1.

Syntax Highlighting We rely on Agda's LaTeX backend to produce syntax highlighting for all the code fragments in this thesis. The convention is as follows: keywords are highlighted in orange, data constructors in green, record fields in pink, types and functions in blue while bound variables are *black*.

Implicit Generalisation The latest version of Agda supports ML-style implicit prenex polymorphism and we make heavy use of it: every unbound variable should be considered implicitly bound at the beginning of the telescope.

2.1 Data and (co)pattern matching

As is customary, we start our introduction to dependently-typed programming with the natural numbers. They are defined as an inductive type with two conctructors: zero and successor.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Record Types Agda also supports record types; they are defined by their list of fields. Unlike inductive types they enjoy η-rules. That is to say that any two values of the same record type are judgmentally equal whenever all of their fields' values are.

We define the unit type (⊤) which has one constructor (tt) but no field together with the pair type _×_ which has an infix constructor _,_ and two fields: its first (fst) and second (snd) projections.

```
record ⊤ : Set where
  constructor tt
```

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field fst : A; snd : B
```

Defining a record type R also results in the definition of a module *R* parameterised by a value of type R and containing a projection function for each field. We can open it to make these projections available to the outside world, or use *R*-qualified names.

Recursive Functions Definitions taking values of an inductive type as argument are defined by pattern matching in a style familiar to Haskell programmers: one lists clauses assuming a first-match semantics. If the patterns on the left hand side are covering all possible cases and the recursive calls are structurally smaller, the function is total. All definitions have to be total in Agda.

The main difference with Haskell is that in Agda, we can perform so-called "large elimination": we can define a Set by pattern-matching on a piece of data. Here we use our unit and pair records to define a tuple of size *n* by recursion over *n*: a (zero -Tuple) is empty whilst a (suc n -Tuple) is a value paired with an (*n* -Tuple).

```
_-Tuple_ : ℕ → Set → Set
zero  -Tuple A = ⊤
suc n -Tuple A = A × (n -Tuple A)
```

Because types can now depend on the shape of values, in a definition by pattern matching each clause has a type *refined* based on the patterns which appear on the left hand side. This will be familiar to Haskell programmers used to manipulating Generalized Algebraic Data Types (GADTs). Let us see two examples of a type being refined based on the pattern appearing in a clause.

First, we introduce replicate which takes a natural number *n* and a value *a* of type *A* and returns an (*n* -Tuple) by duplicating *a*. The return type of replicate reduces to ⊤ when the natural number is zero and (*A* × (*n* -Tuple *A*)) when it is (suc *n*).

```
replicate : ∀ n → A → n -Tuple A
replicate zero    a = tt
replicate (suc n) a = a , replicate n a
```

Second, we define map^-Tuple which takes a function and applies it to each one of the elements in a -Tuple. Both the type of the -Tuple argument and the -Tuple return type are refined based on the pattern the natural number matches. In the second clause, this tells us the -Tuple argument is a pair, allowing us to match on it with the pair constructor _,_.

```
map^-Tuple : ∀ n → (A → B) → n -Tuple A → n -Tuple B
map^-Tuple zero    f tt        = tt
map^-Tuple (suc n) f (a , as) = f a , map^-Tuple n f as
```

Dependent Record Types Record types can be dependent, i.e. the type of later fields can depend on that of former ones. We define a Tuple as a natural number (its length) together with a (length -Tuple) of values (its content).

```
record Tuple (A : Set) : Set where
  constructor mkTuple
  field length  : ℕ
        content : length -Tuple A
```

In Agda, as in all functional programming languages, we can define anonymous functions by using a λ-abstraction. Additionally, we can define anonymous (co)pattern-matching functions by using (λ where) followed by an indented block of clauses. We use here copattern-matching, that is to say that we define a Tuple in terms of the observations that one can make about it: we specify its length first, and then its content. We use postfix projections (hence the dot preceding the field's name).

```
map^Tuple : (A → B) → Tuple A → Tuple B
map^Tuple f as = λ where
  .length  → as .length
  .content → map^-Tuple (as .length) f (as .content)
```

When record values are going to appear in types, it is often a good idea to define them by copattern-matching: this prevents the definition from being unfolded eagerly thus making the goal more readable during interactive development.

Strict Positivity In order to rule out definitions leading to inconsistencies, all datatype definitions need to be strictly positive. Although a syntactic criterion originally (its precise definition is beyond the scope of this discussion), Agda goes beyond by recording internally whether functions use their arguments in a strictly positive manner. This allows us to define types like rose trees where the subtrees of a node are stored in a Tuple, a function using its Set argument in a strictly positive manner.

```
data Rose (A : Set) : Set where
  leaf  : A → Rose A
  node : Tuple (Rose A) → Rose A
```

2.2 Sized Types and Termination Checking

If we naïvely define rose trees like above then we quickly realise that we cannot re-use higher order functions on Tuple to define recursive functions on Rose. As an example, let us consider map^Rose. In the node case, the termination checker does not realise that the partially applied recursive call (map^Rose *f*) passed to map^Tuple will only ever be used on subterms. We need to use an unsafe TERMINATING pragma to force Agda to accept the definition.

```
{-# TERMINATING #-}
map^Rose : (A → B) → Rose A → Rose B
map^Rose f (leaf a)    = leaf (f a)
map^Rose f (node rs) = node (map^Tuple (map^Rose f) rs)
```

This is not at all satisfactory: we do not want to give up safety to write such a simple traversal. The usual solution to this issue is to remove the level of indirection introduced by the calll to map^Tuple by mutually defining with map^Rose an inlined version of (map^Tuple (map^Rose *f*)).

```
mutual

  map^Rose : (A → B) → Rose A → Rose B
  map^Rose f (leaf a)                  = leaf (f a)
  map^Rose f (node (mkTuple n rs)) = node (mkTuple n (map^Roses n f rs))

  map^Roses : ∀ n → (A → B) → n -Tuple (Rose A) → n -Tuple (Rose B)
  map^Roses zero    f rs        = tt
  map^Roses (suc n) f (r , rs) = map^Rose f r , map^Roses n f rs
```

This is, of course, still unsatisfactory: we need to duplicate code every time we want to write a traversal! By using sized types, we can have a more compositional notion of termination checking: the size of a term is reflected in its type. No matter how many levels of indirection there are between the location where we are peeling off a constructor and the place where the function is actually called recursively, as long as the intermediate operations are size-preserving we know that the recursive call will be legitimate.

Writing down the sizes explicitly, we get the following implementation. Note that in (map^Tuple (map^Rose *j f*)), *j* (bound in node) is smaller than *i* and therefore the recursive call is justified.

```
data Rose (A : Set) (i : Size) : Set where
  leaf   : A → Rose A i
  node : (j : Size< i) → Tuple (Rose A j) → Rose A i


map^Rose : ∀ i → (A → B) → Rose A i → Rose B i
map^Rose i f (leaf a)      = leaf (f a)
map^Rose i f (node j rs) = node j (map^Tuple (map^Rose j f) rs)
```

In practice we make the size arguments explicit in the types but implicit in the terms. This leads to programs that look just like our ideal implementation, with the added bonus that we have now *proven* the function to be total.

```
data Rose (A : Set) (i : Size) : Set where
  leaf   : A → Rose A i
  node : {j : Size< i} → Tuple (Rose A j) → Rose A i
```

```
map^Rose : ∀ {i} → (A → B) → Rose A i → Rose B i
map^Rose f (leaf a)     = leaf (f a)
map^Rose f (node rs) = node (map^Tuple (map^Rose f) rs)
```

2.3 Working with Indexed Families

On top of the constructs provided by the language itself, we can define various domain specific languages (DSL) which give us the means to express ourselves concisely. We are going to manipulate a lot of indexed families representing scoped languages so we give ourselves a few combinators corresponding to the typical operations we want to perform on them.

First, noticing that most of the time we silently thread the current scope, we lift the function space pointwise with _⇒_.

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

Second, the _⊢_ combinator makes explicit the *adjustment* made to the index by a function, conforming to the convention (see e.g. Martin-Löf [1982]) of mentioning only context *extensions* when presenting judgements and write $(f \vdash P)$ where f is the modification and P the indexed Set it operates on.

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

Although it may seem surprising at first to define binary infix operators as having arity three, they are meant to be used partially applied, surrounded by ∀[_] which turns an indexed Set into a Set by implicitly quantifying over the index.

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

We make _⇒_ associate to the right as one would expect and give it the highest precedence level as it is the most used combinator. These combinators lead to more readable type declarations. For instance, the compact expression ∀[suc ⊢ (P ⇒ Q) ⇒ R] desugars to the more verbose type ∀ {i} → (P (suc i) → Q (suc i)) → R i.

As the combinators act on the *last* argument of any indexed family, this inform our design: our notions of variables, languages, etc. will be indexed by their kind first and scope second. This will be made explicit in the definition of −Scoped in fig. 3.4.

Part I

Type and Scope Preserving Programs, and Their Proofs

Chapter 3

Intrinsically Scoped and Typed Syntax

A programmer implementing an embedded language with bindings has a wealth of possibilities. However, should they want to be able to inspect the terms produced by their users in order to optimise or even compile them, they will have to work with a deep embedding.

3.1 A Primer on Scope And Type Safe Terms

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

An inductive type is the fixpoint of an endofunctor on Set. Similarly, an inductive family is the fixpoint of an endofunctor on (Set $\to$ Set). Using inductive families to enforce scope safety, we get the following definition of the untyped λ-calculus: $T(F) = \lambda X \in \mathsf{Set}.\ X + (F(X) \times F(X)) + F(1 + X)$. This endofunctor offers a choice of three constructors. The first one corresponds to the variable case; it packages an inhabitant of X, the index Set. The second corresponds to an application node; both the function and its argument live in the same scope as the overall expression. The third corresponds to a λ-abstraction; it extends the current scope with a fresh variable. The language is obtained as the fixpoint of T:

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

Figure 3.1: Well-Scoped Untyped Lambda Calculus as the Fixpoint of a Functor

Since ‘UTLC’ is an endofunction on Set, it makes sense to ask whether it is also a functor and a monad. Indeed it is, as Altenkirch and Reus have shown. The functorial action corresponds to renaming, the monadic 'return' corresponds to the use of variables, and the monadic 'join' corresponds to substitution. The functor and monad laws correspond to well known properties from the equational theories of renaming and substitution. We will revisit these properties below in chapter 9.

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

We pick as our J the category whose objects are inhabitants of (List I) (I is a parameter of the construction) and whose morphisms are thinnings (see section 4.3). This (List I) is intended to represent the list of the sort (/ kind / types depending on the application) of the de Bruijn variables in scope. We can recover an untyped approach by picking I to be the unit type. Given this typed setting, our functors take an extra I argument corresponding to the type of the expression being built. This is summed up by the large type (I –Scoped) defined in fig. 3.4.

3.2 The Calculus and Its Embedding

We work with a deeply embedded simply typed λ-calculus (STλC). It has `'Unit` and `'Bool` 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. We describe the types both as a grammar and the corresponding inductive type in Agda in fig. 3.2.

```
⟨Type⟩  ::=  'Unit | 'Bool
          |  ⟨Type⟩ '→ ⟨Type⟩
```

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

Figure 3.2: Types used in our Running Example

The language's constructs are those one expects from a λ-calculus: variables, application and λ-abstraction. We then have a constructor for values of the unit type but no eliminator (a term of unit type carries no information). Finally, we have two constructors for boolean values and the expected if-then-else eliminator.

```
⟨Term⟩  ::=  x | ⟨Term⟩⟨Term⟩ | λx.⟨Term⟩
          |  ()
          |  'true | 'false | 'if⟨Term⟩then⟨Term⟩else⟨Term⟩
```

Figure 3.3: Grammar of our Language

Well Scoped and Typed by Construction

To talk about the variables in scope and their type, we need *contexts*. We choose to represent them as lists of types; [] denotes the empty list and (σ :: Γ) the list Γ extended with a fresh variable of type σ. Because we are going to work with a lot of well typed and well scoped families, we defined (I –Scoped) as the set of type and scope indexed families.

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

Figure 3.4: Typed and Scoped Definitions

Our first example of a type and scope indexed family is Var, the type of Variables. A variable is a position in a typing context, represented as a typed de Bruijn (1972) index. This amounts to an inductive definition of context membership. We use the combinators defined in section 2.3 to show only local changes to the context.

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

Figure 3.5: Well Scoped and Typed de Bruijn indices

The z (for zero) constructor refers to the nearest binder in a non-empty scope. The s (for successor) constructor lifts an existing variable in a given scope to the extended scope where an extra variable has been bound. The constructors' types respectively normalise to:

```
z : ∀ {σ Γ} → Var σ (σ :: Γ)        s : ∀ {σ τ Γ} → Var σ Γ → Var σ (τ :: Γ)
```

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 ‘lam.

```
data Term : Type −Scoped where
  ‘var  : ∀[ Var σ ⇒ Term σ ]
  ‘app : ∀[ Term (σ ‘→ τ) ⇒ Term σ ⇒ Term τ ]
  ‘lam : ∀[ (σ ::_) ⊢ Term τ ⇒ Term (σ ‘→ τ) ]
  ‘one : ∀[ Term ‘Unit ]
  ‘tt ‘ff : ∀[ Term ‘Bool ]
  ‘ifte  : ∀[ Term ‘Bool ⇒ Term σ ⇒ Term σ ⇒ Term σ ]
```

Figure 3.6: Well Scoped and Typed Calculus

Chapter 4

Refactoring Common Traversals

Once they have a good representation for their language, they will have to (re)implement a great number of traversals doing such mundane things as renaming, substitution, or partial evaluation. Should they want to get help from the typechecker in order to fend off common bugs, they will have opted for inductive families (Dybjer [1991]) to enforce precise invariants. But the traversals now have to be invariant preserving too!

4.1 McBride's Kit

In an unpublished manuscript, McBride (2005) observes the similarity between the types and implementations of renaming and substitution for the simply typed λ-calculus (STλC) in a dependently typed language as shown in fig. 4.1 (we focus only on ‘var, ‘app, and ‘lam for the moment). 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 s (which is essentially weakening for variables) whereas for a substitution we need a weakening operation for terms. It can be given by renaming via the successor constructor (ren (pack s));

3. also in the λ case, when pushing a renaming or a substitution under a binder we must extend it to ensure that the variable bound by the λ is mapped to itself. For renaming this involves its extension by the zeroth variable z whereas for subsitutions we must extend by the zeroth variable seen as a term (‘var z).

McBride then defines a notion of “Kit” abstracting these differences. Rather than considering Var and Tm in isolation as different types of environment values, he considers ♦, an arbitrary (Type −Scoped) and designs three constraints:

```
ren : (Γ −Env) Var Δ → Tm σ Γ → Tm σ Δ
ren ρ (‘var v)   = ‘var (lookup ρ v)
ren ρ (‘app f t) = ‘app (ren ρ f) (ren ρ t)
ren ρ (‘lam b)   = ‘lam (ren ρ′ b)
  where ρ′ = (s <$> ρ) • z

sub : (Γ −Env) Tm Δ → Tm σ Γ → Tm σ Δ
sub ρ (‘var v)   = lookup ρ v
sub ρ (‘app f t) = ‘app (sub ρ f) (sub ρ t)
sub ρ (‘lam b)   = ‘lam (sub ρ′ b)
  where ρ′ = (ren (pack s) <$> ρ) • ‘var z
```

Figure 4.1: Renaming and Substitution for the STλC

1. One should be able to turn any environment value into a term of the same type and defined in the same scope (var);
2. One should be able to craft a fresh environment value associated to the zeroth variable of a scope (zro);
3. One should be able to embed environment values defined in a given scope into ones in a scope extended with a fresh variable (wkn).

```
record Kit (♦ : Type −Scoped) : Set where
  field var  : ∀[ ♦ σ ⇒ Tm σ ]
        zro  : ∀[ (σ ::_) ⊢ ♦ σ ]
        wkn  : ∀[ ♦ τ ⇒ (σ ::_) ⊢ ♦ τ ]
```

Figure 4.2: Kit as a set of constraints on ♦

Whenever these constraints are met we can define a type and scope preserving traversal which is based on an environment of ♦-values. This is the fundamental lemma of Kits stated and proved in fig. 4.3.

Thankfully, we can indeed recover renaming and substitution as two instances of the fundamental lemma of Kits. We start with the Kit for renaming and ren defined this time as a corrolary of kit

Just like we needed ren to define sub, once we have recovered ren we can define the Kit for substitution.

```
kit : Kit ♦ → (Γ –Env) ♦ Δ → Tm σ Γ → Tm σ Δ
kit K ρ (‘var v)   = K .var (lookup ρ v)
kit K ρ (‘app f t) = ‘app (kit K ρ f) (kit K ρ t)
kit K ρ (‘lam b)   = ‘lam (kit K ρ′ b)
  where ρ′ = (K .wkn <$> ρ) • K .zro
```

Figure 4.3: Fundamental lemma of Kit

```
ren^Kit : Kit Var
ren^Kit .var  = ‘var              ren : Thinning Γ Δ → Tm σ Γ → Tm σ Δ
ren^Kit .zro  = z                 ren = kit ren^Kit
ren^Kit .wkn = s
```

Figure 4.4: Kit for Renaming, Renaming as a Corrolary of kit

```
sub^Kit : Kit Tm
sub^Kit .var  = id                sub : (Γ –Env) Tm Δ → Tm σ Γ → Tm σ Δ
sub^Kit .zro  = ‘var z            sub = kit sub^Kit
sub^Kit .wkn = ren (pack s)
```

Figure 4.5: Kit for Substitution, Substitution as a Corrolary of kit

4.2 Opportunities for Further Generalizations

After noticing that renaming and substitution fit the pattern, it is natural to wonder about other traversals.

The evaluation function used in normalization by evaluation, although not fitting *exactly* in the Kit-based approach, relies on the same general structure. The variable case is nothing more than a lookup in the environment; the application case is defined by combining the results of two structural calls; and the lambda case corresponds to the evaluation of the lambda's body in an extended context provided that we can get a value for the newly-bound variable. Ignoring for now the definitions of APP and LAM, we can see the similarities in fig. 4.6.

Outline

Building on this observation, our contributions here are twofold:

- We generalise the “Kit” approach from syntax to semantics bringing operations like normalisation by evaluation (cf. fig. 4.6) but also printing with a name supply, or continutation passing style translation into our framework.

```
nbe : (Γ ─Env) Val Δ → Tm σ Γ → Val σ Δ
nbe ρ (‘var v)   = lookup ρ v
nbe ρ (‘app f t) = APP (nbe ρ f) (nbe ρ t)
nbe ρ (‘lam t)   = LAM (λ re v → nbe ((th^Val re <$> ρ) • v) t)
```

Figure 4.6: Normalisation by Evaluation for the STλC

- We prove generic results about simulations between and fusions of semantics given by, and enabled by, Kit.

We start by introducing 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).

4.3 A Generic Notion of Environment

All the semantics we are interested in defining associate to a term t of type Term σ Γ, a value of type $C\ \sigma\ \Delta$ given an interpretation $\mathcal{V}\ \Delta\ \tau$ for each one of its free variables τ in Γ. We call the collection of these interpretations an $\mathcal{V}$-(evaluation) environment. We leave out $\mathcal{V}$ 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 of evaluation environment generically for any (I ─Scoped) family of values.

Formally, this translates to $\mathcal{V}$-environments being the pointwise lifting of the relation $\mathcal{V}$ 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. identity and select) defined later on is vastly simplified by this rather simple decision.

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 type family the values correspond to and it empowers us to define

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

Figure 4.7: Generic Notion of Environment

environments by copattern-matching (Abel et al. [2013a]) thus defining environments by their use cases.

The definition of the empty environment uses an absurd match (()): given the definition of Var in fig. 3.5, it should be pretty clear that there can never be a value of type (Var σ []).

```
ε : ([] −Env) 𝒱 Δ
lookup ε ()
```

Figure 4.8: Empty Environment

The environment extension definition proceeds by pattern-matching on the variable: if it z then we return the newly-added value, otherwise we are referring to a value in the original environment and can simply look it up.

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

Figure 4.9: Environment Extension

The Category of Thinnings

A key instance of environments playing a predominant role in this paper is the notion of thinning. 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), context inclusions, or just weakenings (Altenkirch et al. [1995]). A thinning (Thinning Γ Δ) is an environment pairing each variable of type σ in Γ to one of the same type in Δ.

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 thinnings 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. The extra flexibility will not get in

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

Figure 4.10: Thinnings: A Special Case of Environments

our way, and permits a representation as a function space which grants us monoid laws “for free” as per Jeffrey’s observation (2011).

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.

```
identity : Thinning Γ Γ                extend : Thinning Γ (σ :: Γ)
lookup identity k = k                  lookup extend v = s v

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

Figure 4.11: Examples of Thinning Combinators

The □ combinator turns any (List I)-indexed Set into one that can absorb thinnings. This is accomplished by abstracting over all possible thinnings from the current scope, akin to an S4-style necessity modality. The axioms of S4 modal logic incite us to observe that the functor □ is a comonad: extract applies the identity Thinning to its argument, and duplicate is obtained by composing the two Thinnings we are given (select defined in fig. 4.11 corresponds to transitivity in the special case where $\mathcal{V}$ is Var). The expected laws hold trivially thanks to Jeffrey’s trick mentioned above.

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

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

Figure 4.12: The □ Functor is a Comonad

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

```
Thinnable : (List I → Set) → Set          th^□ : Thinnable (□ T)
Thinnable T = ∀[ T ⇒ □ T ]                th^□ = duplicate
```

Figure 4.13: Thinning Principle and the Cofree Thinnable □

Constant families are trivially Thinnable. 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.

```
th^const : Thinnable (const A)            th^Var : Thinnable (Var i)
th^const a _ = a                          th^Var v ρ = lookup ρ v

th^Env : (∀ {i} → Thinnable (𝒱 i)) → Thinnable ((Γ –Env) 𝒱)
lookup (th^Env th^𝒱 ρ ren) k = th^𝒱 (lookup ρ k) ren
```

Figure 4.14: Thinnable Instances for Variables and Environments

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.

4.4 Semantics and Their Generic Evaluators

The upcoming sections demonstrate that renaming, substitution, and printing with names 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.

Broadly speaking, a semantics turns our deeply embedded abstract syntax trees into the shallow embedding of the corresponding parametrised higher order abstract syntax term (Chlipala [2008]). We get a choice of useful type-and-scope safe traversals by using different 'host languages' for this shallow embedding.

A semantics is parametrised by two (Type –Scoped) type families $\mathcal{V}$ and $\mathcal{C}$. Realisation of a semantics will produce a computation in $\mathcal{C}$ for every term whose variables are assigned values in $\mathcal{V}$. Just as an environment interprets variables in a model, a computation gives a meaning to terms into a model. We can define –Comp to make this parallel explicit.

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$)\ \mathcal{V}\ \Delta \rightarrow (\Gamma$ –Comp$)\ \mathcal{C}\ \Delta)$. In cases such

```
_−Comp : List Type → Type −Scoped → List Type → Set
(Γ −Comp) 𝒞 Δ = ∀ {σ} → Term σ Γ → 𝒞 σ Δ
```

Figure 4.15: Generic Notion of Computation

as substitution or normalisation by evaluation, 𝒱 and 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.

Concretely, we define Semantics as a record packing the properties these families need to satisfy for the evaluation function to exist.

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

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

```
th^𝒱 : 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 lam. The variable case bridges the gap between the fact that the environment translates variables into values 𝒱 but the evaluation function returns computations 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. This is concisely expressed by the type (□ (𝒱 σ ⇒ 𝒞 τ)).

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 𝒱 = 𝒞 (normalisation by evaluation for instance), we recover the usual Kripke structure.

```
lam : ∀[ □ (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. For instance, the semantical counterpart of application is an operation that takes a representation of a function and a representation of an argument and produces a representation of the result.

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

```
tt   : ∀[ C ‘Bool ]
ff   : ∀[ C ‘Bool ]
ifte : ∀[ C ‘Bool ⇒ C σ ⇒ C σ ⇒ C σ ]
```

The type we chose for lam 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 semantics “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 Fundamental (𝒮 : Semantics 𝒱 C) where
  open Semantics 𝒮

  lemma : (Γ −Env) 𝒱 Δ → (Γ −Comp) C Δ
  lemma ρ (‘var v)      = var (lookup ρ v)
  lemma ρ (‘app t u)  = app (lemma ρ t) (lemma ρ u)
  lemma ρ (‘lam t)      = lam (λ re u → lemma (th^Env th^𝒱 ρ re • u) t)
  lemma ρ ‘one          = one
  lemma ρ ‘tt            = tt
  lemma ρ ‘ff            = ff
  lemma ρ (‘ifte b l r) = ifte (lemma ρ b) (lemma ρ l) (lemma ρ r)
```

Figure 4.16: Fundamental Lemma of Semantics

4.5 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 fundamental lemma of Syntactic, a function turning its inhabitants into associated Semantics.

The shape of lam or one should not trick the reader into thinking that this definition performs some sort of η-expansion: the fundamental lemma indeed only ever uses one of these when the evaluated term’s head constructor is already respectively a ‘lam or a ‘one. It is therefore absolutely possible to define renaming or substitution using this approach. We can now port McBride’s definitions to our framework.

```
record Syntactic (𝒯 : Type −Scoped) : Set where
  field zro   : ∀[ (σ ::_) ⊢ 𝒯 σ ]
        th^𝒯 : Thinnable (𝒯 σ)
        var   : ∀[ 𝒯 σ ⇒ Term σ ]


module Fundamental (𝒮 : Syntactic 𝒯) where

  open Syntactic 𝒮

  lemma : Semantics 𝒯 Term
  Semantics.th^𝒱 lemma = th^𝒯
  Semantics.var   lemma = var
  Semantics.lam   lemma = λ b → ‘lam (b extend zro)
  Semantics.app   lemma = ‘app
  Semantics.one   lemma = ‘one
  Semantics.tt    lemma = ‘tt
  Semantics.ff    lemma = ‘ff
  Semantics.ifte  lemma = ‘ifte
```

Figure 4.17: Every Syntactic gives rise to a Semantics

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 4.3) 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.

```
Syn^Ren : Syntactic Var
Syn^Ren .zro   = z                    th^Term : Thinnable (Term σ)
Syn^Ren .th^𝒯 = th^Var               th^Term t ρ = eval Renaming ρ t
Syn^Ren .var   = ‘var
```

Figure 4.18: Thinning as a Syntactic Instance

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.

```
Syn^Sub : Syntactic Term
Syn^Sub .zro    = ‘var z          sub : (Γ –Env) Term Δ → Term σ Γ → Term σ Δ
Syn^Sub .th^𝒯  = th^Term         sub ρ t = eval Substitution ρ t
Syn^Sub .var    = id
```

Figure 4.19: Parallel Substitution as a Syntactic Instance

4.6 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], Bernardy [2017]). 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 (here we use M for the state monad threading that name supply). 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.

```
Name : I –Scoped                    Printer : I –Scoped
Name σ Γ = String                   Printer σ Γ = M String
```

Figure 4.20: Names and Printer for the Printing Semantics

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.

```
Printing : Semantics Name Printer
```

Because the output type is not scoped in any way, thinning for Names (th^𝒱) is trivial: we return the same name. The variable case (var) is a bit more interesting: after looking up a variable's Name in the environment, we use return to produce the trivial Printer constantly returning that name.

```
Printing .th^𝒱 = th^const
Printing .var   = return
```

As often, the case for λ-abstraction (lam) is the most interesting one. We first use fresh to generate a Name for the newly-bound variable, then run the printer for the body in the environment extended with that fresh name and finally build a string combining the name and the body together.

```
Printing .lam {σ} mb = do
  x ← fresh; b ← mb (bind σ) x
  return $ "λ" ++ x ++ ".  " ++ b
```

We then have a collection of base cases for the data constructors of type ‘Unit and ‘Bool. These give rise to constant printers.

```
Printing .one = return "<>"
Printing .tt  = return "true"
Printing .ff  = return "false"
```

Finally we have purely structural cases: we run the printers for each of the subparts and put the results together, throwing in some extra parenthesis to guarantee that the result is unambiguous.

```
Printing .app mf mt = do
  f ← mf; t ← mt
  return $ parens f ++ "  " ++ t
Printing .ifte mb ml mr = do
  b ← mb; l ← ml; r ← mr
  return $ "if  " ++ parens b ++
           "  then  " ++ parens l ++ "  else  " ++ parens r
```

The fundamental lemma of Semantics 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 functions, i or j 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. We can define it using sequenceA because environments are traversable (McBride and Paterson [2008]). The definition of print follows.

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 identity 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.

```
init : M ((Γ −Env) Name Γ)
init = sequenceA (pack (const fresh))
```

```
printer : Term σ Γ → M String
printer t = do
  ρ ← init
  Fundamental.lemma Printing ρ t
```

```
print : Term σ Γ → String
print t = proj₁ $ printer t names
```

Figure 4.21: Printer

```
_ : print (Term (σ ‘→ α) (α :: β :: []) ∋ ‘lam (‘var (s z))) ≡ "λc.   a"
_ = refl
```

Figure 4.22: Printing an Open Term

Chapter 5

Variations on 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 (Berger and Schwichtenberg [1991], Berger [1993], Coquand and Dybjer [1997], Coquand [2002]). The process is based on a model construction describing a family of types by induction on its Type index. Two procedures are then defined: the first (eval) constructs an element of $C\ \sigma\ \Gamma$ provided a well typed term of the corresponding Term $\sigma\ \Gamma$ type whilst the second (reify) extracts, in a type-directed manner, normal forms Nf $\sigma\ \Gamma$ from elements of the model $C\ \sigma\ \Gamma$. 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.

Reduction Rules

Thanks to Renaming and Substitution respectively, we can formally define η-expansion for functions and β-reduction.

```
eta : ∀[ Term (σ ‘→ τ) ⇒ Term (σ ‘→ τ) ]
eta t = ‘lam (‘app (th^Term t extend) (‘var z))
```

```
_⟨_/0⟩ : ∀[ (σ ::_) ⊢ Term τ ⇒ Term σ ⇒ Term τ ]
t ⟨ u /0⟩ = sub ((‘var <$> identity) • u) t
```

Figure 5.1: η-expansion and β-reduction in terms or th^Term and sub

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 `‘Unit` equal to ‘one.

$$\frac{t : \mathsf{Term}\ (\sigma\ \text{`}\!\!\to \tau)\ \Gamma}{t \rightsquigarrow \mathsf{eta}\ t}\eta_1 \qquad \frac{t : \mathsf{Term}\ \text{`}\mathsf{Unit}\ \Gamma}{t \rightsquigarrow \text{`}\mathsf{one}}\eta_2 \qquad \frac{}{\text{`}\mathsf{app}\ (\text{`}\mathsf{lam}\ t)\ u \rightsquigarrow t\ \langle\ u\ /0\rangle}\beta$$

Figure 5.2: $\beta\eta$ Rules for our Calculus

The β-rule is the main driver for actual computation, but the presence of an inductive data type (‘Bool) and its eliminator (‘ifte) 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.

$$\frac{}{\text{`}\mathsf{ifte}\ \text{`}\mathsf{tt}\ l\ r \rightsquigarrow l}\iota_1 \qquad \frac{}{\text{`}\mathsf{ifte}\ \text{`}\mathsf{ff}\ l\ r \rightsquigarrow r}\iota_2 \qquad \frac{t \rightsquigarrow u}{\text{`}\mathsf{lam}\ t \rightsquigarrow \text{`}\mathsf{lam}\ u}\xi$$

Figure 5.3: $\iota\xi$ Rules for our Calculus

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 $\beta\iota$-short normal forms.

Normal and Neutral Forms

We parametrise the mutually defined inductive families Ne and Nf by a predicate *Eta?* constraining the types at which one may embed a neutral as a normal form. This constraint shows up in the type of ‘neu; it makes it possible to control whether the NBE should η-expands all terms at certain types by prohibiting the existence of neutral terms at said type.

Once more, the expected notions of thinning th^Ne and th^Nf are induced as Ne and Nf 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 will tackle this problem in part II.

5.1 Normalisation by Evaluation for $\beta\iota\xi\eta$

In the case of NBE, the environment values and the computations in the model will both use the same type family Model, defined by induction on the Type argument. The η-rules allow us to represent functions (respetively inhabitants of ‘Unit) in the source language as function spaces (respectively values of type $\top$). Evaluating a ‘Bool may

```
mutual

  data Ne : Type ─Scoped where
    ‘var  : ∀[ Var σ ⇒ Ne σ ]
    ‘app : ∀[ Ne (σ ‘→ τ) ⇒ Nf σ ⇒ Ne τ ]
    ‘ifte  : ∀[ Ne ‘Bool ⇒ Nf σ ⇒ Nf σ ⇒ Ne σ ]

  data Nf : Type ─Scoped where
    ‘neu : Eta? σ → ∀[ Ne σ ⇒ Nf σ ]
    ‘one : ∀[ Nf ‘Unit ]
    ‘tt ‘ff : ∀[ Nf ‘Bool ]
    ‘lam : ∀[ (σ ::_) ⊢ Nf τ ⇒ Nf (σ ‘→ τ) ]
```

Figure 5.4: Neutral and Normal Forms

however yield a stuck term so we can't expect the model to give us anything more than an open term in normal form.

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 (Model ‘Unit Γ) are equal and all elements of (Model (σ ‘→ τ) Γ) are functions in Agda.

```
Model : Type ─Scoped
Model ‘Unit       Γ = ⊤
Model ‘Bool       Γ = Nf ‘Bool Γ
Model (σ ‘→ τ) Γ = □ (Model σ ⇒ Model τ) Γ
```

Figure 5.5: Model for Normalisation by Evaluation

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, (Model σ) is Thinnable.

```
th^Model : ∀ σ → Thinnable (Model σ)
th^Model ‘Unit       = th^const
th^Model ‘Bool       = th^Nf
th^Model (σ ‘→ τ) = th^□
```

Figure 5.6: Values in the Model are Thinnable

Application's semantic counterpart is easy to define: given that $\mathcal{V}$ and $\mathcal{C}$ are equal in this instance definition we can feed the argument directly to the function, with the identity renaming. This corresponds to extract for the comonad □.

```
APP : ∀[ Model (σ ‘→ τ) ⇒ Model σ ⇒ Model τ ]
APP t u = extract t u
```

Figure 5.7: Semantic Counterpart of ‘app

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 interplay between neutral terms, model values and normal forms. reflect performs a form of semantic η-expansion: all stuck ‘Unit terms are equated and all functions are λ-headed. It allows us to define var0, the semantic counterpart of (‘var z).

```
mutual

  var0 : ∀[ (σ ::_) ⊢ Model σ ]
  var0 = reflect _ (‘var z)

  reflect : ∀ σ → ∀[ Ne σ ⇒ Model σ ]
  reflect ‘Unit       t = _
  reflect ‘Bool       t = ‘neu ‘Bool t
  reflect (σ ‘→ τ) t = λ ρ u → reflect τ (‘app (th^Ne t ρ) (reify σ u))

  reify : ∀ σ → ∀[ Model σ ⇒ Nf σ ]
  reify ‘Unit       T = ‘one
  reify ‘Bool       T = T
  reify (σ ‘→ τ) T = ‘lam (reify τ (T extend var0))
```

Figure 5.8: Reify and Reflect

We can then give the semantics of ‘ifte: if the boolean is a value, the appropriate branch is picked; if it is stuck then the whole expression is stuck. It is then turned into a neutral form by reifying the two branches and then reflected in the model.

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 can define a normaliser by kickstarting the evaluation with an environment of placeholder values obtained by reflecting the term's free variables and then reifying the result.

```
IFTE : Model ‘Bool Γ → Model σ Γ → Model σ Γ → Model σ Γ
IFTE ‘tt          l r = l
IFTE ‘ff          l r = r
IFTE (‘neu _ T) l r = reflect σ (‘ifte T (reify σ l) (reify σ r))
```

Figure 5.9: Semantic Counterpart of ‘ifte

```
Eval : Semantics Model Model
Eval .th^𝒱 = th^Model _
Eval .var   = id
Eval .lam   = id
Eval .app   = APP
Eval .one   = _
Eval .tt    = ‘tt
Eval .ff    = ‘ff
Eval .ifte  = IFTE
```

Figure 5.10: Evaluation is a Semantics

```
eval : Term σ Γ → Model σ Γ
eval = Fundamental.lemma Eval (pack (reflect _ ∘ ‘var))

norm : Term σ Γ → Nf σ Γ
norm = reify _ ∘ eval
```

Figure 5.11: Normalisation as Reification of an Evaluated Term

5.2 Normalisation by Evaluation for $\beta\iota\xi$

As seen above, the traditional typed model construction leads to an NBE procedure outputting $\beta\iota$-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 *Eta?* which holds for all types thus allowing us to embed all neutrals into normal forms, and to mutually defining the model (Model) together with the *acting* model (Value).

```
mutual

  Model : Type ─Scoped
  Model σ Γ = Ne σ Γ ⊎ Value σ Γ

  Value : Type ─Scoped
  Value 'Unit       = const ⊤
  Value 'Bool       = const Bool
  Value (σ '→ τ) = □ (Model σ ⇒ Model τ)
```

Figure 5.12: Model Definition for $\beta\iota\xi$

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 ⊤, 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 either the argument or the body may or may not be stuck.

We have once again only used families constant in their scope index, neutral forms or □-closed families. All of these are Thinnable hence Value and Model also are.

```
th^Value : ∀ σ → Thinnable (Value σ)
th^Value 'Unit       = th^const
th^Value 'Bool       = th^const
th^Value (σ '→ τ) = th^□

th^Model : ∀ σ → Thinnable (Model σ)
th^Model σ (inj₁ neu) ρ = inj₁ (th^Ne neu ρ)
th^Model σ (inj₂ val)  ρ = inj₂ (th^Value σ val ρ)
```

Figure 5.13: The Model is Thinnable

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 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 : ∀[ Ne σ ⇒ Model σ ]
reflect = inj₁

var0 : ∀[ (σ ::_) ⊢ Model σ ]
var0 = reflect (‘var z)

mutual

  reify : ∀ σ → ∀[ Model σ ⇒ Nf σ ]
  reify σ (inj₁ neu) = ‘neu _ neu
  reify σ (inj₂ val)  = reify^Value σ val

  reify^Value : ∀ σ → ∀[ Value σ ⇒ Nf σ ]
  reify^Value ‘Unit       _ = ‘one
  reify^Value ‘Bool       b = if b then ‘tt else ‘ff
  reify^Value (σ ‘→ τ) f = ‘lam (reify τ (f extend var0))
```

Figure 5.14: Reflect, Reify and Interpretation for Fresh Variables

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.

```
APP : ∀[ Model (σ ‘→ τ) ⇒ Model σ ⇒ Model τ ]
APP (inj₁ f) t = inj₁ (‘app f (reify _ t))
APP (inj₂ f) t = extract f t
```

Figure 5.15: Semantical Counterpart of ‘app

When defining the semantical counterpart of ‘ifte, the value case is similar to that of the previous section: depending on the boolean value we pick either the left or the right branch which are precisely of the right type already. If the boolean evaluates to a stuck term, we once again reify the two branches and assemble a neutral term. However this time we do not need to η-expand it: it is a perfectly valid inhabitant of the Model as is.

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, normal-

```
IFTE : ∀[ Model ‘Bool ⇒ Model σ ⇒ Model σ ⇒ Model σ ]
IFTE (inj₁ b) l r = inj₁ (‘ifte b (reify _ l) (reify _ r))
IFTE (inj₂ b) l r = if b then l else r
```

Figure 5.16: Semantical Counterpart of ‘ifte

isation is defined by composing reification and evaluation on a diagonal environment made of placeholders.

```
eval : Term σ Γ → Model σ Γ
eval = Fundamental.lemma Eval (pack (reflect ∘ ‘var))

norm : Term σ Γ → Nf σ Γ
norm = reify _ ∘ eval
```

5.3 Normalisation by Evaluation for $\beta\iota$

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.

For that purpose, we introduce an inductive family describing terms in weak-head normal forms.

Weak-Head Normal Forms

A weak-head normal form (respectively a weak-head neutral form) is a term which has been evaluated just enough to reveal a head constructor (respectively to reach a stuck elimination). There are no additional constraints on the subterms: a λ-headed term is in weak-head normal form no matter the shape of its body. Similarly an application composed of a variable as the function and a term as the argument is in weak-head neutral form no matter what the argument looks like. This means in particular that unlike with Ne and Nf there is no mutual dependency between the definitions of WHNE (defined first) and WHNF.

Naturally, it is possible to define the thinnings th^WHNE and th^WHNF as well as erasure functions erase^WHNE and erase^WHNF with codomain Term. We omit their simple definitions here.

Model Construction

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

```
data WHNE : Type −Scoped where
  ‘var  : ∀[ Var σ ⇒ WHNE σ ]
  ‘app : ∀[ WHNE (σ ‘→ τ) ⇒ Term σ ⇒ WHNE τ ]
  ‘ifte  : ∀[ WHNE ‘Bool ⇒ Term σ ⇒ Term σ ⇒ WHNE σ ]

data WHNF : Type −Scoped where
  ‘lam    : ∀[ (σ ::_) ⊢ Term τ ⇒ WHNF (σ ‘→ τ) ]
  ‘one    : ∀[ WHNF ‘Unit ]
  ‘tt ‘ff   : ∀[ WHNF ‘Bool ]
  ‘whne : ∀[ WHNE σ ⇒ WHNF σ ]
```

Figure 5.17: Weak-Head Normal and Neutral Forms

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 subterm needs to be evaluated.

```
mutual

  Model : Type −Scoped
  Model σ Γ = Term σ Γ × (WHNE σ Γ ⊎ Value σ Γ)

  Value : Type −Scoped
  Value ‘Unit       = const ⊤
  Value ‘Bool       = const Bool
  Value (σ ‘→ τ) = □ (Model σ ⇒ Model τ)
```

Figure 5.18: Model Definition for Computing Weak-Head Normal Forms

Thinnable can be defined rather straightforwadly based on the template provided in the previous section: once more all the notions used in the model definition are Thinnable themselves. Reflection and reification also follow the same recipe as in the previous section.

The application and conditional branching rules are more interesting. One important difference with respect to the previous section 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 and let the other subterms untouched.

The semantical counterpart of ‘lam is also slightly trickier than before. Indeed, we need to recover the source term the value corresponds to. Luckily we know it has to be λ-headed and once we have introduced a fresh variable with ‘lam, we can project out the source term of the body evaluated using this fresh variable as a placeholder value.

```
APP : ∀[ Model (σ ‘→ τ) ⇒ Model σ ⇒ Model τ ]
APP (f , inj₁ whne) (t , _) = (‘app f t , inj₁ (‘app whne t))
APP (_ , inj₂ f)     t      = extract f t
```

```
IFTE : ∀[ Model ‘Bool ⇒ Model σ ⇒ Model σ ⇒ Model σ ]
IFTE (b , inj₁ whne) (l , _) (r , _) = (‘ifte b l r , inj₁ (‘ifte whne l r))
IFTE (b , inj₂ v)     l r             = if v then l else r
```

Figure 5.19: Semantical Counterparts of ‘app and ‘ifte

```
LAM : ∀[ □ (Model σ ⇒ Model τ) ⇒ Model (σ ‘→ τ) ]
LAM b = (‘lam (proj₁ (b extend var0)) , inj₂ b)
```

Figure 5.20: Semantical Counterparts of ‘lam

We can finally put together all of these semantic counterparts to obtain a Semantics corresponding to weak-head normalisation. We omit the now self-evident definition of norm^βι as the composition of evaluation and reification.

Chapter 6

CPS Transformations

In their generic account of continuation passing style (CPS) transformations, 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 (ML). 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 always considered to be a computational type.

```
data CType : Set where
  ‘Unit    : CType
  ‘Bool    : CType
  _‘→#_ : (σ τ : CType) → CType
  #_       : CType → CType
```

Figure 6.1: Inductive Definition of Types for Moggi's ML

Then comes the Meta-Language itself (cf. fig. 6.2). It incorporates Term 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 ‘bind make #_ a monad. They can be used to explicitly schedule the evaluation order of various subterms.

As explained in Hatcliff and Danvy's paper, the translation from Type to CType 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 but differ on the way they translate function spaces. In CBN the argument of a function is a computation (i.e. it is wrapped in a #_ type constructor) whilst it is expected to have been fully evaluated in CBV. Let us look more closely at these two translations.

```
data ML : CType –Scoped where
  ‘var  : ∀[ Var σ ⇒ ML σ ]
  ‘app  : ∀[ ML (σ ‘→# τ) ⇒ ML σ ⇒ ML (# τ) ]
  ‘lam  : ∀[ (σ ::_) ⊢ ML (# τ) ⇒ ML (σ ‘→# τ) ]
  ‘one  : ∀[ ML ‘Unit ]
  ‘tt ‘ff  : ∀[ ML ‘Bool ]
  ‘ifte  : ∀[ ML ‘Bool ⇒ ML (# σ) ⇒ ML (# σ) ⇒ ML (# σ) ]
  ‘ret  : ∀[ ML σ ⇒ ML (# σ) ]
  ‘bind : ∀[ ML (# σ) ⇒ ML (σ ‘→# τ) ⇒ ML (# τ) ]
```

Figure 6.2: Definition of Moggi’s Meta Language

6.1 Translation into Moggi’s Meta-Language

Call by Name

We define the translation CBN of Type in a call by name style together with a shorthand for the computational version of the translation #CBN. As explained earlier, base types are kept identical whilst function spaces are turned into function spaces whose domains and codomains are computational.

```
mutual

  #CBN : Type → CType
  #CBN σ = # (CBN σ)

  CBN : Type → CType
  CBN ‘Unit      = ‘Unit
  CBN ‘Bool      = ‘Bool
  CBN (σ ‘→ τ) = #CBN σ ‘→# CBN τ
```

Figure 6.3: Translation of Type in a Call by Name style

Once we know how to translate types, we can start thinking about the way terms are handled. The term’s type will *have* to be computational as there is no guarantee that the input term is in normal form. In a call by name strategy, variables in context are also assigned a computational type.

By definition of Semantics, our notions of environment values and computations will *have* to be of type (Type –Scoped). This analysis leads us to define the generic transformation _^CBN in fig. 6.4.

Our notion of environment values are then (Var ^CBN) whilst computations will be (ML ^CBN). Once these design decisions are made, we can start drafting the semantical counterpart of common combinators.

```
_^CBN : CType −Scoped → Type −Scoped
(T ^CBN) σ Γ = T (#CBN σ) (map #CBN Γ)
```

Figure 6.4: ·−Scoped Transformer for Call by Name

As usual, we define combinators corresponding to the two eliminators first. In these cases, we need to evaluate the subterm the redex is potentially stuck on first. This means evaluating the function first in an application node (which will then happily consume the thunked argument) and the boolean in the case of boolean branching.

```
APP : ∀[ (ML ^CBN) (σ ‘→ τ) ⇒ (ML ^CBN) σ ⇒ (ML ^CBN) τ ]
APP f t = ‘bind f (‘lam (‘app (‘var z) (th^ML t extend)))


IFTE : ∀[ (ML ^CBN) ‘Bool ⇒ (ML ^CBN) σ ⇒ (ML ^CBN) σ ⇒ (ML ^CBN) σ ]
IFTE b l r = ‘bind b (‘lam (‘ifte (‘var z) (th^ML l extend) (th^ML r extend)))
```

Figure 6.5: Semantical Counterparts for ‘app and ‘ifte

Values have a straightforward interpretation: they are already fully evaluated and can thus simply be returned as trivial computations using ‘ret. This gives us everything we need to define the embedding of STLC into Moggi's ML in call by name style.

Call by Value

Call by value follows a similar pattern. As the name suggests, in call by value function arguments are expected to be values already. In the definition of CBV this translates to function spaces being turned into functions spaces where only the *codomain* is made computational.

```
mutual

 #CBV : Type → CType
 #CBV σ = # (CBV σ)

 CBV : Type → CType
 CBV ‘Unit      = ‘Unit
 CBV ‘Bool      = ‘Bool
 CBV (σ ‘→ τ) = CBV σ ‘→# CBV τ
```

Figure 6.6: Translation of Type in a Call by Value style

We can then move on to the notion of values and computations for our call by value Semantics. All the variables in scope should refer to values hence the choice to translate Γ by mapping CBV over it in both cases. As with the call by name translation, we need our target type to be computational: the input terms are not guaranteed to be in normal form.

```
V^CBV : Type −Scoped
V^CBV σ Γ = Var (CBV σ) (map CBV Γ)

C^CBV : Type −Scoped
C^CBV σ Γ = ML (# CBV σ) (map CBV Γ)
```

Figure 6.7: Values and Computations for the CBN CPS Semantics

Albeit being defined at different types, the semantical counterparts of value constructors and ‘ifte are the same as in the call by name case. The interpretation of ‘app is where we can see a clear difference: we need to evaluate the function *and* its argument before applying one to the other. We pick a left-to-right evaluation order but that is arbitrary: another decision would lead to a different but equally valid translation.

```
APP : ∀[ C^CBV (σ ‘→ τ) ⇒ C^CBV σ ⇒ C^CBV τ ]
APP f t = ‘bind f (‘lam (‘bind (th^ML t extend) (‘lam (‘app (‘var (s z)) (‘var z)))))
```

Figure 6.8: Semantical Counterparts for ‘app

Finally, the corresponding Semantics can be defined (code omitted here).

6.2 Translation Back from Moggi's Meta-Language

Once we have picked an embedding from STLC to Moggi's ML, we can kickstart it by using an environment of placeholder values just like we did for normalisation by evaluation. The last thing missing to get the full CPS translation is to have the generic function elaborating terms in ML into STLC ones.

We first need to define a translation of types in Moggi's Meta-Language to types in the simply-typed lambda-calculus. The translation is parametrised by r, the *return* type. Type constructors common to both languages are translated to their direct counterpart and the computational type constructor is translated as double r-negation (i.e. $(\cdot$ ‘$\rightarrow r)$ ‘$\rightarrow r)$.

Once these translations have been defined, we give a generic elaboration function getting rid of the additional language constructs available in Moggi's ML. It takes any term in Moggi's ML and returns a term in STLC where both the type and the context have been translated using (CPS[r]) for an abstract parameter r.

```
mutual

  #CPS[_] : Type → CType → Type
  #CPS[ r ] σ = (CPS[ r ] σ ‘→ r) ‘→ r

  CPS[_] : Type → CType → Type
  CPS[ r ] ‘Bool       = ‘Bool
  CPS[ r ] ‘Unit       = ‘Unit
  CPS[ r ] (σ ‘→# τ) = CPS[ r ] σ ‘→ #CPS[ r ] τ
  CPS[ r ] (# σ)       = #CPS[ r ] σ
```

Figure 6.9: Translating Moggi's ML's Types to STLC Types

```
cps : ML σ Γ → Term (CPS[ r ] σ) (map CPS[ r ] Γ)
```

All the constructors which appear both in Moggi's ML and STLC are translated in a purely structural manner. The only two interesting cases are ‘ret and ‘bind which correpond to the # monad in Moggi's ML and are interpreted as return and bind for the continuation monad in STLC.

First, (‘ret *t*) is interpreted as the function which takes the current continuation and applies it to its translated argument (cps *t*).

```
cps (‘ret t) = ‘lam (‘app (‘var z) (th^Term (cps t) extend))
```

Then, (‘bind *m f*) gets translated as the function grabbing the current continuation *k*, and running the translation of *m* with the continuation which, provided the value *v* obtained by running *m*, runs *f* applied to *v* and *k*. Because the translations of *m* and *f* end up being used in extended contexts, we need to make use of the fact Terms are thinnable.

```
cps (‘bind m f) = ‘lam $ ‘app m’ $ ‘lam $ ‘app (‘app f’ (‘var z)) (‘var (s z))
  where m’ = th^Term (cps m) (pack s)
        f’  = th^Term (cps f) (pack (s ∘ s))
```

By highlighting the shared structure, of the call by name and call by value translations we were able to focus on the interesting part: the ways in which they differ. The formal treatment of the type translations underlines the fact that in both cases the translation of a function's domain is uniform. This remark opens the door to alternative definitions; we could for instance consider a mixed strategy which is strict in machine-representable types thus allowing an unboxed representation (Jones and Launchbury [1991]) but lazy in every other type.

Chapter 7

Conclusion

7.1 Summary

We have now seen how to give an intrinsically well-scoped and well-typed presentation of a simple calculus. We represent it as an inductive family indexed by the term's type and the context it lives in. Variables are represented by typed de Bruijn indices.

To make the presentation lighter, we have made heavy use of a small domain specific language to define indexed families. This allows us to silently thread the context terms live in and only ever explicitly talk about it when it gets extended.

We have seen a a handful of vital traversals such as thinning and substitution which, now that they act on a well-typed and well-scoped syntax need to be guaranteed to be type and scope preserving.

We have studied how McBride (2005) identifies the structure common to thinning and substitution, introduces a notion of Kit and refactors the two functions as instances of a more fundamental Kit-based traversal.

After noticing that other usual programs such as the evaluation function for normalisation by evaluation seemed to fit a similar pattern, we have generalised McBride's Kit to obtain our notion of Semantics. The accompanying fundamental lemma is the core of this whole part. It demonstrates that provided a notion of values and a notion of computations abiding by the Semantics constraints, we can write a scope-and-type preserving traversal taking an environment of values, a term and returning a computation.

Thinning, substitution, normalisation by evaluation, printing with names, and various continuation passing style translations are all instances of this fundamental lemma.

7.2 Related Work

This part of the work which focuses on programming and not (yet!) on proving, can be fully replicated in Haskell. The subtleties of working with dependent types in Haskell (Lindley and McBride [2014]) are outside the scope of this paper.

If the tagless and typeful normalisation by evaluation 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 construction 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.

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 Martin-Löf 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 app 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 (Team [2017]) 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.

7.3 Further Work

There are three main avenues for future work and we will tackle all of these later on this thesis. We could focus on the study of instances of Semantics, the generalisation of Semantics to a whole class of syntaxes with binding rather than just our simple STLC, or proving properties of the traversals that are instances of Semantics.

Other instances

The vast array of traversals captured by this framework from meta-theoretical results (stability under thinning and substitution) to programming tools (printing with names) and compilation passes (partial evaluation and continuation passing style translations) suggests that this method is widely applicable. The quest of ever more traversals to refactor as instances of the fundamental lemma of Semantics is a natural candidate for further work.

We will see later on that once we start considering other languages including variants with fewer invariants baked in, we can find new candidates. The fact that erasure from a language with strong invariants to an untyped one falls into this category may not be too surprising. The fact that the other direction, that is type checking of raw terms or even elaboration of such raw terms to a typed core language also corresponds to a notion of Semantics is perhaps more intriguing.

A Generic Notion of Semantics

If we look closely at the set of constraints a Semantics imposes on the notions of values and computations, we can see that it matches tightly the structure of our language:

- Each base constructor needs to be associated to a computation of the same type;
- Each eliminator needs to be interpreted as a function combining the interpretation of its subterms into the interpretation of the whole;
- The lambda case is a bit special: it uses a Kripke function space from values to computation as its interpretation

We can apply this recipe mechanically to enrich our language with e.g. product and sum types, their constructor and eliminators. This suggests that we ought to be able to give a generic description of syntaxes with binding and the appropriate notion of Semantics for each syntax. We will make this intuition precise in part II.

Properties of Semantics

Finally, because we know e.g. that we can prove generic theorems for all the programs defined using fold (Malcolm [1990]), the fact that all of these traversals are instances of a common fold-like function suggests that we ought to be able to prove general theorems about its computational behaviour and obtain interesting results for each instance as corollaries. This is the topic we will focus on for now.

Chapter 8

The Simulation Relation

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.

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.

8.1 Relations Between Scoped Families

We start by defining what it means to be a relation between two (I –Scoped) families T and U: at every type σ and every context Γ, we expect to have a relation between ($T\,\sigma\,\Gamma$) and ($U\,\sigma\,\Gamma$). We use a record wrapper for two reasons. First, we define the relations we are interested in by copattern-matching thus preventing their eager unfolding by Agda; this makes the goals much more readable during interactive development. Second, it is easier for Agda to recover T and U by unification when they appear as explicit parameters of a record rather than as applied families in the body of the definition.

If we have a relation for values, we can lift it in a pointwise manner to a relation on environment of values. We call this relation transformer All. We also define it using a record wrapper, for the same reasons.

```
record Rel (T U : I −Scoped) : Set₁ where
  constructor mkRel
  field rel : ∀ σ → ∀[ T σ ⇒ U σ ⇒ const Set ]
```

Figure 8.1: Relation Between (I −Scoped) Families

```
record All (ℛ : Rel T U) Γ (ρᵀ : (Γ −Env) T Δ) (ρᵁ : (Γ −Env) U Δ) : Set where
  constructor packᴿ
  field lookupᴿ : (k : Var σ Γ) → rel ℛ σ (lookup ρᵀ k) (lookup ρᵁ k)
```

Figure 8.2: Relation Between Environments of Values

For virtually every combinator on environments, we have a corresponding combinator for All: the empty environment ε is associated to ε^R the proof that two empty environments are always related, to the environment extension _•_ corresponds the relation on environment extension _•ᴿ_ which provided takes a proof that two environments are related and that two values are related and returns the proof that the environments each extended with the appropriate value are both related, etc.

Once we have all of these definitions, we can spell out what it means to simulate a semantics with another.

8.2 Simulation Constraints

The evidence that we have a Simulation between two Semantics is packaged in a record indexed by the semantics as well as two relations. The first one ($\mathcal{V}^R$) relates values and the second one ($\mathcal{C}^R$) describes simulation for computations.

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

The set of simulation constraints is in one-to-one correspondance with that of semantical constructs. We start with value thinnings: provided two values are related, their respective thinnings should still be related.

```
th^𝒱ᴿ : (ρ : Thinning Δ Θ) → ℛⱽ σ vᴬ vᴮ → ℛⱽ σ (𝒮ᴬ.th^𝒱 vᴬ ρ) (𝒮ᴮ.th^𝒱 vᴮ ρ)
```

Our other constraints are going to heavily feature $\mathcal{C}^R$ applied to one term evaluated twice: once by $\mathcal{S}^A$ with the environment of values ρ^A and once by $\mathcal{S}^B$ with ρ^B. To make the types more readable, we introduce an intermediate definition $\mathcal{R}$ making this pattern explicit.

```
ℛ : ∀ {Γ Δ} σ → (Γ −Env) 𝒱ᴬ Δ → (Γ −Env) 𝒱ᴮ Δ → Term σ Γ → Set
ℛ σ ρᴬ ρᴮ t = rel 𝒞ᴿ σ (evalᴬ ρᴬ t) (evalᴮ ρᴮ t)
```

The relational counterpart of ‘var and var is the first field to make use of $\mathcal{R}$: provided that ρ^A and ρ^B carry values related by $\mathcal{V}^R$, the result of evaluating the variable v in each respectively should yield computations related by C^R.

```
var^R : All 𝓥^R Γ ρ^A ρ^B → (v : Var σ Γ) → 𝓡 σ ρ^A ρ^B (‘var v)
```

Value constructors in the language follow a similar pattern: provided that the evaluation environment are related, we expect the computations to be related too.

```
one^R : All 𝓥^R Γ ρ^A ρ^B → 𝓡 ‘Unit ρ^A ρ^B ‘one
tt^R  : All 𝓥^R Γ ρ^A ρ^B → 𝓡 ‘Bool ρ^A ρ^B ‘tt
ff^R  : All 𝓥^R Γ ρ^A ρ^B → 𝓡 ‘Bool ρ^A ρ^B ‘ff
```

Then come the structural cases: for language constructs like ‘app and ‘ifte whose subterms live in the same context as the overall term, the constraints are purely structural. Provided that the evaluation environments are related, and that the evaluation of the subterms in each environment respectively are related then the evaluations of the overall terms should also yield related results.

```
app^R : All 𝓥^R Γ ρ^A ρ^B →
        ∀ f t → 𝓡 (σ ‘→ τ) ρ^A ρ^B f → 𝓡 σ ρ^A ρ^B t →
        𝓡 τ ρ^A ρ^B (‘app f t)
ifte^R : All 𝓥^R Γ ρ^A ρ^B →
        ∀ b l r → 𝓡 ‘Bool ρ^A ρ^B b → 𝓡 σ ρ^A ρ^B l → 𝓡 σ ρ^A ρ^B r →
        𝓡 σ ρ^A ρ^B (‘ifte b l r)
```

Finally, we reach the most interesting case. The semantics attached to the body of a ‘lam is expressed in terms of a Kripke function space. As a consequence, the relational semantics will need a relational notion of Kripke function space (Kripke^R) to spell out the appropriate simulation constraint. This relational Kripke function space states that in any thinning of the evaluation context and provided two related inputs, the evaluation of the body in each thinned environment extended with the appropriate value should yield related computations.

```
Kripke^R : ∀ {Γ Δ} σ τ → (Γ −Env) 𝓥^A Δ → (Γ −Env) 𝓥^B Δ →
           Term τ (σ :: Γ) → Set
Kripke^R {Γ} {Δ} σ τ ρ^A ρ^B b =
  ∀ {Θ} (ρ : Thinning Δ Θ) {u^A u^B} → 𝓡^V σ u^A u^B →
  𝓡 τ (th^Env 𝓢^A.th^𝓥 ρ^A ρ • u^A) (th^Env 𝓢^B.th^𝓥 ρ^B ρ • u^B) b
```

Figure 8.3: Relational Kripke Function Spaces: From Related Inputs to Related Outputs

This allows us to describe the constraint for ‘lam: provided related environments of values, if we have a relational Kripke function space for the body of the ‘lam then both evaluations should yield related results.

```
lam^R : All 𝒱^R Γ ρ^A ρ^B → ∀ b → Kripke^R σ τ ρ^A ρ^B b → ℛ (σ ‘→ τ) ρ^A ρ^B (‘lam b)
```

This specification is only useful if it is accompanied by a a fundamental lemma of simulations stating that the evaluation of a term on related inputs yields related outputs.

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

In a manner reminiscent of our proof of the fundamental lemma of Semantics, we introduce a Fundamental module parametrised by a record packing the evidence that two semantics are in Simulation. 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.

```
module Fundamental (𝒮^R : Simulation 𝒮^A 𝒮^B 𝒱^R 𝒞^R) where

  open Simulation 𝒮^R

  lemma : All 𝒱^R Γ ρ^A ρ^B → ∀ t → ℛ σ ρ^A ρ^B t
  lemma ρ^R (‘var v)      = var^R ρ^R v
  lemma ρ^R (‘app f t)    = app^R ρ^R f t (lemma ρ^R f) (lemma ρ^R t)
  lemma ρ^R (‘lam b)      = lam^R ρ^R b $ λ ren v^R →
                            lemma ((th^𝒱^R ren <$>^R ρ^R) •^R v^R) b
  lemma ρ^R ‘one          = one^R ρ^R
  lemma ρ^R ‘tt           = tt^R ρ^R
  lemma ρ^R ‘ff           = ff^R ρ^R
  lemma ρ^R (‘ifte b l r) = ifte^R ρ^R b l r (lemma ρ^R b) (lemma ρ^R l) (lemma ρ^R r)
```

Figure 8.4: Fundamental Lemma of Simulations

We can now consider the second criterion for usefulness: the existence of interesting instances of a Simulation.

8.4 Syntactic Traversals are Extensional

A first corollary of the fundamental lemma of simulations is the fact that semantics arising from a Syntactic (cf. fig. 4.17) definition are extensional. We can demonstrate this by proving that every syntactic semantics is in simulation with itself. That is to say that the evaluation function yields propositionally equal values provided extensionally equal environments of values.

Under the assumption that *Syn* is a Syntactic instance, we can define the corresponding Semantics $\mathcal{S}$ by setting $\mathcal{S}$ = syntactic *Syn*. Using Eq^R the Rel defined as the pontwise lifting of propositional equality, we can make our earlier claim formal and prove it. All the constraints are discharged either by reflexivity or by using congruence to combine various hypotheses.

```
Syn-ext : Simulation 𝒮 𝒮 Eq^R Eq^R
Syn-ext .th^𝒱^R = λ ρ eq → cong (λ t → th^𝒯 t ρ) eq
Syn-ext .var^R  = λ ρ^R v → cong var (lookup^R ρ^R v)
Syn-ext .lam^R  = λ ρ^R b b^R → cong ‘lam (b^R extend refl)
Syn-ext .app^R  = λ ρ^R f t → cong₂ ‘app
Syn-ext .ifte^R = λ ρ^R b l r → cong₃ ‘ifte
Syn-ext .one^R  = λ ρ^R → refl
Syn-ext .tt^R   = λ ρ^R → refl
Syn-ext .ff^R   = λ ρ^R → refl
```

Figure 8.5: Syntactic Traversals are in Simulation with Themselves

Because the Simulation statement is not necessarily extremely illuminating, we spell out the type of the corollary to clarify what we just proved: whenever two environments agree on each variable, evaluating a term with either of them produces equal results.

```
syn-ext : All Eq^R Γ ρ^l ρ^r → (t : Term σ Γ) → eval 𝒮 ρ^l t ≡ eval 𝒮 ρ^r t
syn-ext = simulation Syn-ext
```

Figure 8.6: Syntactic Traversals are Extensional

This may look like a trivial result however we have found multiple use cases for it during the development of our solution to the POPLMark Reloaded challenge (2018): when proceeding by equational reasoning, it is often the case that we can make progress on each side of the equation only to meet in the middle with the same traversals using two environments manufactured in slightly different ways but ultimately equal. This lemma allows us to bridge that last gap.

8.5 Renaming is a Substitution

Similarly, it is sometimes the case that after a bit of rewriting we end up with an equality between one renaming and one substitution. But it turns out that as long as the substitution is only made up variables, it is indeed equal to the corresponding renaming. We can make this idea formal by introducting the VarTerm^R relation stating that a variable and a term are morally equal like so:

We can then state our result: we can prove a simulation lemma between Renaming and Substitution where values (i.e. variables in the cases of renaming and terms in

```
VarTermᴿ : Rel Var Term
rel VarTermᴿ σ v t = ‘var v ≡ t
```

Figure 8.7: Characterising Equal Variables and Terms

terms of substitution) are related by VarTerm^R and computations (i.e. terms) are related by Eq^R. Once again we proceed by reflexivity and congruence.

```
RenSub^Sim : Simulation Renaming Substitution VarTermᴿ Eqᴿ
RenSub^Sim .th^𝒱ᴿ = λ ρ → cong (λ t → th^Term t ρ)
RenSub^Sim .varᴿ  = λ ρᴿ v → lookupᴿ ρᴿ v
RenSub^Sim .lamᴿ  = λ ρᴿ b bᴿ → cong ‘lam (bᴿ extend refl)
RenSub^Sim .appᴿ  = λ ρᴿ f t → cong₂ ‘app
RenSub^Sim .ifteᴿ = λ ρᴿ b l r → cong₃ ‘ifte
RenSub^Sim .oneᴿ  = λ ρᴿ → refl
RenSub^Sim .ttᴿ   = λ ρᴿ → refl
RenSub^Sim .ffᴿ   = λ ρᴿ → refl
```

Figure 8.8: Renaming is in Simulation with Substitution

Rather than showing one more time the type of the corollary, we show a specialized version where we pick the substitution to be precisely the thinning used on which we have mapped the ‘var constructor.

```
ren-as-sub : (t : Term σ Γ) (ρ : Thinning Γ Δ) → th^Term t ρ ≡ sub (‘var <$> ρ) t
ren-as-sub t ρ = simulation RenSub^Sim (packᴿ (λ v → refl)) t
```

Figure 8.9: Renaming as a Substitution

8.6 The PER for $\beta\iota\xi\eta$-Values is Closed under Evaluation

Now that we are a bit more used to the simulation framework and simulation lemmas, we can look at a more complex example: the simulation lemma relating normalisation by evaluation’s eval function to itself. This may seem bureaucratic but it is crucial: the model definition uses the host language’s function space which contains more functions than just the ones obtained by evaluating a simply-typed λ-term. A value at type (‘Bool ‘→ ‘Bool) may for instance behave like boolean negation on canonical terms but be the constant ‘tt function on neutral value. It does not correspond to any term in the source language: any candidate term would allow us to write expressions not stable under substitution!

```
exotic : ∀[ Model (‘Bool ‘→ ‘Bool) ]
exotic ρ ‘tt          = ‘ff
exotic ρ ‘ff          = ‘tt
exotic ρ (‘neu _ _) = ‘tt
```

Figure 8.10: Exotic Value, Not Quite Equal to Negation

Clearly, these exotic functions have undesirable behaviours and need to be ruled out if we want to be able to prove that normalisation has good properties. This is done by defining a Partial Equivalence Relation (PER) (Mitchell [1996]) on the model which is to say a relation which is symmetric and transitive but may not be reflexive for all elements in its domain. The elements equal to themselves will be guaranteed to be well behaved. We will 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. Values at base types are concrete data: either trivial for values of type ‘Unit or normal forms for values of type ‘Bool. They are considered equal when they are effectively syntactically the same, i.e. propositionally equal. Functions on the other hand are declared equal whenever equal inputs map to equal outputs.

```
PER : Rel Model Model
rel PER ‘Unit       t u = t ≡ u
rel PER ‘Bool       t u = t ≡ u
rel PER (σ ‘→ τ) f g = ∀ {Δ} (ρ : Thinning _ Δ) {t u} →
                            rel PER σ t u → rel PER τ (f ρ t) (g ρ u)
```

Figure 8.11: Partial Equivalence Relation for Model Values

On top of being a PER (i.e. symmetric and transitive), we can prove by a simple case analysis on the type that this relation is also stable under thinning for Model values defined in fig. 5.6.

```
th^PER : ∀ σ {T U} → rel PER σ T U → (ρ : Thinning Γ Δ) →
          rel PER σ (th^Model σ T ρ) (th^Model σ U ρ)
th^PER ‘Unit       _   ρ = refl
th^PER ‘Bool       bᴿ ρ = cong (λ t → th^Nf t ρ) bᴿ
th^PER (σ ‘→ τ) fᴿ  ρ = λ σ → fᴿ (select ρ σ)
```

Figure 8.12: Stability of the PER under Thinning

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.

```
mutual

  reflect^R : ∀ σ {t u : Ne σ Γ} → t ≡ u → rel PER σ (reflect σ t) (reflect σ u)
  reflect^R ‘Unit      _ = refl
  reflect^R ‘Bool      t = cong (‘neu ‘Bool) t
  reflect^R (σ ‘→ τ) f = λ ρ t → reflect^R τ (cong_2 ‘app (cong _ f) (reify^R σ t))

  reify^R : ∀ σ {v w : Model σ Γ} → rel PER σ v w → reify σ v ≡ reify σ w
  reify^R ‘Unit      _   = refl
  reify^R ‘Bool      b^R = b^R
  reify^R (σ ‘→ τ) f^R = cong ‘lam (reify^R τ (f^R extend (reflect^R σ refl)))
```

Figure 8.13: Relational Versions of Reify and Reflect

Just like in the definition of the evaluation function, conditional branching is the interesting case. Provided a pair of boolean values (i.e. normal forms of type ‘Bool) which are PER-equal (i.e. syntactically equal) and two pairs of PER-equal σ-values corresponding respectively to the left and right branches of the two if-then-elses, we can prove that the two semantical if-then-else produce PER-equal values. Because of the equality constraint on the booleans, Agda allows us to only write the three cases we are interested in: all the other ones are trivially impossible.

In case the booleans are either ‘tt or ‘ff, we can immediately conclude by invoking one of the hypotheses. Otherwise we remember from fig. 5.9 that the evaluation function produces a value by reflecting the neutral term obtained after reifying both branches. We can play the same game but at the relational level this time and we obtain precisely the proof we wanted.

```
IFTE^R : (B C : Model ‘Bool Γ) → rel PER ‘Bool B C →
         rel PER σ L S → rel PER σ R T → rel PER σ (IFTE B L R) (IFTE C S T)
IFTE^R ‘tt          ‘tt          _   l^R r^R = l^R
IFTE^R ‘ff          ‘ff          _   l^R r^R = r^R
IFTE^R (‘neu a t) (‘neu b u) b^R l^R r^R =
  reflect^R σ (cong_3 ‘ifte (‘neu-injective b^R) (reify^R σ l^R) (reify^R σ r^R))
```

Figure 8.14: Relational If-Then-Else

This provides us with all the pieces necessary to prove our simulation lemma. The relational counterpart of ‘lam is trivial as the induction hypothesis corresponds precisely to the PER-notion of equality on functions. Similarly the case for ‘app is easily discharged: the PER-notion of equality for functions is precisely the strong induction hypothesis we need to be able to make use of the assumption that the respective function's arguments are PER-equal.

```
Eval^Sim : Simulation Eval Eval PER PER
Eval^Sim .th^𝒱ᴿ = λ ρ EQ → th^PER _ EQ ρ
Eval^Sim .varᴿ  = λ ρᴿ v → lookupᴿ ρᴿ v
Eval^Sim .lamᴿ  = λ ρᴿ b bᴿ → bᴿ
Eval^Sim .appᴿ  = λ ρᴿ f t fᴿ tᴿ → fᴿ identity tᴿ
Eval^Sim .ifteᴿ = λ ρᴿ b l r → IFTEᴿ _ _
Eval^Sim .oneᴿ  = λ ρᴿ → refl
Eval^Sim .ttᴿ   = λ ρᴿ → refl
Eval^Sim .ffᴿ   = λ ρᴿ → refl
```

Figure 8.15: Normalisation by Evaluation is in PER-Simulation with Itself

As a corollary, we can deduce that evaluating a term in two environments related pointwise by PER yields two semantic objects themselves related by PER. Which, once reified, give us two equal terms.

```
normᴿ : All PER Γ ρˡ ρʳ → ∀ t → reify σ (eval ρˡ t) ≡ reify σ (eval ρʳ t)
normᴿ ρᴿ t = reifyᴿ σ (Fundamental.lemma Eval^Sim ρᴿ t)
```

Figure 8.16: Normalisation in PER-related Environments Yields Equal Normal Forms

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

Chapter 9

The Fusion Relation

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 lead 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.

9.1 Fusion Constraints

The evidence that $\mathcal{S}^A$, $\mathcal{S}^B$ and $\mathcal{S}^{AB}$ are such that $\mathcal{S}^A$ followed by $\mathcal{S}^B$ is equivalent to $\mathcal{S}^{AB}$ (e.g. Substitution followed by Renaming can be reduced to Substitution) is packed in a record Fusion indexed by the three semantics but also three relations. The first one ($\mathcal{E}^R$) characterises the triples of environments (one for each one of the semantics) which are compatible. The second one ($\mathcal{V}^R$) states what it means for two environment values of $\mathcal{S}^B$ and $\mathcal{S}^{AB}$ respectively to be related. The last one ($\mathcal{C}^R$) relates computations obtained as results of running $\mathcal{S}^B$ and $\mathcal{S}^{AB}$ respectively.

```
record Fusion
  (𝒮ᴬ : Semantics 𝒱ᴬ 𝒞ᴬ) (𝒮ᴮ : Semantics 𝒱ᴮ 𝒞ᴮ) (𝒮ᴬᴮ : Semantics 𝒱ᴬᴮ 𝒞ᴬᴮ)
  (ℰᴿ : ∀ {Γ Δ Θ} → (Γ −Env) 𝒱ᴬ Δ → (Δ −Env) 𝒱ᴮ Θ → (Γ −Env) 𝒱ᴬᴮ Θ → Set)
  (𝒱ᴿ : Rel 𝒱ᴮ 𝒱ᴬᴮ) (𝒞ᴿ : Rel 𝒞ᴮ 𝒞ᴬᴮ) : 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 placeholder $\mathcal{S}^A$ value when going under a binder. Our first two fields are therefore:

```
reifyᴬ : ∀[ 𝒞ᴬ σ ⇒ Term σ ]
var0ᴬ : ∀[ (σ ::_) ⊢ 𝒱ᴬ σ ]
```

Then come two constraints dealing with the relations talking about evaluation environments. $\_\bullet^R\_$ tells us how to extend related environments: one should be able to push related values onto the environments for $\mathcal{S}^B$ and $\mathcal{S}^{AB}$ whilst merely extending the one for $\mathcal{S}^A$ with the token value $\mathsf{var0}^A$.

$\mathsf{th}\hat{}\mathcal{E}^R$ guarantees that it is always possible to thin the environments for $\mathcal{S}^B$ and $\mathcal{S}^{AB}$ in a $\mathcal{E}^R$ preserving manner.

```
_•ᴿ_ : ℰᴿ ρᴬ ρᴮ ρᴬᴮ → rel 𝒱ᴿ σ vᴮ vᴬᴮ →
        ℰᴿ (th^Env 𝒮ᴬ.th^𝒱 ρᴬ extend • var0ᴬ) (ρᴮ • vᴮ) (ρᴬᴮ • vᴬᴮ)
th^ℰᴿ : ℰᴿ ρᴬ ρᴮ ρᴬᴮ → (ρ : Thinning Θ Ω) →
         ℰᴿ ρᴬ (th^Env 𝒮ᴮ.th^𝒱 ρᴮ ρ) (th^Env 𝒮ᴬᴮ.th^𝒱 ρᴬᴮ ρ)
```

Then we have the relational counterpart of the various term constructors. We can once again introduce an extra definition $\mathcal{R}$ which will make the type of the combinators defined later on clearer. $\mathcal{R}$ relates at a given type 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. Note the use of reify^A to recover a Term from a computation in C^A before using the second evaluation function eval^B.

```
ℛ : ∀ σ → (Γ ─Env) 𝒱ᴬ Δ → (Δ ─Env) 𝒱ᴮ Θ → (Γ ─Env) 𝒱ᴬᴮ Θ →
    Term σ Γ → Set
ℛ σ ρᴬ ρᴮ ρᴬᴮ t = rel 𝒞ᴿ σ (evalᴮ ρᴮ (reifyᴬ (evalᴬ ρᴬ t))) (evalᴬᴮ ρᴬᴮ 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.

```
varᴿ : ℰᴿ ρᴬ ρᴮ ρᴬᴮ → (v : Var σ Γ) → ℛ σ ρᴬ ρᴮ ρᴬᴮ (‘var v)
```

Just like for the simulation relation, the relational counterpart of value constructors in the language state that provided that the evaluation environment are related, we expect the computations to be related too.

```
oneᴿ : ℰᴿ ρᴬ ρᴮ ρᴬᴮ → ℛ ‘Unit ρᴬ ρᴮ ρᴬᴮ ‘one
ttᴿ :  ℰᴿ ρᴬ ρᴮ ρᴬᴮ → ℛ ‘Bool ρᴬ ρᴮ ρᴬᴮ ‘tt
ffᴿ :  ℰᴿ ρᴬ ρᴮ ρᴬᴮ → ℛ ‘Bool ρᴬ ρᴮ ρᴬᴮ ‘ff
```

Similarly, we have purely structural constraints for term constructs which have purely structural semantical counterparts. For ‘app and ‘ifte, provided that the evaluation environments are related and that the evaluation of the subterms in each environment respectively are related then the evaluations of the overall terms should also yield related results.

```
appᴿ : ℰᴿ ρᴬ ρᴮ ρᴬᴮ →
       ∀ f t → ℛ (σ ‘→ τ) ρᴬ ρᴮ ρᴬᴮ f → ℛ σ ρᴬ ρᴮ ρᴬᴮ t →
       ℛ τ ρᴬ ρᴮ ρᴬᴮ (‘app f t)
ifteᴿ  : ℰᴿ ρᴬ ρᴮ ρᴬᴮ →
```

```
    ∀ b l r → 𝓡 ‘Bool ρ^A ρ^B ρ^AB b → 𝓡 σ ρ^A ρ^B ρ^AB l → 𝓡 σ ρ^A ρ^B ρ^AB r →
    𝓡 σ ρ^A ρ^B ρ^AB (‘ifte b l r)
```

Finally, the ‘lam-case puts some 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 var0^A. But it is quite natural to have these restrictions: given that reify^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 $\mathcal{S}^{AB}$ on the other hand can be arbitrarily thinned before being extended with related values to be substituted for the variable bound by the ‘lam.

```
lam^R : 𝓔^R ρ^A ρ^B ρ^AB → ∀ b →
        (∀ {Ω} (ρ : Thinning Θ Ω) {v^B v^AB} → rel 𝓥^R σ v^B v^AB →
          let σ^A   = th^Env 𝓢^A.th^𝓥 ρ^A extend • var0^A
              σ^B   = th^Env 𝓢^B.th^𝓥 ρ^B ρ • v^B
              σ^AB  = th^Env 𝓢^AB.th^𝓥 ρ^AB ρ • v^AB
          in 𝓡 τ σ^A σ^B σ^AB b) →
        𝓡 (σ ‘→ τ) ρ^A ρ^B ρ^AB (‘lam b)
```

9.2 Fundamental Lemma of Fusions

As with simulation, we measure the usefulness 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.

```
module Fundamental (𝓕 : Fusion 𝓢^A 𝓢^B 𝓢^AB 𝓔^R 𝓥^R 𝓒^R) where

  open Fusion 𝓕

  lemma : 𝓔^R ρ^A ρ^B ρ^AB → ∀ t → 𝓡 σ ρ^A ρ^B ρ^AB t
  lemma ρ^R (‘var v)      = var^R ρ^R v
  lemma ρ^R (‘app f t)    = app^R ρ^R f t (lemma ρ^R f) (lemma ρ^R t)
  lemma ρ^R (‘lam b)      = lam^R ρ^R b $ λ ren v^R → lemma (th^𝓔^R ρ^R ren •^R v^R) b
  lemma ρ^R ‘one          = one^R ρ^R
  lemma ρ^R ‘tt           = tt^R ρ^R
  lemma ρ^R ‘ff           = ff^R ρ^R
  lemma ρ^R (‘ifte b l r) = ifte^R ρ^R b l r (lemma ρ^R b) (lemma ρ^R l) (lemma ρ^R r)
```

9.3 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 SynFusion record packs the evidence for Syntactic semantics Syn^A, Syn^B and Syn^{AB}. It is indexed by these three Syntactics as well as two relations ($\mathcal{T}^R$ and $\mathcal{E}^R$) corresponding to the $\mathcal{V}^R$ and $\mathcal{E}^R$ ones of the Fusion framework; C^R will always be Eq^R as we are talking about terms.

```
record SynFusion
  (Syn^A : Syntactic 𝒯^A) (Syn^B : Syntactic 𝒯^B) (Syn^AB : Syntactic 𝒯^AB)
  (ℰ^R : ∀ {Γ Δ Θ} → (Γ ─Env) 𝒯^A Δ → (Δ ─Env) 𝒯^B Θ → (Γ ─Env) 𝒯^AB Θ → Set)
  (𝒯^R : Rel 𝒯^B 𝒯^AB) : Set where
```

The first two constraints _•^{R}_ and th^$\mathcal{E}^R$ are directly taken from the Fusion specification: we still need to be able to extend existing related environment with related values, and to thin environments in a relatedness-preserving manner.

```
_•^R_ : ℰ^R ρ^A ρ^B ρ^AB → rel 𝒯^R σ t^B t^AB →
        ℰ^R (th^Env Syn^A.th^𝒯 ρ^A extend • Syn^A.zro) (ρ^B • t^B) (ρ^AB • t^AB)
th^ℰ^R : ℰ^R ρ^A ρ^B ρ^AB → (ρ : Thinning Θ Ω) →
         ℰ^R ρ^A (th^Env Syn^B.th^𝒯 ρ^B ρ) (th^Env Syn^AB.th^𝒯 ρ^AB ρ)
```

We once again define $\mathcal{R}$, a specialised version of its Fusion counterpart stating that the results of the two evaluations are propositionally equal.

```
ℛ : ∀ σ → (Γ ─Env) 𝒯^A Δ → (Δ ─Env) 𝒯^B Θ → (Γ ─Env) 𝒯^AB Θ →
    Term σ Γ → Set
ℛ σ ρ^A ρ^B ρ^AB t = eval^B ρ^B (eval^A ρ^A t) ≡ eval^AB ρ^AB t
```

Once we have $\mathcal{R}$, we can concisely write down the constraint var^R which is also already present in the definition of Fusion.

```
var^R : ℰ^R ρ^A ρ^B ρ^AB → (v : Var σ Γ) → ℛ σ ρ^A ρ^B ρ^AB (`var v)
```

Finally, we have a fourth constraint (zro^R) saying that Syn^B and Syn^{AB}'s respective zros are producing related values. This will provide us with just the right pair of related values to use in Fusion's lam^R.

```
zro^R : rel 𝒯^R σ {σ :: Γ} Syn^B.zro Syn^AB.zro
```

Everything else is a direct consequence of the fact we are only considering syntactic semantics. Given a SynFusion relating three Syntactic semantics, we get a Fusion relating the corresponding Semantics where C^R is Eq^R, the pointwise lifting of propositional equality. The proof relies on the way the translation from Syntactic to Semantics is formulated in section 4.5.

We are now ready to give our first examples of Fusions. They are the first results one typically needs to prove when studying the meta-theory of a language.

9.4 Interactions of Renaming and Substitution

Renaming and Substitution can interact in four ways: all but one of these combinations is equivalent to a single substitution (the sequential execution of two renamings is

```
module Fundamental (𝓕 : SynFusion Syn^A Syn^B Syn^AB 𝓔^R 𝓣^R) where

  open SynFusion 𝓕

  lemma : Fusion (fromSyn Syn^A) (fromSyn Syn^B) (fromSyn Syn^AB) 𝓔^R 𝓣^R Eq^R
  lemma .Fusion.reify^A = id
  lemma .Fusion.var0^A = Syn^A.zro
  lemma .Fusion._•^R_  = _•^R_
  lemma .Fusion.th^𝓔^R = th^𝓔^R
  lemma .Fusion.var^R  = var^R
  lemma .Fusion.one^R  = λ ρ^R → refl
  lemma .Fusion.tt^R   = λ ρ^R → refl
  lemma .Fusion.ff^R   = λ ρ^R → refl
  lemma .Fusion.app^R  = λ ρ^R f t → cong₂ ‘app
  lemma .Fusion.ifte^R = λ ρ^R b l r → cong₃ ‘ifte
  lemma .Fusion.lam^R  = λ ρ^R b b^R → cong ‘lam (b^R extend zro^R)
```

Figure 9.1: Fundamental Lemma of Syntactic Fusions

equivalent to a single renaming). These four lemmas are usually proven in painful separation. Here we discharge them by rapid successive instantiation of our framework, using the earlier results to satisfy the later constraints. We only present the first instance in full details and then only spell out the SynFusion type signature which makes explicit the relations used to constraint the input environments.

First, we have the fusion of two sequential renaming traversals into a single renaming. Environments are related as follows: the composition of the two environments used in the sequential traversals should be pointwise equal to the third one. The composition operator select is defined in fig. 4.11.

```
RenRen : SynFusion Syn^Ren Syn^Ren Syn^Ren
                   (λ ρ^A ρ^B → All Eq^R _ (select ρ^A ρ^B)) Eq^R
RenRen ._•^R_  = λ ρ^R t^R → pack^R λ where
  z      → t^R
  (s v) → lookup^R ρ^R v
RenRen .th^𝓔^R = λ ρ^R ρ → cong (λ v → th^Var v ρ) <$>^R ρ^R
RenRen .var^R  = λ ρ^R v → cong ‘var (lookup^R ρ^R v)
RenRen .zro^R  = refl
```

Figure 9.2: Syntactic Fusion of Two Renamings

Using the fundamental lemma of syntactic fusions, we get a proper Fusion record on which we can then use the fundamental lemma of fusions to get the renaming fusion law we expect.

```
renren : (t : Term σ Γ) → ren ρ₂ (ren ρ₁ t) ≡ ren (select ρ₁ ρ₂) t
renren = let fus = Syntactic.Fundamental.lemma RenRen
         in Fusion.Fundamental.lemma fus reflᴿ
```

Figure 9.3: Corollary: Renaming Fusion Law

A similar proof gives us the fact that a renaming followed by a substitution is equivalent to a substitution. Environments are once more related by composition.

```
RenSub : SynFusion Syn^Ren Syn^Sub Syn^Sub
                   (λ ρᴬ ρᴮ → All Eqᴿ _ (select ρᴬ ρᴮ)) Eqᴿ
```

```
rensub : (t : Term σ Γ) → sub ρ₂ (ren ρ₁ t) ≡ sub (select ρ₁ ρ₂) t
rensub = let fus = Syntactic.Fundamental.lemma RenSub
         in Fusion.Fundamental.lemma fus reflᴿ
```

Figure 9.4: Renaming - Substitution Fusion Law

For the proof that a substitution followed by a renaming is equivalent to a substitution, we need to relate the environments in a different manner: composition now amounts to applying the renaming to every single term in the substitution. We also depart from the use of Eq^R as the relation for values: indeed we now compare variables and terms. The relation VarTerm^R defined in fig. 8.7 relates variables and terms by wrapping the variable in a ‘var constructor and demanding it is equal to the term.

```
SubRen : SynFusion Syn^Sub Syn^Ren Syn^Sub
                   (λ ρᴬ ρᴮ → All Eqᴿ _ (ren ρᴮ <$> ρᴬ)) VarTermᴿ
```

```
subren : (t : Term σ Γ) → ren ρ₂ (sub ρ₁ t) ≡ sub (ren ρ₂ <$> ρ₁) t
subren = let fus = Syntactic.Fundamental.lemma SubRen
         in Fusion.Fundamental.lemma fus reflᴿ
```

Figure 9.5: Substitution - Renaming Fusion Law

Finally, the fusion of two sequential substitutions into a single one uses a similar notion of composition. Here the second substitution is applied to each term of the first and we expect the result to be pointwise equal to the third. Values are once more considered related whenever they are propositionally equal.

```
SubSub : SynFusion Syn^Sub Syn^Sub Syn^Sub
                 (λ ρ^A ρ^B → All Eq^R _ (sub ρ^B <$> ρ^A)) Eq^R


subsub : (t : Term σ Γ) → sub ρ₁ (sub ρ₂ t) ≡ sub (sub ρ₁ <$> ρ₂) t
subsub = let fus = Syntactic.Fundamental.lemma SubSub
         in Fusion.Fundamental.lemma fus refl^R
```

Figure 9.6: Substitution Fusion Law

As we are going to see in the following section, we are not limited to Syntactic statements.

9.5 Other Examples of Fusions

The most simple example of fusion of two Semantics involving a non Syntactic one is probably the proof that Renaming followed by normalization by evaluation's Eval is equivalent to Eval with an adjusted environment.

Fusion of Renaming Followed by Evaluation

As is now customary, we start with an auxiliary definition which will make our type signatures a lot lighter. It is a specialised version of the relation $\mathcal{R}$ introduced when spelling out the Fusion constraints. Here the relation is PER and the three environments carry respectively Var (i.e. it is a Thinning) for the first one, and Model values for the two other ones.

```
𝓡 : ∀ {Γ Δ Θ} σ (ρ^A : Thinning Γ Δ) (ρ^B : (Δ −Env) Model Θ)
    (ρ^AB : (Γ −Env) Model Θ) → Term σ Γ → Set
𝓡 σ ρ^A ρ^B ρ^AB t = rel PER σ (eval ρ^B (th^Term t ρ^A)) (eval ρ^AB t)
```

We start with the most straigtforward of the non-trivial cases: the relational counterpart of ‘app. The Kripke^R structure of the induction hypothesis for the function has precisely the strength we need to make use of the hypothesis for its argument.

```
APP^R : ∀ f t → 𝓡 (σ ‘→ τ) ρ^A ρ^B ρ^AB f → 𝓡 σ ρ^A ρ^B ρ^AB t →
          𝓡 τ ρ^A ρ^B ρ^AB (‘app f t)
APP^R f t f^R t^R = f^R identity t^R
```

Figure 9.7: Relational Application

The relational counterpart of ‘ifte is reminiscent of the one we used when proving that normalisation by evaluation is in simulation with itself in fig. 8.14: we have two arbitrary boolean values resulting from the evaluation of b in two distinct manners but we know them to be the same thanks to them being PER-related. The canonical cases are trivially solved by using one of the assumptions whilst the neutral case can be proven to hold thanks to the relational versions of reify and reflect.

```
IFTEᴿ : ∀ b l r → 𝓡 'Bool ρᴬ ρᴮ ρᴬᴮ b → 𝓡 σ ρᴬ ρᴮ ρᴬᴮ l → 𝓡 σ ρᴬ ρᴮ ρᴬᴮ r →
          𝓡 σ ρᴬ ρᴮ ρᴬᴮ ('ifte b l r)
IFTEᴿ b l r bᴿ lᴿ rᴿ with eval ρᴮ (th^Term b ρᴬ) | eval ρᴬᴮ b
... | 'tt          | 'tt          = lᴿ
... | 'ff          | 'ff          = rᴿ
... | 'neu _ b₁ | 'neu _ b₂ =
  reflectᴿ σ $ cong₃ 'ifte ('neu-injective bᴿ) (reifyᴿ σ lᴿ) (reifyᴿ σ rᴿ)
```

Figure 9.8: Relational If-Then-Else

The rest of the constraints can be discharged fairly easily; either by using a constructor, combining some of the provided hypotheses or using general results such as the stability of PER-relatedness under thinning of the Model values.

```
RenEval : Fusion Renaming Eval Eval
                   (λ ρᴬ ρᴮ → All PER _ (select ρᴬ ρᴮ)) PER PER
RenEval .reifyᴬ    = id
RenEval .var0ᴬ     = z
RenEval ._•ᴿ_      = λ ρᴿ vᴿ → vᴿ ::ᴿ lookupᴿ ρᴿ
RenEval .th^𝓔ᴿ     = λ ρᴿ ρ → (λ v → th^PER _ v ρ) <$>ᴿ ρᴿ
RenEval .varᴿ      = λ ρᴿ → lookupᴿ ρᴿ
RenEval .oneᴿ      = λ ρᴿ → refl
RenEval .ttᴿ       = λ ρᴿ → refl
RenEval .ffᴿ       = λ ρᴿ → refl
RenEval .appᴿ      = λ ρᴿ → APPᴿ
RenEval .ifteᴿ     = λ ρᴿ → IFTEᴿ
RenEval .lamᴿ      = λ ρᴿ b bᴿ → bᴿ
```

Figure 9.9: Renaming Followed by Evaluation is an Evaluation

By the fundamental lemma of Fusion, we get the result we are looking for: a renaming followed by an evaluation is equivalent to an evaluation in a touched up environment.

This gives us the tools to prove the substitution lemma for evaluation.

```
reneval : (th : Thinning Γ Δ) (ρ : (Δ ─Env) Model Θ) → All PER Δ ρ ρ →
          ∀ t → rel PER σ (eval ρ (th^Term t th)) (eval (select th ρ) t)
reneval th ρ ρ^R t = Fundamental.lemma RenEval (select^R th ρ^R) t
```

Figure 9.10: Corollary: Fusion Principle for Renaming followed by Evaluation

Substitution Lemma for Evaluation

Given any semantics, the substitution lemma (see for instance Mitchell and Moggi [1991]) states that evaluating a term after performing a substitution is equivalent to evaluating the term with an environment obtained by evaluating each term in the substitution. Formally (t is a term, γ a substitution, ρ an evaluation environment, _[_] denotes substitution, and ⟦_⟧_ evaluation):

$$[\![\, t \, [\gamma] \,]\!] \, \rho \;\equiv\; [\![\, t \,]\!] \, ([\![\, \gamma \,]\!] \, \rho)$$

This is a key lemma in the study of a language's meta-theory and it fits our Fusion framework perfectly. We start by describing the constraints imposed on the environments. They may 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 and help root out exotic term (cf. fig. 8.10).

First we expect the two evaluation environments to only contain Model values which are PER-related to themselves. Second, we demand that the evaluation of the substitution in a *thinned* version of the first evaluation environment is PER-related in a pointwise manner to the *similarly thinned* second evaluation environment. This constraint amounts to a weak commutation lemma between evaluation and thinning; a stronger version would be to demand that thinning of the result is equivalent to evaluation in a thinned environment.

```
Sub^R : (Γ ─Env) Term Δ → (Δ ─Env) Model Θ → (Γ ─Env) Model Θ → Set
Sub^R ρ^A ρ^B ρ^AB = All PER Δ ρ^B ρ^B × All PER Γ ρ^AB ρ^AB ×
  (∀ {Ω} (ρ : Thinning Θ Ω) →
  All PER Γ (eval (th^Env (th^Model _) ρ^B ρ) <$> ρ^A)
             (th^Env (th^Model _) ρ^AB ρ))
```

Figure 9.11: Constraints on Triples of Environments for the Substitution Lemma

We can then state and prove the substitution lemma using Sub^R as the constraint on environments and PER as the relation for both values and computations.

The proof is similar to that of fusion of renaming with evaluation in section 9.5: we start by defining a notation $\mathcal{R}$ to lighten the types, then combinators APP^R and IFTE^R. The cases for th^$\mathcal{E}^R$, _•^{R}_, and var^R are a bit more tedious: they rely crucially on the fact that we can prove a fusion principle and an identity lemma for th^Model as well

```
SubEval : Fusion Substitution Eval Eval Sub^R PER PER
```

Figure 9.12: Substitution Followed by Evaluation is an Evaluation

as an appeal to reneval (fig. 9.10) and multiple uses of Eval^Sim (fig. 8.15). Because the technical details do not give any additional hindsight, we do not include the proof here.

Chapter 10

Conclusion

10.1 Summary

We have demonstrated that we can exploit the shared structure highlighted by the introduction of Semantics to 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. We then provided useful instances of this schema thus proving that syntactic traversals are extensional, that renaming is a special case of substitution or even that normalisation by evaluation produces equal normal forms provided PER-related evaluation environments.

A more involved second logical relation gave us a general description of fusion of traversals where we study triples of semantics such that composing the two first ones would yield an instance of the third one. We then saw that the four lemmas about the possible interactions of pairs of renamings and/or substitutions are all instances of this general framework and can be proven sequentially, the later results relying on the former ones. We then went on to proving the substitution lemma for Normalisation by Evaluation.

10.2 Related Work

Benton, Hur, Kennedy and McBride's joint work (2012) was not limited to defining traversals. They proved fusion lemmas describing the interactions of renaming and substitution using tactics rather than defining a generic proof framework like we do. They have also proven the evaluation function of their denotational semantics correct; however they chose to use propositional equality and to assume function extensionality rather than resorting to the traditional Partial Equivalence Relation approach we use.

Throught the careful study of the recursion operator associated to each strictly positive datatype, Malcolm defined proof principles (Malcolm [1990]) which can be also used as optimisation principles, just like our fusion principles. Other optimisations such as deforestation (Wadler [1990]) or transformation to an equivalent but tail-recursive program (Tomé Cortiñas and Swierstra [2018]) have seen a generic treatment.

10.3 Further work

We have now fulfilled one of the three goals we highlighted in chapter 7. The question of finding more instances of Semantics and of defining a generic notion of Semantics for all syntaxes with binding is still open. Analogous questions for the proof frameworks arise naturally.

Other Instances

We have only seen a handful of instances of both the Simulation lemma and the Fusion one. They already give us important lemmas when studying the meta-theory of a language. However there are potential opportunities for more instances to be defined.

We would like to know whether the idempotence of normalisation by evaluation can be proven as a corollary of a fusion lemma for evaluation. This would give us a nice example of a case where the reify^{A} is not the identity and actually does some important work.

Another important question is whether it is always possible to fuse a preliminary renaming followed by a semantics $\mathcal{S}$ into a single pass of $\mathcal{S}$. Note that the Printer semantics guarantees that this cannot be true when the preliminary traversal is a substitution.

Other Proof Frameworks

After implementing Simulation and Fusion, we can wonder whether there are any other proof schemas we can make formal.

As we have explained in chapter 8, Simulation gives the *relational* interpretation of evaluation. Defining a similar framework dealing with a single semantics would give us the *predicate* interpretation of evaluation. This would give a generalisation of the fundamental lemma of logical predicates which, once specialised to substitution, would be exactly the traditional definition one would expect.

Another possible candidate is an Identity framework which would, provided that some constraints hold of the values in the environment, an evaluation is the identity. So far we have only related pairs of evaluation results but to prove an identity lemma we would need to relate the evaluation of a term to the original term itself. Although seamingly devoid of interest, identity lemmas are useful in practice both when proving or when optimising away useless traversals.

We actually faced these two challenges when working on the POPLMark Reloaded challenge (Abel et al. [2018]). We defined the proper generalisation of the fundamental lemma of logical predicates but could only give ad-hoc identity lemmas for renaming (and thus substitution because they are in simulation).

Generic Proof Frameworks

In the next part, we are going to define a universe of syntaxes with binding and a generic notion of semantics over these syntaxes. We naturally want to be able to also prove generic results about these generic traversals. We are going to have to need to generalise the proof frameworks to make them syntax generic.

Part II

A Universe of Well Kinded-and-Scoped Syntaxes with Binding, their Programs and Proofs

Chapter 11

Plea For a Universe of Syntaxes with Binding

Now that we have a way to structure our traversals and proofs about them, we can tackle a practical example. Let us look at the formalisation of an apparently straightforward program transformation: the inlining of let-bound variables by substitution and the proof of a simple correctness lemma. We have two languages: the source (S), which has let-bindings, and the target (T), which only differs in that it does not. We want to write a function elaborating source term into target ones and then prove that each reduction step on the source term can be simulated by zero or more reduction steps on its elaboration.

Breaking the task down, we need to start by defining the two languages. We already now how to do this from chapter 3. The only downside is that we need to write down the same constructor types twice for ‘var, ‘lam, and ‘app.

```
data S : Type −Scoped where
  ‘var  : ∀[ Var σ ⇒ S σ ]
  ‘lam  : ∀[ (σ ::_) ⊢ S τ ⇒ S (σ ‘→ τ) ]
  ‘app  : ∀[ S (σ ‘→ τ) ⇒ S σ ⇒ S τ ]
  ‘let  : ∀[ S σ ⇒ (σ ::_) ⊢ S τ ⇒ S τ ]

data T : Type −Scoped where
  ‘var  : ∀[ Var σ ⇒ T σ ]
  ‘lam  : ∀[ (σ ::_) ⊢ T τ ⇒ T (σ ‘→ τ) ]
  ‘app  : ∀[ T (σ ‘→ τ) ⇒ T σ ⇒ T τ ]
```

Figure 11.1: Source and Target Languages

Ignoring for now the Semantics framework, we jump straight to defining the program transformation we are interested in. We notice immediately that we need to prove T to be Thinnable first so that we may push the environment of inlined terms under binders. We also notice that the only interesting case is the one dealing with ‘let: all the other ones are purely structural.

```
unlet : (Γ –Env) T Δ → S σ Γ → T σ Δ
unlet ρ (‘var v)   = lookup ρ v
unlet ρ (‘lam b)   = ‘lam (unlet (th^Env th^T ρ extend • ‘var z) b)
unlet ρ (‘app f t) = ‘app (unlet ρ f) (unlet ρ t)
unlet ρ (‘let e t) = unlet (ρ • unlet ρ e) t
```

Figure 11.2: Let-Inlining Traversal

We now want to state our correctness lemma: each reduction step on a source term can be simulated by zero or more reduction steps on its elaboration. We need to define an operational semantics for each language. We only show the one for S in fig. 11.3: the one for T is exactly the same minus the ‘let-related rules. We immediately notice that to write down the type of β we need to define substitution (and thus renaming) for each of the languages.

```
data _⊢_∋_⇝S_ : ∀ Γ σ → S σ Γ → S σ Γ → Set where
-- computation
  β     : ∀ (b : S τ (σ :: Γ)) u → Γ ⊢ τ ∋ ‘app (‘lam b) u ⇝S b ⟨ u /0⟩^S
  ζ     : ∀ e (t : S τ (σ :: Γ)) → Γ ⊢ τ ∋ ‘let e t ⇝S t ⟨ e /0⟩^S
-- structural
  ‘lam  : (σ :: Γ) ⊢ τ ∋ b ⇝S c → Γ ⊢ σ ‘→ τ ∋ ‘lam b ⇝S ‘lam c
  ‘appl : Γ ⊢ σ ‘→ τ ∋ f ⇝S g → ∀ t → Γ ⊢ τ ∋ ‘app f t ⇝S ‘app g t
  ‘appr : ∀ f → Γ ⊢ σ ∋ t ⇝S u → Γ ⊢ τ ∋ ‘app f t ⇝S ‘app f u
  ‘letl : Γ ⊢ σ ∋ d ⇝S e → ∀ t → Γ ⊢ τ ∋ ‘let d t ⇝S ‘let e t
  ‘letr : ∀ e → (σ :: Γ) ⊢ τ ∋ t ⇝S u → Γ ⊢ τ ∋ ‘let e t ⇝S ‘let e u
```

Figure 11.3: Operational Semantics for the Source Language

In the course of simply stating our problem, we have already had to define two eerily similar languages, spell out all the purely structural cases when defining the transformation we are interested in and implement four auxiliary traversals which are essentially the same.

In the course of proving the correctness lemma (which we abstain from doin here), we discover that we need to prove eight lemmas about the interactions of renaming, substitution, and let-inlining. They are all remarkably similar, but must be stated and proved separately (e.g, as in Benton et al. [2012]).

Even after doing all of this work, we have only a result for a single pair of source and target languages. If you were to change our languages S or T, we would have to repeat the same work all over again or at least do a lot of cutting, pasting, and editing. And if we add more constructs to both languages, we will have to extend our transformation with more and more code that essentially does nothing of interest.

This state of things is not inevitable. After having implemented numerous semantics in part I, we have gained an important insight: the structure of the constraints telling us how to define a Semantics is tightly coupled to the definition of the language. So much so that we should in fact be able to *derive* them directly from the definition of the language.

This is what we set out to do in this part and in particular in section 14.2 where we define a *generic* notion of let-binding to extend any language with together with the corresponding generic let-inlining transformation.

Chapter 12

A Primer on the Universe of Data Types

Chapman, Dagand, McBride and Morris (CDMM, 2010) defined a universe of data types inspired by Dybjer and Setzer's finite axiomatisation of Inductive-Recursive definitions (1999) and Benke, Dybjer and Jansson's universes for generic programs and proofs (2003). This explicit definition of *codes* for data types empowers the user to write generic programs tackling *all* of the data types one can obtain this way. In this section we recall the main aspects of this construction we are interested in to build up our generic representation of syntaxes with binding.

12.1 Descriptions and Their Meaning as Functors

The first component of CDMM's universe's definition is an inductive type of Descriptions of strictly positive functors from $\mathbf{Set}^J$ to $\mathbf{Set}^I$. It has three constructors: 'σ to store data (the rest of the description can depend upon this stored value), 'X to attach a recursive substructure indexed by J and '■ to stop with a particular index value.

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

Figure 12.1: Datatype Descriptions

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

These constructors give the programmer the ability to build up the data types they are used to. For instance, the functor corresponding to lists of elements in A stores a Boolean which stands for whether the current node is the empty list or not. Depending

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

Figure 12.2: Descriptions’ meanings as Functors

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

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

Figure 12.3: The Description of the base functor for List A

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

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

Figure 12.4: The Description of the base functor for Vec A n

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

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

Figure 12.5: Action on Morphisms of the Functor corresponding to a Description

12.2 Datatypes as Least Fixpoints

All the functors obtained as meanings of Descriptions are strictly positive. So we can build the least fixpoint of the ones that are endofunctors i.e. the ones for which *I* equals *J*. This fixpoint is called μ and its iterator is given by the definition of fold *d*. In fig. 12.6 the Size (Abel [2010]) index added to the inductive definition of μ plays a crucial role in getting the termination checker to see that fold is a total function, just like sizes played a crucial role in proving that map^Rose was total in section 2.2.

```
data μ (d : Desc I I) (s : Size) : I → Set where
  ‘con : {s’ : Size< s} → ⟦ d ⟧ (μ d s’) i → μ d s i

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

Figure 12.6: Least Fixpoint of an Endofunctor and Corresponding Generic Fold

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

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

Chapter 13

A Universe of Scope Safe and Well Kinded Syntaxes

Our universe of scope safe and well kinded syntaxes follows the same principle as CDMM's universe of datatypes, except that we are not building endofunctors on Set^I any more but rather on I –Scoped. We now think of the index type I as the sorts used to distinguish terms in our embedded language. The 'σ and '■ constructors are as in the CDMM Desc type, and are used to represent data and index constraints respectively.

13.1 Descriptions and Their Meaning as Functors

What distinguishes this new universe Desc from that of Section 12 is that the 'X constructor is now augmented with an additional List I argument that describes the new binders that are brought into scope at this recursive position. This list of the kinds of the newly-bound variables will play a crucial role when defining the description's semantics as a binding structure in figs. 13.2, 13.3 and 13.5.

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

Figure 13.1: Syntax Descriptions

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

The astute reader may have noticed that ⟦_⟧ is uniform in X and Γ; however refactoring ⟦_⟧ to use the partially applied X _ _ Γ following this observation would

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

Figure 13.2: Descriptions’ Meanings

lead to a definition harder to use with the combinators for indexed sets described in section 2.3 which make our types much more readable.

If we pre-compose the meaning function ⟦_⟧ with a notion of ‘de Bruijn scopes’ (denoted Scope here) which turns any I −Scoped family into a function of type List $I \to I$ −Scoped by appending the two List indices, we recover a meaning function producing an endofunctor on I −Scoped.

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

Figure 13.3: De Bruijn Scopes

So far we have only shown the action of the functor on objects; its action on morphisms is given by a function fmap defined by induction over the description just like in Section 12. We give fmap the most general type we can, the action of functors is then a specialized version of it.

```
fmap : (d : Desc I) → (∀ Θ i → X Θ i Γ → Y Θ i Δ) → ⟦ d ⟧ X i Γ → ⟦ d ⟧ Y i Δ
fmap (‘σ A d)   f = Prod.map₂ (fmap (d _) f)
fmap (‘X Δ j d) f = Prod.map (f Δ j) (fmap d f)
fmap (‘■ i)     f = id
```

Figure 13.4: Action of Syntax Functors on Morphism

13.2 Terms as Free Relative Monads

The endofunctors thus defined are strictly positive and we can take their fixpoints. As we want to define the terms of a language with variables, instead of considering the initial algebra, this time we opt for the free relative monad (Altenkirch et al. [2010, 2014]) with respect to the functor Var. The ‘var constructor corresponds to return, and we will define bind (also known as the parallel substitution sub) in the next section. We have once more a Size index to get all the benefits of type based termination checking when defining traversals over terms.

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

Figure 13.5: Term Trees: The Free Var-Relative Monads on Descriptions

Because we often use closed terms of size ∞ (that is to say fully-defined) in concrete examples, we name this notion.

```
TM : Desc I → I → Set
TM d i = Tm d ∞ i []
```

Figure 13.6: Type of Closed Terms

Examples of Syntaxes With Binding

Coming back to our original example, we now have the ability to give a code for the intrinsically typed STλC. But we start with a simpler example to lay down the foundations: the well scoped untyped λ-calculus. In both cases, the variable case will be added by the free monad construction so we only have to describe two constructors: application and λ-abstraction.

Untyped languages are, as Harper would say, uni-typed syntaxes and can thus be modelled using descriptions whose kind parameter is the unit type. We take the disjoint sum of the respective descriptions for the application and λ-abstraction constructors by using the classic construction in type theory: a dependent pair of a Bool picking one of the two branches and a second component whose type is either that of application or λ-abstraction depending on that boolean.

Application has two substructures (‘X) which do not bind any extra argument and λ-abstraction has exactly one substructure with precisely one extra bound variable. Both constructors’ descriptions end with (‘■ tt), the only inhabitant of the trivial kind.

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

Figure 13.7: Description of The Untyped Lambda Calculus

Typed syntax comes with extra constraints: our tags need to carry extra information about the types involved so we use the ad-hoc ‘STLC type. The description is then the dependent pairing of an ‘STLC tag together with its decoding defined by a pattern-matching λ-expression in Agda.

Application has two substructures none of which bind extra variables. The first has a function type and the second the type of its domain. The overall type of the branch is enforced to be that of the function's codomain by the ‘■ constructor.

λ-abstraction has exactly one substructure of type τ with a newly-bound variable of type σ. The overall type of the branch is once more enforced by ‘■: it is (σ ‘→ τ).

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

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

Figure 13.8: Description of the Simply Typed Lambda Calculus

For convenience we use Agda's pattern synonyms corresponding to the original constructors in section 3.2: ‘app for application and ‘lam for λ-abstraction. These synonyms can be used when pattern-matching on a term and Agda resugars them when displaying a goal. This means that the end user can seamlessly work with encoded terms without dealing with the gnarly details of the encoding. These pattern definitions can omit some arguments by using “_”, in which case they will be filled in by unification just like any other implicit argument: there is no extra cost to using an encoding! The only downside is that the language currently does not allow the user to specify type annotations for pattern synonyms.

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

‘id : TM STLC (σ ‘→ σ)
‘id = ‘lam (‘var z)
```

Figure 13.9: Recovering Readable Syntax via Pattern Synonyms

13.3 Common Combinators and Their Properties

In order to avoid repeatedly re-encoding the same logic, we introduce combinators demonstrating that descriptions are closed under finite sums and finite products of recursive positions.

Closure under Disjoint Union

As we wrote, the construction used in fig. 13.7 to define the syntax for the untyped λ-calculus is classic. It is actually the third time (the first and second times being the definition of listD and vecD in figs. 12.3 and 12.4) that we use a Bool to distinguish between two constructors.

We define once and for all the disjoint union of two descriptions thanks to the _‘+_ combinator. It comes togeter with an appropriate eliminator case which, given two continuations, picks the one corresponding to the chosen branch.

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


case : (⟦ d ⟧ X i Γ → A) → (⟦ e ⟧ X i Γ → A) → (⟦ d ‘+ e ⟧ X i Γ → A)
case l r (true  , t) = l t
case l r (false , t) = r t
```

Figure 13.10: Descriptions are Closed Under Disjoint Sums

A concrete use case for the disjoint union combinator and its eliminator will be given in section 14.2 where we explain how to seamlessly enrich any existing syntax with let-bindings and how to use the Semantics framework to elaborate them away.

Closure Under Finite Product of Recursive Positions

Closure under product does not hold in general. Indeed, the equality constraints introduced by the two end tokens of two descriptions may be incompatible. So far, a limited form of closure (closure under finite product of recursive positions) has been sufficient for all of our use cases. Provided a list of pairs of context extensions and kinds, we can add to an existing description that many recursive substructures.

```
‘Xs : List (List I × I) → Desc I → Desc I
‘Xs Δjs d = foldr (uncurry ‘X) d Δjs
```

Figure 13.11: Descriptions are Closed Under Finite Product of Recursive Positions

As with coproducts, we can define an appropriate eliminator. The function unXs takes a value in the encoding and extracts its constituents (All *P xs* is defined in Agda's standard library and makes sure that the predicate *P* holds true of all the elements in the list *xs*).

```
unXs : ∀ Δjs → ⟦ ‘Xs Δjs d ⟧ X i Γ →
           All (uncurry $ λ Δ j → X Δ j Γ) Δjs × ⟦ d ⟧ X i Γ

unXs []          v       = [] , v
unXs (σ :: Δ) (r , v) = Prod.map₁ (r ::_) (unXs Δ v)
```

Figure 13.12: Breaking Down a Finite Product of Recursive Positions

We will see in section 14.2 how to define let-bindings as a generic language extension and their inlining as a generic semantics over the extended syntax and into the base one. Closure under a finite product of recursive positions demonstrates that we could extend this construction to parallel (or even mutually-recursive) let-bindings where the number and the types of the bound expressions can be arbitrary. We will not go into the details of this construction as it is essentially a combination of Xs, unXs and the techniques used when defining single let-bindings.

Chapter 14

Generic Scope Safe and Well Kinded Programs for Syntaxes

The set of constraints we called a Semantics in section 4.4 for the specific example of the simply typed λ-calculus could be divided in two groups: the one arising from the fact that we need to be able to push environment values under binders and the ones in one-to-one correspondence with constructors for the language.

Based on this observation, we can define a generic notion of semantics for all syntax descriptions. It is once more parametrised by two (I–Scoped) families $\mathcal{V}$ and C corresponding respectively to values associated to bound variables and computations delivered by evaluating terms.

```
record Semantics (d : Desc I) (𝒱 C : I –Scoped) : Set where
```

These two families have to abide by three constraints. First, values should be thinnable so that we can push the evaluation environment under binders.

```
th^𝒱 : Thinnable (𝒱 σ)
```

Second, values should embed into computations for us to be able to return the value associated to a variable in the environment as the result of its evaluation.

```
var : ∀[ 𝒱 σ ⇒ C σ ]
```

Third, we have a constraint similar to the one needed to define fold in chapter 12 (fig. 12.6). We should have an algebra taking a term whose substructures have already been evaluated and returning a computation for the overall term.

```
alg : ∀[ ⟦ d ⟧ (Kripke 𝒱 C) σ ⇒ C σ ]
```

To make formal this idea of "hav[ing] already been evaluated" we crucially use the fact that the meaning of a description is defined in terms of a function interpreting substructures which has the type (List I → I–Scoped), i.e. that gets access to the current scope but also the exact list of the newly bound variables' kinds.

We define a function Kripke by case analysis on the number of newly bound variables. It is essentially a subcomputation waiting for a value associated to each one of the fresh variables. If it's 0 we expect the substructure to be a computation corresponding to the result of the evaluation function's recursive call; but if there are newly bound variables then we expect to have a function space. In any context extension, it will take an environment of values for the newly-bound variables and produce a computation corresponding to the evaluation of the body of the binder.

```
Kripke : (𝒱 C : I –Scoped) → (List I → I –Scoped)
Kripke 𝒱 C [] j = C j
Kripke 𝒱 C Δ j = □ ((Δ –Env) 𝒱 ⇒ C j)
```

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

```
_–Comp : List I → I –Scoped → List I → Set
(Γ –Comp) C Δ = ∀ {s σ} → Tm d s σ Γ → C σ Δ
```

Figure 14.1: _–Comp: Associating Computations to Terms

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

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

Figure 14.2: Statement of the Fundamental Lemma of Semantics

The proof of semantics is straightforward now that we have clearly identified the problem's structure and the constraints we need to enforce. If the term considered is a variable, we lookup the associated value in the evaluation environment and turn it into a computation using var. If it is a non variable constructor then we call fmap to evaluate the substructures using body and then call the algebra to combine these results.

The auxiliary lemma body distinguishes two cases. If no new variable has been bound in the recursive substructure, it is a matter of calling semantics recursively.

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

Figure 14.3: Proof of the Fundamental Lemma of Semantics – semantics

Otherwise we are provided with a Thinning, some additional values and evaluate the substructure in the thinned and extended evaluation environment thanks to a auxiliary function _>>_ which given two environments (Γ –Env) $\mathcal{V}$ Θ and (Δ –Env) $\mathcal{V}$ Θ produces an environment ((Γ ++ Δ) –Env) $\mathcal{V}$ Θ).

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

Figure 14.4: Proof of the Fundamental Lemma of Semantics – body

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

Because most of our examples involve closed terms (for which we have introduced a special notation in fig. 13.6), we define a specialised of the fundamental lemma of semantics for closed terms and apply it to the empty environment.

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

Figure 14.5: Special Case: Fundamental Lemma of Semantics for Closed Terms

14.1 Our First Generic Programs: Renaming and Substitution

Similarly to section 4.5 renaming and substitutions can be defined generically for all syntax descriptions.

Renaming is a semantics with Var as values and Tm as computations. The first two constraints on Var described earlier are trivially satisfied. Observing that renaming strictly respects the structure of the term it goes through, it makes sense for the algebra to be implemented using fmap. When dealing with the body of a binder, we ‘reify’ the Kripke function by evaluating it in an extended context and feeding it placeholder values corresponding to the extra variables introduced by that context. This is reminiscent both of what we did in section 4.5 and the definition of reification in the setting of normalisation by evaluation (see e.g. Coquand's work 2002).

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

Figure 14.6: Renaming: A Generic Semantics for Syntaxes with Binding

From this instance, we can derive the proof that all terms are Thinnable as a corollary of the fundamental lemma of Semantics.

```
th^Tm : Thinnable (Tm d ∞ σ)
th^Tm t ρ = Semantics.semantics Renaming ρ t
```

Figure 14.7: Corollary: Generic Thinning

Substitution is defined in a similar manner with Tm as both values and computations. Of the two constraints applying to terms as values, the first one corresponds to renaming and the second one is trivial. The algebra is once more defined by using fmap and reifying the bodies of binders. We can, once more, obtain parallel substitution as a corollary of the fundamental lemma of Semantics.

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

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

Figure 14.8: Generic Parallel Substitution for All Syntaxes with Binding

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

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

Figure 14.9: VarLike: Thinnable and with placeholder values

For any VarLike $\mathcal{V}$, we can define fresh^r of type $((\Gamma$ −Env$)\ \mathcal{V}\ (\Delta$ ++ $\Gamma))$ and fresh^l of type $((\Gamma$ −Env$)\ \mathcal{V}\ (\Gamma$ ++ $\Delta))$ by combining the use of placeholder values and thinnings. Hence, we can then write a generic reify (fig. 14.10) turning Kripke function spaces from $\mathcal{V}$ to C into Scopes of C computations.

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

Figure 14.10: Generic Reification thanks to VarLike Values

We can now showcase other usages by providing a catalogue of generic programs for syntaxes with binding.

14.2 Sugar and Desugaring as a Semantics

One of the advantages of having a universe of programming language descriptions is the ability to concisely define an *extension* of an existing language by using Description transformers grafting extra constructors à la Swiestra (2008). This is made extremely simple by the disjoint sum combinator _‘+_ we defined in Section 13.3. An example of such an extension is the addition of let-bindings to an existing language.

Let bindings allow the user to avoid repeating themselves by naming sub-expressions and then using these names to refer to the associated terms. Preprocessors adding these types of mechanisms to existing languages (from C to CSS) are rather popular. We introduce a description of Let-bindings which can be used to extend any language description d to d ‘+ Let (where ‘+ is the disjoint of sum of two descriptions defined in Figure 13.10):

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

Figure 14.11: Description of a Single Let Binding

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

In a dependently typed language, a type may depend on a value which in the presence of let bindings may be a variable standing for an expression. The user naturally does not want it to make any difference whether they used a variable referring to a let-bound expression or the expression itself. Various typechecking strategies can accommodate this expectation: in Coq (Team [2017]) let bindings are primitive constructs of the language and have their own typing and reduction rules whereas in Agda they are elaborated away to the core language by inlining.

This latter approach to extending a language *d* with let bindings by inlining them before typechecking can be implemented generically as a semantics over (*d* '+ Let). For this semantics values in the environment and computations are both let-free terms. The algebra of the semantics can be defined by parts thanks to case defined in section 13.3: the old constructors are kept the same by interpreting them using the generic Substitution algebra; whilst the let-binder precisely provides the extra value to be added to the environment.

```
UnLet : Semantics (d '+ Let) (Tm d ∞) (Tm d ∞)
UnLet .th^𝒱 = th^Tm
UnLet .var = id
UnLet .alg = case (Substitution .alg) λ where
  (_ , e , t , refl) → extract t (ε ∙ e)
```

Figure 14.12: Let-Elaboration as a Semantics

The process of removing let binders is kickstarted with a placeholder environment associating each variable to itself.

```
unlet : ∀[ Tm (d '+ Let) ∞ σ ⇒ Tm d ∞ σ ]
unlet = Semantics.semantics UnLet ('var <$> identity)
```

Figure 14.13: Corollary: Let-Elaboration via Evaluation with Placeholders

In half a dozen lines of code we have defined a generic extension of syntaxes with binding together with a semantics which corresponds to an elaborator translating away this new construct. We have seen in chapter 6 that it is similarly possible to implement a Continuation Passing Style transformation as a semantics for STLC.

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

14.3 (Unsafe) Normalisation by Evaluation

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

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

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

Figure 14.14: Corollary: Let-Elaboration via Evaluation with Placeholders

This datatype definition is utterly unsafe. The more conservative user will happily restrict herself to typed settings where the domain can be defined as a logical predicate or opt instead for a step-indexed approach. But this domain does make it possible to define a generic nbe semantics by only specifying an algebra to evaluate one "layer" of term. This constraint is minimal: there is no way for us to know a priori what a constructor means; a binder could represent λ-abstractions, Σ-types, fixpoints, or anything else.

Thanks to the fact we have picked a universe of finitary syntaxes, we can *traverse* (McBride and Paterson [2008]) the functor to define a (potentially failing) reification function turning elements of the reflexive domain into terms. The Kripke function spaces can themselves be reified: Dm is VarLike thanks to the V constructor.

By composing them, we obtain the normalisation function which gives its name to normalisation by evaluation: provided a term, we produce a value in the reflexive domain by evaluating it in an environment made of placeholder values and then reify it to a (maybe) term.

```
Alg : Desc I → Set
Alg d = ∀ {σ} → ∀[ ⟦ d ⟧ (Kripke (Dm d ∞) (Dm d ∞)) σ ⇒ Dm d ∞ σ ]
```

```
nbe : Alg d → Semantics d (Dm d ∞) (Dm d ∞)
nbe alg .th^𝒱 = th^Dm
nbe alg .var   = id
nbe alg .alg   = alg
```

Figure 14.15: Evaluation as a Semantics

```
reify^Dm : ∀[ Dm d i σ ⇒ Maybe ∘ Tm d ∞ σ ]
reify^Dm ⊥      = nothing
reify^Dm (V k) = just (‘var k)
reify^Dm (C v) = ‘con <$> sequenceA d (fmap d reify^Kripke v)
  where reify^Kripke = λ Θ i kr → reify^Dm (reify vl^Dm Θ i kr)
```

Figure 14.16: Generic Reification via sequenceA

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

Figure 14.17: Normalisation by Evaluation

Example: Evaluator for the Untyped Lambda-Calculus

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

We have not used the ⊥ constructor so *if* the evaluation terminates (by disabling the strict positivity check we have lost all guarantees of the sort) we know we will get a term in normal form. For instance: identity applied twice to itself yield a strongly normalising term and if we run the evaluator we indeed get the identity as an output as demonstrated in fig. 14.19.

```
norm : ∀[ Tm UTLC ∞ _ ⇒ Maybe ∘ Tm UTLC ∞ _ ]
norm = NbyE.norm $ λ where
  (false , b)                      → C (false , b)
  (true , C (false , b , _) , t , _) → b (base vl^Var) (ε • t)
  (true , ft)                      → C (true , ft)
```

Figure 14.18: Normalisation for the Untyped λ-calculus

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

Figure 14.19: Normalization Example

14.4 Printing with Names, Generically

Coming back to our work on (rudimentary) printing with names in section 4.6, we can now give a generic account of it. This is a particularly interesting example because it demonstrates that we may sometimes want to give Desc a different semantics to accomodate a specific use-case: we do not want our users to deal explicitly with name generation, explicit variable binding, etc.

We are going to reuse some of the components defined in section 4.6: we can rely on the same state monad for name generation, the same fresh function and the same notions of Name and Printer for the semantics' values and computations.

The first piece of the puzzle is Pieces. The structure of Semantics would suggest giving our users an interface where sub-structures are interpreted as Kripke function spaces expecting fresh names for the fresh variables and returning a monadic computation delivering a printer. However we can do better: we can preemptively generate a set of fresh names for the newly-bound variables and hand them to the user together with the result of printing the body with these names. As usual we have a special case for the substructures without any newly-bound variable. Note that the specific target context of the environment of Names is only picked for convenience as Name ignores the scope: (Δ ++ Γ) is what fresh^l gives us.

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

Figure 14.20: Interpretation of Recursive Substructures: Printing Pieces

The key component making this work is the reification function reify^M turning the

Kripke spaces we get from the semantics framework into Pieces. This function has to be monadic so that we may generate fresh names. It uses the fact that environments are traversable and that (M Name) is easily proven to be VarLike (fresh generates new names and they are trivially Thinnable as they ignore their scope) to generate an environment of names for the newly-bound variables.

```
reify^M : ∀ Δ i → Kripke Name Printer Δ i Γ → M (Pieces Δ i Γ)
reify^M []             i p = p
reify^M Δ@(_ :: _) i f  = do
  ρ ← sequenceA (fresh^l vl^MName _)
  b ← f (fresh^r vl^Var Δ) ρ
  return (ρ , b)
```

Figure 14.21: Reification: from Kripke Functions to Pieces

We also expect the user to provide us with a syntax-specific (Display *d*) explaining how to print one "layer" of term where the subterms are Pieces i.e. both the names of the variables bound in the subterms and the string representation of the subterms are available. See section 14.4 for a concrete example of such a Display.

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

Figure 14.22: Syntax-Specific Display Instructions

Once we have these key components, we can write our printer as a Semantics. The two first constraints are trivial: Name is constant in its scope argument and therefore trivially thinnable and names (strings) are trivially printers (strings in the M monad). The algebra is trickier to define. But because we know how to convert Kripke function spaces from Names to Printer into M-wrapped Pieces and because the functor induced by a description is traversable, we can get a layer of term where the subterms are Pieces. It is then just a matter of applying the user-supplied Display directive to obtain a string representation of the term.

```
printing : Display d → Semantics d Name Printer
printing dis .th^𝒱 = th^const
printing dis .var   = return
printing dis .alg   = λ v → dis <$> sequenceA d (fmap d reify^M v)
```

Figure 14.23: Printing as a Generic Semantics

Using the closed version of the fundamental lemma of semantics defined in fig. 14.5 and reusing the name supply defined in section 4.6 we obtain a printer for closed terms.

```
print : Display d → TM d σ → String
print dis t = proj₁ $ closed (printing dis) t names
```

Figure 14.24: Printer for closed terms

Example: Printing Terms of STLC

Our only purpose here is to show how one typically defines a Display to define a printer. Remember that we get one "layer" of term as an input, in this specific case it means either an application or a λ-abstraction. We use a pattern-matching lambda to distinguish the two cases. If we have an application then we have already been given a string *f* for the function and *t* for the argument; we use Krivine's convention of writing (*f*)*t* for the application. If we have a λ-abstraction, it means we are handed a pair of a (singleton) environment containing a fresh name *x* for the variable bound by the lambda together with a representation *b* of the body using that name; we return λ*x*.*b*.

```
display^STLC : Display STLC
display^STLC = λ where
  (App _ _ , f , t      , _) → "(" ++ f ++ ") " ++ t
  (Lam _ _ , (x , b) , _) → "λ" ++ lookup x z ++ ".   " ++ b
```

Figure 14.25: Display Directive for STLC

We can of course run the printer and check that it does produce the string we would expect.

```
_ : let f : TM STLC (α ‘→ α); f = ‘app ‘id ‘id
    in print display^STLC f ≡ "(λa.  a) λb.  b"
_ = refl
```

Chapter 15

Typechecking as a Semantics

In the previous chapter we have seen various generic semantics one may be interested in when working on a deeply embedded language: renaming, substitution, desugaring, evaluation, and printing with names. All of these fit neatly in the Semantics framework. Now we wish to study a specific language in particular and see how we can take advantage of the same framework to structure language-specific traversals.

15.1 An Algebraic Approach to Typechecking

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

```
data Mode : Set where
  Check Infer : Mode
```

```
data Type : Set where
  α      : Type
  _‘→_ : Type → Type → Type
```

Figure 15.1: Modes and Types

We define the language of (Mode −Scoped) terms using our language of descriptions. We start once more with LangC, the language's constructors, and dispatch over such constructors using a pattern-matching lambda. Following traditional presentations, eliminators give rise to Inferrable terms under the condition that the term they are eliminating is also Inferrable and the other arguments are Checkable whilst constructors and their arguments are always Checkable. Two extra constructors allow changes of direction: Cut annotates a checkable term with its Type thus making it inferrable whilst Emb embeds inferrables into checkables.

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

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

Figure 15.2: A Bidirectional Simply Typed Language

Both the values and computations will be constant in the scope. The values stored in the environment will be Type information for bound variables. Instead of considering that we get a type no matter what the Mode of the variable is, we enforce the fact that all variables need to be Inferrable.

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

Figure 15.3: Values as Type Assignments for Variables

In contrast, the generated computations will, depending on the mode, either take a type candidate and Check it is valid or Infer a type for their argument. These computations are always potentially failing so we use the Maybe monad.

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

Figure 15.4: Computations as Mode-indexed Type Checking or Inference

Before defining typechecking as a Semantics we need to introduce two simple checks: a first function checking that two types are equal and a second making sure its input is a function type and returning its domain and codomain.

```
_==_ : (σ τ : Type) → Maybe ⊤
α == α = just tt
(σ₁ ‘→ τ₁) == (σ₂ ‘→ τ₂) =
  σ₁ == σ₂ >> τ₁ == τ₂
_ == _ = nothing
```

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

Equipped with these combinators, we can define the two most interesting cases as top-level combinators: application and λ-abstraction. When dealing with an application: infer the type of the function, make sure it is an arrow type, check the argument at the domain's type and return the codomain (_<$_ takes an *A* and a Maybe *B* and returns a Maybe *A* which has the same structure as its second argument).

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

For a λ-abstraction: check the input type is an arrow type and check the body at the codomain type in the extended environment where the newly-bound variable is Inferrable and is assigned the type of the domain.

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

We can now define typechecking itself. Because values are constant in the scope, the th^𝒱 constraint is trivial. The var constraint is also easy: values correspond to Inferrable terms so we can simply return the type looked up in the environment. The algebra describes the algorithm by pieces. We have already handled application and λ-abstraction, a cut always comes with a type candidate against which to check the term and to be returned in case of success. Finally, the change of direction from Inferrable to Checkable is successful when the inferred type is equal to the expected one.

```
Typecheck : Semantics Lang (const ∘ Var-) (const ∘ Type-)
Typecheck .th^𝒱 = th^const
Typecheck .var   = λ where (‘var t) → just t
Typecheck .alg   = λ where
  (App , f , t , refl) → APP f t
  (Lam , b , refl)     → LAM b
  (Cut σ , t , refl)   → σ <$ t σ
  (Emb , t , refl)     → λ σ → t >>= σ ==_
```

Figure 15.5: Typechecking as a Semantics

From this we can derive our type-(Infer/Check) function which takes a closed term and computes either an inferred type or a validation function for a type candidate. We make use of the special case of semantics for closed term introduced in fig. 14.5.

We can run this typechecking function on an example and verify that we do get the type we expect.

```
type- : ∀ m → TM Lang m → Type- m
type- m t = Semantics.closed Typecheck t
```

Figure 15.6: Type Inference and Type Checking as Mode-indexed Semantics

```
_ : let ‘id : TM Lang Check
        ‘id = ‘lam (‘emb (‘var z))
    in type- Infer (‘app (‘cut ((α ‘→ α) ‘→ (α ‘→ α)) ‘id) ‘id) ≡ just (α ‘→ α)
_ = refl
```

We have demonstrated how to define a bidirectional typechecker for this simple language by leveraging the Semantics framework. However the output of this function is not very informative. We can do better.

15.2 An Algebraic Approach to Elaboration

Instead of simply generating a type or checking that a candidate will do, we can use our Descriptions to describe not only the source language but also a language of evidence. During typechecking we generate at the same time an expression's type and a well scoped and well typed term of that type. We use STLC (defined in fig. 13.8) as our *internal* language. That is to say that starting from a Lang term, the typechecking process should generate an STLC term.

Before we can jump right in, we need to set the stage: a Semantics for a Lang term will involve (Mode −Scoped) notions of values and computations but an STLC term is (Type −Scoped). We first introduce a Typing associating types to each of the modes in scope, together with fromTyping extracting the context thus defined.

```
Typing : List Mode → Set            fromTyping : Typing ms → List Type
Typing = All (const Type)           fromTyping []       = []
                                    fromTyping (σ :: Γ) = σ :: fromTyping Γ
```

Figure 15.7: Typing: From Contexts of Modes to Contexts of Types

We can then explain what it means for elaboration to target T a (Type −Scoped) at a type σ: provided a list of modes and a corresponding typing, we should get a T of type σ in the context induced by that Typing.

In particular, our environment values are elaboration functions targetting Var. We expect all values to be in scope i.e. provided any typing of the scope of modes, we are guaranteed to return a type together with a variable of that type in the context induced by the typing. We once more limit environment values to the Infer mode only.

The computations are a bit more tricky. On the one hand, if we are in checking mode then we expect that for any typing of the scope of modes and any type candidate

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

Figure 15.8: Elaboration of a Scoped Family

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

Figure 15.9: Values as Variables and Inference Functions

we can Maybe return a term at that type in the induced context. On the other hand, in the inference mode we expect that given any typing of the scope, we can Maybe return a type together with a term at that type in the induced context.

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

Figure 15.10: Computations as Mode-indexed Elaboration Functions

Because we are now writing a typechecker which returns evidence of its claims, we need more informative variants of the equality and isArrow checks. In the equality checking case we want to get a proof of propositional equality but we only care about the successful path and will happily return nothing when failing. Agda's support for (dependent!) do-notations makes writing the check really easy. For the arrow type, we introduce a family Arrow constraining the shape of its index to be an arrow type and redefine isArrow as a view targetting this inductive family (Wadler [1987]).

```
_==_ : (σ τ : Type) → Maybe (σ ≡ τ)
α == α = just refl
(σ₁ ‘→ τ₁) == (σ₂ ‘→ τ₂) = do
  refl ← σ₁ == σ₂
  refl ← τ₁ == τ₂
  return refl
_ == _ = nothing
```

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

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

Figure 15.11: Informative Equality Check and Arrow View

We now have all the basic pieces and can start writing elaboration code. We once more start by dealing with each constructor in isolation before putting everything together to get a Semantics. These steps are very similar to the ones in the previous section.

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

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

Figure 15.12: Elaboration of Applications

The λ-abstraction case, we start by checking that the type candidate is an Arrow. We can then elaborate the body of the lambda in a context extended with one Infer variable assigned an inference function thanks to the auxiliary function var0. From this we get an internal term corresponding to the body of the λ-abstraction and conclude by returning it wrapped in a ‘lam constructor.

```
VAR0 : Var- Infer (Infer :: ms)
VAR0 = ‘var λ where (σ :: _) → (σ , z)

LAM : ∀[ Kripke Var- Type- (Infer :: []) Check ⇒ Type- Check ]
LAM b Γ σ ‘→τ = do
  (σ ‘→ τ) ← isArrow σ ‘→τ
  B ← b (bind Infer) (ε • VAR0) (σ :: Γ) τ
  return (‘lam B)
```

Figure 15.13: Elaboration of Lambda-Abstraction

This time we also stop to consider the semantical counterpart of the change of direction Emb which turns an inferrable into a checkable. We not only want to check that the inferred type and the type candidate are equal: we need to cast the internal term labelled with the inferred type to match the type candidate. Luckily, Agda's dependent do-notations make once again our job easy: when we make the pattern refl explicit, the equality holds in the rest of the block.

We have almost everything we need to define elaboration as a semantics. Discharging the th^$\mathcal{V}$ constraint is a bit laborious and the proof doesn't yield any additional insight so we leave it out here. The semantical counterpart of variables (var) is fairly

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

Figure 15.14: Elaboration of Embedding

straightforward: provided a Typing, we run the inference and touch it up to return a term rather than a mere variable. Finally we define the algebra (alg) by pattern-matching on the constructor and using our previous combinators; the only case left is Cut whose Type annotation provides precisely the piece of information we need.

```
Elaborate : Semantics Lang Var- Type-
Elaborate .th^𝒱 = th^Var-
Elaborate .var   = λ where (‘var infer) Γ → just (map₂ ‘var (infer Γ))
Elaborate .alg   = λ where
  (App , f , t , refl) → APP f t
  (Lam , b , refl)     → LAM b
  (Emb , t , refl)     → EMB t
  (Cut σ , t , refl)   → λ Γ → (σ ,_) <$> t Γ σ
```

Figure 15.15: Elaboration as a Semantics

Chapter 16

Building Generic Proofs about Generic Programs

We have already shown in chapters 8 and 9 that, for the simply typed λ-calculus, introducing an abstract notion of Semantics not only reveals the shared structure of common traversals, it also allows us to give abstract proof frameworks for simulation or fusion lemmas. These ideas naturally extend to our generic presentation of semantics for all syntaxes.

The most important concept going forward is (Zip d), a relation transformer which characterises structurally equal layers such that their substructures are themselves related by the relation it is passed as an argument. It inherits a lot of its relational arguments' properties: whenever R is reflexive (respectively symmetric or transitive) so is Zip d R.

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

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

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

16.1 Simulation Lemma

We first revisit the Simulation relation defined in chapter 8 for STLC, reusing as much as possible the same notations. A Zip constraint appears naturally when we want to say that a semantics can simulate another one.

```
Zip : (d : Desc I) → (∀ Δ i → ∀[ X Δ i ⇒ Y Δ i ⇒ const Set ])
                  → ∀[ ⟦ d ⟧ X i ⇒ ⟦ d ⟧ Y i ⇒ const Set ]
Zip (‘■ j)       R x       y        = ⊤
Zip (‘X Δ j d) R (r , x)  (r’ , y) = R Δ j r r’ × Zip d R x y
Zip (‘σ A d)   R (a , x) (a’ , y) = Σ[ eq ∈ a’ ≡ a ] Zip (d a) R x (rew eq y)
  where rew = subst (λ a → ⟦ d a ⟧ _ _ _)
```

Figure 16.1: Zip: Characterising Structurally Equal Values with Related Substructures

Given a relation $\mathcal{V}^R$ connecting values in $\mathcal{V}^A$ and $\mathcal{V}^B$, and a relation $\mathcal{C}^R$ connecting computations in $\mathcal{C}^A$ and $\mathcal{C}^B$, we can define Kripke^R relating values Kripke $\mathcal{V}^A$ $\mathcal{C}^A$ and Kripke $\mathcal{V}^B$ $\mathcal{C}^B$ by stating that they send related inputs to related outputs. It is a generalisation of the Kripke^R defined in fig. 8.3 to accomodate non-binders and binders introducing more than one variable.

```
Kripkeᴿ : ∀ Δ i → ∀[ Kripke 𝒱ᴬ 𝒞ᴬ Δ i ⇒ Kripke 𝒱ᴮ 𝒞ᴮ Δ i ⇒ const Set ]
Kripkeᴿ []          σ kᴬ kᴮ = rel 𝒞ᴿ σ kᴬ kᴮ
Kripkeᴿ Δ@(_ :: _) σ kᴬ kᴮ = ∀ {Θ} (ρ : Thinning _ Θ) {vsᴬ vsᴮ} →
                               All 𝒱ᴿ Δ vsᴬ vsᴮ → rel 𝒞ᴿ σ (kᴬ ρ vsᴬ) (kᴮ ρ vsᴮ)
```

Figure 16.2: Relational Kripke Function Spaces: From Related Inputs to Related Outputs

Simulation Constraints

We can then combine Zip and Kripke^R to formulate the core of the Simulation constraint. It is parametrised by a description, two semantics and the two relations over values and computations respectively mentioned earlier.

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

The set of constraint closely matches the ones spelt out in chapter 8. We start with a constraint th^R stating that related values should still be related once thinned.

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

We then expect related values to yield related computations once var-wrapped.

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

Finally, alg^R express the idea that two semantic objects of respective types $[\![\ d\ ]\!]$ (Kripke $\mathcal{V}^A$ C^A) and $[\![\ d\ ]\!]$ (Kripke $\mathcal{V}^B$ C^B) are in simulation by using Zip to force them to be in lock-step and Kripke^R to guarantee the subterms are themselves in simulation.

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

Fundamental Lemma of Simulations

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

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

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

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

Figure 16.3: Fundamental Lemma of Simulations

Applications

Instantiating this generic simulation lemma, we can for instance get that renaming and substitution are extensional (cf. fig. 16.4, given extensionally equal environments they produce syntactically equal terms), or that renaming is a special case of substitution (cf. fig. 16.5). Of course these results are not new but having them generically over all syntaxes with binding is convenient; which we have experienced first hand when tackling the POPLMark Reloaded challenge (2018).

When studying specific languages, new opportunities to deploy the fundamental lemma of simulations arise. Our solution to the POPLMark Reloaded challenge for instance describes the fact that (sub ρ t) reduces to (sub ρ' t) whenever for all v, $\rho(v)$ reduces to ρ'(v) as a Simulation. The main theorem (strong normalisation of STLC via a logical relation) is itself an instance of (the unary version of) the simulation lemma, that is to say the fundamental lemma of logical predicates.

```
RenExt : Simulation d Renaming Renaming Eqᴿ Eqᴿ
RenExt .thᴿ  = λ ρ → cong (lookup ρ)
RenExt .varᴿ = cong ‘var
RenExt .algᴿ = λ _ _ →
  cong ‘con ∘ zip^reify Eqᴿ (reifyᴿ Eqᴿ Eqᴿ (vl^Refl vl^Var)) d
```

```
SubExt : Simulation d Substitution Substitution Eqᴿ Eqᴿ
SubExt .thᴿ  = λ ρ → cong (ren ρ)
SubExt .varᴿ = id
SubExt .algᴿ = λ _ _ →
  cong ‘con ∘ zip^reify Eqᴿ (reifyᴿ Eqᴿ Eqᴿ (vl^Refl vl^Tm)) d
```

Figure 16.4: Self-Simulation: Renaming and Substitution are Extensional

```
RenSub : Simulation d Renaming Substitution VarTmᴿ Eqᴿ
RenSub .varᴿ = id
RenSub .thᴿ  = λ ρ → cong (λ t → th^Tm t ρ)
RenSub .algᴿ = λ _ _ →
  cong ‘con ∘ zip^reify VarTmᴿ (reifyᴿ VarTmᴿ Eqᴿ vl^VarTm) d
```

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

Figure 16.5: Renaming as a Substitution via Simulation

The simulation proof framework is the simplest examples of the abstract proof frameworks we introduced in part 1 for the specific case of STLC. We also explained how a similar framework can be defined for fusion lemmas and deploy it for the renaming-substitution interactions but also their respective interactions with normalisation by evaluation. Now that we are familiarised with the techniques at hand, we can tackle this more complex example for all syntaxes definable in our framework.

16.2 Fusion Lemma

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

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

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

The Fusion Constraints

The notion of fusion is defined for a triple of Semantics; each $\mathcal{S}^i$ being defined for values in $\mathcal{V}^i$ and computations in $\mathcal{C}^i$. The fundamental lemma associated to such a set of constraints will state that running $\mathcal{S}^B$ after $\mathcal{S}^A$ is equivalent to running $\mathcal{S}^{AB}$ only.

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

```
record Fusion (d : Desc I) (𝒮ᴬ : Semantics d 𝒱ᴬ 𝒞ᴬ) (𝒮ᴮ : Semantics d 𝒱ᴮ 𝒞ᴮ)
  (𝒮ᴬᴮ : Semantics d 𝒱ᴬᴮ 𝒞ᴬᴮ)
  (ℰᴿ : ∀ Γ Δ {Θ} → (Γ –Env) 𝒱ᴬ Δ → (Δ –Env) 𝒱ᴮ Θ → (Γ –Env) 𝒱ᴬᴮ Θ → Set)
  (𝒱ᴿ : Rel 𝒱ᴮ 𝒱ᴬᴮ) (𝒞ᴿ : Rel 𝒞ᴮ 𝒞ᴬᴮ) : Set where
```

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

```
reifyᴬ : ∀ σ → ∀[ 𝒞ᴬ σ ⇒ Tm d ∞ σ ]
```

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

```
vl^𝒱ᴬ : VarLike 𝒱ᴬ
```

We can combine these two functions to define the reification procedure we will use in practice when facing Kripke function spaces: quote^A which takes such a function and returns a term by first feeding placeholder values to the Kripke function space and getting a C^A back and then reifying it thanks to reify^A.

```
quote^A : ∀ Δ i → Kripke 𝒱^A C^A Δ i Γ → Tm d ∞ i (Δ ++ Γ)
quote^A Δ i k = reify^A i (reify vl^𝒱^A Δ i k)
```

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

```
th^ℰ^R : ℰ^R Γ Δ ρ^A ρ^B ρ^AB → (ρ : Thinning Θ Ω) →
         ℰ^R Γ Δ ρ^A (th^Env 𝒮^B.th^𝒱 ρ^B ρ) (th^Env 𝒮^AB.th^𝒱 ρ^AB ρ)
```

Remembering that _>>_ is used in the definition of body (cf. fig. 14.4) to combine two disjoint environments (Γ –Env) $\mathcal{V}$ Θ and (Δ –Env) $\mathcal{V}$ Θ into one of type ((Γ ++ Δ) –Env) $\mathcal{V}$ Θ), we mechanically need a constraint stating that _>>_ is compatible with $\mathcal{E}^R$. We demand as an extra precondition that the values ρ^B and ρ^{AB} are extended with are related in a pointwise manner according to $\mathcal{V}^R$. Lastly, for all the types to match up, ρ^A has to be extended with placeholder variables which we can do thanks to the VarLike constraint vl^$\mathcal{V}^A$.

```
_>>^R_ : ℰ^R Γ Δ ρ^A ρ^B ρ^AB → All 𝒱^R Θ vs^B vs^AB →
         let id>>ρ^A = fresh^l vl^𝒱^A Δ >> th^Env 𝒮^A.th^𝒱 ρ^A (fresh^r vl^Var Θ)
         in ℰ^R (Θ ++ Γ) (Θ ++ Δ) id>>ρ^A (vs^B >> ρ^B) (vs^AB >> ρ^AB)
```

We finally arrive at the constraints focusing on the semantical counterparts of the terms' constructors. In order to have readable type we introduce an auxiliary definition $\mathcal{R}$. Just like in chapter 9, it relates at a given type a term and three environments by stating that sequentially evaluating the term in the first and then the second environment on the one hand and directly evaluating the term in the third environment on the other yields related computations.

```
ℛ : ∀ σ → (Γ –Env) 𝒱^A Δ → (Δ –Env) 𝒱^B Θ → (Γ –Env) 𝒱^AB Θ →
    Tm d s σ Γ → Set
ℛ σ ρ^A ρ^B ρ^AB t = rel C^R σ (eval^B ρ^B (reify^A σ (eval^A ρ^A t))) (eval^AB ρ^AB t)
```

As one would expect, the var^R constraint states that from related environments the two evaluation processes described by $\mathcal{R}$ yield related outputs.

```
var^R : ℰ^R Γ Δ ρ^A ρ^B ρ^AB → ∀ v → ℛ σ ρ^A ρ^B ρ^AB (‘var v)
```

The case of the algebra follows a similar idea albeit being more complex. It states that we should be able to prove that a ‘con-headed term's evaluations are related according to $\mathcal{R}$ provided that the evaluation of the constructor's body one way or the other yields structurally similar results (hence the use of the (Zip d) relation transformer defined in chapter 16) where the relational Kripke function space relates the semantical objects one can find in place of the subterms.

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

Fundamental Lemma of Fusion

This set of constraint is enough to prove a fundamental lemma of Fusion stating that from a triple of related environments, one gets a pair of related computations: the composition of $\mathcal{S}^A$ and $\mathcal{S}^B$ on one hand and $\mathcal{S}^{AB}$ on the other.

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

Figure 16.6: Statement of the Fundamental Lemma of Fusion

This lemma is once again proven mutually with its counterpart for Semantics' body's action on Scopes: given related environments and a scope, the evaluation of the recursive positions using $\mathcal{S}^A$ followed by their reification and their evaluation in $\mathcal{S}^B$ should yield a piece of data *structurally* equal to the one obtained by using $\mathcal{S}^{AB}$ instead where the values replacing the recursive substructures are Kripke^R-reladed.

```
body : $\mathcal{E}^R\ \Gamma\ \Delta\ \rho^A\ \rho^B\ \rho^{AB} \rightarrow \forall\ \Delta\ \sigma \rightarrow (b : \mathsf{Scope}\ (\mathsf{Tm}\ d\ s)\ \Delta\ \sigma\ \Gamma) \rightarrow$
       let $v^B\ \ = \mathcal{S}^B.\mathsf{body}\ \rho^B\ \Delta\ \sigma\ (\mathsf{quote}^A\ \Delta\ \sigma\ (\mathcal{S}^A.\mathsf{body}\ \rho^A\ \Delta\ \sigma\ b))$
           $v^{AB} = \mathcal{S}^{AB}.\mathsf{body}\ \rho^{AB}\ \Delta\ \sigma\ b$
       in $\mathsf{Kripke}^R\ \mathcal{V}^R\ \mathcal{C}^R\ \Delta\ \sigma\ v^B\ v^{AB}$
```

Figure 16.7: Statement of the Fundamental Lemma of Fusion for Bodies

The proofs involve two functions we have not mentiond before: zip maps a proof that a property holds for any recursive substructure over the arguments of constructor to obtain a Zip object. The proof we obtain does not exactly match the premise in alg^R; we need to adjust it by rewriting an fmap-fusion equality called fmap^2.

Applications

A direct consequence of this result is the four lemmas collectively stating that any pair of renamings and / or substitutions can be fused together to produce either a renaming (in the renaming-renaming interaction case) or a substitution (in all the other cases).

```
fusion ρ^R (‘var v) = var^R ρ^R v
fusion ρ^R (‘con t) = alg^R ρ^R t (rew (zip d (body ρ^R) t)) where

  eq = fmap² d (𝒮^A.body _) (λ Δ i t → 𝒮^B.body _ Δ i (quote^A Δ i t)) t
  rew = subst (λ v → Zip d (Kripke^R 𝒱^R 𝒞^R) v _) (sym eq)


body ρ^R []       i b = fusion ρ^R b
body ρ^R (σ :: Δ) i b = λ ρ vs^R → fusion (th^ℰ^R ρ^R ρ >>^R vs^R) b
```

Figure 16.8: Proof of the Fundamental Lemma of Fusion

One such example is the fusion of substitution followed by renaming into a single substitution where the renaming has been applied to the environment.

```
subren : (t : Tm d i σ Γ) (ρ₁ : (Γ ─Env) (Tm d ∞) Δ) (ρ₂ : Thinning Δ Θ) →
         ren ρ₂ (sub ρ₁ t) ≡ sub (ren ρ₂ <$> ρ₁) t
```

Figure 16.9: Generic Substitution-Renaming Fusion Principle

All four lemmas are proved in rapid succession by instantiating the Fusion framework four times, using the first results to discharge constraints in the later ones. The last such result is the generic fusion result for substitution with itself.

```
sub² : (t : Tm d i σ Γ) (ρ₁ : (Γ ─Env) (Tm d ∞) Δ) (ρ₂ : (Δ ─Env) (Tm d ∞) Θ) →
       sub ρ₂ (sub ρ₁ t) ≡ sub (sub ρ₂ <$> ρ₁) t
```

Figure 16.10: Generic Substitution-Substitution Fusion Principle

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

Chapter 17

Conclusion

17.1 Summary

In the first half of this thesis, we have expanded on the work published in Allais et al. [2017a]. Starting from McBride's Kit (2005) making explicit the common structure of renaming and substitution, we observed that normalisation by evaluation had a similar shape. This led us to defining a notion of type-and-scope preserving Semantics where, crucially, λ-abstraction is interpreted as a Kripke function space. This pattern was general enough to encompass not only renaming, substitution and normalisation by evaluation but also printing with names, continuation passing style transformations and as we have seen later on even let-inlining, typechecking and elaboration to a typed core language.

Once this shared structure was highlighted, we took advantage of it and designed proof frameworks to prove simulation lemmas and fusion principles for the traversals defined as instances of Semantics. These allowed us to prove, among other things, that syntactic traversals are extensional, that multiple renamings and substitutions can be fused in a single pass and that the substitution lemma holds for NBE's evaluation. Almost systematically, previous results where used to discharge the goals arising in the later proofs.

In the second half, we have built on the work published in Allais et al. [2018a]. By extending Chapman, Dagand, McBride, and Morris' universe of datatype descriptions (2010) to support a notion of binding, we have given a generic presentation of syntaxes with binding. We then defined a generic notion of type-and-scope preserving Semantics for these syntaxes with binding. It captures a large class of scope-and-type safe generic programs: from renaming and substitution, to normalisation by evaluation, the desugaring of new constructors added by a language transformer, printing with names or typechecking.

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

We have put all of these results into practice by using them to solve the (to be published) POPLMark Reloaded challenge which consists in formalising strong normalisation for the simply typed λ-calculus via a logical-relation argument. This also gave us the opportunity to try our framework on larger languages by tackling the challenge's extensions to sum types and Gödel's System T. Compared to the Coq solution contributed by our co-authors, we could not rely on tactics and had to write all proof terms by hand. However the expressiveness of dependently-typed pattern-matching, the power of size-based termination checking and the consequent library we could rely on thanks to the work presented in this thesis meant that our proofs were just as short as the tactics-based ones.

17.2 Further Work

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

Total Compilers with Typed Intermediate Representations

Some of our core examples of generic semantics correspond to compiler passes: desugaring, elaboration to a typed core, type-directed partial evaluation, or CPS transformation. This raises the question of how many such common compilation passes can be implemented generically.

Other semantics such as printing with names or a generic notion of raw terms together with a generic scope checker (not shown here but available in Allais et al. [2018b]) are infrastructure a compiler would have to rely on. Together with our library of *total* parser combinators (Allais [2018]) and our declarative syntax for defining hierarchical command line interfaces (Allais [2017]), this suggests we are close to being able to define an entire (toy) compiler with strong invariants enforced in the successive intermediate representations.

To tackle modern languages with support for implicit arguments, a total account of (higher-order) unification is needed. It would ideally be defined generically over our notion of syntax thus allowing us to progressively extend our language as we see fit without having to revisit that part of the compiler.

Generic Meta-Theory

If we cannot use our descriptions to define an intrinsically-typed syntax for a dependently-typed theory, we can however give a well-scoped version and then define typing judgments. When doing so we have a lot of freedom in how we structure them and a natural question to ask is whether we can identify a process which will always give us judgments with good properties e.g. stability under substitution or decidable typechecking.

We can in fact guarantee such results by carefully managing the flow of information in the judgments and imposing that no information ever comes out of nowhere. This calls for the definition of a universe of typing judgments and the careful analysis of its properties.

A Theory of Ornaments for Syntaxes

The reseach programme introduced by McBride's unpublished paper introducing ornaments for inductive families (2017) allows users to make explicit the fact that some inductive families are refinements of others. Once their shared structure is known, the machine can assist the user in transporting an existing codebase from the weakly-typed version of the datatype to its strongly typed variant (Dagand and McBride [2014]). These ideas can be useful even in ML-style settings (Williams et al. [2014]).

Working out a similar notion of ornaments for syntaxes would yield similar benefits but for manipulating binding-aware families. This is particularly evident when considering the elaboration semantics which given a scoped term produces a scoped-and-typed term by type-checking or type-inference.

If the proofs we developped in this thesis would still be out of reach for ML-style languages, the programming part can be replicated using the usual Generalised Algebraic Data Types (GADTs) based encodings (Danvy et al. [2013], Lindley and McBride [2014]) and could thus still benefit from such ornaments being made first order.

Derivatives of Syntaxes

Our work on the POPLMark Reloaded challenge highlighted a need for a generic definition of evaluation contexts (i.e. terms with holes), congruence closures and the systematic study of their properties. This would build on the work of Huet (1997) and Abbott, Altenkirch, McBride and Ghani (2005) and would allow us to revisit previous work based on concrete instances of our Semantics-based approach to formalising syntaxes with binding such as McLaughlin, McKinna and Stark (2018).

List of Figures

Bibliography

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

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

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, 2013a.

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. pages 27–38, 2013b. URL `http://dl.acm.org/citation.cfm?id=2429069`.

A. Abel, A. Momigliano, and B. Pientka. Poplmark reloaded. *Proceedings of the Logical Frameworks and Meta-Languages: Theory and Practice Workshop*, 2017.

A. Abel, G. Allais, S. Schäfer, A. Hameer, A. Momigliano, B. Pientka, and K. Stark. Poplmark reloaded: Mechanizing proofs by logical relations. Submitted to Journal of Functional Programming, 2018.

G. Allais. agdARGS – Declarative hierarchical command line interfaces. In *TTT : Type Theory Based Tools*, 2017.

G. Allais. agdarsec – Total parser combinators. In *JFLA 2018 Journées Francophones des Langages Applicatifs*, page 45, 2018.

G. Allais, C. McBride, and P. Boutillier. New equations for neutral terms: A sound and complete decision procedure, formalized. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, DTP '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2384-0. doi: 10.1145/2502409.2502411. URL `http://doi.acm.org/10.1145/2502409.2502411`.

G. Allais, J. Chapman, C. McBride, and J. McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 195–207. ACM, 2017a. ISBN 978-1-4503-4705-1. doi: 10.1145/3018610.3018613.

G. Allais, J. Chapman, C. McBride, and J. McKinna. Type-and-scope safe programs and their proofs – agda formalization, 2017b. Also from github `https://github.com/gallais/type-scope-semantics`.

G. Allais, R. Atkey, J. Chapman, C. McBride, and J. McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, July 2018a. ISSN 2475-1421. doi: 10.1145/3236785. URL `http://doi.acm.org/10.1145/3236785`.

G. Allais, R. Atkey, J. Chapman, C. McBride, and J. McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs – agda formalization, 2018b. From github `https://github.com/gallais/generic-syntax`.

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.

T. Altenkirch, J. Chapman, and T. Uustalu. *Monads Need Not Be Endofunctors*, pages 297–311. Springer, 2010. ISBN 978-3-642-12032-9. doi: 10.1007/978-3-642-12032-9_21.

T. Altenkirch, J. Chapman, and T. Uustalu. Relative monads formalised. *Journal of Formalized Reasoning*, 7(1):1–43, 2014. ISSN 1972-5787.

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

F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2):287 – 311, 1994. ISSN 0167-6423.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, Dec. 2003. ISSN 1236-6064. URL `http://dl.acm.org/citation.cfm?id=985799.985801`.

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.

U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *LICS*, pages 203–211. IEEE, 1991.

J.-P. Bernardy. A pretty but not greedy printer (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP):6:1–6:21, Aug. 2017. ISSN 2475-1421. doi: 10.1145/3110250. URL `http://doi.acm.org/10.1145/3110250`.

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

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

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

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14. ACM, 2010. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863547.

J. M. Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham (UK), 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.

P.-E. Dagand and C. McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, 2014. doi: 10.1017/S0956796814000069.

N. A. Danielsson and U. Norell. Parsing mixfix operators. In S.-B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages*, pages 80–99, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24452-0.

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.

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

P. Dybjer and A. Setzer. *A Finite Axiomatization of Inductive-Recursive Definitions*, pages 129–146. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48959-7. doi: 10.1007/3-540-48959-2_11.

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.

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

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. L. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 636–666, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47599-6.

J. Kaiser, S. Schäfer, and K. Stark. Binder aware recursion over well-scoped de bruijn syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 293–306. ACM, 2018. ISBN 978-1-4503-5586-5. doi: 10.1145/3167098. URL `http://doi.acm.org/10.1145/3167098`.

A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. *SIGPLAN Not.*, 48(9):343–350, Sept. 2013. ISSN 0362-1340. doi: 10.1145/2544174.2500618. URL `http://doi.acm.org/10.1145/2544174.2500618`.

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

G. Malcolm. Data structures and program transformation. *Sci. Comput. Program.*, 14 (2-3):255–279, 1990. doi: 10.1016/0167-6423(90)90023-7. URL `https://doi.org/10.1016/0167-6423(90)90023-7`.

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

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

C. McBride. Ornamental algebras, algebraic ornaments. 2017. URL `https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/Ornament.pdf`.

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

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi: 10.1017/S0956796807006326.

C. McLaughlin, J. McKinna, and I. Stark. Triangulating context lemmas. In *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2018, pages 102–114. ACM, 2018. ISBN 978-1-4503-5586-5. doi: 10.1145/3167081. URL `http://doi.acm.org/10.1145/3167081`.

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.

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

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.

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

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

C. Tomé Cortiñas and W. Swierstra. From algebra to abstract machine: A verified generic construction. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2018, pages 78–90, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5825-5. doi: 10.1145/3240719.3241787. URL `http://doi.acm.org/10.1145/3240719.3241787`.

P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41653. URL `http://doi.acm.org/10.1145/41625.41653`.

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

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.

T. Williams, P.-E. Dagand, and D. Rémy. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 15–24, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3042-8. doi: 10.1145/2633628.2633631. URL `http://doi.acm.org/10.1145/2633628.2633631`.