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

March 3, 2021

Contents

Chapter 1

Introduction

In modern typed programming languages, programmers writing embedded Domain Specific Languages (DSLs) (Hudak [1996]) and researchers formalising them can now get help from the host language's type system to make illegal states unrepresentable. Their language's deep embeddings as inductive types can enforce strong invariants, their programs can be proven to be invariant-respecting, and they can even prove theorems about these programs.

1.1 A Motivating Example

The representation of variables and binders gives us a prototypical example of the benefits an astute type-based design can bring. Let us quickly survey different representations of the untyped λ-calculus, informally given by the following grammar:

$$t, u, \ldots ::= x \mid \lambda x.\, t \mid t\, u$$

Naïve Representation

Closely following the informal specification leads us to a naïve solution: introduce an inductive type to represent terms and use strings to represent variables. A variable is bound by the closest enclosing binder that uses the same string. For example in the expression (λx.λx.x) the variable x in the body of the expression is bound by the innermost λ-abstraction.

This means that some terms may be essentially the same despite using different names e.g. the above expression is equivalent to (λy.λx.x). This notion of equivalence up to renaming is called α-equivalence (Barendregt [1985]) and it is clearly going to be a bit tricky to implement: given two terms we may need to rename all of their bound variables to decide whether they are α-equivalent.

Nameless Representation

The problem of having to rename variables to compare terms for equality can be solved by adopting a nameless representation based on a positional system. We can

for instance represent variables as de Bruijn indices (1972): variables are modelled as natural numbers, counting the number of enclosing binders one needs to go through before reaching the binder the variable points to. That is to say that we represent both (λx.λx.x) *and* (λy.λx.x) by the term ($\lambda.\lambda.0$).

This solution is not perfect either: if a natural number is larger than the number of enclosing binders, the variable is free. When substituting a term containing free variables into a term containing binders, we need to remember to increment the natural numbers corresponding to free variables (and only those) every time we push the substitution under a binder. This is quite error prone.

Well Scoped Representation

As we have just seen, directly manipulating a string-based representation or even one using raw de Bruijn indices is error-prone. As a consequence, managing variable binding by making the scope part of the typing discipline is a popular use case (Bellegarde and Hook [1994], Bird and Paterson [1999], Altenkirch and Reus [1999]).

We will study this representation in more details in Section 4 but the key idea is that the number of free variables is made to be part of the type of terms. This means that if the programmer has forgotten to appropriately update a term before pushing it under a binder, they will get a static type error.

The illegal state has been made irrepresentable thus rooting out a source of errors.

1.2 Scope of This Thesis

This thesis focuses on the representations providing users with strong built-in guarantees. It covers the data representations, the programs acting on this data and the proofs that these programs are well-behaved.

Data 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.

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.

Programs 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 scope- and type-safe manner.

Proofs This last problem 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 8), 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.

1.3 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 and second parts of this work focus 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 third 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.

1.4 Source Material

The core of this thesis is based on the content of two fully-formalised published papers. They correspond roughly to part one and two on the one hand ("Type-and-scope Safe Programs and Their Proofs" Allais et al. [2017a,b]) and part three on the other ("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 6 originated from the work on a third paper "New equations for neutral terms" (Allais et al. [2013b]) 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 (Abel et al. [2017, 2019]) for which we formalised a proof of strong normalisation for the simply-typed lambda-calculus (and its extension with sum types, and primitive recursion for the natural numbers).

1.5 Plan

The work in this thesis brings together many strands of research so we provide introductory chapters for the less common topics.

Introductory Materials

We use categorical terms whenever it is convenient to concisely describe some of the key properties of our constructions. So we provide the reader with an **introduction to category theory in Section 2**. A reader familiar with Pierce's 'Basic category theory for computer scientists' may skip this section except perhaps for the definition of relative monads in Section 2.4.

All of the content of this thesis has been fully formalised in Agda (the code snippets have all been checked and typeset by Agda itself) so we provide an **introduction to Agda and the conventions we adopted in Section 3**.

Our work on generic programming with syntaxes with binding is heavily influenced by prior results on generic programming for (indexed) inductive types. We offer a **primer on data-generic programming in Section 13**.

The **conventions and techniques**, and **Agda features** present in these introductory materials are collected in the appendices (Appendices A and B respectively).

Contributions

Part I is dedicated to the study of a scope- and type-safe representation of the simply typed lambda calculus with one sum type (the booleans) and one product type (the unit type). Starting from the inductive family used to represent terms in the language, we study various invariant-respecting traversals and notice that they all share the same structure. This leads us to see how they can all be unified as instances of a single generic traversal.

Part II identifies the constraints under which one can relate different instances of the generic traversal. This yields simulation and fusion lemmas for the generic traversal that can be immediately deployed to obtain classic meta-theoretical results as direct corollaries.

Part III builds on parts I and II and data-generic programming to introduce syntax-generic programming and proving. It provides the user with a whole universe of syntaxes with binding, their invariant-respecting traversals, and proofs about these traversals.

Chapter 2

Introduction to Category Theory

Although this thesis is not about category theory, some categorical notions do show up. They help us expose the structure of the problem and the solutions we designed. Hence this (short) introduction to category theory focused on the notions that will be of use to us. Computer scientists looking for a more substantial introduction to category theory can refer to Pierce's little book (1991) for general notions and Altenkirch, Chapman and Uustalu's papers (2010, 2014) if they are specifically interested in relative monads.

2.1 What even is a Category?

A category is a set of morphisms closed under a well-behaved notion of composition. This is made formal by the following definition.

Definition 1 *A **category** C is defined by a set of objects (written Obj(C)) and a family of morphisms (written C(A,B)) between any two such objects A and B such that:*

1. *For any object A, we have an* identity morphism id_A *in C(A,A)*
2. *Given three objects A, B, and C, and two morphisms f in C(A,B) and g in C(B,C), we have a unique morphism $(g \circ_C f)$ in C(A,C) corresponding to the composition of f and g.*
3. *Composition is associative*
4. *The appropriate identity morphisms are left and right neutral elements for composition*

Convention 1 *When the reader should be able to reconstruct the information from the context, we may write id instead of id_A and $(\_ \circ \_)$ instead of $(\_ \circ_C \_)$ respectively.*

Example 1 (Discrete category) *From any type A we can create the discrete category corresponding to A. Its objects are values of type A and the only morphisms it has are the identity functions from each element to itself. It derives its name from the total lack of morphisms between distinct objects.*

Example 2 (Sets and functions) *In this category called* **Set***, objects are sets and morphisms are total functions between them. Identities and compositions are the usual notions of identity and composition for functions.*

Example 3 (Families and index-respecting functions) *Given a set I, we define* $\mathbf{Set}^I$ *as the category whose objects are families of type (I → Set) and whose morphisms between two families P and Q are index preserving functions i.e. functions of type (∀i → P i → Q i).*

2.2 Functors

A functor is a morphism between two categories that is compatible with their respective notions of composition.

Definition 2 (Functor) *A functor F between two categories C and $\mathcal{D}$ is defined by its action on the objects and morphisms of C. It takes objects in C to objects in $\mathcal{D}$, and morphisms in C(A,B) to morphisms in $\mathcal{D}$(F A, F B) such that the structure of C is preserved. In other words:*

1. *F* id_A *is equal to* id_{FA}
2. *F* $(g \circ_{\mathcal{C}} f)$ *is equal to* $(F\ g \circ_{\mathcal{D}} F\ f)$

Example 4 (Reindexing) *Given two sets I and J, every function f from I to J induces a functor from the category* $\mathbf{Set}^J$ *to* $\mathbf{Set}^I$ *(cf. example 3). We write (f ⊢_) for this functor (we explain this notation in Section 3.5) and define its action on objects and morphisms as follows:*

- *Given an object P i.e. a predicate of type (J → Set), we define (f ⊢ P) to be the family (λi → P (f i)) of type (I → Set)*
- *Given a morphism prf from P to Q (i.e. a function of type (∀j → P j → Q j)), we define (f ⊢ prf) to be the morphism (λ i p → prf (f i) p)*

The constraints spelt out in the definition of a functor are trivially verified.

Programmers may be more familiar with endofunctors: functors whose domain and codomain are the same category.

Definition 3 (Endofunctor) *An endofunctor on a category C is a functor from C to C.*

Example 5 (List) *The inductive type associating to each type A, the type of finite lists of elements of A is an endofunctor on the category of Sets and functions (cf. example 2).*

Its action on morphism is simply to map the function over all of the lists' element. The equality constraints spelt out by the definition of Functor correspond to the classic identity and fusion lemmas for the map function. We see detailed proofs for a similar result in Section 3.4 in the context of proving programs correct in Agda.

2.3 Monads

A monad is an endofunctor whose action on objects is well-behaved under iteration. For any amount of nesting of this action $T^n A$, it has a canonical operation computing a value of type $T\,A$ in a way that is compatible with the functor structure.

Definition 4 (Monad) *A monad is an endofunctor T on C equipped with:*

1. *For any object A, a function η_A of type C(A, T A) (the unit)*
2. *For any objects A and B, a mapping $\_^*$ from C(A, T B) to C(T A, T B) (the Kleisli extension)*

such that:

1. *For any object A, ${\eta_A}^*$ is equal to id_{TA}*
2. *For any objects A and B and f a morphism of type C(A, T B), $f^* \circ \eta_A$ is equal to f*
3. *For any objects A, B, C and two morphisms f in C(A, T B) and g in C(B, T C), $(g^* \circ f)^*$ is equal to $(g^* \circ f^*)$*

Example 6 (List) *We have already seen in example 5 that the List type constructor is a functor. It is additionally a monad. η_A takes an element an returns a singleton list, while $\_^*$ maps its input function on every value in the input list before flattening the result. The equality constraints are easily verified.*

We will refine the following example in the next section and revisit it in spirit throughout this thesis as expressions-with-variables as monads is part of the very foundations of our work.

Example 7 (Arithmetic expressions with free variables) *The type of abstract syntax trees of very simple closed arithmetic expressions involving natural numbers ($\underline{n}$ stands for the embedding of a natural number n) and addition of two expressions can be described by following grammar:*

$$t ::= \underline{n} \mid t + t$$

We can freely add variables to this little language by adding an additional constructor for values in a type of variables I (we write i for any value of type I).

$$t ::= i \mid \underline{n} \mid t + t$$

This new type is an endofunctor on the category of sets and functions: a mapping from I to J can be seen as a variable renaming. It is, additionally, a monad: η is the var constructor itself, $\_^*$ corresponds to parallel substitution.

2.4 Monads need not be Endofunctors

Although monads are defined as endofunctors, we can relax this constraint by studying Altenkirch, Chapman and Uustalu's *relative* monads instead (2010, 2014).

Definition 5 (Relative Monad) *A monad relative to a functor V from $\mathcal{C}$ to $\mathcal{D}$ is a functor T from $\mathcal{C}$ to $\mathcal{D}$ equipped with:*

1. *For any object A, a mapping η_A of type $\mathcal{D}(V\,A,\, T\,A)$ (the unit)*
2. *For any objects A and B, a mapping $\_^*$ from $\mathcal{D}(V\,A,\, T\,B)$ to $\mathcal{D}(T\,A,\, T\,B)$ (the Kleisli extension)*

such that they verify similar constraints to the ones spelt out in the definition of a monad (cf. definition 4).

Example 8 (Arithmetic expressions with finitely many free variables) *This example is a slight modification of example 7. Instead of allowing any type to represent the variables free in a term, we restrict ourselves to using a natural number that represents the number of variables that are in scope.*

A variable in a tree indexed by n is now a natural number m together with a proof that it is smaller than n. This new type is a functor between the discrete category of natural numbers[1] *(cf. example 1) and the category of sets and functions.*

It is also a relative monad with respect to the trivial functor taking a natural number and mapping it to the set of natural numbers that are smaller (our notion of variable). Once again mapping is renaming, η is the var constructor itself, and $\_^*$ corresponds to parallel substitution.

[1] We could pick a source category with more structure and obtain a more interesting result but this is beyond the scope of this introduction

Chapter 3

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.

Agda 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]).

3.1 Set and Universe Levels

Agda is both a dependently typed programming language and a proof assistant. As such, its type system needs to be sound. A direct consequence is that the type of all types cannot itself be a type. This is solved by using a stratified tower of universes.

Feature 1 (Universe Levels) *Agda avoids Russell-style paradoxes by introducing a tower of universes* Set_0 *(usually written* Set*),* Set_1*,* Set_2*, etc. Each* Set_n *does not itself have type* Set_n *but rather* Set_{n+1} *thus preventing circularity.*

In practice most of our constructions will stay at the lowest level Set (commonly called the "type of small sets"), and we will only occasionally mention Set_1. The barest of examples involving both notions is the following definition of ID, the (large) type of the identity function on small sets, together with id the corresponding identity function.

```
ID : Set₁                    id : ID
ID = {A : Set} → A → A       id x = x
```

In the type ID we wrapped the Set argument using curly braces. This is our way of telling Agda that this argument should be easy to figure out and that we do not want to have to write it explicitly every time we call id.

Feature 2 (Implicit Arguments) *Programmers can mark some of a function's arguments as implicit by wrapping them in curly braces. The values of these implicit arguments can be left out at the function's call sites and Agda will reconstruct them by unification (Abel and Pientka [2011]).*

3.2 Data and (co)pattern matching

Sum Types Agda supports inductive families (Dybjer [1991]), a generalisation of the algebraic data types one can find in most functional programming languages.

The plainest of sum types is the empty type ⊥, a type with no constructor. We define it together with ⊥-elim also known as the principle of explosion: from something absurd, we can deduce anything. The absurd pattern () is used to communicate the fact that there can be no value of type ⊥.

```
data ⊥ : Set where
```

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Feature 3 (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.*

The next classic example of a sum type is the type of boolean values. We define it as the datatype with two constructors true and false. These constructors are *distinct* and Agda understands that fact: an argument of type true ≡ false can be dismissed with the same absurd pattern one uses for values of type ⊥.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

```
true≠false : true ≡ false → ⊥
true≠false ()
```

All definitions in Agda need to be total. In particular this means that a function is only considered defined once Agda is convinced that all the possible input values have been covered by the pattern-matching equations the user has written. The coverage checker performs this analysis.

Feature 4 (Coverage Checking) *During typechecking the coverage checker elaborates the given pattern-matching equations into a case tree. It makes sure that all the branches are either assigned a right-hand side or obviously impossible. This allows users to focus on the cases of interest, letting the machine check the other ones.*

The function if_then_else_ is defined by pattern-matching on its boolean argument. If it is true then the second argument is returned, otherwise we get back the third one.

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true  then t else f = t
if false then t else f = f
```

Feature 5 (Mixfix Identifiers) *We use Agda's mixfix operator notation (Danielsson and Norell [2011]) where underscores denote argument positions. This empowers us to define the* if_then_else_ *construct rather than relying on it being built in like in so many other languages. For another example, at the type-level this time, see e.g. the notation* _×_ *for the type of pairs defined in section 3.2.*

Inductive Types As is customary in introductions to dependently typed programming, our first inductive type will be the Peano-style natural numbers.They are defined as an inductive type with two constructors: zero and successor. Our first program manipulating them is addition; it is defined by recursion on the first argument.

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

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

For recursive definitions ensuring that all definitions are total entails an additional check on top of coverage checking: all calls to that function should be terminating.

Feature 6 (Termination Checking) *Agda is equipped with a sophisticated termination checking algorithm (Abel [1998]). For each function it attempts to produce a well-founded lexicographic ordering of its inputs such that each recursive call is performed on smaller inputs according to this order.*

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. As a consequence, the η-rule for ⊤ states that any two values of the unit type are equal to each other. This is demonstrated by the equality proof we provide.

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

```
_ : (t u : ⊤) → t ≡ u
_ = λ t u → refl
```

Feature 7 (Underscore as Placeholder Name) *An underscore used in place of an identifier in a binder means that the binding should be discarded. For instance (λ _ → a) defines a constant function. Toplevel bindings can similarly be discarded which is a convenient way of writing unit tests (in type theory programs can be run at typechecking time) without polluting the namespace.*

The second classic example of a record type is the pair type _×_ which has an infix constructor _,_ and two fields: its first (fst) and second (snd) projections. The η-rule for pairs states that any value is equal to the pairing of its first and second projection. This is demonstrated by the equality proof we provide.

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

```
_ : (p : A × B) → p ≡ (fst p , snd p)
_ = λ p → refl
```

Agda gives us a lot of different ways to construct a value of a record type and to project the content of a field out of a record value. We will now consider the same function swap implemented in many different fashion to highlight all the possibilities offered to us.

If the record declaration introduced a constructor we may use it to pattern-match on values of that record type on the left or construct values of that record type on the right. We can always also simply use the record keyword and list the record's fields. The two following equivalent definitions are demonstrating these possibilities.

```
swap : A × B → B × A
swap (a , b) = record
  { fst = b
  ; snd = a
  }
```

```
swap : A × B → B × A
swap record { fst = a
            ; snd = b
            } = (b , a)
```

From now on, we will only use the constructor _,_ rather than the more verbose definition involving the record keyword. Instead of pattern-matching on a record, we may want to project out the values of its fields. For each field, Agda defines a projection of the same name. They may be used prefix or suffix in which case one needs to prefix the projection's name with a dot. This is demonstrated in the two following equivalent definitions.

```
swap : A × B → B × A
swap p = (snd p , fst p)
```

```
swap : A × B → B × A
swap p = (p .snd , p .fst)
```

Instead of using the record constructor to build a value of a record type, we may define it by explaining what ought to happen when a user attempts to project out the value of a field. This definition style dubbed copattern-matching focuses on the *observations* one may make of the value. It was introduced by Abel, Pientka, Thibodeau, and Setzer (2013b). It can be used both in prefix and suffix style.

```
swap : A × B → B × A
fst (swap (a , b)) = b
snd (swap (a , b)) = a
```

```
swap : A × B → B × A
swap (a , b) .fst = b
swap (a , b) .snd = a
```

Finally, we may use an anonymous copattern-matching function to define the record value by the observation one may make of it. Just like any other anonymous (co)pattern-matching function, it is introduced by the construct (λ where) followed by a block of clauses.

```
swap : A × B → B × A
swap (a , b) = λ where
  .fst  → b
  .snd → a
```

Feature 8 (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. In the above example, A and B are introduced using this mechanism.*

Recursive Functions Definitions taking values of an inductive type as argument are constructed 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 tuple of size zero 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)
```

Technique 1 (Intrinsically Enforced Properties) *In the definition of an n-ary tuple, the length of the tuple is part of the type (indeed the definition proceeds by induction over this natural number). We say that the property that the tuple has length n is enforced* intrinsically. *Conversely some properties are only proven after the fact, they are called* extrinsic.

The choice of whether a property should be enforced intrinsically or extrinsically is for the programmer to make, trying to find a sweet spot for the datastructure and the task at hand. We typically build basic hygiene intrinsically and prove more complex properties later.

Types can now depend on values, as a consequence 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 a tuple of As of size n by duplicating a. The return type of replicate reduces to $\top$ when the natural number is zero and $(A \times (n\ \text{-Tuple}\ A))$ when it is (suc n) thus making the following definition valid.

```
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.

Convention 2 (Caret-based naming) *We use a caret to generate a mnemonic name: map^ refers to the fact that we can map a function over the content of a data structure and -Tuple clarifies that the data structure in question is the family of n-ary tuples.*

We use this convention consistently throughout this thesis, using names such as map^Rose for the proof that we can map a function over the content of a rose tree, th^Var for the proof that var*iables are* th*innable, or vl^Tm for the proof that terms are var-like.*

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
```

Remark 1 (Functor) *In Section 3.4 we will prove that applying map^-Tuple to the identity yields the identity and that map^-Tuple commutes with function composition. That is to say we will prove that (n -Tuple) is an endofunctor (see definition 3) on* **Set**.

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
```

3.3 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 call to map^Tuple by mutually defining with map^Rose an inlined version of (map^Tuple (map^Rose f)). In other words, we have to effectively perform supercompilation (Mendel-Gleason [2012]) by hand.

```
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)
```

3.4 Proving the Properties of Programs

We make explicit some of our programs' invariants directly in their type thus guaranteeing that the implementation respects them. The type of map^-Tuple for instance tells us that this is a length preserving operation. We cannot possibly encode all of a function's properties directly into its type but we can use Agda's to state and prove theorems of interest.

In Agda, programming *is* proving. The main distinction between a proof and a program is a social one: we tend to call programs the objects whose computational behaviour matters to us while the other results correspond to proofs.

We can for instance prove that mapping the identity function over a tuple amounts to not doing anything.

Lemma 1 (Identity lemma for map) *Given a function f extensionally equal to the identity, (map f) is itself extensionally equal to the identity.*

Technique 2 (Extensional Statements) *Note that we have stated the theorem not in terms of the identity function but rather in terms of a function which behaves like it. Agda's notion of equality is intensional, that is to say that we cannot prove that two functions yielding the same results when applied to the same inputs are necessarily equal. In practice this means that stating a theorem in terms of the specific implementation of a function rather than its behaviour limits vastly our ability to use this lemma. Additionally, we can always obtain the more specific statement as a corollary.*

The proof of this statement is a program proceeding by induction on n followed by a case analysis on the n-ary tuple. In the base case, the equality is trivial and in the second case map computes just enough to reveal the constructor pairing $(f\ a)$ together with the result obtained recursively. We can combine the proof that $(f\ a)$ is equal to a and the induction hypothesis by an appeal to congruence for the pair constructor _,_.

```
map-identity :
  ∀ n {f : A → A} → (∀ a → f a ≡ a) →
  ∀ as → map^-Tuple n f as ≡ as
map-identity zero    f-id tt       = refl
map-identity (suc n) f-id (a , as) = cong₂ _,_ (f-id a) (map-identity n f-id as)
```

Technique 3 (Functional Induction) *Whenever we need to prove a theorem about a function's behaviour, it is best to proceed by functional induction. That is to say that the proof and the function will follow the same pattern-matching strategy.*

Lemma 2 (Fusion lemma for map) *Provided a function f from A to B, a function g from B to C, and a function h from A to C extensionally equal to the composition of f followed by g, mapping f and then g over an n-ary tuple as is the same as merely mapping h.*

```
map-fusion :
  ∀ n {f : A → B} {g : B → C} {h} → (∀ a → g (f a) ≡ h a) →
  ∀ as → map^-Tuple n g (map^-Tuple n f as) ≡ map^-Tuple n h as
map-fusion zero    gf≈h tt       = refl
map-fusion (suc n) gf≈h (a , as) = cong₂ _,_ (gf≈h a) (map-fusion n gf≈h as)
```

Lemma 3 (Fusion Lemma for replicate) *Given a function f from A to B, a value a of type A, and a natural number n, mapping f over the n-ary tuple obtained by replicating a is the same as replicating (f a).*

```
map-replicate : ∀ n (f : A → B) (a : A) →
  map^-Tuple n f (replicate n a) ≡ replicate n (f a)
map-replicate zero    f a = refl
map-replicate (suc n) f a = cong (f a ,_) (map-replicate n f a)
```

3.5 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.

These combinators will allow us to write the types for index-preserving operations without having to mention the indices. This will empower us to use very precise indexed types whilst having normal looking type declarations. We have implemented a generalisation of these combinators to n-ary relations (Allais [2019]) but this work is out of the scope of this thesis.

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

```
_⇒_ : (P Q : 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 ⊢ P) where f is the modification and P the indexed Set it operates on. This is the reindexing functor we saw in example 4.

```
_⊢_ : (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 ∀[_] or ∃⟨_⟩. These combinators turn an indexed Set into a Set by quantifying over the index. The function ∀[_] universally quantifies over an implicit value of the right type while ∃⟨_⟩, defined using a dependent record, corresponds to an existential quantifier.

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

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field proj₁ : A
        proj₂ : B proj₁
```

```
∃⟨_⟩ : (A → Set) → Set
∃⟨_⟩ P = Σ _ P
```

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. 4.5.

Part I

Type and Scope Preserving Programs

Chapter 4

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.

4.1 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 (cf. definition 3). Similarly, an inductive family is the fixpoint of an endofunctor on $(\mathsf{Set} \to \mathsf{Set})$. 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 least 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 4.1: 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 : $A \to UTLC\ A$' corresponds to the use of variables, and the monadic 'bind : $UTLC\ A \to (A \to UTLC\ B) \to UTLC\ B$' 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 10.

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 5.3). 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. 4.5.

4.2 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. 4.2.

			data Type : Set where
⟨Type⟩	::=	`Unit` | `Bool`	'Unit 'Bool : Type
	|	⟨Type⟩ → ⟨Type⟩	_'→_ : (σ τ : Type) → Type

Figure 4.2: Types used in our Running Example

Convention 3 (Object and Meta Syntaxes) *When we represent object syntax in the meta language (Agda here) we use backquotes to make explicit the fact that we are manipulating quoted objects.*

The language's constructs are layed out in fig. 4.3. They naturally include the basic constructs of any 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.

This syntax is given a static semantics by the type system for the simply typed lambda calculus described in fig. 4.4. Variables have the type they are assigned by the typing context; an application node only makes sense if the function has an arrow type and the argument's type matches the function's domain; similarly an

$$
\begin{array}{rcl}
\langle \texttt{Term} \rangle & ::= & x \mid \langle \texttt{Term} \rangle \langle \texttt{Term} \rangle \mid \lambda x. \langle \texttt{Term} \rangle \\
& \mid & () \\
& \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if} \langle \texttt{Term} \rangle \texttt{then} \langle \texttt{Term} \rangle \texttt{else} \langle \texttt{Term} \rangle
\end{array}
$$

Figure 4.3: Grammar of our Language

`if_then_else_` construct demands that the condition branched over has type `Bool` and that both branches have the same type.

$$\frac{x : \sigma \in \Gamma}{x : \sigma}\,var \qquad \frac{f : \sigma \to \tau \qquad t : \sigma}{f\,t : \tau}\,app \qquad \frac{x : \sigma \vdash b : \tau}{\lambda x.\, b : \sigma \to \tau}\,lam \qquad \frac{}{() : \texttt{Unit}}\,unit$$

$$\frac{}{\texttt{true} : \texttt{Bool}}\,true \qquad \frac{}{\texttt{false} : \texttt{Bool}}\,false \qquad \frac{b : \texttt{Bool} \qquad l : \sigma \qquad r : \sigma}{\texttt{if}\,b\,\texttt{then}\,l\,\texttt{else}\,r : \sigma}\,ifte$$

Figure 4.4: Typing Rules for our Language

Convention 4 (Assume an Ambient Context) *Following Martin-Löf (1982) we adopt the convention that all of our typing rules are defined in an ambient context* Γ *and only mention the* adjustments *made to it. Crucially, we can see that the only rule which modifies the context is the introduction rule for lambda abstractions: in the premise,* $(x : \sigma \vdash)$ *indicates that the ambient context is extended with a new bound variable* x *of type* σ*.*

Well Scoped and Typed by Construction

We decide to represent this language in our host language not as terms in a raw syntax together with typing derivations ruling out badly behaved expressions but rather as a syntax which is *intrinsically* well scoped and typed by construction.

The first hurdle in our way is the representation of contexts, the notion of scope they induce and the concept of variables in a given scope. This matter has been studied at length and multiple competing solutions have been offered (Aydemir et al. [2005a]). We opt for well scoped and typed de Bruijn indices.

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 $(\sigma :: \Gamma)$ 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 define $(I \to \texttt{Scoped})$ as the set of type and scope indexed families where the argument I plays the role of the set of valid types.

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)

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

Figure 4.5: Typed and Scoped Definitions

index. This amounts to an inductive definition of context membership. We use the combinators defined in section 3.5 to show only local changes to the context.

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

Figure 4.6: 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 the fully expanded types displayed on their right.

The syntax for this calculus, shown in fig. 4.7, guarantees that terms are scope- and type-safe 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.

Convention 5 (Using Comments to Mimick Natural Deduction Rules) *In the definition of the calculus' syntax we use comment lines to mimick inference rules in natural deduction. This definition style is already present in Pollack's PhD thesis (1994).*

```
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    : ∀[
         ---------
         Term ‘Bool ]

  ‘ff    : ∀[
         ---------
         Term ‘Bool ]

  ‘ifte  : ∀[ Term ‘Bool ⇒ Term σ ⇒ Term σ
         ---------------------------
         ⇒ Term σ ]
```

Figure 4.7: Well Scoped and Typed Calculus

Chapter 5

Refactoring Common Traversals

Once they have a good representation for their language, DSL writers 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!

5.1 A Generic Notion of Environment

Renamings and substitutions have in common that they map variables to something: other variables in one case, terms in the other. We can abstract over these differences and provide users with a generic notion of environment that may be used in both cases.

Assuming that we are given an (I-Scoped) family $\mathcal{V}$ corresponding to a notion of values variables should be mapped to, we call $\mathcal{V}$-(evaluation) environment this generalisation of both renamings and substitutions. We leave out $\mathcal{V}$ when it can easily be inferred from the context.

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.

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

Figure 5.1: Generic Notion of Environment

We write ((Γ –Env) $\mathcal{V}$ Δ) for an environment that associates a value $\mathcal{V}$ well scoped and typed in Δ to every entry in Γ. 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 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. 4.6, it should be pretty clear that there can never be a value of type (Var σ []).

```
ε : ([] –Env) 𝒱 Δ
lookup ε ()
```

Figure 5.2: 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 5.3: Environment Extension

We also include the definition of _<$>_, which lifts in a pointwise manner a function acting on values into a function acting on environment of such values.

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

Figure 5.4: Environment Mapping

Now that we have a generic notion of environment, we can focus on refactoring various traversals as instances of a more general one.

5.2 McBride's Kit

McBride observed the similarity between the types and implementations of renaming and substitution for a well scoped language in his thesis (1999) and then for the

simply typed λ-calculus (STλC) in an unpublished manuscript (2005). This work building on Goguen and McKinna's candidates for substitution (1997) and leading to the collaboration with Benton, Hur, and Kennedy (2012) teaches us how to refactor both traversals into a single more general function.

We highlight these similarities for a scope- and type- preserving implementation of renaming and substitution in fig. 5.5, focusing only on ‘var, ‘app, and ‘lam for the moment as they allow us to make all of the needed observations. There are three differences between the two functions.

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 renaming 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 substitutions we must extend by the zeroth variable seen as a term (‘var z).

```
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 5.5: Renaming and Substitution for the STλC

Having spotted the small differences between renaming and substitution, it is now possible to extract the essence of the the two traversals. McBride defines a notion of "Kit" covering 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:

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 5.6: 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. 5.7.

```
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 5.7: Fundamental lemma of Kit

Convention 6 (Programs as lemmas) *It may seem strange for us to call a program manipulating abstract syntax trees a "fundamental lemma". From the Curry-Howard correspondence's point of view, types are statements and programs are proofs so the notions may be used interchangeably.*

We insist on calling this program (and other ones in this thesis) a "fundamental lemma" because it does embody the essence of the abstraction we have just introduced.

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 corollary of kit

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

```
ren^Kit : Kit Var
ren^Kit .var  = ‘var
ren^Kit .zro  = z
ren^Kit .wkn = s

ren : (Γ –Env) Var Δ → Tm σ Γ → Tm σ Δ
ren = kit ren^Kit
```

Figure 5.8: Kit for Renaming, Renaming as a Corollary of kit

```
sub^Kit : Kit Tm
sub^Kit .var  = id
sub^Kit .zro  = ‘var z
sub^Kit .wkn = ren (pack s)

sub : (Γ –Env) Tm Δ → Tm σ Γ → Tm σ Δ
sub = kit sub^Kit
```

Figure 5.9: Kit for Substitution, Substitution as a Corollary of kit

5.3 Opportunities for Further Generalizations

After noticing that renaming and substitution fit the pattern, it is natural to wonder about other traversals. The observations made here will set the agenda for the next stage of the development in Section 5.4.

The evaluation function used in normalization by evaluation (cf. Catarina Coquand's (2002) work for instance), 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. 5.10.

```
nbe : (Γ –Env) Model Δ → Tm σ Γ → Model 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^Model re <$> ρ) • v) t)
```

Figure 5.10: Normalisation by Evaluation for the STλC

Building on this observation, we are going to generalise the "Kit" approach from mere syntactic transformations to full blown semantics producing values in a given model. This generalisation will bring operations like normalisation by evaluation (chapter 6) but also printing with a name supply (section 5.6), or various continuation passing style translations (chapter 7) into our framework.

In normalisation by evaluation, the notion of arbitrary extensions of the context at hand plays a crucial role in the definition of said model. Indeed, computing with *open* terms implies that model values may be used in a scope that is not the one they came from: evaluation happily goes under binders and brings the evaluation environment along. These extensions explain the changes made to the scope between the time a value was constructed and the time it was used. Provided that values can be transported along such extensions, they give us a way to get the values at the scope we need from the ones at the scope we had.

This idea of arbitrary extensions of the context at hand will be crucial to the definition of our generic notion of semantics (section 5.4). To make it formal, we need to study a well known instance of the generic notion of environments introduced in Section 5.1: the category of thinnings.

Thinnings: A Special Case of Environments

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.

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

Figure 5.11: Definition of Thinnings

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

The Category of Thinnings

These simple observations allow us to prove that thinnings form a category (cf. definition 1): objects are scopes and morphisms between them are thinnings; the identity thinning is essentially the identity function on variables; and composition is given by select in the special case where it is fed two thinnings. Both are defined in fig. 5.12.

Jeffrey's observation mentioned above gives us that composition is associative and that the identity thinning acts as a neutral element for it. We can provide the user with

additional useful combinators such as the ones Altenkirch, Hofmann and Streicher's "Category of Weakening" (1995) is based on such as extend defined in fig. 5.12.

```
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 5.12: Examples of Thinning Combinators

The □ comonad

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 □ is not only a functor, as witnessed by map^□, but also 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. 5.12 corresponds to transitivity in the special case where 𝒱 is Var). The expected laws hold trivially thanks to Jeffrey's trick mentioned above.

```
□ : (List I → Set) → (List I → Set)        map^□ : ∀[ S ⇒ T ] → ∀[ □ S ⇒ □ T ]
(□ T) Γ = ∀[ Thinning Γ ⇒ T ]              map^□ f v th = f (v th)

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

Figure 5.13: 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 5.14: 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 {I} (const A)
th^const a _ = a

th^Var : Thinnable (Var i)
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 5.15: 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.

5.4 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 [2008b]). We get a choice of useful scope- and type- safe traversals by using different 'host languages' for this shallow embedding.

In more concrete terms, the semantics of a term is a mapping from the meaning of its variables to the meaning of the overall term itself. In practice, our semantics is parametrised by two (Type −Scoped) type families $\mathcal{V}$ and C used to assign meanings to variables and terms respectively. That is to say: realisation of a semantics will produce a computation in 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.

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

Figure 5.16: Generic Notion of Computation

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 $((\Gamma\ \mathsf{-Env})\ \mathcal{V}\ \Delta \to (\Gamma\ \mathsf{-Comp})\ \mathcal{C}\ \Delta)$. In cases such as substitution or normalisation by evaluation, $\mathcal{V}$ and $\mathcal{C}$ will happen to coincide but keeping these two relations distinct is precisely what makes it possible to go beyond these and also model renaming or printing with names.

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

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

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

In the following paragraphs, we interleave the definition of the record of constraints Semantics and the evaluation function semantics (the fields of the record appear in pink). In practice the definitions do not need to be mutual.

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.

In the variable case, the semantics function looks up the meaning of the variable at hand in the evaluation environment. It gets back a value $\mathcal{V}$ but needs to return a computation $\mathcal{C}$ instead. This gives us our first constraint: we should be able to embed values into computations.

```
    var : ∀[ 𝒱 σ ⇒ 𝒞 σ ]

  semantics ρ (`var v) = var (lookup ρ v)
```

The semantic λ-abstraction is notable for two reasons. First, following Mitchell and Moggi (1991), its $\Box$-structure allowing arbitrary extensions of the context is typical of models à la Kripke one can find in normalisation by evaluation (Berger and Schwichtenberg [1991], Berger [1993], Coquand and Dybjer [1997], Coquand [2002]). 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 $(\Box\ (\mathcal{V}\ \sigma \Rightarrow \mathcal{C}\ \tau))$.

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

This constraint 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. We face one last hurdle: the value handed to us lives in the extended scope while the ones in the evaluation environment live in the original one. Luckily we were also handed a thinning from the original scope to the extended one. This incites us to add a constraint stating that all values should be thinnable.

```
    th^𝒱 : Thinnable (𝒱 σ)
```

Using both lam and th^$\mathcal{V}$ together with th^Env (defined in fig. 5.15) and _•_ (defined in fig. 5.3), we are able to give a semantics to a λ-abstraction by evaluating its body in a thinned and extended evaluation environment.

```
semantics ρ (‘lam t) = lam (λ re u → semantics (th^Env th^𝒱 ρ re • u) t)
```

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. The evaluation function can simply use this field to combine the result of two recursive calls.

```
app : ∀[ C (σ ‘→ τ) ⇒ C σ ⇒ C τ ]

semantics ρ (‘app t u) = app (semantics ρ t) (semantics ρ u)
```

The same goes for the constructor ‘one of values of the unit type.

```
one : ∀[ C ‘Unit ]

semantics ρ ‘one = one
```

Or the constructors true and false for boolean values

```
tt : ∀[ C ‘Bool ]
ff : ∀[ C ‘Bool ]

semantics ρ ‘tt = tt
semantics ρ ‘ff = ff
```

Or the ‘if-then-else’ eliminator for booleans.

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

semantics ρ (‘ifte b l r) = ifte (semantics ρ b) (semantics ρ l) (semantics ρ r)
```

The type we chose for lam makes the Semantics notion powerful enough that even logical predicates are instances of it. And we will indeed exploit this power when defining normalisation by evaluation as a semantics (cf. chapter 6): 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" because it essentially is the computational content that such a proof would implicitly carry.

But before demonstrating that we can encompass new traversals, it is important to verify that we can still capture the same traversals as McBride's Kit.

5.5 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. We say that "syntax is the identity semantics" because 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 we call syntactic, a function producing a Semantics from a Syntactic.

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


module _ (𝒮 : Syntactic 𝒯) where

  open Syntactic 𝒮

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

Figure 5.17: Every Syntactic gives rise to a 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.

Remark 2 *Note that even though we are re-casting the following operations in categorical terms, we will so far only provide the computational parts of the definitions. Proving that the expected laws hold will be possible once we have introduced our proof framework in chapter 10.*

Renaming i.e. Functor structure

Our first example of a Syntactic operation works with variables as environment values. We have already defined thinning earlier (see section 5.3) and we can turn a variable into a term by using the ‘var constructor. As demonstrated in fig. 5.18, the type of semantics specialised to this semantics is then precisely the proof that terms are thinnable.

```
Syn^Ren : Syntactic Var
Syn^Ren .zro    = z
Syn^Ren .th^𝒯 = th^Var
Syn^Ren .var    = ‘var

Renaming : Semantics Var Term
Renaming = syntactic Syn^Ren

th^Term : Thinnable (Term σ)
th^Term t ρ = semantics Renaming ρ t
```

Figure 5.18: Thinning as a Syntactic instance

Term is a functor between the category of thinnings (cf. section 5.3) and the category of indexed sets and functions (cf. example 3). The type constructor gives us the action on objects and renaming, by turning any (Thinning Γ Δ) into a function from (Term σ Γ) to (Term σ Δ), gives its action on morphisms.

Simultaneous Substitution i.e. Monad Structure

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 semantics reveals that it delivers precisely the simultaneous substitution.

```
Syn^Sub : Syntactic Term
Syn^Sub .zro    = ‘var z
Syn^Sub .th^𝒯 = th^Term
Syn^Sub .var    = id

Substitution : Semantics Term Term
Substitution = syntactic Syn^Sub

sub : (Γ –Env) Term Δ → Term σ Γ → Term σ Δ
sub ρ t = semantics Substitution ρ t
```

Figure 5.19: Parallel Substitution as a Syntactic instance

This gives us the Kleisli extension construction necessary to prove that Term is a relative monad with respect to the Var functor. The unit is trivially the ‘var constructor.

5.6 Printing with Names

As a warmup exercise, let us a look at a relatively simple semantics. Before considering the various model constructions involved in defining normalisation functions deciding

different equational theories, we will look at what an *actual* semantics looks like by considering 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.

Our goal is to generate a string representation of the input term. As such, it should not be surprising that the object language (i.e. the type- and scope- families for values and computation) will not do anything involved with their type index. However this example is still 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 while the computations thread a name-supply to be used to generate fresh names for bound variables. If the values in the environment had to be computations too, we would not root out some faulty implementations e.g a program picking a new name each time the same variable is mentioned.

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 [1991b]), or effectful semantics e.g. logging the various function calls.

The full definition of the printer follows.

In the implementation, we define Name and Printer using a Wrapper with a type and a context as phantom types in order to help Agda's inference propagate the appropriate constraints. The wrapper Wrap does not depend on the scope Γ so it is automatically a Thinnable functor, as witnessed by the (used but not shown) definitions map^Wrap (functoriality) and th^Wrap (thinnability). We also call Fresh the State monad threading the name supply which is essentially a stream of distinct strings.

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

Name : I –Scoped
Name = Wrap String

Fresh : Set → Set
Fresh = State (Stream String ∞)

Printer : I –Scoped
Printer = Wrap (Fresh String)
```

Figure 5.20: Names and Printer for the Printing Semantics

The monad Fresh allows us to generate fresh names for newly bound variables. This is demonstrated by the function fresh defined in fig. 5.21.

We define the printing semantics by collecting intermediate definitions in a record. For this semantics' notion of values, the scope and type indices are phantom types. As a consequence thinning for Names (the field th^$\mathcal{V}$) is trivial: we reuse th^Wrap, a function that simply returns the same name with changed phantom indices. We are now going to look in details at the more interesting cases.

```
fresh : ∀ σ → Fresh (Name σ (σ :: Γ))
fresh σ = do
  names ← get
  put (tail names)
  return (MkW (head names))
```

Figure 5.21: Fresh Name Generation

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

Figure 5.22: Printing as a semantics

The variable case (var) is interesting: after looking up a variable's Name in the environment, we use return to produce the trivial Printer constantly returning that name.

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

As often, the case for λ-abstraction (lam) is the most interesting one. We first use fresh defined in fig. 5.21 to generate a Name, x, for the newly bound variable, then run the printer for the body, mb, in the environment extended with that fresh name and finally build a string combining the name and the body together.

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

```
one : ∀[ Printer 'Unit ]
one = MkW (return "()")

tt : ∀[ Printer 'Bool ]
tt = MkW (return "true")

ff : ∀[ Printer 'Bool ]
ff = MkW (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 parentheses to guarantee that the result is unambiguous.

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

ifte : ∀[ Printer ‘Bool ⇒ Printer σ ⇒ Printer σ ⇒ Printer σ ]
ifte mb ml mr = MkW do
  b ← getW mb
  l ← getW ml
  r ← getW mr
  return (unwords ("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 guarantees 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 will see in section 15.2 how to build an initial environment to print open terms using more advanced tools. Here we are satisfied with a simple print function for closed terms.

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

  printer : Printer σ []
  printer = semantics Printing ε t
```

Figure 5.23: Printer

We can observe print's behaviour by writing a test (cf. feature 7); 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.

Example 9 (Printing a Term) *Here we display the identity function.*

```
_ : print (σ ‘→ σ) (‘lam (‘var z)) ≡ "λa.  a"
_ = refl
```

Chapter 6

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]). A classic evaluation function manipulates a term in a purely syntactic manner, firing redexes one after the other until none is left. In comparison, normalisation by evaluation is a semantic technique that translates a term into a program in the host language, relying on the host language's evaluation machinery to produce a value. The challenging aspect of this technique is twofold. First, we need to pick a translation that retains enough information that we may extract a normal form from the returned value. Second, we need to make sure that our leveraging of the host language's evaluation machinery does decide the equational theory we are interested in. In this chapter, we will consider different equational theories and as a consequence different translations.

Interface of a NBE Procedure

The normalisation by evaluation process is based on a model construction inspired by the logical predicates of normalisation proofs. It is essentially the computational part of such proofs.

Such a construction starts by describing a family of types *Model* by induction on its Type index. Two procedures are then defined: the first (eval) constructs an element of (*Model* σ Γ) provided a well typed term of the corresponding (Term σ Γ) type whilst the second (reify) extracts, in a type-directed manner, normal forms (Nf σ Γ) from elements of the model (*Model* σ Γ). NBE composes the two procedures.

The definition of this eval function is a natural candidate for our Semantics framework. We introduce in fig. 6.1) an abstract interface for NBE formalising this observation. We will see in this chapter that the various variations on normalisation by evaluation that we will consider can all be described as instances of this NBE interface.

The NBE interface is parametrised by two (Type −Scoped) families: the notion of model values (*Model*) and normal forms (*Nf*) specific to this procedure. The interface

packages a Semantics working on the *Model*, an embedding of variables into model values and a reification function from model values to normal forms.

```
record NBE (Model : Type −Scoped) (Nf : Type −Scoped) : Set₁ where
  field Sem    : Semantics Model Model
        embed : ∀[ Var σ ⇒ Model σ ]
        reify  : ∀[ Model σ ⇒ Nf σ ]
```

Figure 6.1: NBE interface

From each NBE instance we can derive a normalisation function turning terms into normal forms. The embed constraint guarantees that we can manufacture an environment of placeholder values in which to run our Semantics to obtain an eval function. Composing this evaluation function with the reification procedure yields the normalisation procedure.

```
eval : ∀[ Term σ ⇒ Model σ ]
eval = semantics Sem (pack embed)

nbe : ∀[ Term σ ⇒ Nf σ ]
nbe = reify ∘ eval
```

Figure 6.2: Evaluation and normalisation functions derived from NBE

During the course of this chapter, we will consider the action of nbe over the following term (λb.λu. (λx.x) (`if b then () else u`)) of type (`'Bool '→ 'Unit '→ 'Unit`). In fig. 6.3 we can see how this test is implemented in our formalisation.

```
test : Nf ('Bool '→ 'Unit '→ 'Unit) []
test = nbe ('lam ('lam ('app ('lam ('var z))
                             ('ifte ('var (s z)) 'one ('var z)))))
```

Figure 6.3: Running example

As we have explained earlier, NBE is always defined *for* a given equational theory. We start by recalling the various rules a theory may satisfy.

Reduction Rules

We characterise an equational theory by a set of reduction rules. The equational theory is obtained by taking the congruence closure of these reduction rules under all term constructors except for λ-abstraction. Indeed, we will consider systems with and without reductions under λ-abstractions and can only make this distinction if using the congruence rule for λ (usually called ξ) is made explicit.

Computation rules Our first set of rules describes the kind of computations we may expect. The β rule states that functions applied to their argument may fire. It is the main driver for actual computation, but the presence of an inductive data type and its eliminator means we have further redexes: the ι rules specify that if-then-else conditionals applied to concrete booleans may return the appropriate branch.

$$\frac{}{(\lambda x.t)u \rightsquigarrow t[u/x]}\beta$$

$$\frac{}{\texttt{if true then } l \texttt{ else } r \rightsquigarrow l}\iota_1 \qquad \frac{}{\texttt{if false then } l \texttt{ else } r \rightsquigarrow r}\iota_2$$

Figure 6.4: Computation rules: β and ι reductions

Canonicity rules 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 the unit type equal to `one`.

$$\frac{\Gamma \vdash t : \sigma \to \tau}{t \rightsquigarrow \lambda x.t\, x}\eta_1 \qquad \frac{\Gamma \vdash t : \texttt{Unit}}{t \rightsquigarrow \texttt{one}}\eta_2$$

Figure 6.5: Canonicity rules: η rules for function and unit types

Congruence rule Congruence rules are necessary if we do not want to be limited to only computing whenever the root of the term happens to already be a reducible expression. By deciding which ones are included, we can however control the evaluation strategy of our calculus. We will study the impact of the ξ-rule that lets us reduce under λ-abstractions --- the distinction between *weak-head* normalisation and *strong* normalisation.

$$\frac{t \rightsquigarrow u}{\lambda x.t \rightsquigarrow \lambda x.u}\xi$$

Figure 6.6: Congruence rule: ξ for strong normalisation

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. The last piece of the puzzle we will need before writing our first evaluator is a notion of normal form.

Normal and Neutral Forms

The inductive family of normal forms characterises irreducible terms for a given set of reduction rules. It is mutually defined with the family of neutral terms that characterises stuck computations i.e. a variable with a spine of eliminators (here applications or if-then-else conditionals).

We parametrise these mutually defined inductive families Ne and Nf by a predicate *NoEta* 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.

```
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 : NoEta σ → ∀[ Ne σ ⇒ Nf σ ]
    ‘one : ∀[ Nf ‘Unit ]
    ‘tt ‘ff : ∀[ Nf ‘Bool ]
    ‘lam : ∀[ (σ ::_) ⊢ Nf τ ⇒ Nf (σ ‘→ τ) ]
```

Figure 6.7: Neutral and Normal Forms

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 III.

We start with a standard strong evaluator i.e. an evaluator implementing *strong* normalisation in the sense that it goes under λ-abstractions. This kind of evaluator is used for instance in Coq (Grégoire and Leroy [2002], Boespflug et al. [2011]) although the focus there is different: the host language (OCaml) is not total, evaluation is untyped, and unsafe features are used to maximise the evaluator's performance.

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

As mentioned above, actual proof systems such as Coq rely on evaluation strategies that avoid applying η-rules: unsurprisingly, it is a rather bad idea to η-expand proof terms which are already large when typechecking complex developments.

In these systems, 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. This leads us to using a predicate *NoEta* which holds for all types thus allowing us to embed all neutrals into normal forms.

Now that this is established, we can focus on the model construction. As noted in the definition of the NBE interface (cf. fig. 6.1), the Semantics underlying normalisation by evaluation will use the same type family for environment values and the computations in the model.

Model Construction

This equational theory can be decided with what happens to be the most straightforward model construction described in fig. 6.8: in our Model everything is either a (non expanded) stuck computation (i.e. a neutral term Ne) or a Value arising from a constructor-headed term and whose computational behaviour is described by an Agda value of the appropriate type. Values of the ‘Unit type are modelled by the Agda's unit type, values of the ‘Bool type are Agda booleans and values of functions from σ to τ are modelled by Kripke function spaces from the type of elements of the Model at type σ and ones at type τ. It is important to note that the functional values 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.

```
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 6.8: Model Definition for $\beta\iota\xi$

We can observe that we have only used families constant in their scope index, neutral forms or □-closed families. All of these are Thinnable hence Value and Model also are. We give these proofs in fig. 6.9.

Reify and Reflect

During the definition of our Semantics acting on elements of type Model, we are inevitably going to be faced with a situation where we are eliminating what happens to be a stuck computation (e.g. applying a stuck function to its argument, or branching over a stuck boolean). In this case we should return a stuck computation. By definition

```
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 6.9: The Model is Thinnable

that means we ought to be able to take the eliminator's semantic arguments and turn them into syntactic objects i.e. to *reify* them. For reasons that will become obvious in the definition of reify in fig. 6.11, we will first need to define reflect, the function that reflects neutral terms as model values.

By definition we can trivially embed neutral terms into the model using the first injection into the disjoint sum type. From reflect we can derive var0, the semantic version of the first variable in scope that we will use to reify the body of a λ-abstraction.

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

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

Figure 6.10: Reflect and Fresh Semantic Variables

Reification is quite straightforward too as demonstrated in fig. 6.11. A Model value is either a neutral term that can be trivially turned into a normal form or a Value. Reification of Values proceeds by induction on their type. Unit values are turned into ‘one, the trivial syntactic object of type ‘Unit. Semantic booleans are reified as their syntactic counterpart. Finally semantic functions give rise to lambdas. In the context thus extended we may craft var0, the semantic counterpart of the fresh variable, and apply the semantic function to it before reifying the resulting semantic body to one in normal form.

A Semantics Targetting our Model

Now that we have defined the Model we are interested in and that we have proved that we can both embed stuck computations into it and obtain normal forms from it, it is time to define a Semantics targetting it. We will study the most striking semantic combinators one by one and then put everything together.

```
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 6.11: Reify

Semantic application is perhaps the most interesting of the combinators needed to define our value of type (Semantics Model Model). It follows the case distinction we mentioned earlier: in case the function is a stuck term, we grow its spine by reifying its argument; otherwise we have an Agda function ready to be applied. We use the extract operation of the □ comonad (cf. fig. 5.13) to say that we are using the function in the ambient context, not an extended one.

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

Figure 6.12: Semantical Counterpart of ‘app

When defining the semantical counterpart of ‘ifte, we follow a similar case distinction. The value case is straightforward: 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 follow the same strategy we used for semantic application: we reify the two branches and assemble a neutral term.

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

Figure 6.13: Semantical Counterpart of ‘ifte

Finally, we have all the necessary components to show that evaluating a term in our Model is a perfectly valid Semantics (we call the corresponding Semantics record Eval but leave it out here). As showcased earlier, normalisation is obtained as a direct

corollary of NBE by the composition of reification and evaluation in an environment of placeholder values.

```
nbe : NBE Model Nf
nbe = record
  { Sem   = Eval
  ; embed = reflect ∘ 'var
  ; reify = reify _
  }
```

Now that we have our definition of NBE for the $\beta\iota\xi$ rules, we can run the test defined in fig. 6.3 and, obtaining (λb.λu. `if b then () else` u), observe that we have indeed reduced all of the $\beta\iota$ redexes, even if they were hidden under a λ-abstraction. Note however that we still have a stuck if-then-else conditional even though the return type is a record type with only one inhabitant: we are not performing η-expansion so we cannot expect this type of knowledge to be internalised!

```
_ : test nbe ≡ 'lam ('lam ('neu _ ('ifte ('var (s z)) 'one ('neu _ ('var z)))))
_ = refl
```

Figure 6.14: Running example: the $\beta\iota\xi$ case

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

As we have just seen, we can leverage the host language's evaluation machinery to write a normaliser for our own embedded language. We opted for a normalisation procedure deciding the equational theory corresponding to $\beta\iota\xi$ rules. We can develop more ambitious model constructions that will extend the supported equational theory. In this section we will focus on η-rules but it is possible to go even beyond and build models that decide equational theories incorporating for instance various functor and fusion laws. The interested reader can direct its attention to our previous work formalising such a construction (2013a).

In order to decide $\beta\iota\xi\eta$ rules, we are going to build the Model by induction on its type argument. This time we do not need to distinguish neutral terms from terms that compute: each type may or may not have neutral forms depending on whether it has a set of accompanying η-rules. Functions (respectively inhabitants of 'Unit) in the source language will be represented as function spaces (respectively values of type ⊤) no matter whether they are stuck or not. Evaluating a 'Bool may however yield a stuck term so we cannot expect the model to give us anything more than an open term in normal form. Note that there are advanced constructions adding η-rules for sum types (Ghani [1995], Altenkirch and Uustalu [2004], Lindley [2007]) but they are unwieldy and thus outside the scope of this thesis.

The model construction then follows the established 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      = const ⊤
Model 'Bool      = Nf 'Bool
Model (σ '→ τ) = □ (Model σ ⇒ Model τ)
```

Figure 6.15: 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 6.16: Values in the Model are Thinnable

Application's semantic counterpart is easy to define: given that $\mathcal{V}$ and 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 6.17: Semantic Counterpart of 'app

Conditional branching however is more subtle: the boolean value 'ifte 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).

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

```
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 6.18: Reify and Reflect

```
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 6.19: Semantic Counterpart of ‘ifte

matches precisely the shape expected in a Semantics. Because the environment carries model values, the variable case is trivial.

```
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 6.20: Evaluation is a Semantics

With these definitions in hand, we can instantiate the NBE interface we introduced in fig. 6.1. This gives us access to 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.

```
nbe : NBE Model Nf
nbe = record
  { Sem   = Eval
  ; embed = reflect _ ∘ 'var
  ; reify = reify _
  }
```

Figure 6.21: Normalisation as Reification of an Evaluated Term

We can now observe the effect of this normaliser implementing a stronger equational theory by running our test defined fig. 6.3 and obtaining (λb.λu. one). In the previous section, the normaliser yielded a term still containing a stuck if-then-else conditional that has here been replaced with the only normal form of the unit type.

```
_ : test nbe ≡ 'lam ('lam 'one)
_ = refl
```

Figure 6.22: Running example: the $\beta\iota\xi\eta$ case

6.3 Normalisation by Evaluation for $\beta\iota$

The decision to apply the η-rule lazily as we have done at the beginning of this chapter 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.

```
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 6.23: Weak-Head Normal and Neutral Forms

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 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 6.24: Model Definition for Computing Weak-Head Normal Forms

Thinnable can be defined rather straightforwardly based on the template provided in the previous sections: 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.

```
APP : ∀[ Model (σ ‘→ τ) ⇒ Model σ ⇒ Model τ ]
APP (f , inj₁ whne) (t , _)      = (‘app f t , inj₁ (‘app whne t))
APP (f , inj₂ fun)  T@(t , _) = (‘app f t , proj₂ (extract fun 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 6.25: Semantical Counterparts of ‘app and ‘ifte

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.

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

Figure 6.26: 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 NBE where embed is obtained by using reflect.

We can once more run our test and observe that it simply outputs the term it was fed. Indeed our example is λ-headed, meaning that it is already in weak-head normal form and that the normaliser does not need to do any work.

```
_ : test nbe ≡ ‘lam (‘lam (‘app (‘lam (‘var z))
                                (‘ifte (‘var (s z)) ‘one (‘var z))))
_ = refl
```

Figure 6.27: Running example: the $\beta\iota$ case

Chapter 7

CPS Transformations

A Continuation-Passing Style (CPS) transformation is an operation turning an existing program into one that takes an extra argument (the continuation) that is a reification of what is to be done once the program's computation is finished. This is a powerful technique sequentialising the process of evaluating a term and thus a common compilation pass for functional programming languages.

In the rest of this introduction, we will write $[\![.]\!]$ when talking about a translation functional. It takes a term and produces a terms that starts by binding a continuation. This continuation expects to be passed the value resulting from that term's evaluation.

For instance, the translation of the constant program 3 is as follows: $[\![3]\!] = \lambda k.k(3)$ where k is the usual name given to the continuation. Indeed, 3 already is a value and the only thing left to do is to invoke the continuation with it.

A more complex expression such as $g(f(3,2),5)$ would need to be decomposed into simpler subexpressions thus clarifying the order in which it is to be evaluated. We could for instance write: $[\![g(f(3,2),5)]\!] = \lambda k.[\![f]\!](3,2,\lambda v.[\![g]\!](v,5,k))$.

In the process of writing such a transformation, we have made an arbitrary choice: we have decided to first evaluate $f(3,2)$ and then evaluate g with the resulting value. But this is not the only valid transformation. When translating a function application ($f(t)$) we may also decide to evaluate the function and apply the result straight away without modifying the ambient continuation (i.e. $[\![f(t)]\!] = \lambda k.[\![f]\!](\lambda v_f.v_f([\![t]\!],k))$) or to evaluate the function, then the function's argument and continue by applying one to the other (i.e. $[\![f(t)]\!] = \lambda k.[\![f]\!](\lambda v_f.[\![t]\!](\lambda v_t.v_f(v_t,k)))$).

All of these translations correspond to variants of the call-by-name and call-by-value evaluation strategies. In their generic account of continuation passing style transformations, Hatcliff and Danvy (1994) decompose both call by name and call by value CPS transformations in two phases. We will see that this effort can be refactored via our Semantics framework.

A Common Framework: Moggi's Meta Language

The two phases introduced by Hatcliff and Danvy are as follows: the first one, embedding the source language into Moggi's Meta Language (1991b), 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 7.1: Inductive Definition of Types for Moggi's ML

Then comes the Meta-Language itself (cf. fig. 7.2). 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.

```
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 7.2: Definition of Moggi's Meta Language

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 (CBN) 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.

7.1 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 7.3: 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. 7.4.

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

Figure 7.4: ·−Scoped Transformer for Call by Name

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.

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.

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.

```
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 7.5: Semantical Counterparts for 'app and 'ifte

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 7.6: 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.

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.

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

```
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 7.7: Values and Computations for the CBN CPS Semantics

```
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 7.8: Semantical Counterparts for ‘app

7.2 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. $(\cdot \text{ `}\!\!\rightarrow r) \text{ `}\!\!\rightarrow r$).

```
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 7.9: Translating Moggi's ML's Types to STLC Types

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*.

```
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 correspond 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 (Peyton Jones and Launchbury [1991]) but lazy in every other type.

Chapter 8

Discussion

8.1 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.

8.2 Related Work

This part of the work focuses on programming and not (yet!) on proving, apart from the basic well-formedness properties we can easily enforce in the type: terms are well-scoped, well-typed and when we claim to write a normaliser we do guarantee that the output is a normal form. All of this 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 thesis.

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 [2008b]) 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 Bruijn 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 [2004b]) 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 (Coq Development 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.

8.3 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 datatype-generic description of syntaxes with binding and the appropriate notion of Semantics for each syntax. We will make this intuition precise in part III.

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.

Part II

And Their Proofs

Chapter 9

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 of these semantics' properties. 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.

9.1 Relations and Environment Relation Transformer

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

In Section 5.1 we introduced a generic notion of well typed and scoped environment as a function from variables to values. Its formal definition is given in Figure 5.1 as a record type. This record wrapper helps Agda's type inference reconstruct the type family of values whenever it is passed an environment.

For the same reason, we will use a record wrapper for the concrete implementation of our notion of relation over (I −Scoped) families. A Relation between two such families T and U is a function which to any σ and Γ associates a relation between (T

$\sigma\ \Gamma$) and ($U\ \sigma\ \Gamma$). Our first example of such a relation is Eq^R the equality relation between an (I–Scoped) family T and itself.

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

```
Eq^R : Rel T T
rel Eq^R i = _≡_
```

Figure 9.1: Relation Between I–Scoped Families and Equality Example

Once we know what relations are, we are going to have to lift relations on values and computations to relations on environments or Kripke function spaces. These relation transformers will be instances of what Kawahara calls 'relators'. This is what the rest of this section focuses on.

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

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

Figure 9.2: Relating Γ-Environments in a Pointwise Manner

The first example of two environment being related is refl^R that, to any environment ρ associates a trivial proof of the statement (All $\mathsf{Eq}^R\ \Gamma\ \rho\ \rho$).

```
refl^R : All Eq^R Γ ρ ρ
lookup^R refl^R k = refl
```

Figure 9.3: Pointwise Lifting of refl

The combinators we introduced in Figure 5.4 to build environments (ε, _•_, etc.) have natural relational counterparts.

Convention 7 (Relational counterparts) *We systematically reuse the same names for a constructor or a combinator and its relational counterpart. We simply append an* ^{R} *suffix to the relational version.*

.

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

9.2 Simulation Constraints

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.

The evidence that we have a Simulation between two Semantics ($\mathcal{S}^A$ and $\mathcal{S}^B$) is packaged in a record indexed by the semantics as well as the two relations ($\mathcal{V}^R$ and $\mathcal{C}^R$) mentioned earlier. We start by making formal this idea of related evaluations by introducing $\mathcal{R}$, a specialisation of $\mathcal{C}^R$:

```
ℛ : ∀ σ → (Γ –Env) 𝒱ᴬ Δ → (Γ –Env) 𝒱ᴮ Δ → Term σ Γ → Set
ℛ σ ρᴬ ρᴮ t = rel 𝒞ᴿ σ (semantics 𝒮ᴬ ρᴬ t) (semantics 𝒮ᴮ ρᴮ t)
```

We can now spell out what the simulation constraints are. Following the same presentation we used for Semantics (cf. section 5.4), we will study the constraints field by field and accompany them with comments as well as showing the use we make of these constraints in the proof of the fundamental lemma of Simulation.

We start by giving the two types at hand: the Simulation record parameterised by the two semantics and corresponding relations and the statement of simulation, the fundamental lemma of simulations taking proofs that evaluation environments are related and returning proofs that the respective evaluation functions yield related results.

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

  simulation : All 𝒱ᴿ Γ ρᴬ ρᴮ → ∀ t → ℛ σ ρᴬ ρᴮ t
```

The set of simulation constraints is in one-to-one correspondence with that of semantical constructs. We start with the relational counterpart of ‘var and var. 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 $\mathcal{C}^R$. That is to say that the respective var turn related values into related computations.

```
  varᴿ : All 𝒱ᴿ Γ ρᴬ ρᴮ → (v : Var σ Γ) → ℛ σ ρᴬ ρᴮ (‘var v)

  simulation ρᴿ (‘var v) = varᴿ ρᴿ v
```

Value constructors in the language follow a similar pattern: provided that the evaluation environments 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      simulation ρ^R 'one = one^R ρ^R
tt^R  : All V^R Γ ρ^A ρ^B → R 'Bool ρ^A ρ^B 'tt       simulation ρ^R 'tt  = tt^R ρ^R
ff^R  : All V^R Γ ρ^A ρ^B → R 'Bool ρ^A ρ^B 'ff       simulation ρ^R 'ff  = ff^R ρ^R
```

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)

simulation ρ^R ('app f t)   = app^R ρ^R f t (simulation ρ^R f) (simulation ρ^R t)
simulation ρ^R ('ifte b l r) =
  ifte^R ρ^R b l r (simulation ρ^R b) (simulation ρ^R l) (simulation ρ^R 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 9.4: 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. Together with a constraint guaranteeing that value-relatedness is stable under thinnings, this is enough to discharge the λ case and thus conclude the proof.

```
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)
```

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

simulation ρᴿ (‘lam b) = lamᴿ ρᴿ b $ λ ren vᴿ →
                          simulation ((th^𝒱ᴿ ren <$>ᴿ ρᴿ) •ᴿ vᴿ) b
```

Now that we have our set of constraints together with a proof that they entail the theorem we expected to hold, we can start looking for interesting instances of a Simulation.

9.3 Syntactic Traversals are Extensional

A first corollary of the fundamental lemma of simulations is the fact that semantics arising from a Syntactic (cf. fig. 5.17) 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 pointwise 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 Eqᴿ Eqᴿ
Syn-ext .th^𝒱ᴿ = λ ρ eq → cong (λ t → th^𝒯 t ρ) eq
Syn-ext .varᴿ   = λ ρᴿ v → cong var (lookupᴿ ρᴿ v)
Syn-ext .lamᴿ   = λ ρᴿ b bᴿ → cong ‘lam (bᴿ extend refl)
Syn-ext .appᴿ   = λ ρᴿ f t → cong₂ ‘app
Syn-ext .ifteᴿ  = λ ρᴿ b l r → cong₃ ‘ifte
Syn-ext .oneᴿ   = λ ρᴿ → refl
Syn-ext .ttᴿ    = λ ρᴿ → refl
Syn-ext .ffᴿ    = λ ρᴿ → refl
```

Figure 9.5: 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.

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 (2019): 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.

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

Figure 9.6: Syntactic Traversals are Extensional

More generally when working with higher-order functions in intensional type theory, it is extremely useful to know that from *extensionally* equal inputs we can guarantee that we will obtain *intensionally* equal outputs.

9.4 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 of variables, it is indeed equal to the corresponding renaming. We can make this idea formal by introducing the VarTerm^R relation stating that a variable and a term are morally equal like so:

```
VarTerm^R : Rel Var Term
rel VarTerm^R σ v t = ‘var v ≡ t
```

Figure 9.7: Characterising Equal Variables and Terms

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 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 VarTerm^R Eq^R
RenSub^Sim .th^𝒱^R = λ ρ → cong (λ t → th^Term t ρ)
RenSub^Sim .var^R  = λ ρ^R v → lookup^R ρ^R v
RenSub^Sim .lam^R  = λ ρ^R b b^R → cong ‘lam (b^R extend refl)
RenSub^Sim .app^R  = λ ρ^R f t → cong₂ ‘app
RenSub^Sim .ifte^R = λ ρ^R b l r → cong₃ ‘ifte
RenSub^Sim .one^R  = λ ρ^R → refl
RenSub^Sim .tt^R   = λ ρ^R → refl
RenSub^Sim .ff^R   = λ ρ^R → refl
```

Figure 9.8: 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 (pack^R (λ v → refl)) t
```

Figure 9.9: Renaming as a Substitution

9.5 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. Some properties that may not be provable for arbitrary Agda programs can be proven to hold for the ones obtained by evaluation. In particular, functional extensionality is not provable in intensional type theory but we can absolutely prove that the programs we obtain by evaluating terms with a function type take extensionally equal inputs to extensionally equal outputs.

This strong property will in particular imply that evaluation in environment consisting of *extensionally* equal values will yield *intensionally* equal normal forms as shown in fig. 9.15.

This notion of extensional equality for values in the model is formalised by defining a Partial Equivalence Relation (PER) (Mitchell [1996]) 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.

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. 6.16.

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.

```
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 9.10: Partial Equivalence Relation for Model Values

```
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      b^R ρ = cong (λ t → th^Nf t ρ) b^R
th^PER (σ ‘→ τ) f^R  ρ = λ σ → f^R (select ρ σ)
```

Figure 9.11: Stability of the PER under Thinning

```
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 (cong₂ ‘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 9.12: 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. 6.19 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 (cong₃ 'ifte ('neu-injective b^R) (reify^R σ l^R) (reify^R σ r^R))
```

Figure 9.13: 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^𝒱^R = λ ρ EQ → th^PER _ EQ ρ
Eval^Sim .var^R  = λ ρ^R v → lookup^R ρ^R v
Eval^Sim .lam^R  = λ ρ^R b b^R → b^R
Eval^Sim .app^R  = λ ρ^R f t f^R t^R → f^R identity t^R
Eval^Sim .ifte^R = λ ρ^R b l r → IFTE^R _ _
Eval^Sim .one^R  = λ ρ^R → refl
Eval^Sim .tt^R   = λ ρ^R → refl
Eval^Sim .ff^R   = λ ρ^R → refl
```

Figure 9.14: 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.

```
norm^R : All PER Γ ρ^l ρ^r → ∀ t → reify σ (eval ρ^l t) ≡ reify σ (eval ρ^r t)
norm^R ρ^R t = reify^R σ (simulation Eval^Sim ρ^R t)
```

Figure 9.15: 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 10

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.

10.1 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. As always, provided that these constraints are satisfied we should be able to prove fusion the fundamental lemma of Fusion stating that given related environments we will get related outputs. We will interleave the definition of the record's fields together with the proof of fusion. Our first goal will be to define this notion of relatedness formally.

```
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

fusion : ℰᴿ ρᴬ ρᴮ ρᴬᴮ → ∀ t → ℛ σ ρᴬ ρᴮ ρᴬᴮ t
```

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 a way to extract

a term from a computation returned by $\mathcal{S}^A$. Hence our first field reify^A. Typical examples include the identity function (whenever the first semantics is a Syntactic one, cf. section 10.2) or one of the reify functions followed by an erasure from Nf to Term (whenever the first semantics corresponds to normalisation by evaluation)

$$\mathsf{reify}^A : \forall[\ C^A\ \sigma \Rightarrow \mathsf{Term}\ \sigma\ ]$$

Once we have this key component we can introduce the relation $\mathcal{R}$ 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 semantics $\mathcal{S}^B$.

$$\begin{aligned}
&\mathcal{R} : \forall\ \sigma \to (\Gamma\ \mathsf{-Env})\ \mathcal{V}^A\ \Delta \to (\Delta\ \mathsf{-Env})\ \mathcal{V}^B\ \Theta \to (\Gamma\ \mathsf{-Env})\ \mathcal{V}^{AB}\ \Theta \to \\
&\quad \mathsf{Term}\ \sigma\ \Gamma \to \mathsf{Set} \\
&\mathcal{R}\ \sigma\ \rho^A\ \rho^B\ \rho^{AB}\ t = \mathsf{let}\ v^A \ = \mathsf{semantics}\ \mathcal{S}^A\ \rho^A\ t \\
&\qquad\qquad\qquad\qquad\quad\ v^B \ = \mathsf{semantics}\ \mathcal{S}^B\ \rho^B\ (\mathsf{reify}^A\ v^A) \\
&\qquad\qquad\qquad\qquad\quad\ v^{AB} = \mathsf{semantics}\ \mathcal{S}^{AB}\ \rho^{AB}\ t \\
&\qquad\qquad\qquad\qquad \mathsf{in\ rel}\ C^R\ \sigma\ v^B\ v^{AB}
\end{aligned}$$

As with the previous section, only a handful of the relational counterpart to the term constructors and their associated semantic counterpart 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.

$$\mathsf{var}^R : \mathcal{E}^R\ \rho^A\ \rho^B\ \rho^{AB} \to (v : \mathsf{Var}\ \sigma\ \Gamma) \to \mathcal{R}\ \sigma\ \rho^A\ \rho^B\ \rho^{AB}\ (\text{‘var}\ v)$$

$$\mathsf{fusion}\ \rho^R\ (\text{‘var}\ v) = \mathsf{var}^R\ \rho^R\ v$$

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.

$$\begin{aligned}
&\mathsf{one}^R : \mathcal{E}^R\ \rho^A\ \rho^B\ \rho^{AB} \to \mathcal{R}\ \text{‘Unit}\ \ \rho^A\ \rho^B\ \rho^{AB}\ \text{‘one} \qquad && \mathsf{fusion}\ \rho^R\ \text{‘one} = \mathsf{one}^R\ \rho^R \\
&\mathsf{tt}^R \ : \mathcal{E}^R\ \rho^A\ \rho^B\ \rho^{AB} \to \mathcal{R}\ \text{‘Bool}\ \rho^A\ \rho^B\ \rho^{AB}\ \text{‘tt} && \mathsf{fusion}\ \rho^R\ \text{‘tt} \ = \mathsf{tt}^R\ \rho^R \\
&\mathsf{ff}^R \ : \mathcal{E}^R\ \rho^A\ \rho^B\ \rho^{AB} \to \mathcal{R}\ \text{‘Bool}\ \rho^A\ \rho^B\ \rho^{AB}\ \text{‘ff} && \mathsf{fusion}\ \rho^R\ \text{‘ff} \ = \mathsf{ff}^R\ \rho^R
\end{aligned}$$

Similarly, we have purely structural constraints for term constructs which have purely structural semantic 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.

$$\begin{aligned}
&\mathsf{app}^R : \mathcal{E}^R\ \rho^A\ \rho^B\ \rho^{AB} \to \\
&\qquad \forall\ f\ t \to \mathcal{R}\ (\sigma\ \text{‘}\to\ \tau)\ \rho^A\ \rho^B\ \rho^{AB}\ f \to \mathcal{R}\ \sigma\ \rho^A\ \rho^B\ \rho^{AB}\ t \to \\
&\qquad \mathcal{R}\ \tau\ \rho^A\ \rho^B\ \rho^{AB}\ (\text{‘app}\ f\ t)
\end{aligned}$$

```
ifte^R : E^R ρ^A ρ^B ρ^AB →
         ∀ 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)

fusion ρ^R ('app f t)   = app^R ρ^R f t (fusion ρ^R f) (fusion ρ^R t)
fusion ρ^R ('ifte b l r) = ifte^R ρ^R b l r (fusion ρ^R b) (fusion ρ^R l) (fusion ρ^R r)
```

Finally, the 'lam-case is as always the most complex of all. The constraint we pick 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, a placeholder value which is another parameter of this record. 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.

```
var0^A : ∀[ (σ ::_) ⊢ V^A σ ]

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)
```

This last observation reveals the need for additional combinators stating that the environment-manipulating operations we have used respect the notion of relatedness at hand. We introduce two constraints dealing with the relations talking about evaluation environments. _•^{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 var0^A. th^$\mathcal{E}^R$ 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_ : E^R ρ^A ρ^B ρ^AB → rel V^R σ v^B v^AB →
        E^R (th^Env S^A.th^V ρ^A extend • var0^A) (ρ^B • v^B) (ρ^AB • v^AB)
th^E^R : E^R ρ^A ρ^B ρ^AB → (ρ : Thinning Θ Ω) →
         E^R ρ^A (th^Env S^B.th^V ρ^B ρ) (th^Env S^AB.th^V ρ^AB ρ)
```

Using these last constraint, we can finally write the case of the fusion proof dealing with λ-abstraction.

```
fusion ρ^R ('lam b) = lam^R ρ^R b $ λ ren v^R → fusion (th^E^R ρ^R ren •^R v^R) b
```

As with simulation, we measure the usefulness of this framework not only by our ability to prove its fundamental lemma but also to then obtain useful corollaries. We will start with the special case of syntactic semantics.

10.2 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 fusible 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 $\_\bullet^R\_$ and $\mathsf{th}\hat{}\mathcal{E}^R$ 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 5.5.

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.

```
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 10.1: Fundamental Lemma of Syntactic Fusions

10.3 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 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. 5.12.

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.

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.

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 $\mathsf{VarTerm}^R$ defined in fig. 9.7 relates variables and terms by wrapping the variable in a 'var constructor and demanding it is equal to the term.

Finally, the fusion of two sequential substitutions into a single one uses a similar

```
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 10.2: Syntactic Fusion of Two Renamings

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

Figure 10.3: Corollary: Renaming Fusion Law

```
RenSub : SynFusion Syn^Ren Syn^Sub Syn^Sub
                   (λ ρ^A ρ^B → All Eq^R _ (select ρ^A ρ^B)) Eq^R
```

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

Figure 10.4: Renaming - Substitution Fusion Law

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.

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

10.4 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.

```
SubRen : SynFusion Syn^Sub Syn^Ren Syn^Sub
                   (λ ρ^A ρ^B → All Eq^R _ (ren ρ^B <$> ρ^A)) VarTerm^R


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

Figure 10.5: Substitution - Renaming Fusion Law

```
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.fusion fus refl^R
```

Figure 10.6: Substitution Fusion Law

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 straightforward 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.

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. 9.13: 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.

```
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 10.7: Relational Application

```
IFTE^R : ∀ 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)
IFTE^R b l r b^R l^R r^R with eval ρ^B (th^Term b ρ^A) | eval ρ^AB b
... | 'tt          | 'tt          = l^R
... | 'ff          | 'ff          = r^R
... | 'neu _ b₁ | 'neu _ b₂ =
  reflect^R σ $ cong₃ 'ifte ('neu-injective b^R) (reify^R σ l^R) (reify^R σ r^R)
```

Figure 10.8: 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 ρ^A ρ^B → All PER _ (select ρ^A ρ^B)) PER PER
RenEval .reify^A   = id
RenEval .var0^A    = z
RenEval ._•^R_     = λ ρ^R v^R → v^R ::^R lookup^R ρ^R
RenEval .th^E^R    = λ ρ^R ρ → (λ v → th^PER _ v ρ) <$>^R ρ^R
RenEval .var^R     = λ ρ^R → lookup^R ρ^R
RenEval .one^R     = λ ρ^R → refl
RenEval .tt^R      = λ ρ^R → refl
RenEval .ff^R      = λ ρ^R → refl
RenEval .app^R     = λ ρ^R → APP^R
RenEval .ifte^R    = λ ρ^R → IFTE^R
RenEval .lam^R     = λ ρ^R b b^R → b^R
```

Figure 10.9: 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 = fusion RenEval (select^R th ρ^R) t
```

Figure 10.10: 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.

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 10.11: 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 10.4: we start by defining a notation $\mathcal{R}$ to lighten the types, then combinators APP^R and IFTE^R. The cases for $\mathsf{th}^{\wedge}\mathcal{E}^R$, $\_\bullet^R\_$, 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

SubEval : Fusion Substitution Eval Eval Sub^R PER PER

Figure 10.12: Substitution Followed by Evaluation is an Evaluation

as well as an appeal to reneval (fig. 10.10) and multiple uses of Eval^Sim (fig. 9.14). Because the technical details do not give any additional hindsight, we do not include the proof here.

Chapter 11

Discussion

11.1 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.

11.2 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.

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

11.3 Further work

We have now fulfilled one of the three goals we highlighted in chapter 8. 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 syntactic traversal followed by a semantics $\mathcal{S}$ into a single pass of $\mathcal{S}$.

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 9, 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 seemingly 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. [2019]). 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 III

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

Chapter 12

A 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.

In this exercise 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 terms 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 know how to do this from chapter 4. In a textbook such as Pierce's (2002) we would simply start by defining T and then state that S is T extended with let bindings.

$$\begin{array}{lll} \texttt{<T>} & ::= & x \mid \texttt{<T><T>} \mid \lambda x.\texttt{<T>} \\ \texttt{<S>} & ::= & \cdots \mid \texttt{let}\ x\ =\ \texttt{<S>}\ \texttt{in}\ \texttt{<S>} \end{array}$$

When formalising this definition however we need to write down the same constructor types twice for ‘var, ‘lam, and ‘app as demonstrated in fig. 12.1.

```
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 12.1: Source and Target Languages

Ignoring for now the Semantics framework, we jump straight to defining the program transformation we are interested in. Given an environment of target terms, it will elaborate a source term into a target one.

Informally, we could state that it proceeds by a simple case distinction. On the one hand all the constructors that exist in both languages should be mapped to their immediate counterpart and their subterms recursively simplified. And on the other a let-binding should be elaborated away by elaborating the let-bound expression first, putting the result into the evaluation environment and returning the result of elaborating the body in the thus extended environment.

In practice however things are more complex. 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. And even though our informal description highlighted that all cases but one are purely structural, we still need to painstakingly go through them one by one.

```
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 12.2: 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. 12.3: 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 12.3: 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 doing 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 we 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 16.4 where we define a *generic* notion of let-binding to extend any language with together with the corresponding generic let-inlining transformation.

Chapter 13

A Primer on Universes of Data Types

To achieve a syntax-generic presentation of the work presented in Parts I and II, we need to find a language to talk about syntaxes in general. The solution we adopt is inspired by previous work on data-generic constructions.

There is a long tradition of data-generic developments in dependently typed languages because, as Altenkirch and McBride remark, "generic programming is just programming" within dependent types (2002). This style of programming is typified by Benke, Dybjer and Jansson's universes for generic programs and proofs (2003) or Chapman, Dagand, McBride and Morris' (CDMM, 2010) universe of data types. This last work is inspired by Dybjer and Setzer's finite axiomatisation of Inductive-Recursive definitions (1999).

All of these proceed by an explicit definition of syntactic *codes* for data types that are given a semantics as objects in the host language. Users can perform induction on these codes 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.

13.1 Datatypes as Fixpoints of Strictly Positive Functors

The first component of the definition of CDMM's universe is an inductive type of Descriptions of strictly positive functors and their fixpoints.

Descriptions and Their Meaning as Functors

Our type of descriptions represents functors from $\mathbf{Set}^I$ to $\mathbf{Set}^J$. These functors correspond to I-indexed containers of J-indexed payloads. Keeping these index types distinct prevents mistaking one for the other when constructing the interpretation of descriptions. Later of course we can use these containers as the nodes of recursive datastructures by interpreting some payloads sorts as requests for subnodes (Altenkirch et al. [2015]) i.e. we will identify I and J when the time comes to build a fixpoint.

We will interleave the definition of the type of descriptions and the recursive function $[\![\_]\!]$ assigning a meaning to them provided it is handed the meaning to attach

to the substructures. Interpretation of descriptions gives rise to right-nested tuples terminated by equality constraints.

```
data Desc (I J : Set) : Set₁ where        ⟦_⟧ : Desc I J → (J → Set) → (I → Set)
```

First, ‘σ stores data. This can be used to either attach a payload to a node in which case the rest of the description will be constant in the value stored, or offer a choice of possible descriptions computed from the stored value. It is interpreted as a Σ type in the host language.

```
‘σ : (A : Set) → (A → Desc I J) →        ⟦ ‘σ A d ⟧ X i = Σ A (λ a → ⟦ d a ⟧ X i)
     Desc I J
```

Then ‘X attaches a recursive substructure indexed by J. Its interpretation is unsuprisingly a simple call to the argument giving a meaning to substructures.

```
‘X : J → Desc I J → Desc I J              ⟦ ‘X j d ⟧ X i = X j × ⟦ d ⟧ X i
```

Finally, ‘■ to stop with a particular index value. Its interpretation makes sure that the index we demanded corresponds to the one we were handed.

```
‘■ : I → Desc I J                         ⟦ ‘■ j ⟧ X i = i ≡ j
```

Note that our universe definition precludes higher-order branching *by choice*: we are interested in terms and these are first-order objects.

These constructors give the programmer the ability to build up the data types they are used to. For instance, the functor corresponding to lists of elements in A stores a Boolean which stands for whether the current node is the empty list or not. Depending on its value, the rest of the description is either the “stop” token or a pair of an element in A and a recursive substructure 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 13.1: 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 13.2: The Description of the base functor for Vec A n

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 we provide its definition in fig. 13.3.

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

Figure 13.3: Least Fixpoint of an Endofunctor

Equipped with this fixpoint operator, we can go back to our examples redefining lists and vectors as instances of this framework. We can see in Figure 13.4 that we can recover the types we are used to thanks to this least fixpoint (we only show the list example). Pattern synonyms let us hide away the encoding: users can use them to pattern-match on lists and Agda conveniently resugars them when displaying a goal.

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

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

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

example : List (List Bool)
example = (false :: []) :: (true :: []) :: []
```

Figure 13.4: List Type, Patterns and Example

Convention 8 (Patterns for Layers and Datatypes) *Whenever we work with a specific example in a universe of datatypes or syntaxes, we introduce a set of patterns that work on a single ⟦_⟧-defined layer (the primed version of the pattern) and one that works on full μ-defined values (the classic version of the pattern). In most cases users will only use the classic version of the pattern to write their programs. But when using generic fold-like function (see the next section) they may be faced with a single layer.*

Now that we have the ability to represent the data we are interested it, we are only missing the ability to write programs manipulating it.

13.2 Generic Programming over Datatypes

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 13.5: Action on Morphisms of the Functor corresponding to a Description

Once we take the fixpoint of such functors, we expect to be able to define an iterator. It is given by the definition of fold d, our second generic program over datatypes. In fig. 13.3, we skipped over the Size (Abel [2010]) index added to the inductive definition of μ. It 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 3.3.

```
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 13.6: Eliminator for the Least Fixpoint

Finally, fig. 13.7 demonstrates that we can get our hands on the types' eliminators we are used to by instantiating the generic fold we have just defined. Note that we are here using the primed versions of the patterns: the algebra of a fold takes as argument a single layer of data where the substructures have already been replaced with inductive hypotheses.

We can readily use this eliminator to define e.g. append, and then flatten for lists and check that it behaves as we expect by running it on the example introduced in fig. 13.4.

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 are going to see whether

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

Figure 13.7: The Eliminator for List

```
_++_ : List A → List A → List A
xs ++ ys = foldr _::_ ys xs

flatten : List (List A) → List A
flatten = foldr _++_ []

_ : flatten example ≡ false :: true :: []
_ = refl
```

Figure 13.8: Applications

we could use CDMM's approach to represent abstract syntax trees equipped with a notion of binding.

13.3 A Free Monad Construction

The natural candidate to represent trees with binding and thus variables is to take not the least fixpoint of a strictly positive functor but rather the corresponding free monad. It can be easily defined by reusing the interpretation function, adding a special case corresponding to the notion of variable we may want to use.

```
data Free (d : Desc I I) : Size → (I → Set) → (I → Set) where
  pure : ∀[ X                  ⇒ Free d (↑ s) X ]
  node : ∀[ ⟦ d ⟧ (Free d s X) ⇒ Free d (↑ s) X ]
```

Figure 13.9: Free Indexed Monad of a Description

The pure constructor already gives us a unit for the free monad, all we have left to do is to define the Kleisli extension of a morphism. The definition goes by analysis over the term in the free monad. The pure case is trivial and, once again, the node case essentially amounts to using fmap to recursively call kleisli on all of the subterms of a node.

This gives us a notion of trees with variables in them and a substitution operation replacing these variables with entire subtrees. The functor underlying any well scoped and sorted syntax can be coded as some Desc (I × List I) (I × List I), with the free

```
kleisli : ∀ d → ∀[ X ⇒ Free d ∞ Y ] → ∀[ Free d s X ⇒ Free d ∞ Y ]
kleisli d f (pure x) = f x
kleisli d f (node t) = node (fmap d (kleisli d f) t)
```

Figure 13.10: Kleisli extension of a morphism with respect to Free d

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.

We will see in the next chapter how to modify CDMM's universe construction to account for variable binding and finally obtain a scope-aware universe.

Chapter 14

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 (defined in fig. 4.5).

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.

14.1 Descriptions and Their Meaning as Functors

Following our approach in the previous chapter, we are going to interleave the definition of the indexed datatype of descriptions and the meaning function ⟦_⟧ associated to it. They are essentially identical to their counterparts defining a universe of datatype in chapter 13 except for two aspects.

First, the constructor marking the presence of a recursive substructure will take an extra (List I) argument specifying the sorts of the newly bound variables that will be present in the subterm.

Second, the type of the meaning function is more complex. It does not translate descriptions into 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 demand for recursive substructures. This first list argument corresponds to the newly bound variables that will be present in the subterm and having it as a separate argument will play a crucial role when defining the description's semantics as a binding structure in figs. 14.1 and 14.3.

Without further ado, let us look at the definition. The declaration of both the universe of descriptions and the meaning function associated to them first.

```
data Desc (I : Set) : Set₁ where

⟦_⟧ : Desc I → (List I → I −Scoped) → I −Scoped
```

The ‘σ is exactly identical to that of the universe of data. It also serves the same purpose: either provide the ability to store a payload, or a tag upon which the shape of the rest of the description will depend.

```
‘σ : (A : Set) → (A → Desc I) → Desc I

⟦ ‘σ A d ⟧ X i Γ = Σ A (λ a → ⟦ d a ⟧ X i Γ)
```

The case of interest is the one for recursive substructures. As explained above, it packs an extra argument specifying the number and sorts of the newly bound variables. For a λ-abstraction, this list would be a singleton, meaning that a single extra variable is in scope in the body of the lambda; for both of the arguments of an application node the list would be empty.

```
‘X : List I → I → Desc I → Desc I

⟦ ‘X Δ j d ⟧ X i Γ = X Δ j Γ × ⟦ d ⟧ X i Γ
```

Finally, the stop case is identical to that of the universe of data. And its role is the same: to enforce that the branch selected by the user has the sort that was demanded.

```
‘■ : I → Desc I

⟦ ‘■ j ⟧ X i Γ = i ≡ j
```

The astute reader may have noticed that ⟦_⟧ is uniform in X and Γ; however refactoring ⟦_⟧ to use the partially applied X _ _ Γ following this observation would lead to a definition harder to use with the combinators for indexed sets described in section 3.5 which make our types much more readable.

A Syntactic Meaning: De Bruijn Scopes

If we pre-compose (using _⊢_ defined in section 3.5) the meaning function ⟦_⟧ with a notion of ‘de Bruijn scopes’ (denoted Scope here) that turns any (I −Scoped) family into a function of type (List I → I −Scoped) by appending the two List indices, we recover a meaning function producing an endofunctor on I −Scoped. It corresponds to the syntactic interpretation of the description we expect when building terms: the newly bound variables are simply used to extend the ambient context.

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

Figure 14.1: 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 13. 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 (a , t) = a , fmap (d a) f t
fmap (‘X Δ j d) f (x , t) = f Δ j x , fmap d f t
fmap (‘■ i)     f eq      = eq
```

Figure 14.2: Action of Syntax Functors on Morphism

14.2 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 14.3: 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 14.4: Type of Closed Terms

Three Small Examples of Syntaxes With Binding

Rather than immediately coming back to our original example which has quite a few constructors, we are going to give codes for 3 variations on the λ-calculus to show the whole spectrum of syntaxes: untyped, well sorted, and well typed by construction. We start with the simplest example to lay down the foundations: the well scoped untyped λ-calculus. We then introduce a well scoped and well sorted but not well typed bidirectional language. We finally consider the code for the intrinsically typed STλC.

In all examples, the variable case will be added by the free monad construction so we only have to describe the other constructors.

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 we have already deployed once for lists and once for vectors: 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 sort.

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

Figure 14.5: Description of The Untyped Lambda Calculus

Bidirectional STLC Our second example is a bidirectional (Pierce and Turner [2000]) language hence the introduction of a notion of Mode. Each term is either part of the Synth (the terms whose type can be synthesised) or Check (the terms whose type can be checked) fraction of the language. This language has four constructors which we list in the ad-hoc ‘Bidi type of constructor tags, its decoding Bidi is defined by a pattern-matching λ-expression in Agda. Application and λ-abstraction behave as expected, with the important observation that λ-abstraction binds a Synthesisable term. The two remaining constructors correspond to changes of direction: one can freely Embbed inferrable terms as checkable ones whereas we require a type annotation when forming a Cut.

```
data Mode : Set where
  Check Synth : Mode

data ‘Bidi : Set where
  App Lam Emb : ‘Bidi
  Cut : Type → ‘Bidi
```

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

Figure 14.6: Description for the bidirectional STLC

Convention 9 (Embed and Cut) *When working with a bidirectional syntax, we systematically call the change of direction from a synthesisable to a checkable term an* ***embedding*** *as no extra information is tacked onto the term.*

In the other direction we use terminology from proof theory because we recognise the connection between ***cut****-elimination and* $\beta\iota$ *reductions. Putting a constructor-headed checkable in a synthesisable position is the first step to creating a redex. Hence the analogy.*

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

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 $(\sigma \text{ `}\!\!\rightarrow \tau)$.

```
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 14.7: Description of the Simply Typed Lambda Calculus

For convenience we use Agda's pattern synonyms corresponding to the original constructors in section 4.2: ‘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. We only include examples of pattern synonyms for the two extreme examples, the definition for Bidi are similar.

As a usage example of these pattern synonyms, we define the identity function in all three languages in Figure 14.9, using the same caret-based naming convention we introduced earlier. The code is virtually the same except for Bidi which explicitly records the change of direction from Check to Synth.

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

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

Figure 14.8: Respective Pattern Synonyms for UTLC and STLC

```
‘id : TM UTLC tt
‘id = ‘lam (‘var z)
```

```
id^B : TM Bidi Check
id^B = ‘lam (‘emb (‘var z))
```

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

Figure 14.9: Identity function in all three languages

Porting our Earlier Running Example

To contrast and compare the two approaches, let us write down side by side the original definition of the intrinsincally typed STLC we introduced in fig. 4.7 and its encoding as a description.

We are going to take this definition apart, considering the different classes of constructors that it introduces and how to adequatly represent them using our universe of description. Our representation will once more rely on a type ‘Term of tags listing the available term constructors together with a description Term using a dependent pair to offer users the ability to pick the constructor of their choosing. That is to say that we are considering the following three definitions.

```
data Term : Type −Scoped where
```

```
data ‘Term : Set where

Term : Desc Type
Term = ‘σ ‘Term $ λ where
```

The most basic of constructors are the one with no subterms. The tag associated to them is a simple constructor and their decoding just uses the stop description constructor (‘■) to enforce that their return type matches the one specified in the original definition.

```
‘one : ∀[ Term ‘Unit ]

‘tt  : ∀[ Term ‘Bool ]
‘ff  : ∀[ Term ‘Bool ]
```

```
One TT FF : ‘Term

One → ‘■ ‘Unit
TT  → ‘■ ‘Bool
FF  → ‘■ ‘Bool
```

Next we have the constructors that have subterms in which no extra variable is bound. Their encoding will use the constructor for recursive substructures and apply

it to the empty list to reflect that fact. Additionally, both ‘app and ‘ifte store some additional information: the type of the subterms at hand. We will push this information into the tags just like we did in the previous section. We once more use the stop constructor to enforce that the constructors' respective return types are faithfully reproduced.

```
‘app : ∀[ Term (σ ‘→ τ)          App : Type → Type → ‘Term
      ⇒ Term σ                   Ifte : Type → ‘Term
      ⇒ Term τ ]
‘ifte : ∀[ Term ‘Bool            (App σ τ) → ‘X [] (σ ‘→ τ) (‘X [] σ (‘■ τ))
      ⇒ Term σ ⇒ Term σ          (Ifte σ)  → ‘X [] ‘Bool (‘X [] σ (‘X [] σ (‘■ σ)))
      ⇒ Term σ ]
```

Finally, we have the λ-abstraction constructor. Its body is defined in a context extended with exactly one newly bound variable whose type is the domain of the overall term's function type. This is modelled by applying the constructor for recursive substructure to a singleton list.

```
‘lam : ∀[ (σ ::_) ⊢ Term τ       Lam : Type → Type → ‘Term
      ⇒ Term (σ ‘→ τ) ]
                                 (Lam σ τ) → ‘X (σ :: []) τ (‘■ (σ ‘→ τ))
```

This concludes the translation. It clearly is a very mechanical process and thus could be automated using reflection (van der Walt and Swierstra [2012], Christiansen [2016]). Such automation is however outside the scope of this thesis.

14.3 Common Combinators and Their Properties

In order to avoid repeatedly re-encoding the same logic, we introduce a combinator demonstrating that descriptions are closed under finite sums.

As we wrote, the construction used in fig. 14.5 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. 13.1 and 13.2) 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 in fig. 14.10.

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

Figure 14.10: Descriptions are Closed Under Disjoint Sums

It comes together with an appropriate eliminator case defined in fig. 14.11 which, given two continuations, picks the one corresponding to the chosen branch.

```
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 14.11: Eliminator for _`+_

A concrete use case for the disjoint union combinator and its eliminator will be given in section 16.4 where we explain how to seamlessly enrich any existing syntax with let-bindings and how to use the Semantics framework to elaborate them away.

Chapter 15

Generic Scope Safe and Well Kinded Programs for Syntaxes

The constraints forming what we called a Semantics in section 5.4 have a shape determined by the specific language at hand: the simply typed λ-calculus. These constraints could be divided in two groups: that arising from our need to push environment values under binders and those in one-to-one correspondence with constructors of the language.

By carefully studying the way in which the language-specific constraints are built, we ought to be able to define a language-agnostic notion of Semantics valid for all of the syntaxes with binding our universe of descriptions can accomodate.

Dissecting the Components of A Semantics

Recall that the language we used as our running example was the simply-typed λ-calculus with a unit and boolean type. The language constructs with the simplest associated Semantics constraints were the constructors for constants in the language. We recall both the type of the constructors and that of their semantical counterparts below.

‘one : ∀[Term ‘Unit]	one : ∀[C ‘Unit]
‘tt : ∀[Term ‘Bool]	tt : ∀[C ‘Bool]
‘ff : ∀[Term ‘Bool]	ff : ∀[C ‘Bool]

From this observation, we can derive our first observation: constants in the language are interpreted as constants in the model. Their type is the same except that Term has been replaced by the notion of computation C the model at hand is using. Next come the language constructs that do have subterms but are not binding any additional variables. We have two such examples: application and boolean conditionals. We recall their types as well as that of their semantical counterparts below.

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

These type tell us that the counterparts for recursive substructures living in the same ambient environment as the overall term should be computations of the appropriate type. Finally comes the constraint attached to the λ-abstraction, showing what the semantical counterpart to a binder should be.

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

These types tell us that the semantical counterpart for inductive substructures with newly bound variables are Kripke function spaces whose domain ensures a model-specific value $\mathcal{V}$ of the appropriate type is provided for each newly bound variable and whose codomain is a computation.

This tight coupling between the definition of the calculus and the structure of the Semantics constraints tells us we will be able to define a generic version of this notion. The fact that these constraints have the same shape as their associated constructor suggests we ought to be able to reuse our interpretation function ⟦_⟧ to compute them.

Defining a Generic Notion of Semantics

Based on these observations, we can define a generic notion of semantics for all syntax descriptions. Any pair of (I–Scoped) families $\mathcal{V}$ and C satisfying the Semantics constraints will once more give rise to an evaluation function. We introduce a notion of computation _–Comp analogous to that defined in fig. 5.16. Just as an environment interprets variables in a model, a computation gives a meaning to terms into a model.

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

Figure 15.1: _–Comp: Associating Computations to Terms

We expect the fundamental lemma of Semantics to be a program turning an environment mapping free variables to values into a mapping from terms to computations.

```
semantics : (Γ –Env) 𝒱 Δ → (Γ –Comp) C Δ
```

Now that the goal is clear, let us spell out the Semantics constraints. As always, they are packed in a record parametrised over the description in question and the two (I–Scoped) families $\mathcal{V}$ and C used to interpret values and computations respectively.

```
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 may 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 13 (fig. 13.3). We should have an algebra taking a term whose substructures have already been evaluated and returning a computation for the overall term. We will need a notion (Kripke $\mathcal{V}$ C) describing how the substructures have been evaluated. We will look at this definition shortly; the important thing to have in mind is that it should behave sensibly both for substructures with and without newly bound variables.

```
  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 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 worth noting that we could do away with the special case for subterms with no newly bound variables and give a uniform definition of Kripke. Indeed when evaluating a term we can always absorb thinnings by mapping them over the environment of values thanks to th^$\mathcal{V}$. However this special case is truer to the observations we made earlier and gives users a definition that is easier to use.

We can now recall 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 15.2: 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.

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

Figure 15.3: Proof of the Fundamental Lemma of Semantics (semantics)

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

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

Figure 15.4: 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. 14.4), we immediately introduce closed, a corollary of the fundamental lemma of semantics for the special cases of closed terms in Figure 15.5. Given a Semantics with value type $\mathcal{V}$ and computation type C, we can evaluate a closed term of type σ and obtain a computation of type ($C\ \sigma$ []) by kickstarting the evaluation with an empty environment.

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

Figure 15.5: Special Case: Fundamental Lemma of Semantics for Closed Terms

15.1 Our First Generic Programs: Renaming and Substitution

Similarly to section 5.5 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, we reuse the proof th^Var introduced in fig. 5.15. 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 5.5 and the definition of reification in the setting of normalisation by evaluation (see e.g. Coquand's work 2002). Reification can be defined generically for var-like values (vl^Var is the proof that variables are var-like); we will make this formal in fig. 15.9.

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

Figure 15.6: 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.

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

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

Figure 15.7: Corollary: Generic Thinning

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.

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


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

Figure 15.8: Generic Parallel Substitution for All Syntaxes with Binding

The reification process mentioned in the definition of renaming and substitution can be implemented generically for the Semantics families satisfying a set of conditions. We introduce in fig. 15.9 the VarLike set of constraints to characterise these families. It captures values that are thinnable and such that we can craft placeholder values in non-empty contexts. It is almost immediate that both Var and Tm are VarLike (with proofs vl^Var and vl^Tm, respectively).

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

Figure 15.9: VarLike: Thinnable and with placeholder values

Given a proof that $\mathcal{V}$ is VarLike, we can manufacture several useful $\mathcal{V}$-environments. We provide users with base of type $(\Gamma$ −Env$)$ $\mathcal{V}$ Γ, fresh$^{\mathsf{r}}$ of type $(\Gamma$ −Env$)$ $\mathcal{V}$ $(\Delta$ ++ $\Gamma)$ and fresh$^{\mathsf{l}}$ of type $(\Gamma$ −Env$)$ $\mathcal{V}$ $(\Gamma$ ++ $\Delta)$ by combining the use of placeholder values and thinnings. In the Var case these very general definitions respectively specialise to the identity renaming for a context Γ and the injection of Γ fresh variables to the

right or the left of an ambient context Δ. Similarly, in the Tm case, we can show (base vl^Tm) extensionally equal to the identity environment id^Tm given by lookup id^Tm = 'var, which associates each variable to itself (seen as a term).

Using these definitions, we can then implement reify as in Figure 15.10 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 _)
```

Figure 15.10: Generic Reification thanks to VarLike Values

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

15.2 Printing with Names

Coming back to our work on (rudimentary) printing with names in section 5.6, 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 accommodate a specific use-case: we do not want our users to deal explicitly with name generation, explicit variable binding, etc.

Unlike renaming or substitution, this generic program will require user guidance: there is no way for us to guess how an encoded term should be printed. We can however take care of the name generation, deal with variable binding, and implement the traversal generically. We are going to reuse some of the components defined in section 5.6: we can rely on the same state monad (Fresh) for name generation, the same fresh function and the same notions of Name and Printer for the semantics' values and computations. We want our printer to have type:

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

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

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

Reusing the notion of Name introduced in Section 5.6, we can make Pieces formal. The structure of Semantics would suggest giving our users an interface where substructures are interpreted as Kripke function spaces expecting fresh names for the fresh variables and returning a printer i.e. a monadic computation returning a String. 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. In other words: Pieces states that a subterm has already been printed if we have a string representation of it together with an environment of Names we have attached to the newly bound variables this structure contains.

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

The key observation that will help us define a generic printer is that Fresh composed with Name is VarLike. Indeed, as the composition of a functor and a trivially thinnable Wrapper, Fresh is Thinnable, and fresh (defined in Figure 5.21) is the proof that we can generate placeholder values thanks to the name supply.

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

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

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

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

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

Otherwise we proceed in a manner reminiscent of the pure reification function defined in Figure 15.10. We start by generating an environment of names for the newly bound variables by using the fact that Fresh composed with Name is VarLike together with the fact that environments are Traversable (McBride and Paterson [2008]), and thus admit the standard Haskell-like mapA and sequenceA traversals. We then run the Kripke function on these names to obtain the string representation of the subterm. We finally return the names we used together with this string.

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

We can put all of these pieces together to obtain the Printing semantics presented in Figure 15.11. The first two constraints can be trivially discharged. When defining the algebra we start by reifying the subterms, then use the fact that one "layer" of term of our syntaxes with binding is always traversable to combine all of these results into a value we can apply our display function to.

This allows us to write a printer for open terms as demonstrated in Figure 15.12. We start by using base (defined in Section 15.1) to generate an environment of Names for the free variables, then use our semantics to get a printer which we can run using a stream names of distinct strings as our name supply.

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

Figure 15.11: Printing with Names as a Semantics

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

Figure 15.12: Generic Printer for Open Terms

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

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

As always, these functions are readily executable and we can check their behaviour by writing tests. First, we print the identity function defined in Figure 14.9 in an empty context and verify that we do obtain the string "λa. a". Next, we print an open term in a context of size two and can immediately observe that names are generated for the free variables first, and then the expression itself is printed.

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

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

15.3 (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 (cf. fig. 15.13) 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 (a user may for instance try to add a function and a number).

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

Figure 15.13: Generic Reflexive Domain

This datatype definition is utterly unsafe. The more conservative user will happily restrict themselves to particular syntaxes where the typed settings allows for domain to be defined as a logical predicate or opt instead for a step-indexed approach. We did develop a step-indexed model construction but it was unusable: we could not get Agda to normalise even the simplest of terms.

But this domain does make it possible to define a generic nbe semantics which, given a term, produces a value in the reflexive domain. Thanks to the fact we have picked a universe of finitary syntaxes, we can *traverse* (McBride and Paterson [2008], Gibbons and d. S. Oliveira [2009]) the functor to define a (potentially failing) reification function turning elements of the reflexive domain into terms. By composing them, we obtain the normalisation function which gives its name to normalisation by evaluation.

The user still has to explicitly pass an interpretation of the various constructors because there is no way for us to know what the binders are supposed to represent: they may stand for λ-abstractions, Σ-types, fixpoints, or anything else.

```
reify^Dm : ∀[ Dm d s σ ⇒ Maybe ∘ Tm d ∞ σ ]
nbe      : Alg d (Dm d ∞) (Dm d ∞) → Semantics d (Dm d ∞) (Dm d ∞)

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

Figure 15.14: Generic Normalisation by Evaluation Framework

Example: Evaluator for the Untyped Lambda-Calculus

Using this setup, we can write a normaliser for the untyped λ-calculus by providing an algebra. The key observation that allows us to implement this algebra is that we can turn a Kripke function, f, mapping values of type σ to computations of type τ into an Agda function from values of type σ to computations of type τ. This is witnessed by the application function (_$$_) defined in Figure 15.15: we first use extract (defined in Figure 5.14) to obtain a function taking environments of values to computations. We then use the combinators defined in Figure 5.4 to manufacture the singleton environment ($\varepsilon \bullet t$) containing the value t of type σ.

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

Figure 15.15: Applying a Kripke Function to an argument

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

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

Figure 15.16: Pattern synonyms for UTLC-specific Dm values

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

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

Figure 15.17: Normalisation by Evaluation for the Untyped λ-Calculus

We have not used the $\bot$ constructor so *if* the evaluation terminates (by disabling totality checking we have lost all guarantees of the sort) we know we will get a term in normal form. See for instance in Figure 15.18 the evaluation of an untyped yet normalising term: (λx. x) ((λx. x) (λx. x)) normalises to (λx. x).

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

Figure 15.18: Example of a normalising untyped term

Chapter 16

Compiler Passes as 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, printing with names or evaluation. All of these fit neatly in the Semantics framework.

Now we wish to focus on the kind of traversals one may find in a compiler pipeline such as a scopechecker, an elaboration function from an untyped surface syntax to an intrinsically typed syntax, or an optimisation pass inlining definitions used at most once.

All of the examples but the scopechecker are instances of Semantics. Some, like the scoping (section 16.1), desugaring (section 16.4), and inlining (section 16.5) passes will be fully generic while others like the typechecking (section 16.2) and elaboration (section 16.3) passes will correspond to a specific language and its particular type system.

16.1 Writing a Generic Scope Checker

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

Parsing is currently out of scope for our library; users can write safe ad-hoc parsers for their object language by either using a library of total parser combinators (Danielsson [2010], Allais [2018]) or invoking a parser generator oracle whose target is a total language (Stump [2016]). As we will see shortly, we can write a generic scope checker transforming terms in a raw syntax where variables are represented as strings into a well scoped syntax. We will come back to typechecking with a concrete example in section 16.2 and then discuss related future work in the conclusion.

Our scope checker will be a function taking two explicit arguments: a name for each variable in scope Γ and a raw term for a syntax description d. It will either fail (the Monad Fail granting us the ability to fail is made explicit in Figure 16.3) or return a well scoped and sorted term for that description.

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

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

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

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

Figure 16.1: Associating a raw string to each variable in scope

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

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

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

Figure 16.2: Names and Raw Terms

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

an error, the string that caused the failure and the extra data of type E that accompanied it.

```
data Error : Set where                    Fail : Set → Set
  OutOfScope : Error                      Fail A = (Error × E × String) ⊎ A
  WrongSort  : (σ τ : I) → σ ≢ τ → Error

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

Figure 16.3: Error Type and Scope Checking Monad

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

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

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

Figure 16.4: Variable Resolution

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

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

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

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

Figure 16.5: Generic Scope Checking for Terms and Scopes

16.2 An Algebraic Approach to Typechecking

Following Atkey (2015), we can consider type checking and type synthesis as a possible semantics for a bidirectional (Pierce and Turner [2000]) language. We reuse the syntax introduced in Section 14.2; it gives us a simply typed bidirectional calculus as a two-moded language using a notion of Mode to distinguish between terms for which we will be able to Synthesise the type and the ones for which we will have to Check a type candidate. We can write Type-, the type-level function computing from a Mode the associated type synthesis or checking behaviour we expect.

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

Figure 16.6: Type- Synthesis / Checking Specification

Our goal in this section is to construct of a Semantics whose associated evaluator will correspond to a function implementing this specification. In other words, we want to define a semantics Typecheck such that we can obtain the following type- function as a corrolary.

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

We will first have to decide what the appropriate notions of values and computations for this semantics should be.

Values and Computations for Type Checking

The values stored in the environment of the typechecking function will attach Type information to bound variables whose Mode is Synth, guaranteeing no variable ever uses the Check mode. Hence the definition of Var- in fig. 16.7.

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

Figure 16.7: Var- Relation indexed by Mode

In contrast, the generated computations will, depending on the mode, either take a type candidate and Check it is valid or Synth a type for their argument. This is the logic represented by the Type- relation defined in fig. 16.6. These computations are always potentially failing as terms may not be well typed. As a consequence we use the Maybe monad. In an actual compiler pipeline we would naturally use a different error monad and generate helpful error messages pointing out where the type error occured. The interested reader can see a fine-grained analysis of type errors in the extended example of a typechecker in McBride and McKinna [2004a].

Handling Type Constraints

A change of direction from synthesising to checking will require being able to check that the type that was synthesised and the type that was required agree. So we introduce the function _=?_ that checks two types for equality. Similarly we will sometimes be handed a type that we expect to be a function type. We will have to check that it is and so we introduce isArrow, a function making sure that our candidate's head constructor is indeed an arrow, and returning the domain and codomain.

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

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

Typechecking as a Semantics

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

Application When facing an application: synthesise the type of the function, make sure it is an arrow type, check the argument at the domain's type and return the codomain.

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

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

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

Embedding of Synth into Check The change of direction from Synthesisable to Checkable is successful when the synthesised type is equal to the expected one.

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

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

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

We have defined a bidirectional typechecker for this simple language by leveraging the Semantics framework and can now obtain the type- function we motivated this work with. Defining β to be (α '→ α), we can synthesise the type of the expression (λx. x : β → β) (λx. x).

```
_ : let β = α '→ α in type- Synth ('app ('cut (β '→ β) id^B) id^B) ≡ just β
_ = refl
```

Figure 16.8: Example of Type- Synthesis

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

16.3 An Algebraic Approach to Elaboration

Instead of generating a type or checking that a candidate will do, we can use our language of Descriptions to define not only an untyped source language but also an intrinsically typed internal language. During typechecking we simultaneously generate an expression's type and a well scoped and well typed term of that type. We use STLC (defined in Section 14.2) as our internal language.

Let us start with a simple example reusing the different definitions of the identity function we introduced in fig. 14.9 (id^B for the identity in our little bidirectional language and id^S for the well scoped and typed one in STLC). We want a function that typechecks id^B against the candidate (α `→ α), succeeds and returns the fully annotated id^S.

```
_ : type- Check id^B (α `→ α) ≡ just id^S
_ = refl
```

Figure 16.9: Example Use of our Elaborator

Before we can jump right in, we need to set the stage: a Semantics for a Bidi term will involve (Mode −Scoped) notions of values and computations but an STLC term is (Type −Scoped). We first introduce a Typing associating types to each of the modes in scope, together with an erasure function ⌞_⌟ extracting the context of types implicitly defined by such a Typing. We will systematically distinguish contexts of modes (typically named *ms*) and their associated typings (typically named Γ). Note that we assume that our contexts of modes only contain Synth variables; but we cannot enforce it here: our framework forces us to have the same sorts for terms and variables. Relaxing this constraint is an interesting avenue for further work. In the meantime, we work within these parameters and only enforce this constraint when defining our Semantics thus making it impossible to evaluate invalid terms.

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

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

Figure 16.10: Typing: From Contexts of Modes to Contexts of Types

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

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

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

Figure 16.11: Elaboration of a Scoped Family

that here we do enforce that variables only have a meaning if they have been rightly marked as Synthesisable.

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

Figure 16.12: Values as Elaboration Functions for Variables

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

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

Figure 16.13: Synthesis Function for the 0-th Variable

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

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

Figure 16.14: 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-notation makes writing the check really easy.

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

Figure 16.15: Informative Equality Check

For the arrow type, we introduce a family Arrow constraining the shape of its index to be an arrow type and redefine isArrow as a *view* targeting this inductive family (Wadler [1987], McBride and McKinna [2004a]). We deliberately overload the constructor of the isArrow family by calling it _'→_. This means that the proof that a given type has the shape (σ '→ τ) is literally written (σ '→ τ). This allows us to specify *in the type* whether we want to work with the full set of values in Type or only the subset corresponding to function types and to then proceed to write the same programs a Haskell programmers would, with the added confidence that ours are guaranteed to be total.

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

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

Figure 16.16: Arrow View

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

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

```
app : ∀[ Elab- Synth ⇒ Elab- Check ⇒ Elab- Synth ]
app f t Γ = do
  (arr , F)  ← f Γ
  (σ '→ τ) ← isArrow arr
  T          ← t Γ σ
  return (τ , 'app F T)
```

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

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

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

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

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

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

We have almost everything we need to define elaboration as a semantics. Discharging the th^$\mathcal{V}$ 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 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.

We can once more define a specialised version of the traversal induced by this Semantics for closed terms: not only can we give a (trivial) initial environment (using the closed corollary defined in Figure 15.5) but we can also give a (trivial) initial Typing. This leads to the definitions in Figure 16.18.

Revisiting the example introduced in Section 16.2, we can check that elaborating the expression (λx. x : $\beta \to \beta$) (λx. x) yields the type β together with the term (λx. x) (λx. x) in internal syntax. Type annotations have disappeared in the internal syntax as all the type invariants are enforced intrinsically.

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

Figure 16.17: Elaborate, the elaboration semantics

```
Type- : Mode → Set
Type- Check = ∀ σ → Maybe (TM STLC σ)
Type- Synth  = Maybe (∃ λ σ → TM STLC σ)


type- : ∀ p → TM Bidi p → Type- p
type- Check t = closed Elaborate t []
type- Synth  t = closed Elaborate t []
```

Figure 16.18: Evidence-producing Type (Checking / Synthesis) Function

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

16.4 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 W. Swiestra (2008). This is made extremely simple by the disjoint sum combinator _‘+_ which we defined in Figure 14.10. 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. In Figure 16.19, we introduce a description Let which can be used to extend any language description d to a language with let-bindings (d ‘+ Let).

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

Figure 16.19: Description of a single let binding, associated pattern synonyms

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 (Coq Development Team [2017]) let bindings are primitive constructs of the language and have their own typing and reduction rules whereas in Agda they are elaborated away to the core language by inlining.

This latter approach to extending a language d with let bindings by inlining them before typechecking can be implemented generically as a semantics over (d ‘+ Let). For this semantics, values in the environment and computations are both let-free terms. The algebra of the semantics can be defined by parts thanks to case, the eliminator for _‘+_ defined in Figure 14.10: the old constructors are kept the same by interpreting them using the generic substitution algebra (Substitution); whilst the let-binder precisely provides the extra value to be added to the environment.

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

Figure 16.20: Desugaring as a Semantics

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

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

Figure 16.21: Specialising semantics with an environment of placeholder values

In less than 10 lines of code we have defined a generic extension of syntaxes with binding together with a semantics which corresponds to an elaborator translating away this new construct. In chapter 7 we had focused on STLC only and showed that it is similarly possible to implement a Continuation Passing Style transformation as the composition of two semantics à la Hatcliff and Danvy (1994). The first semantics embeds STLC into Moggi's Meta-Language (1991a) and thus fixes an evaluation order. The second one translates Moggi's ML back into STLC in terms of explicit continuations with a fixed return type.

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

16.5 Reference Counting and Inlining as a Semantics

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

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

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

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

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

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

During the counting phase we need to be particularly careful not to overestimate the contribution of a let-bound expression. If the let-bound variable is not used then we can naturally safely ignore the associated count. But if it used many times then we know we will not inline this let-binding and the count should therefore only contribute once to the running total. We define the control combinator in Figure 16.26 precisely to explicitly handle this subtle case.

The first step is to introduce the Counter additive monoid (cf. Figure 16.22). Addition will allow us to combine counts coming from different subterms: if any of the two counters is zero then we return the other, otherwise we know we have many occurences.

```
data Counter : Set where
  zero  : Counter
  one   : Counter
  many  : Counter

_+_ : Counter → Counter → Counter
zero + n    = n
m    + zero = m
_    + _    = many
```

Figure 16.22: The (Counter, zero, _+_) additive monoid

The syntax extension CLet defined in Figure 16.23 is a variation on the Let syntax extension of Section 16.4, attaching a Counter to each Let node. The annotation process can then be described as a function computing a (d ‘+ CLet) term from a (d ‘+ Let) one.

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

Figure 16.23: Counted Lets

We keep a tally of the usage information for the variables in scope. This allows us to know which Counter to attach to each Let node. Following the same strategy as in Section 16.1, we use the standard library's All to represent this mapping. We say that a scoped value has been Counted if it is paired with a Count.

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

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

Figure 16.24: Counting i.e. Associating a Counter to each Var in scope.

The two most basic counts are described in Figure 16.25: the empty one is zero everywhere and the one corresponding to a single use of a single variable v which is zero everywhere except for v where it's one.

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

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

Figure 16.25: Zero Count and Count of One for a Specific Variable

When we collect usage information from different subterms, we need to put the various counts together. The combinators in Figure 16.26 allow us to easily do so: merge adds up two counts in a pointwise manner while control uses one Counter to decide whether to erase an existing Count. This is particularly convenient when computing the contribution of a let-bound expression to the total tally: the contribution of the let-bound expression will only matter if the corresponding variable is actually used.

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

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

Figure 16.26: Combinators to Compute Counts

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

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

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

Figure 16.27: Purely Structural Case

The Let-to-CLet case in Figure 16.28 is the most interesting one. We start by reifying the *body* of the let binder which gives us a tally cx for the bound variable and

ct for the body's contribution to the ambient environment's Count. We annotate the node with *cx* and use it as a control to decide whether we are going to merge any of the let-bound's expression contribution *ce* to form the overall tally.

```
clet : ⟦ Let ⟧ (Kripke Var (Counted (Tm (d ‘+ CLet) ∞))) σ Γ →
       Counted (⟦ CLet ⟧ (Scope (Tm (d ‘+ CLet) ∞))) σ Γ
clet (στ , (e , ce) , body , eq) = case body extend (ε • z) of λ where
  (t , cx :: ct) → (cx , στ , e , t , eq) , merge (control cx ce) ct
```

Figure 16.28: Annotating Let Binders

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

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

Figure 16.29: Specialising semantics to obtain an annotation function

Using techniques similar to the ones described in Section 16.4, we can write an Inline semantics working on (*d* ‘+ CLet) terms and producing (*d* ‘+ Let) ones. We make sure to preserve all the let-binders annotated with many and to inline all the other ones. By composing Annotate with Inline we obtain a size-preserving generic optimisation pass.

Chapter 17

Other Programs

All the programs operating on syntax we have seen so far have been instances of our notion of Semantics. This shows that our framework covers a lot of useful cases.

An interesting question raised by this observation is whether there are any interesting traversals that are not captured by this setting. At least two types of programs are clearly not instances of Semantics: those that do not assign a meaning to a term in terms of the meaning of its direct subterms, and those whose return type depends on the input term.

In this chapter we study two such programs. We also demonstrate that although they may not fit the exact pattern we have managed to abstract, it is still sometimes possible to take advantage of our data-generic setting and to implement them once and for all syntaxes with binding.

17.1 Big Step Evaluators

Chapman's thesis (2009) provides us with a good example of a function that does not assign a meaning to a term by only relying on the meaning of its direct subterms: a big step evaluator.

Setting aside the issue of proving such a function terminating, a big step evaluator computes the normal form of a term by first recursively computing the normal forms of its subterms and then either succeeding or, having uncovered a new redex, firing it and then normalising the reduct. This is embodied by the case for application (cf. fig. 17.1).

In contrast, the Semantics constraints enforce that the meaning of each term

$$\frac{f \Downarrow \lambda x.b \qquad t \Downarrow v \qquad b[x \mapsto v] \Downarrow nf}{f\ t \Downarrow nf}$$

Figure 17.1: Big step evaluation of an application

constructor is directly expressed in terms of the meaning of its arguments. There is no way to "restart" the evaluator once the redex has been fired. This means we will not be able to make it an instance of our framework and, consequently, we will not be able to use the tools introduced in the next chapters to reason generically over such functions.

17.2 Generic Decidable Equality for Terms

A program deciding whether two terms are equal is clearly another good example of a function that cannot be written as an instance of our notion of Semantics. In type theory we can do better than an uninformative boolean function claiming that two terms are equal: we can implement a decision procedure for propositional equality (Löh and Magalhães [2011]) which either returns a proof that its two inputs are equal or a proof that they cannot possibly be. Such a decidability proof necessarily mentions its inputs in the type of its output while our framework does not allow these type of dependencies.

This kind of boilerplate function is however something users really do not want to write themselves. Haskell programmers for instance are used to receiving help from the 'deriving' mechanism (Hinze and Peyton Jones [2000], Magalhães et al. [2010]) to automatically generate common traversals for every inductive type they define.

Recalling that generic programming is normal programming over a universe in a dependently typed language (Altenkirch and McBride [2002]), we ought to be able to deliver similar functionalities for syntaxes with binding. The techniques used in this concrete example are general enough that they also apply to the definition of an ordering test, a `Show` instance, etc.

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

```
data ⊥ : Set where
```

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

Figure 17.2: Empty Type and Decidability as an Inductive Family

To get acquainted with these new notions we can start by proving that equality of variables is decidable.

Deciding Variable Equality

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

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

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

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

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

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

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

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

Deciding Term Equality

The constructor ‘σ for descriptions gives us the ability to store values of any Set in terms. For some of these Sets (e.g. ($\mathbb{N} \to \mathbb{N}$)), equality is not decidable. As a consequence our decision procedure will be conditioned to the satisfaction of a certain set of Constraints which we can compute from the Desc itself, as show in Figure 17.3. We demand that we are able to decide equality for all of the Sets mentioned in a description.

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

Figure 17.3: Constraints Necessary for Decidable Equality

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

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

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

Figure 17.4: Type of Decidable Equality for Terms and Bodies

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

Chapter 18

Building Generic Proofs about Generic Programs

We have already shown in chapters 9 and 10 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.

18.1 Additional Relation Transformers

During our exploration of generic proofs about the behaviour of Semantics for a concrete object language, we have introduced a notion Rel of relations as well as a relation transformer for environments (cf. section 9.1). Working on a universe of syntaxes, we are going to need to define additional relators.

Kripke relator The Kripke relator is a generalisation of the ad-hoc definition introduced in fig. 9.4. We assume that we have two types of values $\mathcal{V}^A$ and $\mathcal{V}^B$ as well as a relation $\mathcal{V}^R$ for pairs of such values, and two types of computations C^A and C^B whose notion of relatedness is given by C^R. We can define Kripke^R relating Kripke functions of type $(\mathsf{Kripke}\ \mathcal{V}^A\ C^A)$ and $(\mathsf{Kripke}\ \mathcal{V}^B\ C^B)$ respectively by stating that they send related inputs to related outputs. In this definition we use the relation transformer for environment called All and introduced in fig. 9.2.

Desc relator The relator $(\![\ d\ ]\!]^R)$ is a relation transformer which characterises structurally equal layers such that their substructures are themselves related by the relation it is passed as an argument. It inherits a lot of its relational arguments' properties: whenever R is reflexive (respectively symmetric or transitive) so is $(\![\ d\ ]\!]^R\ R)$.

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

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

Figure 18.1: Relational Kripke Function Spaces: From Related Inputs to Related Outputs

- In the stop token case (‘■ *i*), the two layers are two proofs that the branches' respective indices match the specified one. We consider these two proofs to be trivially equal (i.e. the constraint generated is the unit type). This would not hold true in Homotopy Type Theory (Univalent Foundations Program [2013]) but if we were to generalise the work to that setting, we could either explicitly restrict our setup to language whose indices are equipped with a decidable equality or insist on studying the ways in which these proofs can be equal.
- 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 (⟦ *d* ⟧^{R} *R*)
- Two nodes of type (‘σ *A d*) will be related if they both carry the same payload *a* of type *A* and if the rest of the layers are related by (⟦ *d a* ⟧^{R} *R*)

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

Figure 18.2: Relator: Characterising Structurally Equal Values with Related Substructures

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

18.2 Simulation Lemma

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

$\mathcal{V}^B$ and computations C^B, we want to establish the constraints under which these two semantics yield related computations provided they were called with environments of related values. That is to say we want to prove the following generic result:

```
sim : All 𝒱ᴿ Γ ρᴬ ρᴮ → (t : Tm d s σ Γ) →
      rel Cᴿ σ (𝒮ᴬ.semantics ρᴬ t) (𝒮ᴮ.semantics ρᴮ t)
```

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

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

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

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

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

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

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

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

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

The fundamental lemma of simulations is a generic theorem showing that for each pair of Semantics respecting the Simulation constraint, we get related computations given environments of related input values. In Figure 18.3, 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.

Instantiating this generic simulation lemma, we can for instance prove that renaming is a special case of substitution, or that renaming and substitution are extensional i.e. that given environments equal in a pointwise manner they produce syntactically equal terms. Of course these results are not new but having them generically over all syntaxes

```
sim : All 𝒱ᴿ Γ ρᴬ ρᴮ → (t : Tm d s σ Γ) →
      rel Cᴿ σ (𝒮ᴬ.semantics ρᴬ t) (𝒮ᴮ.semantics ρᴮ t)

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

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

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

Figure 18.3: Fundamental Lemma of Simulations

```
RenSub : Simulation d Ren Sub VarTmᴿ Eqᴿ
```

```
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 18.4: Renaming as a Substitution via Simulation

with binding is convenient. We experienced this first hand when tackling the POPLMark Reloaded challenge (2017) where rensub (defined in Figure 18.4) was actually needed.

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

The Simulation proof framework is the simplest example of the abstract proof frameworks introduced in ACMM (2017a). We also explain how a similar framework can be defined for fusion lemmas and deploy it for the renaming-substitution interactions but also their respective interactions with normalisation by evaluation. Now that we are familiarised with the techniques at hand, we can tackle this more complex example for all syntaxes definable in our framework.

18.3 Fusion Lemma

There are abundant result that can be reformulated as the ability to fuse two traversals obtained as Semantics into one. 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 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.

Our fundamental lemma of Fusion states that from a triple of related environments (the exact meaning of "related" will be defined in the next section), one gets a pair of related computations (again the meaning of "related" will be made formal soon):

```
fusion : 𝓔ᴿ Γ Δ ρᴬ ρᴮ ρᴬᴮ → (t : Tm d s σ Γ) → 𝓡 σ ρᴬ ρᴮ ρᴬᴮ t
```

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 (Γ −Env) $\mathcal{V}^A$ Δ, (Δ −Env) $\mathcal{V}^B$ Θ and (Γ −Env) $\mathcal{V}^{AB}$ Θ 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. Nothing prevents proofs, such as the idempotence of NbE, which use a bona fide reification function that extracts terms from model values.

```
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 (defined in fig. 15.10) to get a Scope back which will be more amenable to running the semantics $\mathcal{S}^B$. 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.

```
quoteᴬ : ∀ Δ i → Kripke 𝒱ᴬ Cᴬ Δ i Γ → Tm d ∞ i (Δ ++ Γ)
quoteᴬ Δ i k = reifyᴬ i (reify vl^𝒱ᴬ Δ i k)
```

Still thinking about going under binders: if three evaluation environments ρ^A of type $(\Gamma \text{ −Env})\ \mathcal{V}^A\ \Delta$, ρ^B in $(\Delta \text{ −Env})\ \mathcal{V}^B\ \Theta$, and ρ^{AB} in $(\Gamma \text{ −Env})\ \mathcal{V}^{AB}\ \Theta$ 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^ℰᴿ : ℰᴿ Γ Δ ρᴬ ρᴮ ρᴬᴮ → (ρ : Thinning Θ Ω) →
        ℰᴿ Γ Δ ρᴬ (th^Env 𝒮ᴮ.th^𝒱 ρᴮ ρ) (th^Env 𝒮ᴬᴮ.th^𝒱 ρᴬᴮ ρ)
```

Remembering that _++^Env_ is used in the definition of body (cf. fig. 15.4) to combine two disjoint environments $(\Gamma \text{ −Env})\ \mathcal{V}\ \Theta$ and $(\Delta \text{ −Env})\ \mathcal{V}\ \Theta$ into one of type $((\Gamma \text{ ++ } \Delta) \text{ −Env})\ \mathcal{V}\ \Theta)$, we mechanically need a constraint stating that _++^Env_ 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 $\text{vl}^\wedge\mathcal{V}^A$.

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

We finally arrive at the constraints focusing on the semantical counterparts of the terms' constructors. In order to have readable types we introduce an auxiliary definition $\mathcal{R}$. Just like in chapter 10, 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) 𝒱ᴬ Δ → (Δ −Env) 𝒱ᴮ Θ → (Γ −Env) 𝒱ᴬᴮ Θ →
    Tm d s σ Γ → Set
ℛ σ ρᴬ ρᴮ ρᴬᴮ t = rel Cᴿ σ (evalᴮ ρᴮ (reifyᴬ σ (evalᴬ ρᴬ t))) (evalᴬᴮ ρᴬᴮ t)
```

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 ($[\![\ d\ ]\!]^R$) relation transformer defined in section 18.1) where the relational Kripke function space relates the semantical objects one can find in place of the subterms.

```
alg^R : ℰ^R Γ Δ ρ^A ρ^B ρ^AB → (b : [[ d ]] (Scope (Tm d s)) σ Γ) →
  let b^A : [[ d ]] (Kripke 𝒱^A 𝒞^A) _ _
      b^A  = fmap d (𝒮^A.body ρ^A) b
      b^B  = fmap d (λ Δ i → 𝒮^B.body ρ^B Δ i ∘ quote^A Δ i) b^A
      b^AB = fmap d (𝒮^AB.body ρ^AB) b
  in [[ d ]]^R (Kripke^R 𝒱^R 𝒞^R) b^B b^AB → ℛ σ ρ^A ρ^B ρ^AB (‘con b)
```

Fundamental Lemma of Fusion

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

```
fusion : ℰ^R Γ Δ ρ^A ρ^B ρ^AB → (t : Tm d s σ Γ) → ℛ σ ρ^A ρ^B ρ^AB t
```

Figure 18.5: 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-related.

```
body : ℰ^R Γ Δ ρ^A ρ^B ρ^AB → ∀ Δ σ → (b : Scope (Tm d s) Δ σ Γ) →
  let v^B  = 𝒮^B.body ρ^B Δ σ (quote^A Δ σ (𝒮^A.body ρ^A Δ σ b))
      v^AB = 𝒮^AB.body ρ^AB Δ σ b
  in Kripke^R 𝒱^R 𝒞^R Δ σ v^B v^AB
```

Figure 18.6: Statement of the Fundamental Lemma of Fusion for Bodies

The proofs involve two functions we have not mentioned before: lift^R maps a proof that a property holds for any recursive substructure over the arguments of said constructor to obtain a [[d]]^R 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.

```
fusion ρ^R (‘var v) = var^R ρ^R v
fusion ρ^R (‘con t) = alg^R ρ^R t (rew (lift^R 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 → ⟦ d ⟧^R (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 ++^Env^R vs^R) b
```

Figure 18.7: Proof of the Fundamental Lemma of Fusion

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).

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 18.8: 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 18.9: Generic Substitution-Substitution Fusion Principle

Variations on these results

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 reproduced this result generically for all syntaxes (see accompanying code). The need for functional

extensionality arises in the proof when dealing with subterms which have extra bound variables. These terms are interpreted as Kripke functional spaces in the host language and we can only prove that they take equal inputs to equal outputs. An intensional notion of equality will simply not do here. As a consequence, we refrain from using the generic result in practice when an axiom-free alternative is provable.

Kaiser, Schäfer and Stark's observation naturally raises the question of whether the same semantics are also stable under substitution. Our semantics implementing printing with names is a clear counter-example: a fresh name is generated each time we go under a binder, meaning that the same term will be printed differently dependening on whether it is located under one or two binders. Consequently, printing u and substituting the result for t during the printing of $(\lambda x.t)(\lambda x.\lambda y.t)$ will lead to a different result than printing the term $(\lambda x.u)(\lambda x.\lambda y.u)$ where the substitution has already been performed.

Chapter 19

Conclusion

19.1 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 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.

19.2 Related Work

Variable Binding

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

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

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

Alternative Binding Structures

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

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

Bach Poulsen, Rouvoet, Tolmach, Krebbers and Visser (2018) introduce notions of scope graphs and frames to scale the techniques typical of well scoped and typed deep embeddings to imperative languages. They showcase the core ideas of their work using STLC extended with references and then demonstrate that they can already handle a large subset of Middleweight Java. We have demonstrated that our framework could be used to define effectful semantics by choosing an appropriate monad stack (Moggi [1991a]). This suggests we should be able to model STLC+Ref. It is however clear that

the scoping structures handled by scope graphs and frames are, in their full generality, out of reach for our framework. In constrast, our work shines by its generality: we define an entire universe of syntaxes and provide users with traversals and lemmas implemented *once and for all*.

Many other opportunities to enrich the notion of binder in our library are highlighted by Cheney (2005).As we have demonstrated in Sections 16.4 and 16.5 we can already handle let-bindings generically for all syntaxes. We are currently considering the modification of our system to handle deeply-nested patterns by removing the constraint that the binders' and variables' sorts are identical. A notion of binding corresponding to hierarchical namespaces would be an exciting addition.

Semantics of Syntaxes with Binding

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

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

Meta-Theory Automation via Tactics and Code Generation

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

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

Kaiser, Schäfer, and Stark (2018) build on our previous paper to draft possible theoretical foundations for Autosubst, a so-far untrusted set of tactics. The paper is based on a specific syntax: well scoped call-by-value System F. In contrast, our effort

has been here to carve out a precise universe of syntaxes with binding and give a systematic account of these syntaxes' semantics and proofs.

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

Universes of Syntaxes with Binding

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

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

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

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

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

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

Fusion of Successive Traversals

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

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

19.3 Limitations of the Current Framework

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

Inefficient Environment Weakening Our fundamental lemma of semantics systematically traverses its environment of values whenever it needs to push it under a binder. This means that if we need to push an environment under n successive binders, we will thin each of the values it carries n times. Preliminary work demonstrates that it is possible to avoid these repeated traversals. The key idea is to use an inductive notion of environments in which the thin-and-extend operation used to go under a binder is reified as a constructor. At variable-lookup time, we merge the accumulated thinnings and actually apply them to the original value. Intuitively, going under a binder amounts to pushing a frame consisting of a thinning and an environment of values for the newly bound variables onto the evaluation stack (which is essentially a type-aligned list of frames).

Closure under Products Our current universe of descriptions is closed under sums as demonstrated in Section 14.3. It is however not closed under products: two arbitrary right-nested products conforming to a description may disagree on the sort of the term they are constructing. An approach where the sort is an input from which the description of allowed constructors is computed (à la Dagand [2013] where, for instance, the ‘lam constructor is only offered if the input sort is a function type) would not suffer from this limitation.

Unrestricted Variables Our current notion of variable can be used to form a term of any kind. We remarked in Sections 16.2 and 16.3 that in some languages we want to restrict this ability to one kind in particular. In that case, we wanted users to only be able to use variables at the kind Infer of our bidirectional language. For the time being we made do by restricting the environment values our Semantics use to a subset of the kinds: terms with variables of the wrong kind will not be given a semantics.

Flat Binding Structure Our current setup limits us to flat binding structures: variable and binder share the same kinds. This prevents us from representing languages with binding patterns, for instance pattern-matching let-binders which can have arbitrarily nested patterns taking pairs apart.

Closure under Derivation One-hole contexts play a major role in the theory of programming languages. Just like the one-hole context of a datatype is a datatype (Abbott et al. [2005b]), we would like our universe to be closed under derivatives so that the formalisation of e.g. evaluation contexts could benefit directly from the existing machinery.

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

19.4 Future 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 research program 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 developed 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 (2005a) 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).

Appendix A

Conventions and Techniques

Convention 1 *When the reader should be able to reconstruct the information from the context, we may write id instead of* id_A *and* $(\_ \circ \_)$ *instead of* $(\_ \circ_C \_)$ *respectively.*

Convention 2 (Caret-based naming) *We use a caret to generate a mnemonic name:* `map` *refers to the fact that we can map a function over the content of a data structure and* `-Tuple` *clarifies that the data structure in question is the family of n-ary tuples.*

We use this convention consistently throughout this thesis, using names such as `map^Rose` *for the proof that we can map a function over the content of a rose tree,* `th^Var` *for the proof that* variables are thinnable, *or* `vl^Tm` *for the proof that terms are var-like.*

Convention 3 (Object and Meta Syntaxes) *When we represent object syntax in the meta language (Agda here) we use backquotes to make explicit the fact that we are manipulating quoted objects.*

Convention 4 (Assume an Ambient Context) *Following Martin-Löf (1982) we adopt the convention that all of our typing rules are defined in an ambient context* Γ *and only mention the* adjustments *made to it. Crucially, we can see that the only rule which modifies the context is the introduction rule for lambda abstractions: in the premise,* $(x : \sigma \vdash)$ *indicates that the ambient context is extended with a new bound variable x of type* σ*.*

Convention 5 (Using Comments to Mimick Natural Deduction Rules) *In the definition of the calculus' syntax we use comment lines to mimick inference rules in natural deduction. This definition style is already present in Pollack's PhD thesis (1994).*

Convention 6 (Programs as lemmas) *It may seem strange for us to call a program manipulating abstract syntax trees a "fundamental lemma". From the Curry-Howard correspondence's point of view, types are statements and programs are proofs so the notions may be used interchangeably.*

We insist on calling this program (and other ones in this thesis) a "fundamental lemma" because it does embody the essence of the abstraction we have just introduced.

Convention 7 (Relational counterparts) *We systematically reuse the same names for a constructor or a combinator and its relational counterpart. We simply append an* ^{R} *suffix to the relational version.*

Convention 8 (Patterns for Layers and Datatypes) *Whenever we work with a specific example in a universe of datatypes or syntaxes, we introduce a set of patterns that work on a single* $[\![\_]\!]$*-defined layer (the primed version of the pattern) and one that works on full* μ*-defined values (the classic version of the pattern). In most cases users will only use the classic version of the pattern to write their programs. But when using generic fold-like function (see the next section) they may be faced with a single layer.*

Convention 9 (Embed and Cut) *When working with a bidirectional syntax, we systematically call the change of direction from a synthesisable to a checkable term an* ***embedding*** *as no extra information is tacked onto the term.*

In the other direction we use terminology from proof theory because we recognise the connection between ***cut****-elimination and* $\beta\iota$ *reductions. Putting a constructor-headed checkable in a synthesisable position is the first step to creating a redex. Hence the analogy.*

Technique 1 (Intrinsically Enforced Properties) *In the definition of an n-ary tuple, the length of the tuple is part of the type (indeed the definition proceeds by induction over this natural number). We say that the property that the tuple has length n is enforced* intrinsically. *Conversely some properties are only proven after the fact, they are called* extrinsic.

The choice of whether a property should be enforced intrinsically or extrinsically is for the programmer to make, trying to find a sweet spot for the datastructure and the task at hand. We typically build basic hygiene intrinsically and prove more complex properties later.

Technique 2 (Extensional Statements) *Note that we have stated the theorem not in terms of the identity function but rather in terms of a function which behaves like it. Agda's notion of equality is intensional, that is to say that we cannot prove that two functions yielding the same results when applied to the same inputs are necessarily equal. In practice this means that stating a theorem in terms of the specific implementation of a function rather than its behaviour limits vastly our ability to use this lemma. Additionally, we can always obtain the more specific statement as a corollary.*

Technique 3 (Functional Induction) *Whenever we need to prove a theorem about a function's behaviour, it is best to proceed by functional induction. That is to say that the proof and the function will follow the same pattern-matching strategy.*

Appendix B

Agda Features

Feature 1 (Universe Levels) *Agda avoids Russell-style paradoxes by introducing a tower of universes* Set_0 *(usually written* Set*),* Set_1*,* Set_2*, etc. Each* Set_n *does not itself have type* Set_n *but rather* Set_{n+1} *thus preventing circularity.*

Feature 2 (Implicit Arguments) *Programmers can mark some of a function's arguments as implicit by wrapping them in curly braces. The values of these implicit arguments can be left out at the function's call sites and Agda will reconstruct them by unification (Abel and Pientka [2011]).*

Feature 3 (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.*

Feature 4 (Coverage Checking) *During typechecking the coverage checker elaborates the given pattern-matching equations into a case tree. It makes sure that all the branches are either assigned a right-hand side or obviously impossible. This allows users to focus on the cases of interest, letting the machine check the other ones.*

Feature 5 (Mixfix Identifiers) *We use Agda's mixfix operator notation (Danielsson and Norell [2011]) where underscores denote argument positions. This empowers us to define the* `if_then_else_` *construct rather than relying on it being built in like in so many other languages. For another example, at the type-level this time, see e.g. the notation* `_×_` *for the type of pairs defined in section 3.2.*

Feature 6 (Termination Checking) *Agda is equipped with a sophisticated termination checking algorithm (Abel [1998]). For each function it attempts to produce a well-founded lexicographic ordering of its inputs such that each recursive call is performed on smaller inputs according to this order.*

Feature 7 (Underscore as Placeholder Name) *An underscore used in place of an identifier in a binder means that the binding should be discarded. For instance (*λ `_` $\rightarrow$ *a) defines a constant function. Toplevel bindings can similarly be discarded which is a*

convenient way of writing unit tests (in type theory programs can be run at typechecking time) without polluting the namespace.

Feature 8 (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. In the above example, A and B are introduced using this mechanism.*

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, 2005a.

M. G. Abbott, T. Altenkirch, C. McBride, and N. Ghani. δ for data: Differentiating data structures. *Fundam. Inform.*, 65(1-2):1--28, 2005b. URL `http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-02`.

A. Abel. foetus -- termination checker for simple functional programs. Technical report, 1998.

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 and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In C. L. Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 10--26. Springer, 2011. ISBN 978-3-642-21690-9. doi: 10.1007/978-3-642-21691-6_5. URL `https://doi.org/10.1007/978-3-642-21691-6_5`.

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, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark. Poplmark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, dec 2019.

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. Generic level polymorphic n-ary functions. In D. Darais and J. Gibbons, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019*, pages 14--26. ACM, 2019. ISBN 978-1-4503-6815-5. doi: 10.1145/3331554.3342604. URL https://doi.org/10.1145/3331554.3342604.

G. Allais, C. McBride, and P. Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In S. Weirich, editor, *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, pages 13--24. ACM, 2013a. ISBN 978-1-4503-2384-0. doi: 10.1145/2502409.2502411. URL https://doi.org/10.1145/2502409.2502411.

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, 2013b. 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 C. McBride. Generic programming within dependently typed programming. In J. Gibbons and J. Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany*, volume 243 of *IFIP Conference Proceedings*, pages 1--20. Kluwer, 2002. ISBN 1-4020-7374-7.

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

T. Altenkirch and T. Uustalu. Normalization by evaluation for lambda-2. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, volume 2998 of *Lecture Notes in Computer Science*, pages 260--275. Springer, 2004. ISBN 3-540-21402-X. doi: 10.1007/978-3-540-24754-8_19. URL https://doi.org/10.1007/978-3-540-24754-8_19.

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.

T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. *J. Funct. Program.*, 25, 2015. doi: 10.1017/S095679681500009X. URL https://doi.org/10.1017/S095679681500009X.

A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *J. Funct. Program.*, 7(5):515--540, 1997. URL http://journals.cambridge.org/action/displayAbstract?aid=44115.

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

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMark Challenge. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50--65. Springer, 2005a. ISBN 3-540-28372-2. doi: 10.1007/11541868_4. URL https://doi.org/10.1007/11541868_4.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, pages 50--65. Springer, 2005b. ISBN 978-3-540-31820-0.

C. Bach Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*,

2(POPL):16:1--16:34, Jan. 2018. ISSN 2475-1421. doi: 10.1145/3158104. URL http://doi.acm.org/10.1145/3158104.

H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

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.

M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362--377. Springer, 2011. ISBN 978-3-642-25378-2. doi: 10.1007/978-3-642-25379-9_26. URL https://doi.org/10.1007/978-3-642-25379-9_26.

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. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363--408, Oct 2012. ISSN 1573-0670. doi: 10.1007/s10817-011-9225-2. URL https://doi.org/10.1007/s10817-011-9225-2.

J. Cheney. Toward a general theory of names: binding and scope. In R. Pollack, editor, *ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2005, Tallinn, Estonia, September 30, 2005*, pages 33--40. ACM, 2005. doi: 10.1145/1088454.1088459. URL https://doi.org/10.1145/1088454.1088459.

A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143--156. ACM, 2008a. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411226. URL https://doi.org/10.1145/1411204.1411226.

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

D. Christiansen. *Practical Reflection and Metaprogramming for Dependent Types*. PhD thesis, Denmark, 2016.

E. Copello. *On the Formalisation of the Metatheory of the Lambda Calculus and Languages with Binders*. PhD thesis, Universidad de la República (Uruguay), 2017.

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

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.

C. T. Cortiñas and W. Swierstra. From algebra to abstract machine: a verified generic construction. In R. A. Eisenberg and N. Vazou, editors, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018*, pages 78--90. ACM, 2018. doi: 10.1145/3240719.3241787. URL https://doi.org/10.1145/3240719.3241787.

P. Dagand. *A cosmology of datatypes : reusability and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013. URL http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713.

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. Total parser combinators. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 285--296. ACM, 2010. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863585. URL https://doi.org/10.1145/1863543.1863585.

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.

J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 281--292. ACM, 2004. ISBN 1-58113-729-X. doi: 10.1145/964001.964025. URL http://doi.acm.org/10.1145/964001.964025.

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.

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

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding (extended abstract). In *Proc. 14th LICS Conf.*, pages 193--202. IEEE, Computer Society Press, 1999.

M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax with Variable Binding. 13(3--5):341--363, July 2001. doi: 10.1007/s001650200016.

N. Ghani. $\beta\eta$-equality for coproducts. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*, pages 171--185. Springer, 1995. ISBN 3-540-59048-X. doi: 10.1007/BFb0014052. URL https://doi.org/10.1007/BFb0014052.

J. Gibbons and B. C. d. S. Oliveira. The essence of the Iterator pattern. *J. Funct. Program.*, 19(3-4):377--402, 2009. doi: 10.1017/S0956796809007291. URL https://doi.org/10.1017/S0956796809007291.

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.

B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In M. Wand and S. L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 235--246. ACM, 2002. ISBN 1-58113-487-8. doi: 10.1145/581478.581501. URL https://doi.org/10.1145/581478.581501.

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.

M. Hedberg. A coherence theorem for Martin-Löf's type theory. *J. Funct. Program.*, 8(4):413--436, 1998. URL http://journals.cambridge.org/action/displayAbstract?aid=44199.

R. Hinze and S. L. Peyton Jones. Derivable type classes. *Electr. Notes Theor. Comput. Sci.*, 41(1):5--35, 2000. doi: 10.1016/S1571-0661(05)80542-0. URL https://doi.org/10.1016/S1571-0661(05)80542-0.

M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 208--212. IEEE Computer Society, 1994. ISBN 0-8186-6310-3. doi: 10.1109/LICS.1994.316071. URL https://doi.org/10.1109/LICS.1994.316071.

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.

P. Jander. Verifying type-and-scope safe program transformations. Master's thesis, University of Edinburgh, 2019.

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

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. Keuchel. Generic programming with binders and scope. Master's thesis, Utrecht University, 2011.

S. Keuchel and J. Jeuring. Generic conversions of abstract syntax representations. In A. Löh and R. Garcia, editors, *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2012, Copenhagen, Denmark, September 9-15, 2012*, pages 57--68. ACM, 2012. ISBN 978-1-4503-1576-0. doi: 10.1145/2364394.2364403. URL https://doi.org/10.1145/2364394.2364403.

S. Keuchel, S. Weirich, and T. Schrijvers. Needle & Knot: Binder boilerplate tied up. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 419--445. Springer-Verlag New York, Inc., 2016. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_17. URL http://dx.doi.org/10.1007/978-3-662-49498-1_17.

G. Lee, B. C. D. S. Oliveira, S. Cho, and K. Yi. GMeta: A generic formal metatheory framework for first-order representations. In H. Seidl, editor, *Programming Languages and Systems*, pages 436--455. Springer, 2012. ISBN 978-3-642-28869-2.

S. Lindley. Extensional rewriting with sums. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 255--271. Springer, 2007. ISBN 978-3-540-73227-3. doi: 10.1007/978-3-540-73228-0_19. URL https://doi.org/10.1007/978-3-540-73228-0_19.

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

A. Löh and J. P. Magalhães. Generic programming with indexed functors. In J. Järvi and S. Mu, editors, *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 1--12. ACM, 2011. ISBN 978-1-4503-0861-8. doi: 10.1145/2036918.2036920. URL https://doi.org/10.1145/2036918.2036920.

J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In J. Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 37--48. ACM, 2010. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863529. URL https://doi.org/10.1145/1863523.1863529.

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. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

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. *J. Funct. Program.*, 14(1): 69--111, 2004a. doi: 10.1017/S0956796803004829. URL https://doi.org/10.1017/S0956796803004829.

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

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.

G. Mendel-Gleason. *Types and verification for infinite state systems*. PhD thesis, Dublin City University (IE), 2012.

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. *Inf. Comput.*, 93(1):55--92, 1991a. doi: 10.1016/0890-5401(91)90052-4. URL https://doi.org/10.1016/0890-5401(91)90052-4.

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

P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *Types for Proofs and Programs*, pages 252--267. Springer, 2006. ISBN 978-3-540-31429-5.

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

S. L. Peyton 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, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 636--666. Springer, 1991. ISBN 3-540-54396-1. doi: 10.1007/3540543961_30. URL https://doi.org/10.1007/3540543961_30.

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

B. C. Pierce. *Basic category theory for computer scientists*. Foundations of computing. MIT Press, 1991. ISBN 978-0-262-66071-6.

B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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

R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

E. Polonowski. Automatically generated infrastructure for de Bruijn syntaxes. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 402--417. Springer, 2013. ISBN 978-3-642-39634-2.

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

A. Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016. ISBN 978-1-97000-127-3.

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

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

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.

T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

P. van der Walt and W. Swierstra. Engineering proof by reflection in agda. In R. Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 157--173. Springer, 2012. ISBN 978-3-642-41581-4. doi: 10.1007/978-3-642-41582-1_10. URL https://doi.org/10.1007/978-3-642-41582-1_10.

P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 307--313. ACM Press, 1987. ISBN 0-89791-215-2. doi: 10.1145/41625.41653. URL https://doi.org/10.1145/41625.41653.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231--248, 1990a. doi: 10.1016/0304-3975(90)90147-A. URL https://doi.org/10.1016/0304-3975(90)90147-A.

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

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

S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 333--345. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034818. URL https://doi.org/10.1145/2034773.2034818.

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.