Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

COSTRUZIONI MODULARI IN LINGUAGGI CON TIPI DIPENDENTI

Tesi di Laurea Magistrale

Relatore:
Dott. Claudio Sacerdoti Coen

Correlatore:
Dott. Lorenzo Malatesta

Presentata da:
Matteo Acerbi

Sessione III Anno Accademico 2012/2013

Sintesi

In questa tesi si indaga come è possibile strutturare in modo modulare programmi e prove in linguaggi con tipi dipendenti. Il lavoro è sviluppato nel linguaggio di programmazione con tipi dipendenti Agda.

Il fine è quello di tradurre l'approccio *Datatypes à la carte* [34], originariamente formulato per Haskell, in Type Theory: puntiamo ad ottenere un simile *embedding* di una nozione di sottotipaggio per tipi ricorsivi, che permetta sia la definizione di programmi con side-effect dove i diversi effetti sono definiti modularmente, che la modularizzazione di sintassi, semantica e ragionamento relativi a descrizioni di linguaggi.

Nella soluzione Haskell di Wouter Swierstra, il sottotipaggio [32] è ottenuto richiedendo coercizioni parametriche, inferite attraverso il meccanismo delle type class. Nella nostra codifica Agda, la relazione di sottotipaggio è definita come una famiglia induttiva relativamente a ben note costruzioni di universi di funtori [8, 10]. La disponibilità di questi codici di sottotipaggio permette di definire costruzioni generiche per induzione, mentre gli *instance arguments* [14] sono utilizzati per recuperare un supporto di base alla loro inferenza automatica. Questo metodo non richiede estensioni al linguaggio sottostante e permette di modularizzare anche *prove* che procedono per eliminazione dipendente.

Dimostriamo i seguenti esempi di applicazione:

- **Effetti modulari sicuri**

  Si mostra come è possible fare uso della monade libera indicizzata, assieme alla codifica del sottotipaggio di cui sopra, per describere effetti modulari. Come nel paper di Swierstra, definiamo le *effect signature* `Teletype` e `FileSystem`, ma raffiniamo il secondo tramite indicizzazione ([25, 10]), in modo tale da permettere solamente la costruzione di programmi che obbediscono ad un particolare protocollo ``sicuro'' rispetto al filesystem. Portando l'esempio originale di Swierstra (UNIX `cat`) nel nostro framework, mostriamo come assicurare proprietà di sicurezza ``lightweight'' tramite la disciplina di indicizzazione sia un approccio compatibile con la modularizzazione degli effetti.

- **Compilazione modulare e verificata**

  Si definisce un tipo di *compilatori modulari verificati* tra due *linguaggi*, cioè codici per tipi di induttivi. La nostra soluzione permette una definizione modulare della funzione di traduzione, della semantica del linguaggio sorgente e della prova di correttezza. Dimostriamo come è possibile instanziare il tipo astratto con compilatori verificati per frammenti di semplici linguaggi di espressioni.

Nella conclusione confrontiamo il nostro approccio con ricerche sviluppate contemporaneamente da altri autori, sintetizziamo quali consideriamo le contribuzioni principali fornite da questo lavoro e indichiamo diverse possibili direzioni di ricerca ulteriore.

Abstract

In this thesis we investigate how programs and proofs in a language with dependent types can be structured in modular style. Our development is carried out in the dependently typed programming language Agda.

The goal is to translate the *Datatypes à la carte* approach [34] from Haskell to Type Theory: we aim at obtaining a similar embedding of a notion of *subtyping* for recursive datatypes, allowing both the definition of side-effecting programs where effects are modularly defined, and the modularisation of syntax, semantics and reasoning on language descriptions.

In Wouter Swierstra's Haskell solution, subtyping [32] is mimicked by requiring parametric coercions via the type class mechanism. In our Agda encoding, the subtyping relation is defined as an inductive family and can be reasoned about in type theory, on top of well-known universe constructions [8, 10]. The availability of these subtyping codes allows for generic constructions to be defined by induction, while *instance arguments* [14] are employed to recover a minimal support for their automatic inference. The method does not require extensions to the underlying language and allows *proofs* by dependent elimination to be modularised as well.

We provide the following example applications of the above technique:

- **Modular safe effects**

  We show how to make use of the (indexed) free monad construction together with our encoding of subtyping in order to describe modular effects. We define `Teletype` and `FileSystem` effect signatures as in Swierstra's paper, but we refine the latter by indexing ([25, 10]) in order to only allow the construction of programs that obey a ``safe'' protocol with respect to the filesystem. By porting Swierstra's original example (UNIX `cat`) to our framework, we demonstrate how ensuring lightweight behavioural correctness properties via the indexing discipline is compatible with the modularisation of effects.

- **Modular and verified compilation**

  We define a type of *modular verified compilers* between two *languages*: in our setting, these are just codes for inductive types. Our solution allows for a modular definition of the translation function, source semantics and the correctness proof. We show how to instantiate the abstract type with verified compilers for fragments of simple expression languages.

In the last chapter we compare our approach with similar research that was concurrently developed by other authors, we summarise what we consider the main contributions provided by this work and point to several possible directions of further research.

Contents

1 Introduction

1.1 The Expression Problem

This main objective of this thesis is that of finding solutions to the *expression problem* [35] in dependently typed languages. Languages based on dependent type theory usually offer the possibility to introduce new types via `data` definitions satisfying a particular syntactic schema [15], similar to that of the *algebraic data types* of more mainstream functional programming languages such as Haskell or ML. These are, essentially, isorecursive variant types [32]: the user can provide a finite number of (possibly recursive) constructors in a given datatype definition, which will be the only permitted ways of introducing terms of the types being defined.

For what concerns functional programming languages, the expression problem consists in the lack of *extensibility* of datatype definitions which follow these schemas, and in particular in the consequent lack of *reusability* for functions defined over a given datatype, when an extension of such a datatype with more *constructors* is considered.

We can define, for example, a type for a language of arithmetical expressions:

```
data Exp : Set where
  [_] :         ℕ → Exp
  _⊕_ : Exp → Exp → Exp
```

`Exp` is a language of numerals closed under a binary sum operator, corresponding to the grammar:

$$Exp \quad ::= \quad \mathbb{N} \mid Exp \oplus Exp$$

Let us now consider a function that computes the semantics of these expressions, as a natural number:

```
⟦_⟧Exp : Exp → ℕ
⟦ [ n ]   ⟧Exp = n
⟦ e1 ⊕ e2 ⟧Exp = ⟦ e1 ⟧Exp + ⟦ e2 ⟧Exp
```

For some reason, we might successively need to extend this syntax with a new binary operator *times* ⊗:

$$Exp \quad ::= \quad \mathbb{N} \mid Exp \oplus Exp \mid Exp \otimes Exp$$

If we want to keep the original datatype definition `Exp` and the semantics function `⟦_⟧Exp` unmodified (they might, for instance, be part of a separate module upon which other code depends), we necessarily need to define a completely new datatype `Exp'`:

```
data Exp' : Set where
  [_] :          ℕ → Exp'
  _⊕_ : Exp' → Exp' → Exp'
  _⊗_ : Exp' → Exp' → Exp'
```

While the language we are using (Agda [30]) allows to overload the constructor identifiers `[_]` and `_⊕_`, it is not possible to define `Exp'` as an *extension* of `Exp`: the constructors for `Exp` are totally disjoint from those for `Exp'`, which are simply two different types.

The *Expression Problem* consists, in particular, in the impossibility of reusing the code for `⟦_⟧Exp` when defining the extended semantic function `⟦_⟧Exp'`. While it is now possible to describe how to interpret the new construct(or) `_⊗_`, the algorithm relative to the `_⊕_` and `[_]` cases needs to be reimplemented:

```
⟦_⟧Exp' : Exp' → ℕ
⟦ [ n ]   ⟧Exp' = n
⟦ e1 ⊕ e2 ⟧Exp' = ⟦ e1 ⟧Exp' + ⟦ e2 ⟧Exp'
⟦ e1 ⊗ e2 ⟧Exp' = ⟦ e1 ⟧Exp' * ⟦ e2 ⟧Exp'
```

The first two clauses are identical to those in the definition of `⟦_⟧Exp`: however, there is no way to recycle that implementation because types would not match.

It is therefore impossibile to combine separately implemented *modules*, containing definitions related to fragments of a syntax, to build a single module with the functions for the union of these fragments: the non-extensibility of the syntactic schema for datatype definitions makes modular developments impossible.

1.2 Encoding modular inductive types

In the following chapters we will see a possible approach to the expression problem in dependent type theory. We will not try to extend the rules of type theory with syntactic support akin to *polymorphic variants* [18], which would require major changes to the implementation and formal metatheoretical justification. We will show, instead, how to *encode* a class of datatypes which can be freely composed inside the Agda system, demonstrating how this construction enables the reusability and modularity that usual `data` definitions inhibit. We will restrict to encoding modular *inductive* definitions, leaving coinduction (which is still evolving in Agda itself) for later work.

The categorical semantics of inductive types inspired, in functional programming languages with sufficiently expressive type systems such as Haskell, an approach to datatype definitions based on type-level fixpoint operators [28], where the definition of recursive functions is obtained by means of combinators (e.g., *fold*) which implement fixed *recursion schemes*. For example, we can replace the definition of an inductive type such as `Exp` with that of its *base* $Set \to Set$ *functor*:

```
data ExpF (X : Set) : Set where
  [_] :     ℕ → ExpF X
  _⊕_ : X → X → ExpF X
```

The extension `_⊗_` can be described as another functor:

```
data TimesF (X : Set) : Set where
  _⊗_ : X → X → TimesF X
```

Given a generic fixpoint operator `μ`,

```
data μ (F : Set → Set) : Set where
  In : F (μ F) → μ F
```

a type isomorphic to `Exp` can be obtained as follows:

```
Exp = μ ExpF
```

Note that we can form the *coproduct* of any pair of operators:

```
_:+:_ : (Set → Set) → (Set → Set) → Set → Set
(F :+: G) X = F X + G X
```

Extensibility of `Exp` to `Exp'` is a consequence of the fact that the coproduct of functors `ExpF` and `TimesF` gives a functor which is isomorphic to the base functor of `Exp'`:

```
Exp' = μ (ExpF :+: TimesF)
```

Now, if the operator `F` admits a covariant map function, we can build a generic *fold* or *catamorphism* `cata`, *folding* an algebra `F X → X` with a generic type `X` as carrier over the input `μ F`, to obtain an output of type `X`:

```
cata : ∀ {F}⦃ map : ∀ {A B} → (A → B) → F A → F B ⦄{X} → (F X → X) → μ F → X
cata ⦃ map ⦄ α (In xs) = α (map (cata α) xs)
```

Notice that the `map` function in the above definition is required as an *instance argument* [14], which are a special type of implicit argument, notated with bold curly braces **{}**, whose inference is performed via a search in the current environment: if a unique inhabitant of the required type - here, `∀ {A B} → (A → B) → F A → F B` - is found, that term is implicitly applied to the occurrence of the function (in this case, `cata`).

We can check that the `ExpF` and `TimesF`, being functors, are covariant, and that a coproduct of two covariant operators is covariant (implementations are hidden for brevity):

```
mapExpF   : ∀ {A B} → (A → B) → ExpF   A → ExpF   B
mapTimesF : ∀ {A B} → (A → B) → TimesF A → TimesF B
map+      : ∀ {F G} → (∀ {A B} → (A → B) → F A → F B)
                    → (∀ {A B} → (A → B) → G A → G B)
                    →  ∀ {A B} → (A → B) → (F :+: G) A → (F :+: G) B
```

If we describe the first evaluation function as an `ExpF`-algebra with carrier `ℕ`,

```
algExpF : ExpF ℕ → ℕ
algExpF [ n ]   = n
algExpF (l ⊕ r) = l + r
```

it is possible to implement `⟦_⟧Exp` as a catamorphism:

```
⟦_⟧Exp : Exp → ℕ
⟦_⟧Exp = cata algExpF
```

Note that for any two functors `F` and `G`, it is possible to take the *coproduct* (`+.[_,_]`) of an `F`-algebra and a `G`-algebra which share the same carrier `X`, constructing a `(F :+: G)`-algebra; intuitively, this describes a recursive program which behaves as the `F`-algebra on the left-injected `F`-shaped data, and as the `G`-algebra on the right-injected `G`-shaped data.

We might now want to obtain an equivalent to `⟦_⟧Exp'`: as opposed to the monolithic case where we had to reimplement the logic for the numerals and sum case on their own, here we can reuse the `algExpF` implementation. We simply need to take the coproduct of `algExpF` with an algebra `algTimesF` encoding the semantics for the `_⊗_` extension[1]:

```
algTimesF : TimesF ℕ → ℕ
algTimesF (l ⊗ r) = l * r

⟦_⟧Exp' : Exp' → ℕ
⟦_⟧Exp' = let _ = map+ mapExpF mapTimesF in
          cata +.[ algExpF , algTimesF ]
```

[1]Note how we had to hint the *instance arguments* mechanism [14] to provide the right instance for `cata`: inference of instance arguments in Agda is limited to lookup in the current typing context, which makes it much less powerful than the typeclass mechanisms of languages like Haskell, Idris and Coq.

The *Datatypes à la carte* paper by Wouter Swierstra [34] builds on this idea to show how, in the context of Haskell, quantification over parametric transformations - i.e., functions of type `∀ X. F X → G X` for some functors `F` and `G` - can be used to simulate *bounded quantification*. For example, the following code defines constructors for all the *superfunctor* of a functor:

```
[[_]] : ∀ {F}⦃ inj : ∀ {X} → ExpF X → F X ⦄ → ℕ → μ F
[[_]] ⦃ inj ⦄ n = In (inj [ n ])

_[⊕]_ : ∀ {F}⦃ inj : ∀ {X} → ExpF X → F X ⦄ → μ F → μ F → μ F
_[⊕]_ ⦃ inj ⦄ e1 e2 = In (inj (e1 ⊕ e2))

_[⊗]_ : ∀ {F}⦃ inj : ∀ {X} → TimesF X → F X ⦄ → μ F → μ F → μ F
_[⊗]_ ⦃ inj ⦄ e1 e2 = In (inj (e1 ⊗ e2))
```

Swierstra calls these functions *smart constructors*. As they can be flexibly adapted to any *superfunctor*, we can easily build terms in the composed language:

```
example : Exp'
example = ([[ 1 ]] [⊕] [[ 2 ]]) [⊗] [[ 3 ]]
```

This, however, requires some help to the inference of instance arguments:

```
 where
  a : ∀ {X} → ExpF   X → (ExpF :+: TimesF) X ; a = inl
  b : ∀ {X} → TimesF X → (ExpF :+: TimesF) X ; b = inr
```

In the original Haskell solution inference of these *injections* is taken care by the type class mechanism (see the definition of the class `:<:` in Swierstra's paper), where predefined - but *open* - instances are provided in order to help inference.

1.3 From inductive types to inductive families

We have seen how to approach the expression problem for inductive types: a very useful feature of dependent type theories, however, is that they support the more expressive schema of inductive *families* [15], which are families of types (i.e., types of type `I → Set`, where `I` is a type of *indices*) that are inductively defined. This schema is one of the cornerstones of *dependently typed programming*: it allows to represent the rules of inductively defined relations as constructors of `data` definitions.

A prominent application of inductive families is the direct encoding of well-typed syntax, which is obtained by indexing terms of an embedded language by their type; to continue our expression-related example, given an index type `I` containing codes for natural numbers and booleans,

```
data I : Set where
 nat bool : I
```

and a semantic interpretation for `I`,

```
Ty : I → Set
Ty nat  = ℕ
Ty bool = Bool
```

we can define a (by construction) well-typed `Exp`, which we rename `A` (for arithmetic), as follows:

```
data A : I → Set where
 [_] : ∀ {i} → Ty i  → A i
 _⊕_ : A nat → A nat → A nat
```

Semantics can be given as an indexed function:

```
⟦_⟧A : ∀ {i} → A i → Ty i
⟦ [ x ]   ⟧A = x
⟦ e1 ⊕ e2 ⟧A = ⟦ e1 ⟧A + ⟦ e2 ⟧A
```

We could, in this case, consider the problem of extending `A` to `A+L` (arithmetic + logic), adding `=?` (equality test) and `if_then_else_` constructs to the expression language:

```
data A+L : I → Set where
 [_]       : ∀ {i}  → Ty i → A+L i
 _⊕_       : A+L nat → A+L nat → A+L nat
 _=?_      : ∀ {i} → A+L i → A+L i → A+L bool
 if_th_el_ : ∀ {i} → A+L bool → A+L i → A+L i → A+L i
```

By dependent pattern matching, defining a total semantics for this family as an indexed function is straightforward:

```
⟦_⟧A+L : ∀ {i} → A+L i → Ty i
⟦ [ x ]            ⟧A+L = x
⟦ e1 ⊕ e2          ⟧A+L = ⟦ e1 ⟧A+L + ⟦ e2 ⟧A+L
⟦ e1 =? e2         ⟧A+L = ⟦ e1 ⟧A+L ≟  ⟦ e2 ⟧A+L
⟦ if b th e1 el e2 ⟧A+L = if ⟦ b ⟧A+L then ⟦ e1 ⟧A+L else ⟦ e2 ⟧A+L
```

If our syntax had been untyped, the previous definition would have been cluttered with typechecking tests: using inductive families saved us a lot of work, also reducing the opportunities for introducing mistakes. But how to make inductive families *modular*, then? We simply need to switch from endofunctors over Set to endofunctors over Set^I, which in type-theoretical terms are functions of type `(I → Set) → (I → Set)`. We describe these functions as inductive families

```
data AF (X : I → Set) : I → Set where
 [_] : ∀ {i} → Ty i  → AF X i
 _⊕_ : X nat → X nat → AF X nat

data LF (X : I → Set) : I → Set where
 _=?_      : ∀ {i} → X i → X i → LF X bool
 if_th_el_ : ∀ {i} → X bool → X i → X i → LF X i
```

and take their *indexed* least fixpoint:

```
data μ (F : (I → Set) → I → Set)(i : I) : Set where
  In : F (μ F) i → μ F i
```

It is now easy to implement the algebras corresponding to ⟦_⟧A and the extension to the `if_th_el_` case:

```
algAF : ∀ {i} → AF Ty i → Ty i
algAF   [ x ] = x
algAF (l ⊕ r) = l + r

algLF : ∀ {i} → LF Ty i → Ty i
algLF (l =? r        ) = l ≟ r
algLF (if x th l el r) = if x then l else r
```

The reader is invited to check that functor coproduct and the generic fold can be recovered for the indexed inductive families case: ⟦_⟧A and ⟦_⟧A+L can be modularly defined as we saw for the unindexed case.

Unfortunately, we must now stop and consider the problem that makes the `μ` fixpoints we have been defining until now *unsafe*.

1.4 Problem: unrestricted fixpoints are unsafe

A well-known restriction of the schema of inductive families [15] is that a syntactic condition, known as *strict positivity*, must be met by all definitions.

This corresponds to the fact that in the types of constructors, the recursive occurrence of the datatype being defined can never occur to the left of a (possibly dependent) arrow: this rules out datatypes that would correspond to fixpoints of functors with negative occurrences (e.g., $FX = X \to X$), or even of covariant functors with *doubly negated* occurrences (for example, $GX = (X \to 2) \to 2$), which still do not admit a set-theoretical model. In the case of G, for example, the existence of a fixpoint X would imply X to be isomorphic to 2^{2^X}, which is false by cardinality arguments.

For these reasons, all `data` definitions must pass a strict-positivity syntactic check that makes sure that datatypes are valid inductive definitions: omitting this test could also break the normalisation property of the calculus. Note that the syntactic test is conservative (i.e., it does not aim at completeness), and the various dependently typed systems (Agda, Coq, Matita, Idris, etc.) implement it in different ways: one of the reasons why we chose Agda for this work is that its strict positivity check is particularly tolerant and would support our constructions and related experimentations with no ad-hoc modification. [2]

[2] It should also be noted that it is not easy to verify that the implementation of these checks are correct, and in the past bugs have been found that allowed to define unsafe datatypes, possibly leading to non-termination: the definitions we will be using, however, rely on the reflection of strict positivity *itself* in syntax, hence are uncontroversially considered safe in the related literature.

To typecheck the code in this chapter, we need to pass the `--no-positivity-check` and `--no-termination-check` options to the Agda program: however, if this is done in a formalised development, no proof can be believed to be correct. In order to proceed with the following chapters, we need to find a solution to this problem.

1.5 Universes of strictly-positive functors

A standard solution to the problem of representing datatypes as (generic) fixpoints of functors is to not define the functors directly, but simply encode a type of codes for a class of strictly-positive functors, together with a semantic interpretation function that actually builds the required operators: such constructions are called *universes*.

For example we might restrict to the following simple universe of functors `(U , El)`, with codes for the identity `` `I `` and constant unit `` `1 `` functors, and closed under binary coproduct and product:

```
data U : Set where
 `I   : U
 `1   : U
 _`+_ : U → U → U
 _`×_ : U → U → U
```

The semantic interpretation is straightforward:

```
El : U → Set → Set
El `I       X = X
El `1       X = ⊤
El (L `+ R) X = El L X + El R X
El (L `× R) X = El L X × El R X
```

The following fixpoint operator is *safe*, and it is accepted without disabling any check:

```
data μ (F : U) : Set where
 In : El F (μ F) → μ F
```

The same approach can be used for inductive families: in this case, a type of codes with an interpretation as *indexed* functors is required.

Universes of strictly-positive indexed functors have been studied in depth under many aspects: well-known constructions for the literature are Indexed Containers [3], Dependent Polynomials [17], Dybjer and Setzer's Inductive-Recursive codes [16], Indexed Inductive Definitions [7] and Strictly Positive Families [29]. The works [7] and [29], in particular, stress how universe constructions allow *generic programming* in dependent type theories to be completely first-class, and not really distinguishable from usual programming.

In chapters 2 and 4 we will see how a universe similar to those cited above can be adopted to encode inductive families in a very intuitive way [8]; in chapter

4 we will build an encoding of subtyping for these codes. On top of this framework, in the last two chapters of the thesis we will build examples of modular constructions.

2 Codes for strictly-positive functors

As discussed at the end of the last chapter, in *safe* Agda we cannot take fixpoints of arbitrary operators: we must restrict to endofunctors which are strictly-positive, and in order to satisfy this syntactic condition we must reflect it into a syntax for operators which Agda recognizes as strictly-positive.

In this chapter we show how to capture a large class of strictly-positive functors with a *universe* construction which is standard in the literature [8]: first we define a parametrised type of codes `De`, then we show how to assign a functorial semantic interpretation `⟦_⟧` to it. On top of this semantics, we define some generic constructions that will prove useful in the following chapters.

2.1 Syntax

The codes we adopt are called *indexed descriptions* [8, 10]. For convenience, we shorten the original name of the type, `IDesc`, to `De`, and rename `` `X `` to `` `I ``. The type `De` is obtained as a `data` definition, parameterised by a type of indices `I`:

```
data De (I : Set lI) : Set (S lI) where
```

- identity

  ```
  `I : (i : I) → De I
  ```

- dependent sum and product

  ```
  `Σ `Π : (S : Set lI)(T : S → De I) → De I
  ```

- constant unit

  ```
  `1   : De I
  ```

- binary product

  ```
  _`×_ : De I → De I → De I
  ```

We will also often use the following derived code, for constant functors with value a given type `S`:

```
`K : ∀ {I} → Set lI → De I
`K S = `Σ S λ _ → `1
```

For example, if we wanted to define a functor that denotes the `ExpF` datatype from the introduction (see 1.2), we would use a code such as the following:

```
`ValF `AddF `ExpF : De ⊤
`ValF = `K (^ _ ℕ)
`AddF = `I _ `× `I _
`ExpF = `Σ Two +.[ κ `ValF , κ `AddF ]
```

2.2 Semantics

In this section we will consider a functorial semantics for the given type of codes `De I`.

2.2.1 Function extensionality

Function extensionality is the principle that allows to prove that two functions are equal from a proof that they are *extensionally* so:

```
Ext : ∀ l1 l2 → Set _
Ext l1 l2 = {X : Set l1}{Y : Set^ X l2}{f g : (x : X) → Y x} → (∀ x → f x ≡ g x) → f ≡ g
```

In some cases `Ext` will be required in order to prove equalities on a generic `D` of type `De I`: usually, however, we will only need extensionality when `D` contains a `` `Π `` code. Therefore, we compute a more precise statement `ExtFor D` to be used as a weaker assumption for those lemmas:

```
ExtFor : ∀ {I} → De I → ∀ lE → Set _
ExtFor (`I k  ) _  = ⊤
ExtFor (`Σ S T) lE = Π S λ s → ExtFor (T s) lE
ExtFor (`Π S T) lE = Ext lI lE
ExtFor (`1    ) _  = ⊤
ExtFor (L `× R) lE = ExtFor L lE × ExtFor R lE
```

With `extFor`, instead, we simply weaken a proof of extensionality to an `ExtFor D`, for a given description `D`.

```
extFor : ∀ {I lE} → Ext lI lE → (D : De I) → ExtFor D lE
extFor e (`I k  ) = tt
extFor e (`Σ S T) = λ s → extFor e (T s)
extFor e (`Π S T) = e
extFor e (`1    ) = tt
extFor e (L `× R) = extFor e L , extFor e R
```

2.2.2 $Set^I \to Set$ functors

A `De I` can be interpreted as a $Set^I \to Set$ operator:

```
⟦_⟧ : ∀ {I} → De I → ∀ {lX} → Set^ I lX → Set (lX ⊔ lI)
⟦ `I i   ⟧ X = ^ lI (X i)
⟦ `Σ S T ⟧ X = Σ S λ s → ⟦ T s ⟧ X
⟦ `Π S T ⟧ X = Π S λ s → ⟦ T s ⟧ X
⟦ `1     ⟧ _ = ⊤
⟦ L `× R ⟧ X = ⟦ L ⟧ X × ⟦ R ⟧ X
```

Notice that the identity code is actually interpreted by a universe-level lifting wrapper in Agda, which does not have a *cumulative hierarchy* of universes, this seems the only available option to make the semantic function *universe polymorphic*.

Back to our example, we can obtain the functor `ExpF`, isomorphic to the one shown in the introduction, by applying the `⟦_⟧` to the `` `ExtF`` code:

```
ExpF : Set lI → Set _
ExpF X = ⟦ `ExpF ⟧ λ _ → X
```

The two constructors `[_]` and `_⊕_`

```
 [_] : ∀ {X} → ℕ → ExpF X
 _⊕_ : ∀ {X} → X → X → ExpF X
```

can now be implemented as follows:

```
 [ n ] = inl _ , ↑ n , _
 l ⊕ r = inr _ , ↑ l , ↑ r
```

To simplify presentation, we choose to not refer to the `` `1`` and ``_`×_`` cases in the following proofs. While in an intensional theory such as the one implemented in Agda it proves useful to have those codes, ideally, in a theory with function extensionality one could adopt the following equivalent[1] encoding:

```
 ′1 : De I
 ′1 = `Π ⊥ ⊥-elim

 _′×_ : De I → De I → De I
 L ′× R = `Π Two +.[ κ L , κ R ]
```

The following translation functions show why we consider `` `1`` and ``_`×_`` redundant:

[1]The equivalence concerns propositional equality, as, for definitional equality to be decidable, and for eliminators to compute as expected, this is not a possibility even in the presence of function extensionality [24].

```
`1→´1 : ⟦ `1 ⟧ X → ⟦ ´1 ⟧ X
`1→´1 _ = ⊥-elim

´1→`1 : ⟦ ´1 ⟧ X → ⟦ `1 ⟧ X
´1→`1 _ = _

`×→´× : ∀ {L R} → ⟦ L `× R ⟧ X → ⟦ L ´× R ⟧ X
`×→´× (l , r) = +.[ κ l , κ r ]

´×→`× : ∀ {L R} → ⟦ L ´× R ⟧ X → ⟦ L `× R ⟧ X
´×→`× p = p (inl _) , p (inr _)
```

We leave to the reader to verify that, in a theory with function extensionality, the above logical equivalences are (natural) isomorphisms: the class of functors captured by `De` via the interpretation function `⟦_⟧` is the same as the one captured by the {`` `I ``, `` `Σ ``, `` `Π ``} fragment.

For this reason, we prefer to hide the pattern-matching clauses for `` `1 `` and ``_`×_`` from the text of the following definitions: the interested reader can find the missing code in the Agda source files.

2.2.3 The generic `map`

To convince ourselves that `De I` really encodes a family of functors, we proceed by computing the generic `map`, taking morphisms in Set^I, i.e. indexed functions (`X ⇒ Y ≡ ∀ i → F i → G i`), to morphisms in Set:

```
⟦_⟧map : (D : De I) → X ⇒ Y → ⟦ D ⟧ X → ⟦ D ⟧ Y
⟦ `I k   ⟧map f (↑ x)   = ↑ (f k x)
⟦ `Σ S T ⟧map f (s , t) =   s , ⟦ T s ⟧map f t
⟦ `Π S T ⟧map f g       = λ s → ⟦ T s ⟧map f (g s)
```

Instantiating `⟦_⟧map` to our running example, one has:

```
mapExpF : ∀ {X Y} → (X → Y) → ExpF X → ExpF Y
mapExpF f = ⟦ `ExpF ⟧map (κ f)
```

With similar termination-checked recursive definitions we can prove that the interpretation function `⟦_⟧` is actually a functor.

First, we need to check that the identity morphism is preserved:

```
⟦_,_⟧map-id⇒ : (D : De I)(eD : ExtFor D l) → ⟦ D ⟧map (id⇒ {X = X}) Π≡ id
⟦ `I n   , _   ⟧map-id⇒ (↑ x)   = <>
⟦ `Σ S T , eT  ⟧map-id⇒ (s , t) = ,_ $≡ ⟦ T s , eT s ⟧map-id⇒ t
⟦ `Π S T , e   ⟧map-id⇒ f       = e λ s → ⟦ T s , extFor e (T s) ⟧map-id⇒ (f s)
```

Second, we prove that also morphism composition is preserved:

```
⟦_,_⟧map-∘⇒ : (D : De I)(e : ExtFor D l) →
              ⟦ D ⟧map f ∘ ⟦ D ⟧map g Π≡ ⟦ D ⟧map (f ∘⇒ g)
⟦ `I n   , _   ⟧map-∘⇒ (↑ x)   = <>
⟦ `Σ S T , eT  ⟧map-∘⇒ (s , t) = ,_ $≡ ⟦ T s , eT s ⟧map-∘⇒ t
⟦ `Π S T , e   ⟧map-∘⇒ f       = e λ s → ⟦ T s , extFor e (T s) ⟧map-∘⇒ (f s)
```

In the above proofs extensionality was only needed in the `` `Π `` case, motivating the use of `ExtFor` and `extFor`.

2.2.4 Functors and predicates

Given a description `D : De I` and an *indexed type* `X : Set^ I`, we will often be interested in reasoning about properties concerning functorial values `xs : ⟦ D ⟧ X`.

We need therefore to understand how to *transform* [20] a predicate of type `Set^Σ X ≡ Set^ (Σ I X)` to a predicate of type `Set^ (⟦ D ⟧ X)`: alternatively, we could describe this operation as lifting [21] the functor (code) `D` from the category of (indexed) types to that of predicates over them. Again, we implement this operation by induction over the syntax of the codes.

```
□ : (D : De I) → Set^Σ X lP → Set^ (⟦ D ⟧ X) l
□ (`I k)   P (↑ x)   = ^ (S lI) (P (, x))
□ (`Σ S T) P (s , t) = □ (T s) P t
□ (`Π S T) P f       = Π S λ s → □ (T s) P (f s)
```

The `□` notation (from [29]) is inspired by the *box* operator of *modal logic*. Other names which are used in the literature are `All` in [8] and `Everywhere` in [10]. The intuitive meaning is that `□ D P xs` *holds* when `P : Set^Σ X` holds on every elements of `X` occurring in `xs : ⟦ D ⟧ X`.

One should not forget, however, that we are discussing *relevant* predicates: a proof that a predicate holds may carry information of computational interest. This relevance is clearly preserved by `□`.

A useful property of `□` is that `⟦ D ⟧ (Σ X P)` and `Σ (⟦ D ⟧ X) (□ D P)` are logically equivalent.

```
to : (D : De I) → ⟦ D ⟧ (Σ/ X P) → Σ (⟦ D ⟧ X) (□ D P)
to (`I k)   (↑ (x , p)) = ↑ x , ↑ p
to (`Σ S T) (s , t)     = (s , fst rec) , snd rec
                          where rec = to (T s) t
to (`Π S T) f           = (fst ∘ rec)   , snd ∘ rec
                          where rec = λ s → to (T s) (f s)
```

```
fr : (D : De I) → Σ (⟦ D ⟧ X) (□ D P) → ⟦ D ⟧ (Σ/ X P)
fr (`I k  ) (↑ x , ↑ ih)   = ↑ (x , ih)
fr (`Σ S T) ((s , t) , ih) = s , fr (T s) (t , ih)
fr (`Π S T) (f , ih)       = λ s → fr (T s) (f s , ih s)
```

The equivalence is *morally* an isomorphism, but merely assuming function extensionality does not seem sufficient to prove that `to ∘ fr` is the identity in the `` `Π `` case [2]. However, we will not need the roundtrip properties in the following.

As a consequence we can derive the following function, which returns a functorial value from a proof of a non-dependent predicate `Y` (i.e., an indexed type in `Set^ I`):

[2]See the sources for the hidden details.

```
□→⟦⟧ : (D : De I) → ∀ i → □ {X = X} D (Y ∘ fst) i → ⟦ D ⟧ Y
□→⟦⟧ D = cu (⟦ D ⟧map snd/ ∘ fr _ D)
```

Notice that the predicate lifting of a strictly-positive functor is witnessed by a generic *proof transformer*:

```
▫ : (D : De I) → Π _ P → Π _ (□ D P)
▫ (`I i  ) f (↑ xs ) = ↑ f (, xs)
▫ (`Σ S T) f (s , t) = ▫ (T s) f t
▫ (`Π S T) f g       = λ s → ▫ (T s) f (g s)
```

2.2.5 $Set^O \to Set^N$ functors

The target category of a `De I`-encoded functor is Set: to encode a $Set^O \to Set^N$ functor, we notice that

$$(Set^O \to Set^N) \simeq (N \to Set^O \to Set)$$

Hence a N-indexed family of `De O` codes [3]

```
_▻_ = λ (O N : Set lI) → N → De O
```

can be used to implement a functor of type $Set^O \to Set^N$:

```
⟦_⟧ : ∀ {O N} → (F : O ▻ N) → ∀ {lX} → Set^ O lX → Set^ N _
⟦ F ⟧ X n = ⟦ F n ⟧ X
```

We can now encode endofunctors over Set^I, whose type of codes we refer to as `En I`:

```
En = λ I → I ▻ I
```

Here is, for example, a code for the the `AF` functor from 1.3:

```
`AF : En I
`AF i = `Σ Two +.[ κ (`K (Ty i)) , κ (`K (i == nat) `× `I nat `× `I nat) ]

AF : Set^ I lI → Set^ I lI
AF = ⟦ `AF ⟧
```

The constructors for `AF` can be implemented as follows:

```
[_] : ∀ {X i} → Ty i → AF X i
[ x ] = inl _ , x , _

_⊕_ : ∀ {X} → X nat → X nat → AF X nat
l ⊕ r = inr _ , (refl , _) , ↑ l , ↑ r
```

The previous generic constructions carry over from `De` to `▻` codes:

[3]In the Agda models of [10] `O ▻ N` is called `func O N`.

```
⟦_⟧map : (F : O ▻ N) → X ⇒ Y → ⟦ F ⟧ X ⇒ ⟦ F ⟧ Y
⟦ F ⟧map f i = ⟦ F i ⟧map f

□/ : (F : O ▻ N) → Set^Σ X lP → Set^Σ (⟦ F ⟧ X) _
□/ F X (n , xs) = □ (F n) X xs

ExtFor/ = λ (F : O ▻ N) l → ∀ i → ExtFor (F i) l
```

2.2.6 Algebras

We define the type of *algebras* for an `En I`-encoded endofunctor as follows:

```
_alg>_ : (F : En I) → ∀ {lY} → Set^ I lY → _
F alg> Y = ⟦ F ⟧ Y ⇒ Y

_-ALG_ : En I → ∀ lY → Set _
F -ALG lY = Σ (Set^ _ lY) (_alg>_ F)
```

Most of the time, we will use the relation `_alg>_` instead of the predicate `_-ALG`.

2.2.7 Natural transformations

We now define a type of natural transformations between functors captured by the universe of descriptions: we will refer back to this definition in paragraph 4.2.2.

We first define the family of parametric transformations between `O ▻ N`-encoded $Set^O \to Set^N$ functors:

```
_pt[_]>_ : (F : O ▻ N) → ∀ l (G : O ▻ N) → Set _
F pt[ l ]> G = (X : Set^ O l) → ⟦ F ⟧ X ⇒ ⟦ G ⟧ X
```

Naturality of a parametric transformation is captured by the following predicate:

```
isNat : (F G : O ▻ N) → ∀ {l} → F pt[ l ]> G → Set _
isNat F G α = ∀ {A B}(f : A ⇒ B) → α B ∘⇒ ⟦ F ⟧map f ⇒≡ ⟦ G ⟧map f ∘⇒ α _
```

Natural transformations are defined by refinement as those parametric transformations which are natural:

```
_nt[_]>_ : ∀ (F : O ▻ N) l (G : N → De O) → Set _
F nt[ l ]> G = Σ (F pt[ l ]> G) (isNat F G)
```

3 Initial algebra

In this chapter, for every functor corresponding to a code F as defined in the last chapter, we define an inductive family `μ F`, corresponding to a fixpoint for the functor `⟦ F ⟧`; we will see that this is indeed the initial algebra for those codes, and that it is possible to define a generic elimination principle.

From a more operational point of view, in this chapter we switch from the non-recursive functors captured by descriptions, to actual inductive datatypes. Informally, we will sometimes refer to inhabitants of our inductive types as *trees*: indeed, in Martin-Löf's Type Theory inhabitants of the W-type, a type our `μ` can be reduced to [2, 10], are conventionally called *wellfounded trees*.

The content of this chapter is standard in the literature about the universe of descriptions [8, 10]: our only contribution is the proof of initiality.

3.1 A fixpoint for *descriptions*

For the purpose of our development, the most important property of the universe discussed in the previous chapter is the fact that it consists of *strictly positive* functors. In particular, it is recognized by the checker that all operators in `⟦_⟧`'s range are strictly positive: this allows to take a *least fixpoint* as a `data` definition

```
data μ (F : En I)(i : I) : Set lI where
  ⟨_⟩ : ⟦ F ⟧ (μ F) i → μ F i
```

As usual we indicate with `In` the *algebra* corresponding to `⟨_⟩`:

```
In : {F : En I} → F alg> μ F
In _ = ⟨_⟩
```

For example, we can construct the least fixpoint for the code `` `AF`` from last chapter, obtaining an inductive family isomorphic to `A` from 1.3:

```
A : Set^ I _
A = μ `AF
```

Here, `[_]` and `_⊕_` are the constructors for the inductive family `A`:

```
[_] : ∀ {i} → Ty i → A i
[ x ] = ⟨ inl _ , x , _ ⟩

_⊕_ : A nat → A nat → A nat
l ⊕ r = ⟨ inr _ , (refl , _) , ↑ l , ↑ r ⟩
```

3.2 Initiality

We will now prove that, for any description `F`, the object `(μ F , In)` is initial in the category of `⟦ F ⟧`-algebras.

First, we need to define the generic *catamorphism* to an `⟦ F ⟧`-algebra `(Y , α)`.

```
module Cata {F : En I}{lY}{Y : Set^ I lY}(α : F alg> Y) where
```

As a first attempt we could try to define

```
  cata : μ F ⇛ Y
  cata i ⟨ xs ⟩ = α i (⟦ F ⟧map cata i xs)
```

but, unfortunately, this definition would not pass termination-checking.

We need to resort to structural induction over the code `F` code to define a function which, while being extensionally equivalent to `⟦ F ⟧map cata`, is also accepted as total:

```
 mapCata  : (D : De I)(xs : ⟦ D ⟧ (μ F)) → ⟦ D ⟧ Y
 mapCata (`I k)   (↑ ⟨ xs ⟩) = ↑ (α k (mapCata (F k) xs))
 mapCata (`Σ S T) (s , t)    = , mapCata (T s) t
 mapCata (`Π S T)  f         = λ s → mapCata (T s) (f s)
```

We can then obtain the catamorphism by evaluating `mapCata` at the identity code:

```
 cata : μ F ⇛ Y
 cata i x = ↓ mapCata (`I i) (↑ x)
```

Assuming `ExtFor/ F` (2.2.5) we can prove, by induction over the codes, that any algebra homomorphism `k` from `In` to `α` is equal to the catamorphism:

```
 module Init (eF : ExtFor/ F _)
             (k : μ F ⇛ Y)
             (h : α ∘⇛ ⟦ F ⟧map k ⇛≡ k ∘⇛ In) where

  mapInit : (D : De I)(eD : ExtFor D (lY ⊔ lI)) → mapCata D Π≡ ⟦ D ⟧map k
  mapInit (`I j  ) _   (↑ ⟨ xs ⟩) = ↑_ $≡ (α j $≡ mapInit (F j) (eF j) xs ⊚ h (, xs))
  mapInit (`Σ S T) eT  (s , t)    = _,_ s $≡ mapInit (T s) (eT s) t
  mapInit (`Π S T) e   f          = e λ s → mapInit (T s) (extFor e (T s)) (f s)


  init : cata ⇛≡ k
  init (i , ⟨ xs ⟩) = α i $≡ mapInit (F i) (eF i) xs ⊚ h (i , xs)
```

In a theory with function extensionality, *initiality* is enough to obtain a `cata`-based *dependent* eliminator: this, in turn, suffices for all the structurally-recursive dependently typed definitions for the inductive types captured by `μ` [19, 9].

However, in an intensional type theory such as Agda the function extensionality hypothesis cannot be witnessed by a canonical object. While this could be acceptable for *irrelevant* proofs, defining functions by appealing to abstract assumptions destroys their computational properties. If functions are *stuck* at typechecking time it is often impossible to prove properties about them: we must therefore find an alternative way to obtain dependent elimination for the `μ` family.

3.3 Dependent elimination

In order to define a *generic* dependent eliminator, we first define the types of the generic target predicate `Mot`, also known as *motive* [26], followed by that of the generic induction step `_me>_`, or (elimination) *method* [26]. The latter is based on the predicate lifting operator defined in 2.2.4: here it serves the purpose of capturing the generic *induction hypothesis*.

```
Mot = λ F lP → Set^Σ (μ F) lP

_me>_ : ∀ (F : En I){lP} → Mot F lP → Set (lP ⊔ S lI)
F me> P = □/ F P ⇛ P ∘ Σ.< fst , ⟨_⟩ ∘ snd >

_-ME_ : En I → ∀ lP → Set _
F -ME lP = Σ (Mot F lP) (_me>_ F)
```

A method `F -ME` can *intuitively* be considered the dependently-typed version of an algebra `F -ALG`. We will usually refer to `Mot` and `_me>_` individually.

We now proceed to define the generic eliminator by induction over the code. As for the catamorphism, we would like to define `elim` directly, in this case by applying `□` - playing the role of a *dependent* `map` - to `elim` itself:

```
module Elim (F : En I){lP}(P : Mot F lP)(m : F me> P) where

  elim : Π _ P
  elim (i , ⟨ xs ⟩) = m (, xs) (□ (F i) elim xs)
```

Similarly to what happened for the catamorphism in 3.2, Agda's termination checker rejects this definition.

The alternative, again, consists in giving an auxiliary definition `□Elim` by structural induction over the code. In this case, `□Elim` inhabits the lifting of the description `D` applied to the predicate `P`:

```
 □Elim  : (D : De I)(xs : ⟦ D ⟧ (μ F)) → □ D P xs
 □Elim (`I k  ) (↑ ⟨ xs ⟩) = ↑ (m (, xs) (□Elim (F k) xs))
 □Elim (`Σ S T) (s , t)    = □Elim (T s) t
 □Elim (`Π S T) f          = λ s → □Elim (T s) (f s)
```

By post-composing the method - which, in this case, is also applied to the actual input `xs` - we obtain the eliminator.

```
elim : Π (Σ I (μ F)) P
elim (i , ⟨ xs ⟩) = m (, xs) (□Elim (F i) xs)
```

Back to our running example, we can check that the generic eliminator `elim` can be instantiated to give an induction principle for `A`:

```
elimA : (P : {i : I} → A i → Set) →
        (m[] : {i : I}(x : Ty i) → P {i} [ x ]) →
        (m⊕  : {l r : A nat} → P l → P r → P (l ⊕ r)) →
        {i : I}(x : A i) → P x
elimA P m[] m⊕ x = Elim.elim `AF M m (, x) where
  M : Mot `AF _
  M (i , a) = P {i} a
  m : `AF me> M
  m (_    , inl _ , (x    ,  _    ))               _  = m[] x
  m (.nat , inr _ , (refl ,  _) , _) (_ , ↑ hl , ↑ hr) = m⊕ hl hr
```

The type of `elimA` confirms that `[_]` and `_⊕_` can be correctly regarded as inductive constructors for `A`: indeed, it is enough to supply the corresponding methods `m[]` and `m⊕` to prove that the target predicate `P` holds on every inhabitant of `A`.

3.3.1 `cata` from `elim`

The equivalence in section 2.2.4 allows the embedding of algebras in methods:

```
alg→me : F alg> Y → F me> (Y ∘ fst)
alg→me α (i , xs) = α i ∘ □→⟦⟧ (F i) xs
```

We can then redefine the catamorphism in terms of the eliminator:

```
cata' : F alg> Y → μ F ⇒ Y
cata' = cu ∘ Elim.elim F _ ∘ alg→me
```

The interested reader can easily prove `cata'` to be pointwise equal to `cata` by induction over the code.

In a finitely-axiomatised theory, i.e. a system whose set of rules is finite and inductive types are obtained from a distinguished type former (for example, W in Martin-Lof's type theory), one usually considers a single elimination principle for each type former: in Agda, however, we can simply rely on termination checking and adopt several independent definitions. In the following we choose to use both `cata` and `elim` in order to obtain more readable proofs.

3.4 Example: natural numbers

As a simple example we can show how to encode the inductive type of natural numbers:

```
data NatTags : Set where `zero `suc : NatTags

NatF : En ⊤
NatF _ = `Σ NatTags λ { `zero → `1 ; `suc → `I _ }

Nat : Set
Nat = μ NatF _
```

The constructors can be recovered as follows:

```
zero : Nat
zero = ⟨ `zero , tt ⟩

suc : Nat → Nat
suc n = ⟨ `suc , ↑ n ⟩
```

We can now try to define functions over natural numbers by structural recursion. A common example is binary addition `_+_`, which can be captured by an algebra with carrier `Nat → Nat`:

```
+alg : NatF alg> κ (Nat → Nat)
+alg _ (`zero , _) = id
+alg _ (`suc  , m) = suc ∘ ↓ m
```

To obtain the required function, we apply the catamorphism:

```
_+_ : Nat → Nat → Nat
_+_ = cata +alg _
```

We can rely on the typechecker to test this program on example inputs:

```
2+1≡3 : suc (suc zero) + suc zero ≡ suc (suc (suc zero))
2+1≡3 = <>
```

Most importantly, specialising the generic eliminator to work on the type `Nat` by simple application to the `NatF` code, we can inhabit any dependent predicate over `Nat` which is provable by induction. For example, here we prove that the above defined `_+_` is *associative*:

```
+-associative : ∀ m n o → (m + n) + o ≡ m + (n + o)
+-associative = elim NatF P me ∘ ,_
 where P  : ⊤ × Nat → Set
       P (_ , m) = ∀ n o → (m + n) + o ≡ m + (n + o)
       me : NatF me> P
       me (α , `zero , xs) ih n o = <>
       me (α , `suc  , xs) ih n o = suc $≡ ↓_ ih n o
```

With similar applications of the generic eliminator, it is not difficult to prove `Nat` isomorphic to the standard library's type `Data.Nat.ℕ`.

4 Encoding subtyping

An inductive family definition can be seen as a finite coproduct or sum of functors. Indeed, as we have seen in the introduction, for any subset of the *constructors* of an inductive definition we can generate the corresponding base functor, and obtain the family as the initial algebra of the *coproduct* of these functors. In addition, in Agda and similar systems the constructor is introduced as a constant with a distinguished name or *label*.

In this chapter we will see how it is possible to capture labelled finite sums of descriptions by defining an ad-hoc syntax for them, with an interpretation as standard descriptions: on top of these new codes, we will define inductively a subtyping relation similar to that defined by Swierstra in [34] using typeclasses, together with some generic tools that will be useful in the following chapters.

4.1 Labelled finite sums of codes

The representation of labelled finite sums of codes we will be using can be captured by a simple grammar,

$$\begin{aligned} \blacktriangleright &::= A \,,\, \triangleright \\ \triangleright &::= C \mid \blacktriangleright \oplus \blacktriangleright \end{aligned}$$

where A is the type of names and C is the type of codes.

The corresponding inductive family is the following:

```
mutual

 record _[_]►_ (O : Set lI)(A : Set lA)(N : Set lI) : Set (S lI ⊔ lA) where
  constructor _⟩_
  field
    name : 1+ A
    code : O [ A ]▻ N

 data _[_]▻_ (O : Set lI)(A : Set lA)(N : Set lI) : Set (S lI ⊔ lA) where
   [_] : (F   :   N → De O) → O [ A ]▻ N
   _⊕_ : (L R : O [ A ]► N) → O [ A ]▻ N
```

We will assign semantics to the above language by translation to the underlying codes. A simple way to translate these `A`-labelled finite sums codes `O [ A ]► N` back to `O ▻ N` is the following:

```
 _`$_ : {A : Set lA}{O N : Set lI} → O [ A ]► N → O ▻ N
 (_ ⟩ [ F ]  ) `$ n = F n
 (_ ⟩ (L ⊕ R)) `$ n = `Σ (name L +∋ name R) +.[ (λ _ → L `$ n) , (λ _ → R `$ n) ]
```

The output is a code for a (possibly empty) right-nested dependent tuple of *boolean* values, ending with an actual `O ▻ N` code. The *bits* denote the path from the root of the input binary tree to the code: we use binary sums of *contractible* manifest types instead of actual booleans because we want the type of each bit to be informative, but names will be erased during compilation[1].

The translation above is simple to understand but, in actual functorial values - and, by consequence, in inductive trees - it has the disadvantage of *slowing* the access to the `⟦ F ⟧`-shaped *payload*. As a first optimisation, we fix the actual data representation to a top-level dependent pair: the first component is a *tag* consisting of the full list of (labelled) path-selecting booleans, while the second, now directly accessible, is the actual payload.

```
tag▻ : O [ A ]▻ N → Tree▻ (1+ A)
tag▻ [ F   ] = []
tag▻ (L ⊕ R) = (name L , tag▻ (code L)) ** (name R , tag▻ (code R))

code▻ : (F : O [ A ]▻ N) → ⟦ tag▻ F ⟧Tree▻ → N → De O
code▻ [   F ]     t  n = F n
code▻ (L ⊕ R) (« Ls) n = code▻ (code L) Ls n
code▻ (L ⊕ R) (» Rs) n = code▻ (code R) Rs n

_`$_ : O [ A ]► N → O ▻ N
F `$ n = `Σ (⟦ tag▻ (code F) ⟧Tree▻) (code▻ (code F) § n)

to▻ = _`$_
```

The definition above makes use of the following type for labelled binary trees `Tree►`, which can be find in our base library:

```
mutual
 Tree► = Σ A λ a → Tree▻

 data Tree▻ : ★ lA where
  []   :                Tree▻
  _**_ : Tree► → Tree► → Tree▻
```

The family of labelled paths indexed by labelled trees, from the same project, is called `⟦_⟧Tree►`:

[1]The interested reader can check the definitions in module `AD.Manifest` of the adaptation layer [1] to better understand this technical point.

```
⟦_⟧Tree▻ : Tree▻ → ★ lI
⟦ []                ⟧Tree▻ = ⊤
⟦ (a , l) ** (b , r) ⟧Tree▻ = Σ (a +∋ b) +.[ nκ ⟦ l ⟧Tree▻ , nκ ⟦ r ⟧Tree▻ ]

⟦_⟧Tree► : Tree► → ★ lI
⟦ _ , t ⟧Tree► = ⟦ t ⟧Tree▻
```

While one could argue that, asymptotically, it is reasonable to consider the length of the boolean lists as a runtime constant, we think that not needing to always traverse a list of booleans in order to get to the data has some advantages. For example, in the next section we will see that transporting a functorial value or an inductive tree along a subtyping witness results in data with longer encodings of tags: with this representation, applying the coercion does not make accessing the payload more costly.

In a more low-level implementation, we would expect the depth of a `O [ A ]▻ N` to not exceed the size of a machine word: having split the two parts of the data, the tag in the first component could more easily be encoded as a list of words, most often of size 1.

4.1.1 Helpers

In the following, we will often use the coproduct operator `[_,_]⟦⟧`:

```
[_,_]⟦⟧ : ⟦ to▻ F               ⟧ X ⇛ Y →
          ⟦ to▻ G               ⟧ X ⇛ Y →
          ⟦ to▻ (a ⟩ (F ⊕ G)) ⟧ X ⇛ Y
[ α , β ]⟦⟧ n (« x , p) = α n (x , p)
[ α , β ]⟦⟧ n (» x , p) = β n (x , p)
```

The following syntax will be used to construct *unlabelled* codes and binary sums:

```
<_> : ∀ {A O N} → O [ A ]▻ N → O [ A ]► N
<_> = _⟩_ ε

infixr 5 _[⊕]_

_[⊕]_ : ∀ {A O N} → O [ A ]► N → O [ A ]► N → O [ A ]► N
L [⊕] R = < L ⊕ R >
```

4.2 The subtyping relation

We define a *subtyping* relation between codes for (sums of) endofunctors as an inductive family[2]

```
data _<:_ : En A I → En A I → Set (lA ⊔ S lI) where
  [] : ∀ {F}                            → F  <: F
  <[ : ∀ {L1 L2 R}{n : 1+ A} → L1 <: L2 → L1 <: n ⟩ (L2 ⊕ R )
  ]> : ∀ {L R1 R2}{n : 1+ A} → R1 <: R2 → R1 <: n ⟩ (L  ⊕ R2)
```

A term of type `F <: G` *encodes*, with a first-order representation [3], the computationally relevant path to subcode `F` in code `G`: we view this path as a *subtyping witness*.

Let us see some examples using abstract codes. `B<A+B+C+D` witnesses the fact that `B` is a subtype of `A [⊕] B [⊕] C [⊕] D`:

```
B<A+B+C+D : B <: A [⊕] B [⊕] C [⊕] D
B<A+B+C+D = ]> (<[ [])
```

The labelled codes syntax allows for the same code to appear more than once in a sum: in situations such as the following we might have two subtyping witnesses:

```
F<F+F₁ : F <: F [⊕] F
F<F+F₁ = <[ []

F<F+F₂ : F <: F [⊕] F
F<F+F₂ = ]> []
```

If we had not added labels but simply used descriptions, we would not be able to rely on inference of subtyping witnesses in cases where there are identical codes in a sum; instead, by pairing codes with a label, different usages of the same functor can be distinguished.

For example, if `F` is a code for a binary product functor, we can use it both for an addition and a multiplication operator of a grammar. If we use different labels, the types of `Plus<:Plus+Times` and `Times<:Plus+Times` will not be convertible, and each of them will only have one possible inhabitant:

```
Plus  = i `plus  ⟩ F
Times = i `times ⟩ F

Plus<:Plus+Times : Plus  <: Plus [⊕] Times
Plus<:Plus+Times = <[ []

Times<:Plus+Times : Times <: Plus [⊕] Times
Times<:Plus+Times = ]> []
```

[2]The use of an inductive definition in current Agda gives a fake universe-level dependency on `lA`, i.e., the family `_<:_` is in `Set (lA ⊔ S lI)` instead of `Set (S lI)`: this could be avoided by defining `_<:_` by recursion, but we did not consider it necessary for our current purpose.

[3]In fact, all arguments to constructors `<[` and `]>` are *forced*.

4.2.1 Properties of the subtyping relation

It is possible to prove several properties of the subtyping relation. Note that, as `_<:_` is *morally* expressed via equalities over types (terms of type `En A I` usually contain terms of type `Set`), we will be exploiting the Uniqueness of Identity Proofs principle, as validated by Agda's pattern matching.

- Transitivity (composition)

```
_<:∘_ : {F G H : En A I} → F <: G → G <: H → F <: H
p <:∘ []   = p
p <:∘ <[ q = <[ (p <:∘ q)
p <:∘ ]> q = ]> (p <:∘ q)
```

- Associativity of composition

```
<:∘-assoc : {F1 F2 F3 F4 : En A I}
            (p : F1 <: F2)(q : F2 <: F3)(r : F3 <: F4) →
            (p <:∘ q) <:∘ r ≡ p <:∘ (q <:∘ r)
<:∘-assoc p q []     = <>
<:∘-assoc p q (<[ r) = <[ $≡ <:∘-assoc p q r
<:∘-assoc p q (]> r) = ]> $≡ <:∘-assoc p q r
```

- Antisymmetry

  We first prove a pair of inversion lemmas:

```
<:-invL : {F G H : En A I}{n : 1+ A} → (n ⟩ (F ⊕ G)) <: H → F <: H
<:-invL < = <[ [] <:∘ <
<:-invR : {F G H : En A I}{n : 1+ A} → (n ⟩ (F ⊕ G)) <: H → G <: H
<:-invR < = ]> [] <:∘ <
```

  Inversion lemmas allow to prove a *no-cycle* property for `_<:_`:

```
¬<:L : {F G : En A I}{n : 1+ A} → n ⟩ (F ⊕ G) <: F → [0]
¬<:R : {F G : En A I}{n : 1+ A} → n ⟩ (F ⊕ G) <: G → [0]
¬<:L {F = n ⟩ (L ⊕ R)} (<[  p) = ¬<:L (<:-invL p)
¬<:L {F = n ⟩ (L ⊕ R)} (]>  p) = ¬<:R (<:-invL p)
¬<:R {G = n ⟩ (L ⊕ R)} (<[  p) = ¬<:L (<:-invR p)
¬<:R {G = n ⟩ (L ⊕ R)} (]>  p) = ¬<:R (<:-invR p)
```

  We can now prove that `_<:_` enjoys antisymmetry:

```
<:-antisym : {F G : En A I} → F <: G → G <: F → F ≡ G
<:-antisym []         []  = <>
<:-antisym []      (<[  q) = <>
<:-antisym []      (]>  q) = <>
<:-antisym (<[  p)      q  = magic (¬<:L (q <:∘ p))
<:-antisym (]>  p)      q  = magic (¬<:R (q <:∘ p))
```

To wrap up, as `_<:_` is reflexive by construction (`[] : F <: F`), we can conclude that it is a (relevant) *partial order* relation.

4.2.2 Semantics of subtyping

We introduced the subtyping relation as an inductive family: we assign a semantic interpretation to its inhabitants in terms of the semantics of the codes they relate, i.e., in terms of functors.

Subtyping witnesses of type `F <: G` can be interpreted as *natural transformations* from `⟦ F ⟧` to `⟦ G ⟧`:

```
inj : ∀ {l}{F G : En A I}⦃ p : F <: G ⦄ → F pt[ l ]> G
inj ⦃ []   ⦄ _ _ xs = xs
inj ⦃ <[ p ⦄ _ _ xs = let t , ys = inj _ _ xs in « t , ys
inj ⦃ ]> p ⦄ _ _ xs = let t , ys = inj _ _ xs in » t , ys

injNat : ∀ {l}{F G : En A I}⦃ p : F <: G ⦄ → isNat F G (inj {l} ⦃ p ⦄)
injNat ⦃ []   ⦄ f _        = <>
injNat ⦃ <[ p ⦄ f (i , xs) = Σ.map (λ x → « x) id $≡ injNat ⦃ p ⦄ f (, xs)
injNat ⦃ ]> p ⦄ f (i , xs) = Σ.map (λ x → » x) id $≡ injNat ⦃ p ⦄ f (, xs)

inj# : ∀ {l}{F G : En A I}⦃ p : F <: G ⦄ → F nt[ l ]> G
inj# ⦃ p ⦄ = inj ⦃ p ⦄ , injNat ⦃ p ⦄
```

It is possible to prove that these natural transformations enjoy some additional properties. An important one is that they are *injective*, hence they preserve all the input information.

```
injectivity-inj : {F G : En A I}⦃ p : F <: G ⦄ →
                  ∀ {as bs} → inj ⦃ p ⦄ X i as ≡ inj ⦃ p ⦄ X i bs → as ≡ bs
injectivity-inj                        ⦃ []   ⦄                  h = h
injectivity-inj {F} {n ⟩ (L2 ⊕ R)} ⦃ <[ p ⦄ {tx , x} {ty , y} h =
  injectivity-inj ⦃ p ⦄ (Σ≡→≡ (Σ.map «-inj (λ {a} p → help a p ⊚ p) (≡→Σ≡ h)))
  where help = λ a p → coe-erasable ((λ s → ⟦ code▻ (code L2) s i ⟧ X) $≡ «-inj a)
                                    ((λ s → ⟦ code▻ (L2 ⊕ R ) s i ⟧ X) $≡       a)
                                    (snd (inj X i (tx , x)))
injectivity-inj {F} {n ⟩ (L ⊕ R2)} ⦃ ]> p ⦄ {tx , x} {ty , y} h =
  injectivity-inj ⦃ p ⦄ (Σ≡→≡ (Σ.map »-inj (λ {a} p → help a p ⊚ p) (≡→Σ≡ h)))
  where help = λ a p → coe-erasable ((λ s → ⟦ code▻ (code R2) s i ⟧ X) $≡ »-inj a)
                                    ((λ s → ⟦ code▻ (L  ⊕ R2) s i ⟧ X) $≡       a)
                                    (snd (inj X i (tx , x)))
```

Note that the proof above makes use of UIP (`coe-erasable`). The following lemma states that injection along a composition of subtyping codes equals the composition of injections:

```
inj≡inj∘inj : ∀ {F G H}(p : F <: G)(q : G <: H) →
                inj ⦃ p <:∘ q ⦄ X ⇒≡ inj ⦃ q ⦄ _ ∘⇒ inj ⦃ p ⦄ _
inj≡inj∘inj p []      (i , xs) = <>
inj≡inj∘inj p (<[  q) (i , xs) = Σ.map (λ x → « x) id $≡ inj≡inj∘inj p q (, xs)
inj≡inj∘inj p (]>  q) (i , xs) = Σ.map (λ x → » x) id $≡ inj≡inj∘inj p q (, xs)
```

4.2.3 Smart constructors

In our examples we will often build *smart constructors* [34] using the following helper:

```
=> : ∀ {F G : En A I}{ p : F <: G }{i} → ⟦ F ⟧ (μ G) i → μ G i
=> xs = ⟨ inj _ _ xs ⟩
```

Our `|=>|` corresponds to Swierstra's `inject`.

4.2.4 Bounded quantification

Bounded quantification can be simulated by quantifying over a `Σ`-type `Sub`:

```
Sub : (F : En A I) → Set _
Sub F = Σ _ λ G → G <: F
```

For example, if we want to define a function `f` which takes any subtype of `μ G` and returns an `X`, we will give a definition with the following type:

```
f : (F : Sub G) → μ (fst F) ⇒ X
```

Subtyping witnesses can be mapped over `Sub`:

```
mapSub : {G H : En A I} → G <: H → Sub G → Sub H
mapSub p (_ , q) = _ , q <:∘ p
```

An important fact is that we can *compute* all the subcodes of a given code:

```
subs : (F : En A I) → List (Sub F)
subs (n ⟩   [ F ]) = (_ , []) ∷ []
subs (n ⟩ (L ⊕ R)) = (_ , []) ∷    mapL (mapSub (<[ [])) (subs L)
                                 ++ mapL (mapSub (]> [])) (subs R)
```

We can similarly define *supercodes*:

```
Sup : (F : En A I) → Set _
Sup F = Σ _ λ G → F <: G

pamSup : {H F : En A I} → H <: F → Sup F → Sup H
pamSup p (_ , q) = _ , p <:∘ q
```

In this case the predicates are contravariant (`pam` instead of `map`): there is an (uncountable) infinity of supercodes, so we cannot compute them.

4.2.5 Pointed types from subcodes

A term inhabiting the type `⋆• ≡ Σ Set id`, consisting in a pair of a type and one of its inhabitants, is called *pointed type*. A finite set of dependently-typed definitions can be represented as a list of pointed types, i.e. `List ⋆•`.

In our `adapter` library we are using we defined some modules (`AD.Misc.*Points`) containing a fixed number of definitions, and whose body is taken from the parameter of type `List ⋆•`: both the type and the term are read from the parameter.

In this way, we can instantiate a module with any list of definitions we can compute, up to a constant size (currently, 32). When opening the module we can bring these definitions into scope, in this way we can *hint* the instance argument inference with the computed definitions.

In our case, we will often deal with functions defined by universal quantification over subtyping witnesses with instance arguments, e.g.:

```
f : ⦃ _ : F <: G ⦄ → X ⇒ μ G
```

When using the function `f`, we might want to have in scope the specific subtyping code, so that we do not have to specify it ourselves: the following functions `smartSubs<:` and `smartSubs`, computing pointed sets from subtyping witnesses, can be used together with the modules `*Points` from `AD.Misc` to ensure that the right subtyping witnesses are made available to the instance arguments mechanism.

```
smartSubs<: : ∀ {F G} → F <: G → List ⋆•
smartSubs<: F<G = mapL (λ { (H , H<G) → _ , H<G }) (mapL (mapSub F<G) (subs _))

smartSubs : (F : En A I) → List ⋆•
smartSubs F = smartSubs<: {F = F} []
```

4.2.6 Algebra subtyping

In this section we adopt an idea from [13] (paragraph 4.1, class `WF_Eval`), namely that of *algebra delegation*.

```
 _<:alg[_]_ : F alg> X → F <: G → G alg> X → Set _
 α <:alg[ < ] β = β ∘⇒ inj _ ⇒≡ α
```

`α <:alg[ p ] β` means that `β : G alg> X` *delegates* to `α : F alg> X` with respect to the injection encoded by `p`: in other words, it means that `β` behaves as `α` for what concerns the `⟦ F ⟧`-shaped input that is injected via `p`. We can also read this as a lifting of our subtyping relation to algebras: in this case, we simply say that `α` is a *subalgebra* of `β`.

5 Modular safe effects

In this chapter we construct the *free monad* [34, 25] for the strictly-positive Set^I *endofunctors* captured by the universe of descriptions (2.2.5). By integrating this type with the embedding of subtyping that was developed in chapter 4, we show how the original example of modularisation of monadic *effects* from [34] can also be obtained in a dependently-typed language: an advantage is that indexing, acting as an embedded type system, restricts the programs the user can construct to those that respect some safety conditions.

While all the main ideas can be traced back to the cited literature, our contribution consists in showing how these techniques can be implemented on top of the constructions from the last chapters.

5.1 Free monad

In category theory it is said that an endofunctor F on a category $\mathcal{C}$ has a *free monad* when the forgetful functor that returns the carrier of any of its algebras $U : F\text{-alg} \to \mathcal{C}$ has a left adjoint. If F is a *polynomial functor*, however, there always exists a monad $F\star$ with this property, which can be constructed as follows [17]:[1]

$$F \star X \triangleq \mu\,(F + K_X)$$

Despite the categorial origin of the concept, we stick to a type-theoretical interpretation of this construction, restricting our interest to the polynomial functors over Set^I which, as discussed in [10], are captured by the universe of descriptions: for this reason, as the initial algebra of these functors is a wellfounded *tree* [31], we will regard `F ⋆ X` as the `I`-indexed family of trees which admit *holes* of type `X i`, for some `i : I`. Another common view is that of seeing `F ⋆ X` as *open* terms of language with signature `F`, with variables in the sort `X`.

For any code for a polynomial functor F, we can construct the code for $F + K_X$:

```
_⋆_ : En A I → Set^ I lI → En A I
```

[1] K_X is the constant functor with value the object X.

```
F ⋆ X = ε ⟩ (F ⊕ ε ⟩ [ `K ∘ X ])
```

We obtain an operator of type $Set^I \to Set^I$ by applying the fixpoint `μ`:

```
infixr 8 _∗_

_∗_ : (F : En A I) → Op I I lI lI
F ∗ X = μ (F ⋆ X)
```

We use the identifiers `roll` and `var` for the two constructors:

```
roll : {F : En A I}{X : Set^ I lI} → F alg> F ∗ X
roll i (t , xs) = ⟨ (« t , xs) ⟩

var  : {F : En A I}{X : Set^ I lI} → X ⇒ F ∗ X
var i v = ⟨ (» _ , v , _) ⟩
```

The constructor `roll` is used to introduce `⟦ F ⟧`-shaped data; `var` allows to include occurrences of *variables* of type `X i` in a term of type `(F ∗ X) i`.

We define a free monad-specific catamorphism, where the two components of the algebras are separately required:

```
 cata∗ : F alg> Y → X ⇒ Y → F ∗ X ⇒ Y
 cata∗ α ξ = cata [ α , ξ ∘⇒ fst/ ∘⇒ snd/ ]⟦⟧
```

The following parametric function correspond to the natural transformation η of the (indexed) free monad: we borrow from Haskell the `return` notation.

```
return : ∀ {F X i} → X i → F ∗ X $ i
return = var _
```

In order to obtain monadic *binding* we follow [25] by first defining the *extension operator* for the Kleisli category associated to the free monad:

```
extend : ∀ {F A B} → (A ⇒ F ∗ B) → F ∗ A ⇒ F ∗ B
extend = cata∗ roll
```

Obtaining an infix binding operator with Haskell syntax `_>>=_`, which in the cited paper Conor McBride defines *demonic*, simply requires flipping the arguments and using an implicit quantification over `I` for the index `i`:

```
_>>=_ : ∀ {F A B i} → F ∗ A $ i → (A ⇒ F ∗ B) → F ∗ B $ i
m >>= f = extend f _ m
```

`m >>= f` is an operator which, for any `i : I`, replaces all holes `a : A i` in the tree `m` with another tree `f i a : F ∗ B $ i`: it can be seen as a *grafting* operation, where the actual data contained in the hole selects, from the family `f`, the subtree to be plugged in at its position.

If we only have a continuation `f : A j → F ✶ B $ j` which is restricted to a particular fixed value `j : I`, we need to refine the family of holes `A` to also force the indices to be equal to `j`. By pattern matching on the equality proof we can reuse `_>>=_` to implement what McBride calls *angelic* binding:

```
_=>=_ : ∀ {F A B i j} → F ✶ [ A := j ] $ i  →
                (A j → F ✶ B         $ j) →
                       F ✶ B         $ i
m =>= f = m >>= λ { ._ (<> , a) → f a }
```

Note that in the library we are importing `[ A := j ]` is defined as `λ i → i ≡ j × A i`.

The monad's μ - called, as in Haskell, `join` - and the action on morphisms of the underlying functor can also be defined in terms of `extend`:

```
join : ∀ {F X} → F ✶ (F ✶ X) ⇒ F ✶ X
join = extend (λ _ → id)
```

```
fmap : ∀ {F A B} → (A ⇒ B) → F ✶ A ⇒ F ✶ B
fmap f = extend (var ∘⇒ f)
```

5.2 Modular effects

In this chapter the indexed free monad construction is applied to the problem of representing effectful programs: for a *signature* functor F encoding the API to some side-effecting operations, we can build $F \star X$ *computation trees* which can be easily composed by leveraging the monadic binding operators.

The index of a free-monadic computation should be seen as an input state, hence we use the following view of a monadic computation as a judgment:

```
_⊢_↓_ = λ {A I : Set}(F : En A I)(s : I)(X : Set^ I _) → F ✶ X $ s
```

`F ⊢ s ↓ P` is the type of free monadic computations for the functor (code) `F` which, starting from state `s`, terminate in a state that satisfies predicate `P`.

We need some auxiliary definitions. We will be using `[[_]]` to embed sets in predicates:

```
[[_]] : {I : Set} → Set → Set^ I _
[[ X ]] _ = X
```

The following synonyms will make code more understandable:

```
Contents = String
FilePath = String
```

`State` is employed to represent the state of a file from the point of view of a process:

```
data State : Set where
 Open Closed : State
```

We adopt the standard library's `IO.Primitive` module:

```
open import IO.Primitive as IO using (IO)
```

`IO` can be extended by postulating new primitives. We need:

- two simple methods for dealing with standard I/O

  ```
  postulate
   getCh   : IO Char
   putCh   : Char → IO [1]
  ```

  `getCh` and `putCh` are meant to correspond to Haskell's `System.IO.getChar` and `System.IO.putChar`.

- methods for opening, reading and closing a single file

  ```
   openFi  : FilePath → IO State
   readFi  : IO Contents
   closeFi : IO [1]
  ```

  `openFi` should return monadically whether the given path could be opened or not; `readFi` is a *partial* operation returning the whole content of the file as a string, yielding an error in case no file was opened; `closeFi` closes the file, if any was opened.

The interface is kept abstract and serves as an intuitive reference: one could implement it via Agda's Haskell FFI, using a mutable reference to carry a file handle when one is available.

5.3 The `Teletype` effect

In this section we show how an interface for getting and putting characters can be easily encoded as a `En I` code for any type `I`: in this case indexing is not used as standard input and output are always possible. The example is the transposition to our framework of the `Teletype` effect in Swierstra's *Datatypes à la carte* [34].

```
module Teletype {A : Set}(`TT `getChar `putChar : A)(I : Set) where
```

The signatures are very simple: while `GetCharF` requires a continuation for any input `Char`, `PutCharF` asks for the character to be output.

```
 GetCharF PutCharF : En A I
 GetCharF = ¡ `getChar  ⟩ [ (λ i → `Π Char λ _ → `I i) ]
 PutCharF = ¡ `putChar  ⟩ [ (λ i → `Σ Char λ _ → `I i) ]
```

`TeletypeF` is simply the coproduct of the above codes:

```
 TeletypeF : En A I
 TeletypeF = ¡ `TT ⟩ (GetCharF ⊕ PutCharF)
```

The following *smart constructors* will be the modular methods that adapt to any monad `F *`, as long as it supports `GetCharF` (for `getChar`) or `PutCharF` (for `putChar`).

```
 getChar : ∀ {F} → ⦃ _ : GetCharF <: F ⦄ → ∀ {i} → F ⊢ i ↓ (≡ i ×/ [[ Char ]])
 getChar = =>* (_ , λ c → ↑return (<> , c))

 putChar : ∀ {F} → ⦃ _ : PutCharF <: F ⦄ → Char → ∀ {i} → F ⊢ i ↓ (≡ i ×/ [[ ⊤ ]])
 putChar c = =>* (_ , c , ↑return (<> , _))
```

A monadic interpretation of the whole `TeletypeF` effect signature can be given as an algebra:

```
 evalAlg : ∀ {A} → TeletypeF ⋆ A alg> IO ∘ A
 evalAlg i (« « t , f      ) = getCh   IO.>>= ↓_ ∘ f
 evalAlg i (« » t , c , ↑ x) = putCh c IO.>>= λ _ → x
 evalAlg i (» _   , x , _  ) = IO.return x
```

5.4 The *safe* `FileSystem` effect

The safe filesystem effect is taken from the Agda development[2] accompanying Pierre-Evariste Dagand's PhD thesis [10]. In this case, the API concerns file operations (opening, reading and closing), where for simplicity at most a single file is accessed at any time.

```
module FileSystem {A : Set}(`FS `openFile `readFile `closeFile : A) where

 OpenFileF ReadFileF CloseFileF : En A State
```

Opening a file means sending a path - `` `Σ `` - to the operating system and waiting for it to communicate - `` `Π `` - whether it really succeeded - `s : State` - in reserving the resource. An important aspect of `OpenFileF` is the fact that, by concluding with the identity code `` `I `` at `s`, we are forcing the ensuing continuation to be of the sort `s`: this means that whatever program will be called after this action, it will have to support both possible states.

```
 OpenFileF = ¡ `openFile  ⟩ [ > ] where
   > : _ → _
   > Closed = `Σ FilePath λ _ → `Π State λ s → `I s
   > _      = `K ⊥
```

[2]Module `Chapter6.IDesc.FreeMonad.Examples.FileSystem`.

Reading the file amounts to asking - `` `Π `` - for its contents, which may or may not be returned. We ask for the whole contents at once: if `Nothing` is returned, the resource is no more available.

```
ReadFileF = i `readFile  ⟩ [ > ] where
  > : _ → _
  > Open = `Π (1+ Contents) λ { (i xs) → `I Open
                             ; ε      → `I Closed }
  > _    = `K ⊥
```

Closing the file is the easiest: we just need to make sure that the current state is `Open` and restrict to continuations that assume the file to be closed. Whatever the state of the file descriptor is, the subsequent code will not have direct access to it, so we can consider it closed immediately, with no need to synchronize.

```
CloseFileF = i `closeFile  ⟩ [ > ] where
  > : _ → _
  > Open = `I Closed
  > _    = `K ⊥
```

We can group the above codes in a single one, representing the whole safe filesystem API:

```
FileSystemF : En A State
FileSystemF = i `FS ⟩ (OpenFileF [⊕] ReadFileF ⊕ CloseFileF )
```

The modular methods have very precise indexing, agreeing with the specifications in the codes:

```
openFile : ∀ {F}⦃ p : OpenFileF <: F ⦄ → FilePath → F ⊢ Closed ↓ [[ ⊤ ]]
openFile path = =>* (_ , path , λ _ → ↑return tt)

readFile : ∀ {F}⦃ p : ReadFileF <: F ⦄ → F ⊢ Open ↓ [κ String := Open ] +/ ≡ Closed
readFile ⦃ p ⦄ = =>* ⦃ p ⦄ (_ , λ { (i x) → ↑return (inl (<> , x))
                                 ;    ε  → ↑return (inr  <>     ) })

closeFile : ∀ {F}⦃ p : CloseFileF <: F ⦄ → F ⊢ Open ↓ ≡ Closed
closeFile = =>* (_ , ↑ return <>)
```

Again, we can provide an interpretation algebra for this fragment:

```
evalAlg : {A : Set} → FileSystemF ⋆ [[ A ]] alg> [[ IO A ]]
evalAlg Closed (« « « _ , fn , f) = openFi fn IO.>>= ↓_ ∘ f
evalAlg Open   (« « « _ , () , f)
evalAlg Closed (« « » _ , () , _)
evalAlg Open   (« « » _ , f     ) = readFi  IO.>>= ↓_ ∘ f ∘ i
evalAlg Closed (« » _   , () , _)
evalAlg Open   (« » _   , m     ) = closeFi IO.>>= λ _ → ↓ m
evalAlg _      (» _     , a , _ ) = IO.return a
```

While program construction is regulated by indices, evaluation only concerns programs whose family of *holes* `A` is non-dependent.

5.5 Modular effects for a safe `cat`

A simple instantiation of the two effects we have demonstrated is the following implementation of a simplistic version of the UNIX `cat` program: the example, originally from *Datatypes à la carte*, is here rebuilt by using the safe filesystem interface.

```
module Cat {A : Set}
           (`TT `getChar  `putChar             : A)
           (`FS `openFile `readFile `closeFile : A)
           where

 open module F = FileSystem `FS `openFile `readFile `closeFile
 open module T = Teletype   `TT `getChar  `putChar  State
```

The type of `cat` indicates that both the `Teletype` and `Filesystem` effects are used, and that the computation starts and ends with a closed file.

```
 cat : ∀ {F} ⦃ p : FileSystemF <: F ⦄ ⦃ q : TeletypeF <: F ⦄ →
       FilePath → F ⊢ Closed ↓ ≡ Closed
```

When implementing `cat`, we must satisfy the indexing constraints of the `FilesystemF` interface. At every program point, we are obliged to face the possible states the file handle can go through and must comply to the usage protocol for the resource. *Angelic binding* (`=>=`) helps in combining monadic operations with continuations at specific indices.

```
 cat {F} ⦃ p ⦄ ⦃ q ⦄ fp =
   openFile fp >>= λ { Closed x → putChars openErrorMsg =>= λ _ →
                                  return <>
                     ; Open   x → readFile >>= postRead }
  where

   open 16Points (smartSubs<: p List.++ smartSubs<: q) 0

   openErrorMsg = S.toList "Error while opening the file!\n"
   readErrorMsg = S.toList "Error while reading the file!\n"

   putChars : List Char → F ⊢ Closed ↓ ≡ Closed ×/ [[ ⊤ ]]
   putChars []       = return (<> , tt)
   putChars (x ∷ xs) = putChar x =>= λ _ → putChars xs

   postRead : ≡ Open ×/ [[ String ]] +/ ≡ Closed ⇒ F ✶ ≡ Closed
   postRead  Closed (inl (() , _)  )
   postRead .Closed (inr <>        ) = putChars readErrorMsg =>= λ _ →
                                       return <>
   postRead .Open   (inl (<> , str)) = closeFile >>= λ { .Closed <> →
                                       putChars (S.toList str) =>= λ _ →
                                       return <> }
```

6 Modular verified compilers

6.1 Introduction

In this chapter we show how it is possible to capture a notion of *modular verified compiler* with a single dependent type.

We consider a *verified compiler* a function which translates a source *indexed* type into another,

```
comp : So ⇛ Ta
```

together with a proof that some given relation `_ℝ_` holds over the interpretations (`eval` and `exec`) of the two languages in their respective domains:

```
eval : So ⇛ DSo
exec : Ta ⇛ DTa
ok   : ∀ {o}{x : So o} → eval x ℝ exec (comp x)
```

$$\begin{tikzcd}
So \arrow[r, "comp"] \arrow[d, "eval"] & Ta \arrow[d, "exec"] \\
D_{So} \arrow[r, leftrightarrow, "\mathcal{R}"] & D_{Ta}
\end{tikzcd} \tag{6.1}$$

As the index types can differ, we allow the user to provide a functor `H` to fix the mismatch. Given

```
So : Set^ O
Ta : Set^ N
H  : Functor N O
```

we would encode a generic verified compiler as follows:

```
comp : So ⇛ ∣ H ∣ Ta
ok   : ∀ {o}{x : So o} → eval x ℝ ∣ H ∣map exec (comp x)
```

$$\begin{tikzcd} So \arrow[r, "comp"] \arrow[d, "eval"] & H\ Ta \arrow[d, "H\ exec"] \\ D_{So} \arrow[r, leftrightarrow, "\mathcal{R}"] & H D_{Ta} \end{tikzcd} \tag{6.2}$$

This representation, while generic and reusable, still does not allow to compose (*sum*) verified compilers for different fragment source languages.

As in [12], we use the term *modular* to refer to the compilation style that allows independent compilation of different fragments of the syntax, so that it is possible to build a larger compiler by assembling independent components, each possibly concerning a different language *feature*. In the cited paper, Day and Hutton tackle this problem by describing evaluation `eval` and compilation `comp` by means of functor algebras: composition of two fragments is the binary coproduct of their functors, while the composed program is obtained by folding the coproduct of the algebras.

In the development below, we build on the ideas of the cited paper, with some differences:

- we do not consider the exceptions feature, but limit ourselves to simple arithmetical and logical expressions;
- we treat a generic semantic relation instead of the commutativity of a simple diagram (`extr ∘ exec ∘ comp ≡ eval`);
- we deal with (codes for) indexed functors, with possibly *different* index sets for source and target, instead of simple $Set \to Set$ functors;
- we take advantage of dependent types to also consider verification, and also suggest a *modular* representation of the *elimination method* for the correctness proof: we can therefore build *verified* compilers for different fragments in isolation, allowing a user to freely sum the needed ones in order to build the required compiler, together with its correctness proof.

6.2 Languages

We first define what we consider a *language definition*: given a semantic domain `Dom : Set^ I`, we request a code for an inductive type `F : En A I` together with an `⟦ F ⟧`-algebra with carrier `Dom`, representing the semantic interpretation.

```
record Lang {I : Set lI}{lD}(Dom : Set^ I lD) : Set (S lI ⊔ lD ⊔ lA) where
  constructor mk
  field
    lang : En A I
    alg  : lang alg> Dom
open Lang public
```

Languages are closed under coproduct: we need to take the coproduct of both the codes and the algebras:

```
_[_]Lang+_ : {I : Set lI}{D : Set^ I lI} → Lang D → 1+ A → Lang D → Lang D
L1 [ a ]Lang+ L2 = mk (a ⟩ (lang L1 ⊕ lang L2)) [ alg L1 , alg L2 ]⟦⟧
```

6.3 Modular verified compilers

`MVC So H Ta _ℝ_` is a type of *modular verified compilers* from language `So` to language `Ta` with *effects* captured by `H`, and satisfying the correctness property encoded in the relation `_ℝ_`.

It contains a function (the *compiler*) from all supertypes (or, better, *superfamilies*) of `μ (lang So)` to `∣ H ∣ (μ (lang Ta))`, with a modular (i.e., reusable over any such superfamilies) proof that the relation `_ℝ_` holds between the evaluation of the (`μ (lang So)`-inhabiting) input of the compiler and the (`H`-mapped) execution of the output of the compiler for that input.

```
record MVC {O N : Set lI}{DSo : Set^ N lI}{DTa : Set^ O lI}
           (So : Lang DSo)(H : Functor O N lI lI)(Ta : Lang DTa)
           {lP}(_ℝ_ : ∀ {i} → DSo i → ∣ H ∣ DTa i → Set lP)
           : Set (S lI ∪ S lP ∪ lA) where

  constructor mk

  private
    SoF = lang So ; module So = Cata {F = SoF}
    TaF = lang Ta ; module Ta = Cata {F = TaF}
```

Agda's records are special module definitions, hence we can define the generic source and target semantic functions in the local context, which we respectively call `eval` and `exec`:

```
  eval : μ SoF ⇒ DSo
  eval = So.cata (alg So)

  exec : μ TaF ⇒ DTa
  exec = Ta.cata (alg Ta)
```

The first field is the compilation algebra from `So` to `Ta`.

```
  field
    compAlg : lang So alg> ∣ H ∣ (μ (lang Ta))
```

The actual compiler is exposed as `comp`:

```
  comp : μ SoF ⇒ ∣ H ∣ (μ TaF)
  comp = So.cata compAlg
```

The other field is the proof method. In order to make it *composable* we use bounded quantification over all supercodes of the source code, obtaining a code `G`. The induction step, then, is required to hold over all superalgebras (4.2.6) of `alg So` and `compAlg`[1]:

```
  field
    OK : (G : Sup SoF) → let G , p = G in
         (β : G alg> _) → alg So  <:alg[ p ] β →
         (γ : G alg> _) → compAlg <:alg[ p ] γ →
         let module G = Cata {F = G} in
         let P = uc λ n x → G.cata β n x ℝ ∣ H ∣map exec _ (G.cata γ _ x) in
         ∀ xs → let n , xs = xs in
         □/ SoF P (n , xs) →
```

`OK`'s conclusion *should* consist of the application of the predicate `P` to the injected value:

```
         P n (=> xs)
```

However, we found that avoiding calls to `=>`, by performing some substitutions in advance as justified by naturality of `inj` and the two delegation hypothesis, could considerably simplify proofs. For this reason, instead of `P n (=> xs)`, we ask for the following equivalent statement:

```
           alg So _ (G.mapCata β (SoF `$ n) xs)
         ℝ ∣ H ∣map exec _ (compAlg _ (G.mapCata γ (SoF `$ n) xs))
```

The correctness proof, validating the correctness of the above alternative, is exposed as a method `ok`:

```
  ok : (n : N)(x : μ SoF n) → eval _ x  ℝ  ∣ H ∣map exec _ (comp n x)
  ok = cu $ Elim.elim SoF _ (OK (SoF , []) (alg So) (λ _ → <>) compAlg (λ _ → <>))
```

6.4 Summing modular verified compilers

It is now possible to sum modular compilers which share the same target language `T`.

```
module MVC+ {O N : Set lI}{DSo : Set^ N lI}{DTa : Set^ O lI}
            (S1 S2 : Lang DSo)(H : Functor O N _ lI)(T : Lang DTa)
            {lP}(_ℝ_ : ∀ {i} → DSo i → ∣ H ∣ DTa i → Set lP)
            (a : 1+ A) where
```

We define some local shortcuts:

[1]Note that, however, due to the way we define the goal, the additional *algebra delegation* hypothesis in the type of `OK` might not be needed by specific proofs, such as the ones in our example: however, the availability of these additional hypothesis does not break the composability of proofs, and we expect them to be useful in some cases.

```
private
 S12 = S1 [ a ]Lang+ S2; F1 = lang S1; F2 = lang S2; F12 = a ⟩ (F1 ⊕ F2)
 TF = lang T
```

The output compiler algebra is simply the coproduct of the algebras of the two input compilers:

```
_MVC+_ : MVC S1 H T _ℝ_ → MVC S2 H T _ℝ_ → MVC S12 H T _ℝ_
(mk c1 m1) MVC+ (mk c2 m2) = mk [ c1 , c2 ]⟦⟧ m where
```

The composed proof method, instead, both requires the inversion lemmas for `_<:_` (4.2.1) and simple equality proofs based on `inj≡inj∘inj` (4.2.2):

```
 m : ∀ G β _ γ _ xs → _ → _
 m (G , <) β <β γ <γ (n , « _ , ls) ih =
  m1 (G , <:-invL <) β (uc λ n xs →   β n $≡ (inj≡inj∘inj _ < (n , xs))
                                     ⊚ <β (n , « _ , snd xs))
                    γ (uc λ n xs →   γ n $≡ (inj≡inj∘inj _ < (n , xs))
                                     ⊚ <γ (n , « _ , snd xs))
                    (n , _ , ls) ih
 m (G , <) β <β γ <γ (n , » _ , rs) ih =
  m2 (G , <:-invR <) β (uc λ n xs →   β n $≡ (inj≡inj∘inj _ < (n , xs))
                                    ⊚ <β (n , » _ , snd xs))
                    γ (uc λ n xs →   γ n $≡ (inj≡inj∘inj _ < (n , xs))
                                    ⊚ <γ (n , » _ , snd xs))
                    (n , _ , rs) ih
```

In the following section we will see how this generic operation can be used to build a verified compiler for a language by composing verified compilers for its fragments.

6.5 Examples

In this section we provide an example use of the construction we have just presented. We will define minimal well-typed by construction expression and stack-machine languages *modularly*, i.e., by encoding each grammar production individually: we will specify how to interpret each fragment in a semantic domain (filling in the `Lang`'s `alg` field), and provide a compiler (`compAlg` field of `MVC`) from fragments of the source language to a fixed target obtained by composing the stack machine language fragments. We will also build a proof of correctness for each compilation unit by inhabiting the `OK` field of `MVC`.

6.5.1 Well-typed representation

With *well-typed by construction* we mean the standard approach of representing terms of a typed languages by an inductive family indexed by type (expressions) [4].

Let us assume that `I : Set` contains the types for the expression language, for example `` `nat : I `` or `` `bool : I ``. The stack machine expressions, instead, will be indexed by pairs of lists of these types: `(ss , ts) : J`, where `J ≡ List I`.

For this reason, to adapt to the `MVC` schema, we will need the following $Set^J \to Set^I$ functor `H`:

```
HF : Op (List I × List I) I Z Z
HF X i = (is : List I) → X (is , i ∷ is)

Hmap : ∀ {A B} → A ⇒ B → HF A ⇒ HF B
Hmap f i = mapΠ λ x → f (x , i ∷ x)

H : Functor (List I × List I) I _ _
H = record { RF          = mk HF Hmap
           ; |_|map-id⇒ = λ _ → <>
           ; |_|map-∘⇒  = λ _ → <> }
```

`H` suggests the relation between the types for the source expression language and those for the target stack machine language: an expression of type `i : I` will be compiled into a stack program that, for any context of source types `is : List I`, will take a stack of type `is` to one of type `i ∷ is`.

6.5.2 Stack transformations

We need to define a type of stacks and stack transformations. We assume as given a `Ty : I → Set` semantic function:

```
module Stack {I : Set}(Ty : Set^ I Z) where
```

We describe a stack by recycling the lifting of `List` from our base library, using the `Ty` decoding function as predicate.

```
Stack : List I → Set
Stack = □List Ty
```

For stack transformations we employ a functional representation:

```
ST : List I × List I → Set
ST (ss , ts) = Stack ss → Stack ts
```

The `ST` family is indexed by a pair of lists: the first component is the index of the source stack of the transformation, the second is the index for the target stack.

We require

- a function `addTy?` which decides whether two types admit addition and, when they do, the type of the result;
- a function `_+_` which performs the sum when it is possible.

```
module Add {addTy? : I → I → 1+ I}
           (_+_ : ∀ {i j k}⦃ q : addTy? i j ≡ i k ⦄ → Ty i → Ty j → Ty k)
           where
```

With these assumptions, it is easy to implement a generic (total) *addition* primitive for stacks, taking a stack with two values `v2 : Ty t2` and `v1 : Ty t1` at the top, and producing a stack with those two entries replaced by a single one, whose value is their sum `(v1 + v2) : Ty t`.

```
  stackAdd : {t1 t2 t : I}⦃ q : addTy? t1 t2 ≡ i t ⦄
             (ss : List I) → ST (t2 :: t1 :: ss , t :: ss)
  stackAdd _ (v2 , v1 , vs) = v1 + v2 , vs
```

We can already describe the relation that must hold between the semantics of the input expression (of type `Ty i`) and the family of stack transformations of type `∣ H ∣ ST i ≡ (is : List I) → ST (is , i :: is)` which is obtained by interpreting the compiled stack machine code for that expression:

```
_ℝ_ : {i : I} → Ty i → ∣ H ∣ ST i → Set
e ℝ f = ∀ ss xs → e , xs ≡ f ss xs
```

`_ℝ_` specifies the correctness statement of a modular compiler in the two semantic domains: we require that for any stack `xs`, pushing the semantic interpretation of an expression `e` on top of it will give the same result as applying the corresponding stack transformation `f` to `xs`.

6.5.3 Fragments

In the following we will limit ourselves to showing that a compiler for a straightforward grammar exp

$$\begin{aligned} n &::= 0 \mid 1 \mid 2 \mid \ldots \\ exp &::= n \mid exp + exp \end{aligned}$$

can be expressed as a *sum* of a modular compiler for val

$$\begin{aligned} n &::= 0 \mid 1 \mid 2 \mid \ldots \\ val &::= n \end{aligned}$$

and a modular compiler for $plus$

$$plus ::= plus + plus$$

The target, in both cases, is a fixed language for stack machine programs, i.e.:

$$PROG ::= PUSH\; n \mid ADD \mid NIL \mid PROG\, ; PROG$$

6.5.4 Stack machine programs

We first encode the target language, as it is required by both source fragments. The definition is part of a standalone module `Prog`, parametrised by some tags and by the same assumptions already introduced for `Stack`:

```
module Prog {I A : Set}(`PUSH `+++ `<> `ADD : A)
            (addTy? : I → I → 1+ I)
            (Ty     : Set^ I Z)
            (_+_    : ∀ {i j k}⦃ q : addTy? i j ≡ i k ⦄ → Ty i → Ty j → Ty k)
            where
```

The family of stack machine programs is indexed by pairs of lists of type codes in `I`: in fact, each stack machine program `μ Stack (ss , ts)` will be shown to denote a stack transformation `ST (ss , ts)`. We use `J` as a synonym for the type of such pairs:

```
 private J = List I × List I
```

The `Push`, `Append`, `Nil` and `Add` codes represent the constructs of the stack machine language; being a well-typed representation, they actually capture their *typing rules* directly. The first, `Push`, requires a proof that the type of the output stack is equal to `t :: ts`, where `ts` is the type of the input stack, together with an inhabitant of `Ty t`.

```
PushF : J → De J
PushF (ss , []     ) = `K ⊥
PushF (ss , t ∷ ts) = `K (ss ≡ ts × Ty t)

Push : En A J
Push = i `PUSH ⟩ [ PushF ]
```

The `Append` code requires a middle stack type `bs` and two subprograms with types `(as , bs)` and `(bs , cs)`: it clearly corresponds to sequential composition.

```
Append : En A J
Append = i `+++ ⟩ [ (λ { (as , cs) → `Σ _ λ bs → `I (as , bs) `× `I (bs , cs) }) ]
```

The `Nil` code only requires a proof that the source and target stacks have the same type.

```
Nil : En A J
Nil = i `<> ⟩ [ `K ∘ uc _≡_ ]
```

The `Add` code builds a stack machine program that, if `addTy? t1 t2 ≡ i t`, takes a stack with a pair of elements of types `t1` and `t2` on top and returns a stack where they are substituted by a single element of type `t`:

```
AddF : J → De J
AddF (t1 ∷ t2 ∷ ss , t ∷ ts) = `K (ss ≡ ts × addTy? t1 t2 ≡ i t)
AddF                      _  = `K ⊥

Add : En A J
Add = i `ADD ⟩ [ AddF ]
```

The corresponding *smart* constructors can be defined as follows:

```
PUSH : ∀ {F}⦃ p : Push <: F ⦄{bs t} → Ty t → μ F (bs , t ∷ bs)
PUSH n = => $ _ , (<> , n) , _

infixr 5 _+++_

_+++_ : ∀ {F}⦃ p : Append <: F ⦄{as bs cs} →
        μ F (as , bs) → μ F (bs , cs) → μ F (as , cs)
p1 +++ p2 = => $ _ , _ , ↑ p1 , ↑ p2

<-> : ∀ {F}⦃ p : Nil <: F ⦄{ss} → μ F (ss , ss)
<-> = => $ _ , <> , _

ADD : ∀ {F}⦃ p : Add <: F ⦄{t t1 t2 ts}⦃ q : addTy? t1 t2 ≡ i t ⦄ →
      μ F $ t1 ∷ t2 ∷ ts , t ∷ ts
ADD ⦃ q = q ⦄ = => $ _ , (<> , q) , _
```

The following algebras define the semantics of stack machine programs as stack transformations:

```
 open Stack Ty

 evalPush : Push alg> ST
 evalPush (_  ,    []) (_ , ()       , _)
 evalPush (._ , _ ∷ _) (_ , (<> , t) , _) = _,_ t

 evalNil : Nil alg> ST
 evalNil _ (_ , q , _) = rew Stack q

 evalAppend : Append alg> ST
 evalAppend _ (_ , _ , ↑ st1 , ↑ st2) = st2 ∘ st1

 evalAdd : Add alg> ST
 evalAdd (t1 ∷ t2 ∷ .ts , t ∷ ts) (_ , (<> , p), _) (x1 , x2 , xs) = (x1 + x2) , xs
 evalAdd ([]     , _ ∷ _) (_ , () , _)
 evalAdd (_ ∷ [] , _ ∷ _) (_ , () , _)
 evalAdd (_      ,    []) _ _ = _
```

`Prog` is the code of the base functor for the family of stack machine programs:

```
 Prog : En A J
 Prog = Push [⊕] Append [⊕] Nil [⊕] Add
```

Pairing the code with its semantics, we obtain a `Lang ST`:

```
 ProgL : Lang ST
 ProgL = mk Prog [ evalPush , [ evalAppend , [ evalNil , evalAdd ]⟦⟧ ]⟦⟧ ]⟦⟧
```

In the following two subsections we will see a couple of source language fragments, for which not only the semantics will be defined, but also an instance of a modular verified compiler to the common target `Prog`.

6.5.5 The *val* language

The source language fragment `Val`, containing a construct that allows to embed an element of any type captured by `I` via a family `Ty`, has a simple definition. The code simply requires one such element:

```
module Val {I A : Set}(`val : A)(Ty : Set^ I Z) where

 ValF : En A I
 ValF = i `val ⟩ [ `K ∘ Ty ]

 val : {F : En A I}⦃ p : ValF <: F ⦄ → {i : I} → Ty i → μ F i
 val v = => $ _ , v , _
```

Evaluation for an embedded value is the identity:

```
 evalAlg : ValF alg> Ty
 evalAlg _ (_ , a , _) = a

 ValL : Lang Ty
 ValL = mk ValF evalAlg
```

Compilation makes use of `PUSH` and `<->` to build the program which simply pushes a value `v` on the stack[2].

```
module Comp (`PUSH `+++ `<> `ADD : A) addTy?
            (_+_ : ∀ {i j k}⦃ q : addTy? i j ≡ i k ⦄ → Ty i → Ty j → Ty k)
            where

  open Prog `PUSH `+++ `<> `ADD addTy? Ty _+_
  open 8Points (smartSubs Prog) 0

  compAlg : ValF alg> ∣ H ∣ (μ Prog)
  compAlg i (_ , v , _) _ = PUSH v +++ <->
```

A modular verified compiler from `ValL` to `ProgL` with `H`-encoded effects and respecting `_ℝ_` can be very easily implemented by `compAlg`, and verified by reflexivity:

```
  open Stack Ty

  mvc : MVC ValL H ProgL _ℝ_
  mvc = mk compAlg λ _ _ _ _ _ _ _ _ _ _ → <>
```

6.5.6 The *plus* language

The module corresponding to the `Plus` fragment is, again, parameterised by `addTy?` and `_+_`:

```
module Plus {I A : Set}(`plus : A)
           (addTy? : I → I → 1+ I)(Ty : Set^ I Z)
           (_+_ : ∀ {i j k}⦃ q : addTy? i j ≡ i k ⦄ → Ty i → Ty j → Ty k)
           where
```

The code is a recursive pair of codes (`` `I i `× `I j ``), paired with (propositional) evidence that types `i` and `j` can be added to obtain an element of type `k`:

```
Plus : En A I
Plus = i `plus ) [ (λ k → `Σ _ λ i → `Σ _ λ j →
                          `Σ (addTy? i j ≡ i k) λ _ → `I i `× `I j) ]
```

Exactly like `ADD`, the smart constructor `_plus_` requires the additional type compatibility hypothesis `ijk` to be satisfied via the instance arguments mechanism:

```
module _ {F}⦃ p : Plus <: F ⦄{i j k}⦃ ijk : addTy? i j ≡ i k ⦄ where

  infixr 4 _plus_

  _plus_ : μ F i → μ F j → μ F k
  x plus y = => $ _ , _ , _ , ijk , ↑ x , ↑ y
```

Given `_+_`, the semantics is straightforward:

[2]Note how the instance arguments for `PUSH` and `<->` are inferred thanks to the `8Points` module, which adds to the context the possible subtyping witnesses for `Prog`.

```
evalAlg : Plus alg> Ty
evalAlg _ (_ , _ , _ , _ , ↑ x , ↑ y) = x + y

PlusL : Lang Ty
PlusL = mk Plus evalAlg
```

The compilation algebra for `Plus` consists of the concatenation of the subprograms corresponding to the two subexpressions, followed by the addition instruction.

```
module Comp (`PUSH `+++ `<> `ADD : A) where

 open Prog `PUSH `+++ `<> `ADD addTy? Ty _+_
 open 8Points (smartSubs Prog) 0

 compAlg : Plus alg> ∣ H ∣ (μ Prog)
 compAlg i (_ , _ , _ , _ , ↑ c2 , ↑ c1) is = c1 is +++ c2 _ +++ ADD
```

The `MVC` implementation for `PlusL` is more complicated than that for `ValL`, which is a consequence of the fact that this constructor is recursive. The Agda proof is verbose and its typechecking is slow, but it ultimately consists in rewriting by the inductive hypotheses `ihL` and `ihR`.

```
 open Stack Ty

 mvc : MVC PlusL H ProgL _ℝ_
 mvc = mk compAlg meth where
  meth : ∀ G β _ γ _ xs ih is ss → _
  meth G β _ γ _ (k , t , i , j , p , ↑ e1 , ↑ e2) (↑ ihL , ↑ ihR) is ss =
   goal where
    module G = Cata {F = fst G}
    module P = Cata {F = Prog }
    d1 : Ty i ; d1 = G.cata β i e1
    d2 : Ty j ; d2 = G.cata β j e2
    c1 : ∀ is → ST (is , i ∷ is)
    c1 = ∣ H ∣map (P.cata (alg ProgL)) i (G.cata γ i e1)
    c2 : ∀ is → ST (is , j ∷ is)
    c2 = ∣ H ∣map (P.cata (alg ProgL)) j (G.cata γ j e2)
    lhs = d1 , d2 , ss
    rhs = c1 (j ∷ is) (c2 is ss)
    rec : lhs ≡ rhs
    rec = ihL (j ∷ is) _ ⊚ c1 (j ∷ is) $≡ ihR is ss
    infix 4 _≣_
    _≣_ = Id (Stack (k ∷ is))
    goal : d1 + d2 , ss ≣ head rhs + head (tail rhs) , tail (tail rhs)
    goal = rew (λ s →   (d1 + d2 , ss)
                      ≣ (head s + head (tail s) , tail (tail s)))
               rec <>
```

6.5.7 Instantiating $val \mid plus$

In order to give a concrete instance of a modular verified compiler for the `Exp` language, we just need to give the required closed definitions:

```
module Val+Plus where
```

```
data Names : Set where `val `plus `PUSH `+++ `<> `ADD : Names
data Types : Set where `nat : Types

Ty : Types → Set
Ty t = ℕ

addTy? : Types → Types → 1+ Types
addTy? `nat `nat = i `nat

open Stack Ty
```

With some slightly verbose commands for which some syntactic sugar could surely be desirable, we can now wire the language modules to the concrete tags and auxiliary definitions:

```
module V  = Val    `val                        Ty     ; open V
module VC = V.Comp `PUSH `+++ `<> `ADD addTy?    _+_
module P  = Plus   `plus               addTy? Ty _+_ ; open P
module PC = P.Comp `PUSH `+++ `<> `ADD
open        Prog   `PUSH `+++ `<> `ADD addTy? Ty _+_
```

Generating the concrete modular verified compiler `mvc` for the union of `ValL` and `PlusL` is now a matter of opening the `MVC+` module and calling `_MVC+_`:

```
open MVC+ ValL PlusL H ProgL (λ {i} → _ℝ_ {i}) ε

mvc = VC.mvc MVC+ PC.mvc
```

`testOK` shows what correctness proof is included in `mvc`:

```
open MVC mvc
ExpL = ValL [ ε ]Lang+ PlusL

private
 testOK : (t : Types)(x : μ (lang ExpL) t)
             (ss : List Types)(xs : Stack ss) →
                eval _ x , xs ≡ exec _ (comp _ x ss) xs
 testOK = ok
```

Let us now check the (verified) compilation on an example expression, which we construct by using smart constructors `val` and `_plus_`.

```
exp : μ (lang ExpL) `nat
exp = val 20 plus (val 8 plus val 4) where
      open 8Points (smartSubs (lang ExpL)) 0
```

We can see how source evaluation, compilation and target evaluation compute on closed terms by simply rechecking `ok` on `exp`. The typechecker shows us that both sides of the equation reduce to a stack with the natural number `32` as its only element.

```
 private
  test-exp-0 : eval _ exp , tt ≡ exec _ (comp _ exp []) tt
  test-exp-0 = ok _ exp _ _

  test-exp-1 : 32 , tt ≡ 32 , tt
  test-exp-1 = test-exp-0
```

6.5.8 Adding logical expressions

In this part we sketch how the two constructs for the grammar of logical expressions

$$exp \quad ::= \quad if\ exp\ th\ exp\ el\ exp \mid exp\ ?=\ exp$$

can also be described as standalone components.

The following is a module with the code and evaluation algebra for the *if-then-else* construct:

```
module IfThEl {A I : Set}{`ite : A}{`bool : I} where

 IfThEl : En A I
 IfThEl = ¡ `ite ⟩ [ (λ i → `I `bool `× `I i `× `I i) ]

 if_th_el_ : {F : En A I}⦃ p : IfThEl <: F ⦄
             {i : I}(b : μ F `bool)(e1 e2 : μ F i) → μ F i
 if b th e1 el e2 = => $ _ , ↑ b , ↑ e1 , ↑ e2

 module _ {Ty}⦃ p : Ty `bool ≡ Bool ⦄
          ⦃ _≟_ : ∀ {i}(x y : Ty i) → Dec (x ≡ y) ⦄ where

  evalAlg : IfThEl alg> Ty
  evalAlg i (_ , ↑ b , ↑ e1 , ↑ e2) = if (rew id p b) then e1 else e2

  IfThElL : Lang Ty
  IfThElL = mk IfThEl evalAlg
```

Here is, instead, a similar module for the equality check:

```
module Equals {A I : Set}{`ε `== : A}{`bool : I} where

 Equals : En A I
 Equals = ¡ `== ⟩ [ (λ i → `Σ (i ≡ `bool) λ _ → `Σ _ λ j → `I j `× `I j) ]

 _?=_ : {F : En A I}⦃ p : Equals <: F ⦄ → ∀ {j} → μ F j → μ F j → μ F `bool
 e1 ?= e2 = => $ _ , <> , _ , ↑ e1 , ↑ e2

 module _ {Ty : I → Set}{true false : Ty `bool}
          ⦃ _≟_ : ∀ {i}(x y : Ty i) → Dec (x ≡ y) ⦄ where

   evalEquals : Equals alg> Ty
   evalEquals ._ (_ , <> , _ , ↑ v1 , ↑ v2) with v1 ≟ v2
   evalEquals ._ (_ , <> , _ , ↑ v1 , ↑ v2) | yes ,  p = true
   evalEquals ._ (_ , <> , _ , ↑ v1 , ↑ v2) | no  , ¬p = false

   EqualsL : Lang Ty
   EqualsL = mk Equals evalEquals
```

Implementing the modular verified compilers, however, would require extending the target instruction set `Prog`, which does not contain instructions which implement checks and jumps: the `MVC` type we have shown does not support modular target languages, unless one reimplements all modules with some further parameterisations. For example, for `Val` we would need to implement a compiler to an abstract target which is a supertype of `Ta`,

```
module _ {Ta : En A _}⦃ <1 : PushF <: TaF ⦄⦃ <2 : NilF  <: TaF ⦄ where

 compAlg = ...
```

while the proof should be made in terms of an abstract algebra which delegates to both `evalPush` and `evalNil`:

```
 module _ (α : Ta alg> ST)
        ⦃ <alg1 : evalPush <:alg[ <1 ] α ⦄
        ⦃ <alg2 : evalNil  <:alg[ <2 ] α ⦄ where

  mvc = ...
```

When we tried to solve the problem in this way, the proof for `Val` went through, while for `Plus` the goal turned out to be completely unreadable, and typechecking was so slow that we had to desist: at the present time we do not know whether this approach can be made to work, possibly in a system which supports more advanced pretty-printing and automation facilities.

Conclusion

The work we presented in this thesis, which was started while the author was a guest of the Functional Programming Laboratory at the University of Nottingham, shows practical techniques for the modularisation of dependently typed programs and proofs.

6.6 Related work

The contents of the first three chapters are taken from the cited literature.

We claim originality for our encoding of labeled finite sums and subtyping in chapter 4.

While developing our solution, we were only aware of Delaware et al.'s work on *Metatheory à la Carte* [13] in Coq, from which we inherited the concept of *algebra delegation*: this very large development uses *impredicative* Church-encodings to represent data, with ad-hoc techniques for modular reasoning that account for the lack of induction principles, and typeclasses to describe subtyping and help modular reasoning. While the size of our developments cannot be compared to that work, we consider our universe-based approach much more direct, allowing for first-order and predicative encodings of data, which capture representations that are not so different from those that one normally uses in functional programming languages.

An Agda embedding of subtyping very similar to ours was recently published by Schwaab and Siek [33]: still, it is based on a simple universe of (unindexed) *normal* functors, similar to the example universe in 1.5, and not allowing the description of inductive families. We recently discovered that also Keuchel and Schrijvers [22] followed Schwaab and Siek's solution, in their case using Coq. In their framework they could modularly formalise several interesting metatheoretical developments, similar to those formalised in *Metatheory à la Carte*.

A universe such as that used in those works, however, is not adequate for our purposes: we want to encode indexed functors that allow the use of indexing typical of dependently-typed programming, which is not available with such a simple first-order type of codes. Furthermore, no `Π or `→ code is provided, which is a useful simplification if one is only interested in metatheory of first-order embeddings of programming languages, but is not expressive enough if one also wants to define modular effectful programs in the style of *Datatypes à la carte*, as we did in chapter 5.

For what concerns the free monad implementation and the modular effects examples in chapters 5.1 and 5, they should be considered demonstrations of techniques from the cited literature [34, 25, 10]: our contribution is the demonstration of how these ideas can be integrated on top of our framework.

The last chapter contains an attempt at the axiomatisation of a type of *modular verified compilers* that, to our knowledge, has never been studied in similar terms.

6.7 Future work

We are interested in continuing the work on these topics, trying both to overcome the current limitations of the shown applications and to extend the framework to support new applications.

For what concerns the description of modular effects interfaces, we would like to try and apply the approach of chapter 5 to more complex domains. For example, we think it would be very interesting to describe safe *concurrency* primitives modularly: we have already done some work - `http://ma82.github.io/html/Session.html` - towards the definition of a well-typed by construction domain specific language with dependent session types, but we have not yet described the syntax as codes of the universe of descriptions.

We would like to solve the problem of describing modular verified compilers with a modular *target* language: we did implement a possible solution based on *tree homomorphisms* [6], but the work on better understanding the expressivity of this representation is still ongoing. We also implemented an `MVC` type based on a modular version of paramorphisms [27], which are more efficient and more expressive than algebras (with a fixed carrier); for reasons similar to those we documented at the end of chapter 6 for our functor algebra-based version, however, we could not implement concrete proof methods.

The languages we considered for modular compilation were very simple: we are surely interested in treating more complex languages, supporting binding or graph structures, but we would like to do so by simply extending the syntax of the universe instead of changing the fixpoint type in an ad-hoc way, as has been done, for example, in recent work by Day and Bahr [11, 5].

In our work we have not attempted to modularly describe the operational semantics of a programming language, hence we would like to also understand the specific problems involved in this task: an approach based on strong theoretical foundations seems to be that developed by Madlener, Smetsers and van Eekelen [23].

6.8 Acknowledgments

I am grateful to Franca Alicò, Thorsten Altenkirch, Valentina Bagnacani, Florent Balestrieri, Joey Capper, Pietro Capra, Venanzio Capretta, Paolo Capriotti, Claudio Sacerdoti Coen, Laurence Day, Bas van Gijzel, Graham Hutton, Nicolai Kraus, Lorenzo Malatesta, Peter Morris, Li Nuo, Neil Sculthorpe, and all the others who offered help or encouragement.

Bibliography

[1] Matteo Acerbi. `adapter` - Agda library. `http://github.com/ma82/adapter`.

[2] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. Manuscript, available online, February 2006.

[3] Thorsten Altenkirch and Peter Morris. Indexed Containers. In *LICS*, page 277–285. IEEE Computer Society, 2009.

[4] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453--468, 1999.

[5] Patrick Bahr. Proving correctness of compilers using structured graphs. FLOPS '14, to appear, February 2014.

[6] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, page 83–94, New York, NY, USA, 2011. ACM.

[7] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[8] James Chapman, Pierre Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, page 3–14. ACM, 2010.

[9] Pierre-Evariste Dagand. Induction from fold. `http://gallium.inria.fr/~pdagand/stuffs/notes/html/InductionFromFold.html`.

[10] Pierre-Evariste Dagand. *Reusability and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[11] Laurence E. Day and Patrick Bahr. Pick'n'fix: Modular control structures. Submitted to Modularity '14, October 2013.

[12] Laurence E. Day and Graham Hutton. Towards Modular Compilers for Effects. In *Proceedings of the 12th International Conference on Trends in Functional Programming*, TFP 2011, page 49–64, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207--218, New York, NY, USA, 2013. ACM.

[14] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143--155, New York, NY, USA, 2011. ACM.

[15] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440--465, 1997.

[16] Peter Dybjer and Anton Setzer. A Finite Axiomatization of Inductive-Recursive Definitions. In Jean-Yves Girard, editor, *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, page 129–146. Springer, 1999.

[17] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 210--225. Springer, 2003.

[18] Jacques Garrigue. Programming with Polymorphic Variants. In *In ML Workshop*, 1998.

[19] Neil Ghani, Patricia Johann, and Clément Fumex. Generic Fibrational Induction. *Logical Methods in Computer Science*, 8(2), 2012.

[20] Peter Hancock. *Ordinals and interactive programs*. PhD, Laboratory for Foundations of Computer Science, University of Edinburgh, 2000.

[21] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145:107--152, 1997.

[22] Steven Keuchel and Tom Schrijvers. Generic datatypes à la carte. In *9th ACM SIGPLAN Workshop on Generic Programming (WGP)*, 2013.

[23] Ken Madlener, Sjaak Smetsers, and Marko Eekelen. Modular Bialgebraic Semantics and Algebraic Laws. In AndréRauber Bois and Phil Trinder, editors, *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, page 46–60. Springer Berlin Heidelberg, 2013.

[24] Conor McBride. W-types: good news and bad news (post on epigram blog). `https://web.archive.org/web/20100918110849/http://www.e-pig.org/epilogue/?p=324.`

[25] Conor McBride. Kleisli arrows of outrageous fortune, 2011. `https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf`.

[26] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69--111, January 2004.

[27] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413--424, 1992.

[28] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire, 1991.

[29] Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

[30] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[31] Kent Petersson and Dan Synek. A set constructor for inductive sets in Martin-Löf's type theory. In DavidH. Pitt, DavidE. Rydeheard, Peter Dybjer, AndrewM. Pitts, and Axel Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 128--140. Springer Berlin Heidelberg, 1989.

[32] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[33] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 3--12, New York, NY, USA, 2013. ACM.

[34] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

[35] Philip Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.