Monadically Making Modules

3-for-1 Monadic Notation: Do-it-yourself module types

MUSA AL-HASSY, JACQUES CARETTE, WOLFRAM KAHL

Can parameterised records and algebraic datatypes be derived from one pragmatic declaration?

Record types give a universe of discourse, parameterised record types fix parts of that universe ahead of time, and algebraic datatypes give us first-class syntax, whence evaluators and optimisers.

The answer is in the affirmative. Besides a practical shared declaration interface, which is extensible in the language, we also find that common data structures correspond to simple theories.

ACM Reference Format:

Musa Al-hassy, Jacques Carette, Wolfram Kahl. 2020. Monadically Making Modules 3-for-1 Monadic Notation: Do-it-yourself module types. 1, 1 (March 2020), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We routinely write algebraic datatypes to provide a first-class syntax for record values. We work with semantic values, but need syntax to provide serialisation and introspection capabilities. A concept is thus rendered twice, once at the semantic level using records and again at the syntactic level using algebraic datatypes. Even worse, there is usually a need to expose fields of a record at the type level and so yet another variation of the same concept needs to be written. Our idea is to unify the various type declarations into one —using monadic do-notation and in-language meta-programming combinators to then extract possibly parameterised records and algebraic data types.

For example, there are two ways to implement the type of graphs in the dependently-typed language Agda [4, 11]: Having the vertices be a parameter or having them be a field of the record. Then there is also the syntax for graph vertex relationships.

```
record Graph₀ : Set₁ where
  constructor ⟨_,_⟩₀
```

Author's address: Musa Al-hassy, Jacques Carette, Wolfram Kahl.

Manuscript submitted to ACM

```
  field
    Vertex : Set
    Edges : Vertex → Vertex → Set

record Graph₁ (Vertex : Set) : Set₁ where
  constructor ⟨_⟩₁
  field
    Edges : Vertex → Vertex → Set

data Graph (Vertex : Set) : Set where
  ⟨_,_⟩ₛ : Vertex → Vertex → Graph Vertex
```

To illustrate the difference of the first two, consider the function `comap`, which relabels the vertices of a graph, using a function `f` to transform vertices:

```
comap₀ : {A B : Set}
       → (f : A → B)
       → (Σ G : Graph₀ • Vertex G ≡ B)
       → (Σ H : Graph₀ • Vertex H ≡ A)
comap₀ {A} f (G , refl) = ⟨ A , (λ x y → Edges G (f x) (f y)) ⟩₀ , refl

comap₁ : {A B : Set}
       → (f : A → B)
       → Graph₁ B
       → Graph₁ A
comap₁ f ⟨ edges ⟩₁ = ⟨ (λ x y → edges (f x) (f y)) ⟩₁
```

In $\mathtt{comap}_0$, the input graph `G` and the output graph `H` have their vertex sets constrained to match the type of the relabelling function `f`. Without the constraints, we could not even write the function for $\mathtt{Graph}_0$. With such an importance, it is surprising to see that the occurrences of the constraint proofs are uninsightful `refl`-exivity proofs. In contrast, $\mathtt{comap}_1$ does not carry any excesses baggage at the type level nor at the implementation level.

We will show an automatic technique for obtaining the above three definitions of graphs from a single declaration using similar notation. Our contributions are to show:

(1) Languages with sufficiently powerful type systems and meta-programming can conflate record and termtype declarations into one practical interface. We identify the problem and the subtleties in shifting between representations in Section 2.
(2) Parameterised records can be obtained on-demand from non-parameterised records (Section 3).

- As with $\mathtt{Graph}_0$, the traditional [7] approach to unbundling a record requires the use of transport along propositional equalities, with trivial `refl`-exivity proofs. In Section 3, we develop a combinator, `_:waist_`, which removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

(3) Programming with fixed-points of unary type constructors can be made as simple as programming with termtypes (Section 4).

- Astonishingly, we mechanically regain ubiquitous data structures such as `ℕ, Maybe, List` as the termtypes of simple pointed and monoidal theories.

As an application, in Section 5 we show that the resulting setup applies as a semantics for a declarative pre-processing tool that accomplishes the above tasks.

2 THE PROBLEMS

There are a number of problems, with the number of parameters being exposed being the pivotal concern. To exemplify the distinctions at the type level as more parameters are exposed, consider the following approaches to formalising a dynamical system —a collection of states, a designated start state, and a transition function.

```
record DynamicSystem₀ : Set₁ where
  field
    States : Set
    start  : States
    next   : States → States

record DynamicSystem₁ (States : Set) : Set where
  field
    start : States
    next  : States → States

record DynamicSystem₂ (States : Set) (start : States) : Set where
  field
    next : States → States
```

Each $\mathtt{DynamicSystem}_i$ is a type constructor of `i`-many arguments; but it is the types of these constructors that provide insight into the sort of data they contain:

Type	Kind
$\mathtt{DynamicSystem}_0$	$\mathtt{Set}_1$
$\mathtt{DynamicSystem}_1$	`Π X : Set • Set`
$\mathtt{DynamicSystem}_2$	`Π X : Set • Π x : X • Set`

We shall refer to the concern of moving from a record to a parameterised record as **the unbundling problem** [5]. For example, moving from the *type* $\mathtt{Set}_1$ to the *function type* `Π X : Set • Set` gets us from $\mathtt{DynamicSystem}_0$ to something resembling $\mathtt{DynamicSystem}_1$, which we arrive at if we can obtain a *type constructor* `λ X : Set • ⋯`. We shall refer to the latter change as *reification* since the result is more concrete, it can be applied; it will be denoted by Π→λ. To clarify this subtlety, consider the following forms of the polymorphic identity function. Notice that $\mathtt{id}_i$ *exposes* i-many details at the type level to indicate the sort it consists of. However, notice that $\mathtt{id}_0$ is a type of functions whereas $\mathtt{id}_1$ is a function on types. Indeed, the latter two are derived from the first one: $\mathtt{id}_{i+1}$ = Π→λ $\mathtt{id}_i$ —this is proven by reflexivity in the appendices.

```
id₀ : Set₁
id₀ = Π X : Set • Π e : X • X

id₁ : Π X : Set • Set
id₁ = λ (X : Set) → Π e : X • X

id₂ : Π X : Set • Π e : X • Set
id₂ = λ (X : Set) (e : X) → X
```

Of-course, there is also the need for descriptions of values, which leads to the following termtypes. We shall refer to the shift from record types to algebraic data types as **the termtype problem**.

```
data DSTerms₀ : Set where
  start : DSTerms₀
  next  : DSTerms₀ → DSTerms₀

data DSTerms₁ (States : Set) : Set where
  start : States → DSTerms₁ States
  next  : DSTerms₁ States → DSTerms₁ States

data DSTerms₂ (States : Set) (start : States) : Set where
  next : DSTerms₂ States start → DSTerms₂ States start
```

Table 1. Contexts embody all kinds of grouping mechanisms

Concept	Concrete Syntax	Description
Context	`do S ← Set; s ← S; n ← (S → S); End`	"name-type pairs"
Record Type	`Σ S : Set • Σ s : S • Σ n : S → S • 1`	"bundled-up data"
Function Type	`Π S • Σ s : S • Σ n : S → S • 1`	"a type of functions"
Type constructor	`λ S • Σ s : S • Σ n : S → S • 1`	"a function on types"
Algebraic datatype	data D : Set where s : D; n : $D \to D$	"a descriptive syntax"

Our aim is to obtain all of these notions —of ways to group data together— from a single user-friendly context declaration, using monadic notation.

3 MONADIC NOTATION

There is little use in an idea that is difficult to use in practice. As such, we conflate records and termtypes by starting with an ideal syntax they would share, then derive the necessary artefacts that permit it. Our choice of syntax is monadic do-notation [10?]:

```
DynamicSystem : Context ℓ₁
DynamicSystem = do X ← Set
                   z ← X
                   s ← (X → X)
                   End
```

Here `Context`, `End`, and the underlying monadic bind operator are unknown. Since we want to be able to *expose* a number of fields at will, we may take `Context` to be types indexed by a number denoting exposure. Moreover, since records are a product type, we expect there to be a recursive definition whose base case will be the essential identity of products, the unit type 1.

Exposure	Elaboration
0	Σ X : Set • Σ z : X • Σ s : (X → X) • 1
1	Π X : Set • Σ z : X • Σ s : (X → X) • 1
2	Π X : Set • Π z : X • Σ s : (X → X) • 1
3	Π X : Set • Π z : X • Π s : (X → X) • 1

With these elaborations of `DynamicSystem` to guide the way, we resolve two of our unknowns.

```
{- “Contexts” are exposure-indexed types -}
Context = λ ℓ → ℕ → Set ℓ

{- Every type is a context -}
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

{- The “empty context” is the unit type -}
End : ∀ {ℓ} → Context ℓ
End = ‘ 1
```

It remains to identify the definition of the underlying bind operation >>=. Classically, for a type constructor `m`, bind is typed `∀ {X Y : Set} → m X → (X → m Y) → m Y`. It allows one to “extract an `X`-value for later use” in the `m Y` context. Since our `m = Context` is from levels to types, we need to slightly alter bind’s typing.

```
_>>=_ : ∀ {a b}
      → (Γ : Context a)
      → (∀ {n} → Γ n → Context b)
      → Context (a ⊎ b)
(Γ >>= f) ℕ.zero  = Σ γ : Γ 0 • f γ 0
(Γ >>= f) (suc n) = (γ : Γ n) → f γ n
```

The definition here accounts for the current exposure index: If zero, we have *record types*, otherwise *function types*. Using this definition, the above dynamical system context would need to be expressed using the lifting quote operation.

```
‘ Set >>= λ X → ‘ X >>= λ z → ‘ (X → X) >>= End
{- or -}
do X ← ‘ Set
   z ← ‘ X
   s ← ‘ (X → X)
   End
```

Interestingly [3, 8], use of do-notation in preference to bind, >>=, was suggested by John Launchbury in 1993 and was first implemented by Mark Jones in Gofer. Anyhow, with our goal of practicality in mind, we shall “build the lifting quote into the definition” of bind:

```
_>>=_ : ∀ {a b}
      → (Γ : Set a)   -- Main difference
      → (Γ → Context b)
      → Context (a ⊎ b)
(Γ >>= f) ℕ.zero  = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = (γ : Γ) → f γ n
```

With this definition, the above declaration `DynamicSystem` typechecks. However, `DynamicSystem` $i \ncong$ `DynamicSystem`$_i$, instead `DynamicSystem` i are “factories”: Given i-many arguments, a product value is formed. What if we want to *instantiate* some of the factory arguments ahead of time?

```
𝒩₀ : DynamicSystem 0  {- ≈ Σ X : Set  • Σ z : X  • Σ s : (X → X)  • 1 -}
𝒩₀ = ℕ , 0 , suc , tt

𝒩₁ : DynamicSystem 1 {- ≈ Π X : Set  • Σ z : X  • Σ s : (X → X)  • 1 -}
𝒩₁ = λ X → ??? {- Impossible to complete if X is empty! -}

{- “Instantiaing” X to be ℕ in “DynamicSystem 1” -}
```

```
𝒩₁’ : let X = ℕ in Σ z : X  • Σ s : (X → X)  • 1
𝒩₁’ = 0 , suc , tt
```

It seems what we need is method, say Π→λ, that takes a Π-type and transforms it into a λ-expression. One could use a universe, an algebraic type of codes denoting types, to define Π→λ. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is not practical nor pragmatic.

As such, we are left with pattern matching on the language's type formation primitives as the only reasonable approach. The method Π→λ is thus a macro that acts on the syntactic term representations of types.

```
Π→λ (Π a : A • τ) = (λ a : A • τ)
{- One then extends this homomorphically over all possible term formers. -}
```

That is, we walk along the term tree replacing occurrences of Π with λ. For example,

```
  Π→λ (Π→λ (DynamicSystem 2))
≡{- Definition of DynamicSystem at exposure level 2 -}
  Π→λ (Π→λ (Π X : Set • Π s : X  • Σ n : X → X  • 1))
≡{- Definition of Π→λ -}
  Π→λ (λ X : Set • Π s : X  • Σ n : X → X  • 1)
≡{- Homomorphy of Π→λ -}
  λ X : Set • Π→λ (Π s : X  • Σ n : X → X  • 1)
≡{- Definition of Π→λ -}
≡ λ X : Set • λ s : X  • Σ n : X → X  • 1
```

For practicality, `_:waist_` is a macro acting on contexts that repeats Π→λ a number of times in order to lift a number of field components to the parameter level.

```
τ :waist n      = Π→λⁿ n (τ n)

Π→λⁿ 0         τ = τ
Π→λⁿ (n + 1) τ = Π→λⁿ n (Π→λ τ)
```

We can now "fix arguments ahead of time". Before such demonstration, we need to be mindful of our practicality goals: One declares a grouping mechanism with `do . . . End`, which in turn has its instance values constructed with `⟨ . . . ⟩`.

```
-- Expressions of the form “⋯ , tt” may now be written “⟨ ⋯ ⟩”
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → 1 {ℓ}
⟨⟩ = tt
```

```
⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × (1 {ℓ})
s ⟩ = s , tt
```

The following instances of grouping types demonstrate how information moves from the body level to the parameter level.

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0 suc
𝒩³ = ⟨⟩
```

Using `:waist` i we may fix the first i-parameters ahead of time. Indeed, the type `(DynamicSystem :waist 1) ℕ` is *the type of dynamic systems over carrier* ℕ, whereas `(DynamicSystem :waist 2) ℕ 0` is *the type of dynamic systems over carrier* ℕ *and start state 0.*

Examples of the need for such on-the-fly unbundling can be found in numerous places in the Haskell standard library. For instance, the standard libraries have two isomorphic copies of the integers, called `Sum` and `Prod`, whose reason for being is to distinguish two common monoids: The former is for *integers with addition* whereas the latter is for *integers with multiplication.* An orthogonal solution would be to use contexts:

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              Id      ← Carrier
              _⊕_     ← (Carrier → Carrier → Carrier)
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

With this context, (`Monoid` ℓ_0 `:waist 2) M` ⊕ is the type of monoids over *particular* types `M` and *particular* operations ⊕. Of-course, this is orthogonal, since traditionally unification on the carrier type `M` is what makes typeclasses and canonical structures [9] useful for ad-hoc polymorphism.

4 TERMTYPES AS FIXED-POINTS

We have a practical monadic syntax for possibly parameterised record types that we would like to extend to termtypes. Algebraic data types are a means to declare concrete representations of the least fixed-point of a functor.

In particular, the description language D for dynamical systems, **??**, declares concrete constructors for the fixpoint of `F`:

```
F : Set → Set
F = λ (D : Set) → D ⊎ D
```

That is, D `≅ Fix F` where:

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

The problem is whether we can derive `F` from `DynamicSystem`. Let us attempt a quick calculation.

```
  do X ← Set; z ← X; s ← (X → X); End
⟶{- Use existing interpretation to obtain a record. -}
  Σ X : Set • Σ z : X • Σ s : (X → X) • 1
⟶{- Pull out the carrier, “:waist 1”, to obtain a type constructor using “Π→λ”. -}
  λ X : Set • Σ z : X • Σ s : (X → X) • 1
⟶{- Termtype constructors target the declared type, so only their sources matter.
   E.g., ‘z : X’ is a nullary constructor targeting the carrier ‘X’.
   This introduces 1 types, so any existing occurances are dropped via 0.
 -}
  λ X : Set • Σ z : 1 • Σ s : X • 0
⟶{- Termtypes are sums of products. -}
  λ X : Set •       1    ⊎      X  ⊎ 0
⟶{- Termtypes are fixpoints of type constructors. -}
  Fix (λ X • 1 ⊎ X)  -- i.e., D
```

Listing 1. Guide to termtypes

Since we may view an algebraic data-type as a fixed-point of the functor obtained from the union of the sources of its constructors, it suffices to treat the fields of a record as constructors, then obtain their sources, then union them. That is, since algebraic-datatype constructors necessarily target the declared type, they are determined by their sources. For example, considered as a unary constructor `op : A → B` targets the type termtype `B` and so its source is `A`.

```
⇊ τ = “reduce all de brujin indices by 1”

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (⇊ Ba)

sources (λ x : (Π a : A • Ba) • τ) = (λ x : A • sources τ)
sources (λ x : A              • τ) = (λ x : 1 • sources τ)

{- Extend “sources, Σ→⊎” homomorphicly to other syntactic constructs -}

termtype τ = Fix (Σ→⊎ (sources τ))
```

It is instructive to visually see how D is obtained from `termtype` in order to demonstrate that this approach to algebraic data types is practical.

```
D = termtype (DynamicSystem :waist 1)

-- Pattern synonyms for more compact presentation
pattern startD   = μ (inj₁ tt)       -- : D
pattern nextD e = μ (inj₂ (inj₁ e)) -- : D → D
```

With the `pattern` declarations, we can actually use these more meaningful names, when pattern matching, instead of the seemingly daunting μ-`inj`-ections. For instance, we can immediately see that the natural numbers act as the description language for dynamical systems:

```
to : D → ℕ
to startD    = 0
to (nextD x) = suc (to x)

from : ℕ → D
from zero    = startD
from (suc n) = nextD (from n)
```

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if C `:` `Set → Context` ℓ_0 then C`’ = λ X → termtype (`C` X :waist 1)` can be used to form ‘free, lawless, C-instances’.

Table 2. Data strcutrues as free theories

Theory	Termtype
Dynamical Systems	ℕ
Pointed Structures	Maybe
Monoids	Binary Trees

The final entry in the table is a well known correspondence, that we can, not only formally express, but also prove to be true. We present the setup and leave it as an instructive exercise to the reader to present a bijective pair of functions between M and `TreeSkeleton`. Hint: Interactively case-split on values of M until the declared patterns appear, then associate them with the constructors of `TreeSkeleton`.

```
M : Set
M = termtype (Monoid ℓ₀ :waist 1)

-- Pattern synonyms for more compact presentation
pattern emptyM      = μ (inj₁ tt)                       -- : M
pattern branchM l r = μ (inj₂ (inj₁ (l , r , tt)))      -- : M → M → M
pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a))))  -- absurd values of 0

data TreeSkeleton : Set where
  empty  : TreeSkeleton
  branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

To obtain trees over some 'value type' Ξ, one must start at the theory of "monoids containing a given set Ξ". Similarly, by starting at "theories of pointed sets over a given set Ξ", the resulting termtype is the `Maybe` type constructor —another instructive exercise to the reader: Show P ≅ `Maybe`.

```
PointedOver   : Set → Context (ℓsuc ℓ₀)
PointedOver Ξ    = do Carrier ← Set ℓ₀
                      point   ← Carrier
                      embed   ← (Ξ → Carrier)
                      End

P : Set → Set
P X = termtype (PointedOver X :waist 1)

-- Pattern synonyms for more compact presentation
pattern nothingP = μ (inj₁ tt)        -- : P
pattern justP e  = μ (inj₂ (inj₁ e)) -- : P → P
```

5 RELATED WORKS

Surprisingly, conflating parameterised and non-parameterised record types with termtypes *within a language in a practical fashion* has not been done before.

The PackageFormer [1, 2] editor extension reads contexts —in nearly the same notation as ours— enclosed in dedicated comments, then generates and imports Agda code from them seamlessly in the background whenever typechecking transpires. The framework provides a fixed number of meta-primitives for producing arbitrary notions of grouping mechanisms, and allows arbitrary Emacs Lisp [6] to be invoked in the construction of complex grouping mechanisms.

Table 3. Comparing the in-language Context mechanism with the PackageFormer editor extension

	PackageFormer	Contexts
Type of Entity	Preprocessing Tool	Language Library
Specification Language	Lisp + Agda	Agda
Well-formedness Checking	✗	✓
Termination Checking	✓	✓
Elaboration Tooltips	✓	✗
Rapid Prototyping	✓	✓ (Slower)
Usability Barrier	None	None
Extensibility Barrier	Lisp	Weak Metaprogramming

The original PackageFormer paper provided the syntax necessary to form useful grouping mechanisms but was shy on the semantics of such constructs. We have chosen the names of our combinators to closely match those of PackageFormer's with an aim of furnishing the mechanism with semantics by construing the syntax as semantics-functions; i.e., we have a shallow embedding of PackageFormer's constructs as Agda entities:

Table 4. Contexts as a semantics for PackageFormer constructs

Syntax	Semantics
`PackageFormer`	`Context`
`:waist`	`:waist`
⟴	Forward function application
`:kind`	`:kind`, see below
`:level`	Agda built-in
`:alter-elements`	Agda macros

PackageFormer's `_:kind_` meta-primitive dictates how an abstract grouping mechanism should be viewed in terms of existing Agda syntax. However, unlike PackageFormer, all of our syntax consists of legitimate Agda terms. Since language syntax is being manipulated, we are forced to define it as a macro:

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind

𝒞 :kind ‘record    = 𝒞 0
```

```
𝒞 :kind ‘typeclass = 𝒞 :waist 1
𝒞 :kind ‘data      = termtype (𝒞 :waist 1)
```

We did not expect to be able to assign a full semantics to PackageFormer's syntactic constructs due to Agda's substantially weak metaprogramming mechanism. However, it is important to note that PackageFormer's Lisp extensibility expedites the process of trying out arbitrary grouping mechanisms —such as partial-choices of pushouts and pullbacks along user-provided assignment functions— since it is all either string or symbolic list manipulation. On the Agda side, using contexts, it would require exponentially more effort due to the limited reflection mechanism and the intrusion of the stringent type system.

6 CONCLUSION

Starting from the insight that related grouping mechanisms could be unified, we showed how related structures can be obtained from a single declaration using a practical interface. The resulting framework, based on contexts, still captures the familiar record declaration syntax as well as the expressivity of usual algebraic datatype declarations —at the minimal cost of using `pattern` declarations to aide as user-chosen constructor names. We believe that our approach to using contexts as general grouping mechanisms *with* a practical interface are interesting contributions.

We used the focus on practicality to guide the design of our context interface, and provided interpretations both for the rather intuitive "contexts are name-type records" view, and for the novel "contexts are fixed-points" view for termtypes. In addition, to obtain parameterised variants, we needed to explicitly form "contexts whose contents are over a given ambient context" —e.g., contexts of vector spaces are usually discussed with the understanding that there is a context of fields that can be referenced— which we did using monads.

To those interested in exotic ways to group data together —such as, mechanically deriving product types and homomorphism types of theories— we offer an interface that is extensible using Agda's reflection mechanism. In comparison with, for example, special-purpose preprocessing tools, this has obvious advantages in accessibility and semantics.

To Agda programmers, this offers a standard interface for grouping mechanisms that had been sorely missing, with an interface that is so familiar that there would be little barrier to its use. In particular, as we have shown, it acts as an in-language library for exploring relationships between free theories and data structures. As we have only presented the high-level definitions of the core combinators, leaving the Agda-specific details to the appendices, it is also straightforward to translate the library into other dependently-typed languages.

7 APPENDICES

Below is the entirety of the Context library discussed in the paper proper.

```
module Context where
```

.1 Imports

```
open import Level renaming (_⊔_ to _⊎_; suc to ℓsuc; zero to ℓ₀)
open import Relation.Binary.PropositionalEquality
open import Relation.Nullary

open import Data.Nat
open import Data.Fin  as Fin using (Fin)
open import Data.Maybe  hiding (_>>=_)

open import Data.Bool using (Bool ; true ; false)
open import Data.List as List using (List ; [] ; _∷_ ; _∷ʳ_; sum)

ℓ₁   = Level.suc ℓ₀
```

.2 Quantifiers Π:•/Σ:• and Products/Sums

We shall using Z-style quantifier notation [12] in which the quantifier dummy variables are separated from the body by a large bullet.

In Agda, we use \: to obtain the "ghost colon" since standard colon : is an Agda operator.

```
open import Data.Empty using (⊥)
open import Data.Sum
open import Data.Product
open import Function using (_∘_)

Σ:• : ∀ {a b} (A : Set a) (B : A → Set b) → Set _
Σ:• = Σ

infix -666 Σ:•
syntax Σ:• A (λ x → B) = Σ x : A • B

Π:• : ∀ {a b} (A : Set a) (B : A → Set b) → Set _
Π:• A B = (x : A) → B x

infix -666 Π:•
syntax Π:• A (λ x → B) = Π x : A • B

record ⊤ {ℓ} : Set ℓ where
```

```
  constructor tt

1 = ⊤ {ℓ₀}
0 = ⊥
```

.3 Reflection

We form a few metaprogramming utilities we would have expected to be in the standard library.

```
import Data.Unit as Unit
open import Reflection hiding (name; Type) renaming (_>>=_ to _>>=ₘ_)
```

.3.1 *Single argument application.*

```
_app_ : Term → Term → Term
(def f args) app arg' = def f (args ::ʳ arg (arg-info visible relevant) arg')
(con f args) app arg' = con f (args ::ʳ arg (arg-info visible relevant) arg')
{-# CATCHALL #-}
tm app arg' = tm
```

Notice that we maintain existing applications:

```
quoteTerm (f x) app quoteTerm y  ≈  quoteTerm (f x y)
```

.3.2 *Reify ℕ term encodings as ℕ values.*

```
toℕ : Term → ℕ
toℕ (lit (nat n)) = n
{-# CATCHALL #-}
toℕ _ = 0
```

.3.3 *The Length of a Term.*

```
arg-term : ∀ {ℓ} {A : Set ℓ} → (Term → A) → Arg Term → A
arg-term f (arg i x) = f x

{-# TERMINATING #-}
lengthₜ : Term → ℕ
lengthₜ (var x args)      = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (con c args)      = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (def f args)      = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (lam v (abs s x)) = 1 + lengthₜ x
lengthₜ (pat-lam cs args) = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (Π[ x : A ] Bx)    = 1 + lengthₜ Bx
```

```
{-# CATCHALL #-}
-- sort, lit, meta, unknown
lengthₜ t = 0
```

Here is an example use:

```
_ : lengthₜ (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
_ = refl
```

.3.4 Decreasing Debrujin Indices. Given a quantification (⊕ x : τ • fx), its body fx may refer to a free variable x. If we decrement all Debrujin indices fx contains, then there would be no reference to x.

```
var-dec₀ : (fuel : ℕ) → Term → Term
var-dec₀ zero t  = t
-- Let's use an “impossible” term.
var-dec₀ (suc n) (var zero args)      = def (quote ⊥) []
var-dec₀ (suc n) (var (suc x) args)   = var x args
var-dec₀ (suc n) (con c args)         = con c (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (def f args)         = def f (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (lam v (abs s x))    = lam v (abs s (var-dec₀ n x))
var-dec₀ (suc n) (pat-lam cs args)    = pat-lam cs (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec₀ n A) ] var-dec₀ n B
{-# CATCHALL #-}
-- sort, lit, meta, unknown
var-dec₀ n t = t
```

In the paper proper, var-dec was mentioned once under the name ⇊.

```
var-dec : Term → Term
var-dec t = var-dec₀ (lengthₜ t) t
```

Notice that we made the decision that x, the body of (⊕ x • x), will reduce to 0, the empty type. Indeed, in such a situation the only Debrujin index cannot be reduced further. Here is an example:

```
_ : ∀ {x : ℕ} → var-dec (quoteTerm x) ≡ quoteTerm ⊥
_ = refl
```

.4 Context Monad

```
Context = λ ℓ → ℕ → Set ℓ

infix -1000 ‘_
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S
```

```
End : ∀ {ℓ} → Context ℓ
End = ‘ ⊤

End₀ = End {ℓ₀}

_>>=_ : ∀ {a b}
      → (Γ : Set a)  -- Main diference
      → (Γ → Context b)
      → Context (a ⊍ b)
(Γ >>= f) ℕ.zero  = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = (γ : Γ) → f γ n
```

.5 ⟨⟩ Notation

As mentioned, grouping mechanisms are declared with do . . . End, and instances of them are constructed using ⟨ . . . ⟩.

```
-- Expressions of the form “··· , tt” may now be written “⟨ ··· ⟩”
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → ⊤ {ℓ}
⟨⟩ = tt

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × ⊤ {ℓ}
s ⟩ = s , tt
```

.6 DynamicSystem Context

```
DynamicSystem : Context (ℓsuc Level.zero)
DynamicSystem = do X ← Set
                   z ← X
                   s ← (X → X)
                   End {Level.zero}

-- Records with n-Parameters, n : 0..3
A B C D : Set₁
```

```
A = DynamicSystem 0 -- Σ X : Set  • Σ z : X  • Σ s : X → X  • ⊤
B = DynamicSystem 1 --  (X : Set) → Σ z : X  • Σ s : X → X  • ⊤
C = DynamicSystem 2 --  (X : Set)    (z : X) → Σ s : X → X  • ⊤
D = DynamicSystem 3 --  (X : Set)    (z : X) →  (s : X → X) → ⊤

_ : A ≡ (Σ X : Set  • Σ z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : B ≡ (Π X : Set  • Σ z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : C ≡ (Π X : Set  • Π z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : D ≡ (Π X : Set  • Π z : X  • Π s : (X → X)  • ⊤) ; _ = refl

stability : ∀ {n} →   DynamicSystem (3 + n)
                    ≡ DynamicSystem  3
stability = refl

B-is-empty : ¬ B
B-is-empty b = proj₁( b ⊥)

𝒩₀ : DynamicSystem 0
𝒩₀ = ℕ , 0 , suc , tt

𝒩 : DynamicSystem 0
𝒩 = ⟨ ℕ , 0 , suc ⟩

B-on-ℕ : Set
B-on-ℕ = let X = ℕ in Σ z : X  • Σ s : (X → X)  • ⊤

ex : B-on-ℕ
ex = ⟨ 0 , suc ⟩
```

.7 Π→λ

```
Π→λ-helper : Term → Term
Π→λ-helper (pi  a b)         = lam visible b
Π→λ-helper (lam a (abs x y)) = lam a (abs x (Π→λ-helper y))
{-# CATCHALL #-}
Π→λ-helper x = x

macro
```

```
  Π→λ : Term → Term → TC Unit.⊤
  Π→λ tm goal = normalise tm >>=ₘ λ tm' → unify (Π→λ-helper tm') goal
```

.8 _:waist_

```
waist-helper : ℕ → Term → Term
waist-helper zero t    = t
waist-helper (suc n) t = waist-helper n (Π→λ-helper t)

macro
  _:waist_ : Term → Term → Term → TC Unit.⊤
  _:waist_ t n goal =      normalise (t app n)
                      >>=ₘ λ t' → unify (waist-helper (toℕ n) t') goal
```

.9 DynamicSystem :waist i

```
A' : Set₁
B' : ∀ (X : Set) → Set
C' : ∀ (X : Set) (x : X) → Set
D' : ∀ (X : Set) (x : X) (s : X → X) → Set

A' = DynamicSystem :waist 0
B' = DynamicSystem :waist 1
C' = DynamicSystem :waist 2
D' = DynamicSystem :waist 3

𝒩⁰ : A'
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : B' ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : C' ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : D' ℕ 0 suc
𝒩³ = ⟨⟩
```

It may be the case that Γ `0` ≡ Γ `:waist 0` for every context Γ.

```
_ : DynamicSystem 0 ≡ DynamicSystem :waist 0
_ = refl
```

.10 Field projections

```
Field₀ : ℕ → Term → Term
Field₀ zero c    = def (quote proj₁) (arg (arg-info visible relevant) c :: [])
Field₀ (suc n) c = Field₀ n (def (quote proj₂) (arg (arg-info visible relevant) c :: []))

macro
  Field : ℕ → Term → Term → TC Unit.⊤
  Field n t goal = unify goal (Field₀ n t)
```

.11 Termtypes

Using the guide, ??, outlined in the paper proper we shall form $\mathtt{D}_i$ for each stage in the calculation.

.11.1 Stage 1: Records.

```
D₁ = DynamicSystem 0

1-records : D₁ ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤)
1-records = refl
```

.11.2 Stage 2: Parameterised Records.

```
D₂ = DynamicSystem :waist 1

2-funcs : D₂ ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • ⊤)
2-funcs = refl
```

.11.3 Stage 3: Sources. Let's begin with an example to motivate the definition of `sources`.

```
_ :   quoteTerm (∀ {x : ℕ} → ℕ)
    ≡ pi (arg (arg-info hidden relevant) (quoteTerm ℕ)) (abs "x" (quoteTerm ℕ))
_ = refl
```

We now form two sources-helper utilities, although we suspect they could be combined into one function.

```
sources₀ : Term → Term
-- Otherwise:
sources₀ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) =
    def (quote _×_) (vArg A
                :: vArg (def (quote _×_)
```

```
                        (vArg (var-dec Ba) :: vArg (var-dec (var-dec (sources₀ Cab))) :: []))
            :: [])
sources₀ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 0
sources₀ (Π[ x : arg i A ] Bx) = A
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₀ t = quoteTerm 1

{-# TERMINATING #-}
sources₁ : Term → Term
sources₁ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 0
sources₁ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) = def (quote _×_) (vArg A ::
  vArg (def (quote _×_) (vArg (var-dec Ba) :: vArg (var-dec (var-dec (sources₀ Cab))) :: [])) :: [])
sources₁ (Π[ x : arg i A ] Bx) = A
sources₁ (def (quote Σ) (ℓ₁ :: ℓ₂ :: τ :: body))
    = def (quote Σ) (ℓ₁ :: ℓ₂ :: map-Arg sources₀ τ :: List.map (map-Arg sources₁) body)
-- This function introduces 1s, so let's drop any old occurances a la 0.
sources₁ (def (quote ⊤) _) = def (quote 0) []
sources₁ (lam v (abs s x))      = lam v (abs s (sources₁ x))
sources₁ (var x args) = var x (List.map (map-Arg sources₁) args)
sources₁ (con c args) = con c (List.map (map-Arg sources₁) args)
sources₁ (def f args) = def f (List.map (map-Arg sources₁) args)
sources₁ (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources₁) args)
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₁ t = t
```

We now form the macro and some unit tests.

```
macro
  sources : Term → Term → TC Unit.⊤
  sources tm goal = normalise tm >>=ₘ λ tm' → unify (sources₁ tm') goal

_ : sources (ℕ → Set) ≡ ℕ
_ = refl

_ : sources (Σ x : (ℕ → Fin 3) • ℕ) ≡ (Σ x : ℕ • ℕ)
_ = refl
```

```
_ : ∀ {ℓ : Level} {A B C : Set}
  → sources (Σ x : (A → B) • C) ≡ (Σ x : A • C)
_ = refl

_ : sources (Fin 1 → Fin 2 → Fin 3) ≡ (Σ _ : Fin 1 • Fin 2 × 1)
_ = refl

_ : sources (Σ f : (Fin 1 → Fin 2 → Fin 3 → Fin 4) • Fin 5)
  ≡ (Σ f : (Fin 1 × Fin 2 × Fin 3) • Fin 5)
_ = refl

_ : ∀ {A B C : Set} → sources (A → B → C) ≡ (A × B × 1)
_ = refl

_ : ∀ {A B C D E : Set} → sources (A → B → C → D → E)
                          ≡ Σ A (λ _ → Σ B (λ _ → Σ C (λ _ → Σ D (λ _ → ⊤))))
_ = refl
```

Design descision: Types starting with implicit arguments are *invariants*, not *constructors*.

```
-- one implicit
_ : sources (∀ {x : ℕ} → x ≡ x) ≡ 0
_ = refl

-- multiple implicits
_ : sources (∀ {x y z : ℕ} → x ≡ y) ≡ 0
_ = refl
```

The third stage can now be formed.

```
D₃ = sources D₂

3-sources : D₃ ≡ λ (X : Set) → Σ z : 1 • Σ s : X • 0
3-sources = refl
```

.11.4 *Stage 4: Σ→⊎ –Replacing Products with Sums.*

```
{-# TERMINATING #-}
Σ→⊎₀ : Term → Term
Σ→⊎₀ (def (quote Σ) (h₁ :: h₀ :: arg i A :: arg i₁ (lam v (abs s x)) :: []))
```

```
  =  def (quote _⊎_) (h₁ :: h₀ :: arg i A :: vArg (Σ→⊎₀ (var-dec x)) :: [])
-- Interpret “End” in do-notation to be an empty, impossible, constructor.
Σ→⊎₀ (def (quote ⊤) _) = def (quote ⊥) []
 -- Walk under λ's and Π's.
Σ→⊎₀ (lam v (abs s x)) = lam v (abs s (Σ→⊎₀ x))
Σ→⊎₀ (Π[ x : A ] Bx) = Π[ x : A ] Σ→⊎₀ Bx
{-# CATCHALL #-}
Σ→⊎₀ t = t

macro
  Σ→⊎ : Term → Term → TC Unit.⊤
  Σ→⊎ tm goal = normalise tm >>=ₘ λ tm’ → unify (Σ→⊎₀ tm’) goal

-- Unit tests
_ : Σ→⊎ (Π X : Set • (X → X))      ≡ (Π X : Set • (X → X)); _ = refl
_ : Σ→⊎ (Π X : Set • Σ s : X • X) ≡ (Π X : Set • X ⊎ X)  ; _ = refl
_ : Σ→⊎ (Π X : Set • Σ s : (X → X) • X) ≡ (Π X : Set • (X → X) ⊎ X)  ; _ = refl
_ : Σ→⊎ (Π X : Set • Σ z : X • Σ s : (X → X) • ⊤ {ℓ₀}) ≡ (Π X : Set • X ⊎ (X → X) ⊎ ⊥)  ; _ = refl

D₄ = Σ→⊎ D₃

4-unions : D₄ ≡ λ X → 1 ⊎ X ⊎ 0
4-unions = refl
```

.11.5 Stage 5: Fixpoint and proof that $D \cong \mathbb{N}$.

```
{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
  μ : F (Fix F) → Fix F

D = Fix D₄

-- Pattern synonyms for more compact presentation
pattern zeroD  = μ (inj₁ tt)        -- : D
pattern sucD e = μ (inj₂ (inj₁ e)) -- : D → D

to : D → ℕ
to zeroD     = 0
```

```
to (sucD x) = suc (to x)

from : ℕ → D
from zero    = zeroD
from (suc n) = sucD (from n)

to∘from : ∀ n → to (from n) ≡ n
to∘from zero    = refl
to∘from (suc n) = cong suc (to∘from n)

from∘to : ∀ d → from (to d) ≡ d
from∘to zeroD    = refl
from∘to (sucD x) = cong sucD (from∘to x)
```

.11.6 `termtype` *and* `Inj` *macros.* We summarise the stages together into one macro.

```
macro
  termtype : Term → Term → TC Unit.⊤
  termtype tm goal =
                normalise tm
          >>=ₘ λ tm' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sources₁ tm'))) :: []))
```

It is interesting to note that in place of `pattern` clauses, say for languages that do not support them, we would resort to "fancy injections".

```
Inj₀ : ℕ → Term → Term
Inj₀ zero c    = con (quote inj₁) (arg (arg-info visible relevant) c :: [])
Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) :: [])

-- Duality!
-- i-th projection: proj₁ ∘ (proj₂ ∘ ⋯ ∘ proj₂)
-- i-th injection:  (inj₂ ∘ ⋯ ∘ inj₂) ∘ inj₁

macro
  Inj : ℕ → Term → Term → TC Unit.⊤
  Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

With this alternative, we regain the "user chosen constructor names" for D:

```
startD : D
startD = Inj 0 (tt {ℓ₀})
```

```
nextD' : D → D
nextD' d = Inj 1 d
```

.12 Monoids

.12.1 Context.

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              Id      ← Carrier
              _⊕_     ← (Carrier → Carrier → Carrier)
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

.12.2 Termtypes.

```
M : Set
M = termtype (Monoid ℓ0 :waist 1)
{- ie Fix (λ X → 1           -- Id, nil leaf
              ⊎ X × X × 1 -- _⊕_, branch
              ⊎ 0          -- src of leftId
              ⊎ 0          -- src of rightId
              ⊎ X × X × 0 -- src of assoc
              ⊎ 0)         -- the “End {ℓ}”
-}

-- Pattern synonyms for more compact presentation
pattern emptyM      = μ (inj1 tt)                      -- : M
pattern branchM l r = μ (inj2 (inj1 (l , r , tt)))     -- : M → M → M
pattern absurdM a   = μ (inj2 (inj2 (inj2 (inj2 a))))  -- absurd values of 0

data TreeSkeleton : Set where
  empty  : TreeSkeleton
  branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

.12.3 M ≅ TreeSkeleton.

```
M→Tree : M → TreeSkeleton
M→Tree emptyM = empty
M→Tree (branchM l r) = branch (M→Tree l) (M→Tree r)
M→Tree (absurdM (inj₁ ()))
M→Tree (absurdM (inj₂ ()))

M←Tree : TreeSkeleton → M
M←Tree empty = emptyM
M←Tree (branch l r) = branchM (M←Tree l) (M←Tree r)

M←Tree∘M→Tree : ∀ m → M←Tree (M→Tree m) ≡ m
M←Tree∘M→Tree emptyM = refl
M←Tree∘M→Tree (branchM l r) = cong₂ branchM (M←Tree∘M→Tree l) (M←Tree∘M→Tree r)
M←Tree∘M→Tree (absurdM (inj₁ ()))
M←Tree∘M→Tree (absurdM (inj₂ ()))

M→Tree∘M←Tree : ∀ t → M→Tree (M←Tree t) ≡ t
M→Tree∘M←Tree empty = refl
M→Tree∘M←Tree (branch l r) = cong₂ branch (M→Tree∘M←Tree l) (M→Tree∘M←Tree r)
```

.13 :kind

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind

{- Nope: Since :waist may return type constructors, not sets!
_:kind_ : ∀ {ℓ} → Context ℓ → Kind → Set ℓ
C :kind ‘record    = C :waist 0
C :kind ‘typeclass = C :waist 1
C :kind ‘data      = termtype (C :waist 1)
-}
macro
  _:kind_ : Term → Term → Term → TC Unit.⊤
  _:kind_ t (con (quote ‘record) _)    goal = normalise (t app (quoteTerm 0))
                        >>=ₘ λ t’ → unify (waist-helper 0 t’) goal
  _:kind_ t (con (quote ‘typeclass) _) goal = normalise (t app (quoteTerm 1))
```

```
                         >>=ₘ λ t' → unify (waist-helper 1 t') goal
  _:kind_ t (con (quote ‘data) _) goal = normalise (t app (quoteTerm 1))
                         >>=ₘ λ t' → normalise (waist-helper 1 t')
                         >>=ₘ λ t'' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sources₁ t''))) :: []))
  _:kind_ t _ goal = unify t goal

-- _⟴_ : ∀ {a b} {A : Set a} {B : Set b} → A → (A → B) → B
-- x ⟴ f = f x
```

REFERENCES

[1] Musa Al-hassy. The next 700 module systems: Extending dependently-typed languages to implement module system features in the core language, 2019. URL https://alhassy.github.io/next-700-module-systems-proposal/thesis-proposal.pdf.

[2] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. A language feature to unbundle data at will (short paper). In Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, pages 14–19. ACM, 2019. ISBN 978-1-4503-6980-0. doi: 10.1145/3357765.3359523. URL https://doi.org/10.1145/3357765.3359523.

[3] Richard Bird. Thinking functionally with haskell. 2009. doi: 10.1017/cbo9781316092415. URL http://dx.doi.org/10.1017/cbo9781316092415.

[4] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — A functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings*, pages 73–78, 2009. doi: 10.1007/978-3-642-03359-9_6.

[5] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer. URL https://hal.inria.fr/inria-00368403.

[6] Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, USA, 1995. ISBN 0133708756.

[7] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq, 2014.

[8] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.

[9] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer. doi: 10.1007/978-3-642-39634-2_5. URL https://hal.inria.fr/hal-00816703.

[10] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4. URL https://doi.org/10.1016/0890-5401(91)90052-4.

[11] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, September 2007.

[12] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., USA, 1996. ISBN 0139484728.