Modular Type-Safety Proofs in Agda

Christopher Schwaab Jeremy G. Siek
University of Colorado at Boulder
{schwaab,jeremy.siek}@colorado.edu

Abstract

Methods for reusing code are widespread and well researched, but methods for reusing proofs are still emerging. We consider the use of dependent types for this purpose, introducing a modular approach for composing mechanized proofs. We show that common techniques for abstracting algorithms over data structures naturally translate to abstractions over proofs. We introduce a language composed of a series of smaller language components, each defined as functors, and tie them together by taking the fixed point of their sum [Malcom, 1990]. We then give proofs of type preservation for each language component and show how to compose these proofs into a proof for the entire language, again by taking the fixed point of a sum of functors.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Extensible Languages; D.3.2 [*Formal Definitions and Theory*]: Semantics

General Terms Languages, Theory

Keywords the expression problem, Agda, modularity, meta-theory, dependent types, type-safety

1. Introduction

The POPLmark challenge is a set of common programming language problems meant to test the utility of modern proof assistants and techniques for mechanized metatheory. In an effort to ease the mechanization of metatheory of programming languages, especially regarding variable binding [2], we've made great strides. However, little progress has been made in the direction of modularity: it is still difficult to separately develop the definitions and meta-theory of language fragments.

Dependent types have formed the foundation of a broad range of type systems that allow freely mixed types and values and types. Programmers can express propositions as types or sets, and proofs as programs that produce inhabitants of those sets; and when searching for general solutions to theorem construction the style suggests the application of familiar programming abstractions. Rather than relying on semi-automated proof search such as Coq's Ltac we propose a method of proof composition using simple abstractions whereby language components are defined piecewise and tied together at the end using a wrapper datatype acting as a tagged union.

We use a modular approach to define a programming language in which components of the language are defined separately from one another and are composed along with their proofs.

The language we present is one of simple expressions implemented in the proof assistant Agda. Agda is a dependently typed language with a logical foundation based on Martin Löf type theory. We begin by defining a series of language syntaxes for addition, options, and arrays. Arrays are included because not only can they result in runtime errors, which requires the *Option* type, but like addition, they use the natural numbers, forcing consideration of how value types can be shared across otherwise isolated components. We continue by defining a (small-step) reduction semantics and type system. The language is defined piecewise, each component is built in isolation alongside a proof of type preservation. We conclude by composing these components into a proof of type preservation for the combined language. Our technique is drawn from a solution to the expression problem where languages are defined as the disjoint sum of smaller languages defined using parameterized recursion. We show that this idea can be recast from types and terms, to proofs.

2. Review of the Expression Problem

Extending both data structures and the functions that operate on them in a modular fashion is challenging, this is sometimes referred to as *the expression problem*. In most functional languages, it is easy to add functions that operate on existing data structures but it is difficult to extend a data type with new constructors. On the other hand, in object-oriented languages, it is easy to extend data structures by subclassing, but it is difficult to add new functions to existing classes.

While many solutions to the expression problem have been proposed over the years, here we make use of the method described by Malcom [9] which generalizes recursion operators such as fold from lists to polynomial types. The expression problem in functional languages arises as a result of algebraic data types being *closed*: once the type has been declared, no new constructors for the type may be added without amending the original declaration. Malcom's solution is to remove immediate recursion and split a monolithic datatype into parameterized components that can later be collected under the umbrella of a disjoint sum (i.e., a tagged union).

Throughout this paper we work with a simple language over natural numbers and basic arithmetic operators. In

PLPV'13, January 22, 2013, Rome, Italy.

Agda, new algebraic data types are introduced with the **data** facility. The basic type of types is Set, whose type is Set_1, which in turn has type Set_2, and so on. We might first consider a simple but familiar looking datatype over sums of natural numbers

```
data Expr+ : Set where
  atom : ℕ → Expr+
  _+_ : Expr+ → Expr+ → Expr+
```

The two underscores in the above definition introduce an infix operator, with each underscore representing the location of a parameter. Agda generalizes this notation by allowing mixfix operators that take arbitrarily many parameters.

The definition of Expr+ has the advantage of being direct and simple, however a problem lies within the explicit recursion; to later extend expressions with arrays and option types, we can make no reuse of Expr+ due to the closed nature of algebraic data types. Instead, extending Expr+ requires the declaration of a new data type as in the following definition of MonolithicExpr.

```
data MonolithicExpr : Set where
  atom : ℕ → MonolithicExpr
  esome : MonolithicExpr → MonolithicExpr
  enone : MonolithicExpr
  nil [] : MonolithicExpr
  _!!_ : MonolithicExpr → MonolithicExpr
    → MonolithicExpr
  _[_]:=_ : MonolithicExpr → MonolithicExpr
    → MonolithicExpr → MonolithicExpr
  _+_ : MonolithicExpr → MonolithicExpr
    → MonolithicExpr
fromExpr+ : Expr+ → MonolithicExpr
fromExpr+ (atom n) = atom n
fromExpr+ (n + m) = fromExpr+ n + fromExpr+ m
```

Suppose instead we begin with a variant of Expr+, let's call it Expr+$_2$, that does not recursively refer to itself. Instead Expr+$_2$ is parameterized by the type variable A and the recursive references in the constructor parameters are replaced by A.

```
data Expr+₂ (A : Set) : Set where
  _+_ : A → A → Expr+₂ A
  atom : ℕ → Expr+₂ A
```

Continuing in this way we similarly treat the remaining language fragments.

```
data Expr [] ₂ (A : Set) : Set where
  nil [] : Expr [] ₂ A
  _!!_ : A → A → Expr [] ₂ A
  _[_]:=_ : A → A → A → Expr [] ₂ A
data ExprOption (A : Set) : Set where
  esome : A → ExprOption A
  enone : ExprOption A
```

Recursion is reintroduced by combining components as a disjoint sum, written $- \uplus -$ in Agda.

```
data RecExpr : Set where
  expr : Expr+₂ RecExpr
    ⊎ Expr [] ₂ RecExpr
    ⊎ ExprOption RecExpr
    → RecExpr
```

More generally, this type of data can be captured using a "categorical approach" where recursion is introduced as the fixed point of a functor. This uniformity of shape not only allows us to avoid boilerplate but, in a proof assistant like Agda, can give nice termination properties for free. There is a great deal of interesting work on modularity of inductive types and transformations over "regular" descriptions of data [4, 6, 10, 11].

In a language like Agda where datatypes are required to be strictly positive, we must restrict the set of functors before finding an admissible definition. We proceed by describing a restricted universe where datatypes are coded as *polynomial functors* over which we can safely introduce a type of *least fixed points*.

2.1 Functors and Agda

Functors are a special mapping defined over both types and functions satisfying the so called *functor laws*; a functor F

1. assigns to each type A, a type $F\,A$
2. assigns to each function $f : A \to B$, a function $\mathrm{map}\,f : FA \to FB$

such that

1. identity is preserved: $\mathrm{map}\,\mathrm{id} = \mathrm{id}$, and
2. when $f \circ g$ is defined: $\mathrm{map}\,(f \circ g) = \mathrm{map}\,f \circ \mathrm{map}\,g$.

One familiar example is the *List* functor mapping each type A to $List\,A$ and each function $f : A \to B$ to the function $\mathrm{map}\,f : List\,A \to List\,B$ which applies f to each element of a list. Here we are concerned with the least fixed point over a restricted class of functors called the *polynomial functors*. Polynomial functors are functors built up from the constant, identity, sum, and product functors. To review, the action of these functors on types is defined as follows.

$$\begin{aligned} Const_B\ A &= B \\ Id\ A &= A \\ (F + G)\ A &= F\ A + G\ A \\ (F \times G)\ A &= F\ A \times G\ A \end{aligned}$$

The analogy to ordinary polynomials can be seen if we write X for Id and A_i for $Const_{A_i}$. Then every polynomial functor has the form

$$\sum_{n \in N} A_n X^n$$

In Agda, Ulf Norell [14] expresses polynomial functors as a datatype Functor along with an interpretation as a set $[-]-$

```
infixl 6 _⊕_
infixr 7 _⊗_
data Functor : Set₁ where
  Id : Functor
  Const : Set → Functor
  _⊕_ : Functor → Functor → Functor
  _⊗_ : Functor → Functor → Functor
[_] : Functor → Set → Set
[Id] B = B
[Const C] B = C
[F ⊕ G] B = [F] B ⊎ [G] B
[F ⊗ G] B = [F] B × [G] B
```

The least fixed point of a polynomial function can then be defined in Agda with the following datatype declaration.

```
data μ_ (F : Functor) : Set where
  inn : [F] (μ F) → μ F
```

For example, the familiar *List* datatype can then be defined as the least fixed point of the following L functor.

$$L_A = Const_{Unit} + (Const_A \times Id)$$
$$List\ A = \mu\ L_A$$

Users of Coq might wonder why the definition of μ is accepted by Agda; Coq would reject the above definition of μ because it does not pass Coq's conservative check for positivity. In this case, Agda's type-checker inspects the behavior of the second argument to $[-]-$ building a usage graph and determines that μF will occur positively in $[-]-$, $- \uplus -$, and $- \times -$.

To re-express RecExpr as a polynomial functor we use sum $- \oplus -$ to define cases within a type and product $- \otimes -$ to represent arguments of a particular case. In Agda, the unit type is written $\top$ and has only one member: *tt*. The type $\top$ is used to represent constructors that take no arguments such as *nil*, the empty list.

```
Option : Functor
Option = Id ⊕ Const ⊤
Array : Functor
Array = (Id ⊗ Id ⊗ Id) ⊕ Const ⊤ ⊕ (Id ⊗ Id)
Sum : Functor
Sum = Id ⊗ Id
FExpr : Functor
FExpr = Const ℕ ⊕ Option ⊕ Sum ⊕ Array
Expr : Set
Expr = μ FExpr
```

Unfolding Expr yields the same value calculated above—as we should hope!

```
Expr = μ FExpr
  = [Const ℕ ⊕ Option ⊕ Sum ⊕ Array] FExpr
  = [Const ℕ] (μ FExpr)
  ⊎ [Option] (μ FExpr)
  ⊎ [Sum] (μ FExpr)
  ⊎ [Array] (μ FExpr)
  = ℕ
  ⊎ (μ FExpr ⊎ ⊤)
  ⊎ (μ FExpr × μ FExpr)
  ⊎ ((μ FExpr × μ FExpr × μ FExpr) ⊎
    ⊤ ⊎
    (μ FExpr × μ FExpr))
```

What do values in Expr look like? Written directly they appear nonsensical. Consider the following encoding of $6+7$.

```
six-plus-seven : Expr
six-plus-seven = inn (inj₁ (inj₂ (
  (inn (inj₁ (inj₁ (inj₁ 6))))
  , (inn (inj₁ (inj₁ (inj₁ 7)))))))
```

Traditionally we would provide a unique name for each branch in an algebraic datatype, however here we only have two names inj_1 and inj_2 so instead we rely on nesting to create unique prefixes. Once we have tagged a value we must give it a well known type so that parent expressions can expect a common child type; this is the role of *inn*. Although cumbersome we can hide much of the complexity in constructing values using the right abstractions, consider the following functions used to tag values of Expr. Deconstructing the maybe type gives two constructors: some, which wraps a single expression; and none taking no arguments.

```
none₁ : Expr
none₁ = inn (inj₁ (inj₁ (inj₂ (inj₂ tt))))
some₁ : Expr → Expr
some₁ = inn ∘ inj₁ ∘ inj₁ ∘ inj₂ ∘ inj₁
```

Giving a convenient constructor for $- + -$ is similarly straightforward.

```
enat : ℕ → Expr
enat = inn ∘ inj₁ ∘ inj₁ ∘ inj₁
_+̇_ : Expr → Expr → Expr
e₁ +̇ e₂ = inn (inj₁ (inj₂ (e₁, e₂)))
```

For arrays, we have an assignment operator that takes an array, an index, and a value to assign at that index; `nil`, the empty array; and lookup which accepts an array and an index.

```
_[_]:=₁_ : Expr → Expr → Expr → Expr
a [ i ]:=₁ e = inn (inj₂ (inj₁ (inj₁ (a, i, e))))
nil₁ : Expr
nil₁ = inn (inj₂ (inj₁ (inj₂ tt)))
_!₁_ : Expr → Expr → Expr
a !₁ i = inn (inj₂ (inj₂ (a, i)))
```

These functions make the earlier defined sum significantly easier to read.

```
six-plus-seven' : Expr
six-plus-seven' = enat 6 +̇ enat 7
```

Unfortunately, Agda provides no way of abstracting pattern matching, so for analyzing terms we are forced to use the more obscure syntax.

3. Implicits and Indexed Types

Two important language features used throughout our framework are *implicits* and *indexed types*. One of the major features of dependently typed languages is their support for indexed types which in our case serve as the primary mechanism for expressing well-typed terms and reduction relations. Implicits are a convenient way to introduce polymorphism and they also prove to be an indispensable tool for quantifying over indexed types.

3.1 Implicits

To define the polymorphic identity function we might first write a function of arity two accepting the type of the value to return and the actual value to return.

```
over_identity_ : (A : Set) → A → A
over A identity x = x
```

However the extra type parameter is inconvenient so we can use Agda's support for implicit parameters, which are wrapped in {} rather than ().

```
identity : {A : Set} → A → A
identity x = x
the-answer : ℕ
the-answer = identity 42
```

Here the type checker tries and succeeds at inferring $\mathsf{A} = \mathbb{N}$. We make heavy use of implicits but Agda won't always be able to determine their values. In these cases we must help out by explicitly matching on and supplying implicits. The following shows two definitions of a polymorphic identity function, the first with an implicit type parameter and an implicit term parameter, whereas the second has only an implicit term parameter. The definition of ident calls implicit-ident, providing explicit arguments.

```
implicit-ident : {A : Set} {x : A} → A
implicit-ident {A} {x} = x
ident : {A : Set} → A → A
ident {A} y = implicit-ident {A} {y}
```

Here Agda has little hope of inferring the value of x, so we're forced to explicitly state the value y. Rather than naming each implicit explicitly we can also select only those we are interested in and let Agda fill in the rest.

```
ident₂ : {A : Set} → A → A
ident₂ y = implicit-ident {x = y}
```

3.2 Indexed Datatypes

In Agda it is possible to declare a family of types that can be viewed as a function returning a type. This allows us to refine our view of an existing type by mapping its elements to a new type using a restricted set of constructors. Consider as an example the non-zero natural numbers.

```
data _IsNotZero : ℕ → Set where
  suc-d-is-not-zero : {d : ℕ} → (suc d) IsNotZero
safe-div : {d : ℕ} → ℕ → d IsNotZero → ℕ
safe-div {suc d} n suc-d-is-not-zero = n / suc d
```

Here we've restricted ourselves to only a single constructor that accepts any natural number d and asserts that $d+1$ is non-zero. We will later use this idea to restrict terms of our expression AST to only those that are well-typed.

As a slightly more sophisticated example, we can extend the list type by noting its length, we call this family the vectors of A's.

```
data Vector (A : Set) : ℕ → Set where
  [] : Vector A 0
  _::_ : {n : ℕ} → A → Vector A n
    → Vector A (suc n)
```

In the declaration above, the term $(A : Set)$ is called a *parameter* while the term $\mathbb{N}$ to its right is called an *indice*. We refer to Vector A as a family of types indexed by the naturals. Drawing our attention to its constructors: the first constructor declares the empty vector [] as a Vector A of size zero; the second constructor states that the consing of an A onto a Vector A of size n is a Vector A of size $n+1$.

By using knowledge of a term's length we can create functions that do not need to consider empty lists, such as the following *head* function.

```
head : {n : ℕ} {A : Set} → Vector A (suc n) → A
head (x :: _) = x
```

Agda is smart enough here to notice that there is no $n \in \mathbb{N}$ such that $0 = n + 1$ and considers the single case an exhaustive list of possible inputs.

4. Syntax and Reduction Semantics

In this section we define a simple language and its operational semantics. The language is small, including just sums, an option type, and an array with assignment and lookup.

So far the definition of our syntax has used fairly standard techniques but we have not yet given any sort of meaning to these expressions. We first define a monolithic static and dynamic semantics for this language, then show how to modularize their definitions later in this section. Figure 1 defines a simple set of typing rules using metavariables e to range over expressions and n to range over values; Figure 1b defines a reduction semantics.

While Agda is expressive enough to implement these rules directly, and indeed, they are nearly a direct reflection of that implementation, recall that our goal is to create several independent languages each carrying their own semantics. We begin by defining monolithic semantics for Expr and proceed to determine points of failure and to dissect the definition into independent constituents. To simplify things we define our notion of Type as a closed ADT.

```
data Type : Set where
  TArray : Type
  TOption : Type
  TNat : Type
```

The definitions of the monolithic type system and evaluation relation are as follows.

```
data Welltyped : Expr → Type → Set₁ where
  ok-value : {n : ℕ} → Welltyped (enat n) TNat
  ok-sum : {e₁ e₂ : Expr}
    → Welltyped e₁ TNat → Welltyped e₂ TNat
    → Welltyped (e₁ + e₂) TNat
  ok-nil : Welltyped nil₁ TArray
  ok-lookup : {a e : Expr}
    → Welltyped a TArray
    → Welltyped e TNat
    → Welltyped (a !₁ e) TOption
  ok-ins : {a e n : Expr}
    → Welltyped a TArray
    → Welltyped e TNat
    → Welltyped n TNat
    → Welltyped (a [n] :=₁ e) TArray
infix 2 _⟼E_
data _⟼E_ : Expr → Expr → Set where
  stepl : {e₁ e₁' e₂ : Expr}
    → (e₁ ⟼E e₁')
    → (e₁ + e₂ ⟼E e₁' + e₂)
  stepr : {n₁ : ℕ} {e₂ e₂' : Expr}
    → (e₂ ⟼E e₂')
    → (enat n₁ + e₂ ⟼E enat n₁ + e₂')
  sum : {n₁ n₂ : ℕ}
    → (enat n₁ + enat n₂ ⟼E enat (n₁ +ℕ n₂))
  stepi : {e e' a : Expr}
    → (e ⟼E e')
    → (a !₁ e ⟼E a !₁ e')
  lookup : {a n : Expr}
    → (a !₁ n ⟼E L⟦ a, n ⟧₁)
```

The function $L[\![-,-]\!]_1$ is the lookup function that evaluates to some a_n when a_n has been defined and *none* otherwise. We do not restrict the values of n enough in the *ok-ins* rule; our typing rules require that n be a value while in Agda we have only required it be an expression. Some notion of value is needed and a common solution is to add a tag Value to the Expr type and pattern match; here Value is called [Const $\mathbb{N}$] and in a dependently typed context we might then define a predicate over Value. However because the sum type has only one type of value, a number, it is simpler to use enat directly.

This method for defining semantics is direct and concise, but similar to our first implementation of Expr+ and MonolithicExpr above, there is no simple mechanism for code reuse. The answer is again to delay recursion.

$$
\begin{aligned}
- \dot{+} - &\in Expr \to Expr \to Expr \\
n &\in \mathbb{N} \in Expr \\
-[-] := - &\in Expr \to Expr \to Expr \to Expr \\
-!- &\in Expr \to Expr \to Expr \\
nil &\in Expr
\end{aligned}
$$

(a) Syntax

$$
\text{(stepl+)}\frac{e_1 \longmapsto e_1'}{e_1 \dot{+} e_2 \longmapsto e_1' \dot{+} e_2} \qquad \text{(stepr+)}\frac{e_2 \longmapsto e_2'}{n_1 \dot{+} e_2 \longmapsto n_1 \dot{+} e_2'}
$$

$$
\text{(sum)}\frac{}{n_1 \dot{+} n_2 \longmapsto n_1 + n_2}
$$

$$
\text{(stepi)}\frac{e \longmapsto e'}{a!e \longmapsto a!e'} \qquad \text{(lookup)}\frac{}{a!n \longmapsto L[\![a, n]\!]}
$$

(b) Reduction Semantics

$$
\text{(ok-value)}\frac{}{n : Nat} \qquad \text{(ok-sum)}\frac{e_1 : Nat \quad e_2 : Nat}{e_1 \dot{+} e_2 : Nat}
$$

$$
\text{(ok-nil)}\frac{}{nil : Array} \qquad \text{(ok-lookup)}\frac{a : Array \quad e : Nat}{a!e : Option} \qquad \text{(ok-ins)}\frac{a : Array \quad n : Nat \quad e : Nat}{a[n] := e : Array}
$$

(c) Typing Rules

Figure 1: The syntax, semantics, and type-system of expressions.

4.1 Dissecting the Step Relation

To modularize the evaluation rules, we define a separate step relation for each functor making up our Expr type. First note that $-\dot{+}-$ does not make use of *how* the step from e_1 to e_2 occurs so we can factor this top-level relation as follows.

```
data _⟼⁺_ {_⟼_ : Expr → Expr → Set}
  : Expr → Expr → Set where
  stepl : {e₁ e₁' e₂ : Expr}
    → (e₁ ⟼ e₁') → (e₁ ∔ e₂ ⟼⁺ e₁' ∔ e₂)
  stepr : {n₁ : ℕ} {e₂ e₂' : Expr}
    → (e₂ ⟼ e₂')
    → (enat n₁ ∔ e₂ ⟼⁺ enat n₁ ∔ e₂')
  sum : {n₁ n₂ : ℕ}
    → (enat n₁ ∔ enat n₂ ⟼⁺ enat (n₁ +ℕ n₂))
```

While this is better there is still an undesirable reference to the datatype Expr. Applying the same factorization here to the underlying functor requires parameterization by two extra coercion functions, these are the $-\dot{+}-$ and enat functions defined previously. The new names lift^+ and liftℕ used here are meant to imply that a subtype is being "lifted" into its supertype

```
data _⟼⁺_ {E : Functor}
  {_⟼_ : μ E → μ E → Set}
  {lift⁺ : [Sum] (μ E) → μ E} {liftℕ : ℕ → μ E}
  : μ E → μ E → Set where
  stepl : {e₁ e₁' e₂ : μ E}
    → (e₁ ⟼ e₁') → (lift⁺ (e₁, e₂) ⟼⁺ lift⁺ (e₁', e₂))
  stepr : {n₁ : ℕ} {e₂ e₂' : μ E}
    → (e₂ ⟼ e₂')
    → (lift⁺ (liftℕ n₁, e₂) ⟼⁺ lift⁺ (liftℕ n₁, e₂'))
  sum : {n₁ n₂ : ℕ}
    → (lift⁺ (liftℕ n₁, liftℕ n₂) ⟼⁺ liftℕ (n₁ +ℕ n₂))
```

Unfortunately, continuing with this definition will lead us into a dead end. When we lift terms into the expression type μE, as in the resultant types above, Agda will be working with the evaluation of lift^+ or liftℕ rather than an application tree such as $\mathsf{lift}^+(e_1, e_2)$.

An intelligent human can use their global understanding to peel away lift^+ and see that *stepl* takes the sum (e_1, e_2) to (e_1', e_2) because he or she is staring at the inputs. However Agda needs explicit proof that the function lift^+ doesn't irreversibly destroy the pair (e_1, e_2). This seems reasonable because it will be operating over the *evaluated* term and not much ingenuity is required to find a function destructively mapping sums into Expr. Consider taking $E = \mathsf{FExpr}$ so that $\mu E = \mathsf{Expr}$.

```
forgetful-lift⁺ : [Sum] Expr → Expr
forgetful-lift⁺ (e₁, e₂) = enat 5
```

To see why this would later be a problem, consider a statement about a well-typed sum (e_1, e_2) that steps via *stepl*. The terms representing (e_1, e_2)'s well-typedness and the step it takes will be distinct. Moreover if lift^+ is irreversible Agda can't determine from this information that it's actually (e_1, e_2) stepping at all! Instead it seems to be the case that $5 \longmapsto^+ 5$ despite what's apparent from the type $\mathsf{lift}^+(e_1, e_2) \longmapsto^+ \mathsf{lift}^+(e_1', e_2)$.

The problem is that our abstraction is too general. We need to show that $[\mathsf{Sum}](\mu E)$ and $\mathbb{N}$ are subtypes of the top-level expression datatype μE. In this way, not only will our lift functions prove to be reversible, but their inverses will be highly regular. The solution to the problem is drawn from the notion of a categorical subobject [7].

5. Subobjects and Lazy Coercions

A subobject of a type T is a left invertible function. Being restricted to polynomial functors, we know that all our

subobjects lift : $S \to$ Expr will be some composition of inn, inj_1 and inj_2 so a proof that S is a subtype of Expr is merely a description of which direction to move at each point in a disjoint sum.

```
infix 3 _⊑_
data _⊑_ (F : Functor) : Functor → Set₁ where
  refl : F ⊑ F
  left : {A B : Functor} → F ⊑ A ⊕ B → F ⊑ A
  right : {A B : Functor} → F ⊑ A ⊕ B → F ⊑ B
```

Now we can define containment on a functor's interpretation as a set.

```
infix 3 _<:_
data _<:_ : Set → Set → Set₁ where
  inj : {F A : Functor}
    → F ⊑ A → [A] (μ F) <: (μ F)
```

And we can define conversion functions as follows.

```
upcast : ∀ {F A} → F ⊑ A → [A] (μ F) → μ F
upcast refl = inn
upcast (left t) = upcast t ∘ inj₁
upcast (right t) = upcast t ∘ inj₂

apply : {A B : Set} → (A <: B) → A → B
apply (inj t) = upcast t
```

Recall the two goals we have in mind.

First, we wish to gain access to the lift functions' arguments, in the case of $- + -$ these were e_1 and e_2. By representing containment as a delayed application of a subobject—because the constructor's arguments are stored as a part of the coercion—we can simply use pattern matching to discover that lift is to be applied to e_1 and e_2.

Second, it will be convenient to treat all the constituents of our Expr type uniformly. This allows us to quantify over any value that can be injected into Expr. The type of our step relation is indexed by two expressions: $(e_1 : \text{Expr}) \longmapsto_E (e_2 : \text{Expr})$. We should expect the same of the final abstraction over step relations; however rather than using a number of explicit existentials as in $\exists E_1, E_2 \subseteq \text{Expr}.(e_1 : E_1) \longmapsto (e_2 : E_2)$, we introduce a simpler type LazyCoercion that hides the details.

To delay function application and introduce a uniform type for subsets of Expr we define a LazyCoercion datatype from type A to B representing the *intention* of coercing an object $a \in A$ while thinking of it as a member of its containing type B. A lazy coercion is then an injection $A <: B$ along with an object in A

```
data LazyCoercion (B : Set) : Set₁ where
  delay : {A : Set} → (A <: B)
    → A → LazyCoercion B
coerce : {B : Set} → LazyCoercion B → B
coerce (delay f e) = apply f e
```

We seem to be close to a modular step relation $- \longmapsto^+ -$, defining at each point another level of abstraction to delay immediate application. To modularize datatypes, recursion is delayed and types are viewed as polynomial functors, then to modularize step relations, reduction is parameterized and expression upcasts are delayed.

6. Defining a Modular Step Relation

Attempting again to define a step relation for addition we find very little has changed.

```
data _⟼⁺_ {E : Functor}
  {_⟼_ : μ E → μ E → Set₁}
  {lift⁺ : [Sum] (μ E) <: μ E}
  {liftℕ : ℕ <: μ E}
  : LazyCoercion (μ E) → LazyCoercion (μ E) → Set₁
  where
  stepl : {e₁ e₁' e₂ : μ E}
    → (e₁ ⟼ e₁')
    → (delay lift⁺ (e₁, e₂) ⟼⁺ delay lift⁺ (e₁', e₂))
  stepr : {e₁ e₂ e₂' : μ E}
    → (e₂ ⟼ e₂')
    → (delay lift⁺ (e₁, e₂) ⟼⁺ delay lift⁺ (e₁, e₂'))
  stepv : {n m : ℕ}
    → (delay lift⁺ (apply liftℕ n, apply liftℕ m)
      ⟼⁺ delay liftℕ (n +ℕ m))
```

It appears we've littered an otherwise simple definition with delay but we've replaced our arbitrary arrows with objects having constructors we can match on. Using the above techniques we can modularize the well-typing relation over sums for free.

```
data WtSum {E : Functor}
  {Wt : μ E → Type → Set₁}
  {lift⁺ : [Sum] (μ E) <: μ E}
  : LazyCoercion (μ E) → Type → Set₁ where
  ok-sum : {e₁ e₂ : μ E}
    → Wt e₁ TNat → Wt e₂ TNat
    → WtSum (delay lift⁺ (e₁, e₂)) TNat
```

The above definitions nearly wrote themselves. The simplicity comes from the fact we are just abstracting as many terms as possible, keeping in mind we can fill them in naturally later because the abstraction is so general there are few options available.

6.1 Arrays

We proceed by defining the step and well-typedness relations on arrays, which can be combined with the relations on sums. The definitions for evaluation and well-typedness should look similar to those for sums.

```
data _⟼[]_{E : Functor}
  {_⟼_ : μ E → μ E → Set₁}
  {liftA : [Array] (μ E) <: μ E}
  {liftℕ : ℕ <: μ E}
  {liftO : [Option] (μ E) <: μ E}
  : LazyCoercion (μ E) → LazyCoercion (μ E) → Set₁
  where
  stepi : {e e' a : μ E} → e ⟼ e'
    → (delay liftA (a ! e) ⟼ [] delay liftA (a ! e'))
  lookup : {a : [Array] (μ E)} {n : ℕ}
    → (delay liftA (apply liftA a ! apply liftℕ n)
      ⟼ [] delay liftO L⟦ a, n ⟧)
```

To define the typing relation we again follow the format of WtSum above and we are done.

```
data WtArray {E : Functor}
  {Wt : μ E → Type → Set₁}
  {liftA : [Array] (μ E) <: (μ E)}
  {liftℕ : ℕ <: μ E}
  : LazyCoercion (μ E) → Type → Set₁ where
  ok-nil : WtArray (delay liftA nil) TArray
  ok-ins : {a e n : μ E}
    → Wt a TArray → Wt e TNat → Wt n TNat
```

```
    → WtArray (delay liftA (a [n] := e)) TArray
  ok-lookup : {e a : μ E}
    → Wt a TArray → Wt e TNat
    → WtArray (delay liftA (a ! e)) TOption
```

7. Proving Type Preservation

The type preservation lemma states that if a term is well-typed and can step, then the type of the term is preserved after the step.

$$e \longmapsto e' \wedge e : T \Rightarrow e' : T \qquad \text{(type-preservation)}$$

Prior to considering how type preservation might look for each of the previously defined components, we should review what type preservation looks like for the MonolithicExpr language. The following proof is standard, proceeding by structural induction on the shape of the well-typing tree.

```
pres-MonolithicExpr : ∀ {e e'} {τ}
  → (e ⟼C e')
  → WtMonolithicExpr e τ
  → WtMonolithicExpr e' τ
pres-MonolithicExpr (stepl ste₁) (ok-sum wte₁ wte₂)
  = ok-sum (pres-MonolithicExpr ste₁ wte₁) wte₂
pres-MonolithicExpr (stepr ste₂) (ok-sum wte₁ wte₂)
  = ok-sum wte₁ (pres-MonolithicExpr ste₂ wte₂)
pres-MonolithicExpr
  (stepv {n} {m}) (ok-sum wtn wtm)
  = ok-nat (n +ℕ m)
pres-MonolithicExpr (stepi ste) (ok-lookup wta wte)
  = ok-lookup wta (pres-MonolithicExpr ste wte)
pres-MonolithicExpr
  (lookup {a} {n}) (ok-lookup wta wtn)
  = proj₂ LC⟦ a, n ⟧
```

There are three items worth noting here: the first is the use of the function $LC[\![-,-]\!] : \mathsf{MonolithicExpr} \to \mathbb{N} \to \exists e.\mathsf{WtMonolithicExpr}\ e\ \mathsf{TOption}$ which we have assumed produces a pair whose first component is an expression and second component is a proof that the expression is a well-typed option; the second item worth noting is that recursion acts as our induction hypothesis; and finally note that Agda is smart enough to prove there is only a single possible well-typing constructor for each step constructor—which is good because Agda requires all functions to be total.

We should expect the modular type preservation lemmas to look similar because there is little global knowledge involved. The induction hypothesis and values aside, each case is "contained within its own world" in the sense that each evaluation rule relies only on the fact that subterms are well-typed but ignores the *reason* they are well-typed. To show type preservation for sums we might start with

```
pres-Sum₁ : {τ : Type} {E : Functor}
    {e e' : LazyCoercion (μ E)}
    → (e ⟼+ e')
    → WtSum e τ
    → WtSum e' τ
pres-Sum₁
  (stepl {e₁} {e₁'} {e₂} ste₁) (ok-sum wte₁ wte₂)
    = *
```

however recall that $- \longmapsto^+ -$ requires the top-level step relation and proof that E contains both sums and naturals. There is a second mistake in writing preservation this way—we would like to show that e' is well-typed in the expression language, not just necessarily in the modular sum language, this reflects our desire to expose as little about each component as possible. A second formulation might then begin as follows but we again fail.

```
pres-Sum₂ : {τ : Type}
  {E : Functor}
  {_⟼_ : μ E → μ E → Set₁}
  {lift⁺ : [Sum] (μ E) <: μ E}
  {liftℕ : ℕ <: μ E}
  {Wt : μ E → Type → Set₁}
  {e e' : LazyCoercion (μ E)}
  → _⟼+_ {E} {_⟼_} {lift⁺} {liftℕ} e e'
  → WtSum {E} {Wt} {lift⁺} e τ
  → Wt (coerce e') τ
pres-Sum₂ (stepl ste₁) (ok-sum wte₁ wte₂)
  = * (ok-sum * wte₂)
pres-Sum₂ (stepr ste₁) (ok-sum wte₁ wte₂)
  = * (ok-sum wte₁ *)
pres-Sum₂ stepv (ok-sum wte₁ wte₂)
  = * (n +ℕ m)
```

It seems we're only missing two pieces: we need to be able to lift well-typed sums and naturals into Wt; and we need some way of expressing the induction hypothesis which states that because e_1 is well-typed and stepped, e_1' is well-typed too. The induction hypothesis is slightly stranger than was the case in our MonolithicExpr's because we know e_1 and e_1' are well-typed despite the fact that they are arbitrary expressions, not necessarily just sums. This motivates our solution which takes the induction hypothesis as an explicit assumption.

```
pres-Sum : {τ : Type}
  {E : Functor}
  {_⟼_ : μ E → μ E → Set₁}
  {lift⁺ : [Sum] (μ E) <: μ E}
  {liftℕ : ℕ <: μ E}
  {Wt : μ E → Type → Set₁}
  {a b : LazyCoercion (μ E)}
  → ((n : ℕ) → Wt (apply liftℕ n) TNat)
  → (∀ {δ} {e}
      → WtSum {E} {Wt} {lift⁺} (delay lift⁺ e) δ
      → Wt (apply lift⁺ e) δ)
  → (∀ {δ} {e e'} → (e ⟼ e') → Wt e δ → Wt e' δ)
  → _⟼+_ {E} {_⟼_} {lift⁺} {liftℕ} a b
  → WtSum {E} {Wt} {lift⁺} a τ
  → Wt (coerce b) τ
pres-Sum wtnat wt IH
  (stepl ste₁) (ok-sum wte₁ wte₂)
  = wt (ok-sum (IH ste₁ wte₁) wte₂)
pres-Sum wtnat wt IH
  (stepr ste₂) (ok-sum wte₁ wte₂)
  = wt (ok-sum wte₁ (IH ste₂ wte₂))
pres-Sum wtnat wt IH
  (stepv {n} {m}) (ok-sum wte₁ wte₂)
  = wtnat (n +ℕ m)
```

We are pleased with how similar this is to the original, monolithic formulation. Notice again that the solution was to factor out assumptions about the outside world, similar to the previous abstractions. Proving type preservation for arrays is natural:

```
pres-Array : {τ : Type}
  {E : Functor}
  {_⟼_ : μ E → μ E → Set₁}
  {liftA : [Array] (μ E) <: (μ E)}
  {liftℕ : ℕ <: μ E}
  {liftO : [Option] (μ E) <: (μ E)}
  {Wt : μ E → Type → Set₁}
  {a b : LazyCoercion (μ E)}
  → ((m : [Option] (μ E))
    → Wt (apply liftO m) TOption)
  → (∀ {δ} {e}
    → WtArray {E} {Wt} {liftA} {liftℕ}
      (delay liftA e) δ
    → Wt (apply liftA e) δ)
  → (∀ {δ} {e e'} → (e ⟼ e') → Wt e δ → Wt e' δ)
  → _⟼ [] _ {E} {_⟼_} {liftA} {liftℕ} {liftO}
    a b
  → WtArray {E} {Wt} {liftA} {liftℕ} a τ
  → Wt (coerce b) τ
pres-Array wtopt wt IH
  (stepi ste) (ok-lookup wta wte)
  = wt (ok-lookup wta (IH ste wte))
pres-Array wtopt wt IH
  (lookup {a} {n}) (ok-lookup wta wte)
  = wtopt L⟦ a, n ⟧
```

It would seem we are nearly done and the final pieces should be entirely guided by the selected abstractions. The lift functions each have a unique solution:

```
lift⁺ : [Sum] Expr <: Expr
lift⁺ = inj (right (left (refl)))

liftℕ : ℕ <: Expr
liftℕ = inj (left (left (left refl)))

liftO : [Option] Expr <: Expr
liftO = inj (right (left (left refl)))

liftA : [Array] Expr <: Expr
liftA = inj (right refl)
```

But how should we define well-typedness for Expr? Again the notion of what it means to be well-typed has already been defined and we simply need to "tie the knot" as RecExpr did above

```
data WtExpr : Expr → Type → Set₁ where
  lift-wt-nat : (n : ℕ)
    → WtExpr (apply liftℕ n) TNat
  lift-wt-option : (m : [Option] Expr)
    → WtExpr (apply liftO m) TOption
  lift-wt-sum : {τ : Type} {e : [Sum] Expr}
    → WtSum {FExpr} {WtExpr} {lift⁺}
      (delay lift⁺ e) τ
    → WtExpr (apply lift⁺ e) τ
  lift-wt-array : {τ : Type} {e : [Array] Expr}
    → WtArray {FExpr} {WtExpr} {liftA} {liftℕ}
      (delay liftA e) τ
    → WtExpr (apply liftA e) τ
```

To define a step relation on Expr, $- \longmapsto -$ we provide a similar wrapping for each language component

```
data _⟼_ : Expr → Expr → Set₁ where
  step⁺ : {e : [Sum] Expr}
    {e' : LazyCoercion Expr}
    → _⟼⁺_ {FExpr} {_⟼_} {lift⁺} {liftℕ}
      (delay lift⁺ e) e'
    → (apply lift⁺ e ⟼ coerce e')
  step [] : {e : [Array] Expr}
    {e' : LazyCoercion Expr}
    → _⟼ [] _
      {FExpr} {_⟼_} {liftA} {liftℕ} {liftO}
      (delay liftA e) e'
    → (apply liftA e ⟼ coerce e')
```

The only piece remaining is to prove type preservation. We begin in the same way we have for each of the previous proofs using the step relation's constructors as a guide. The type signature should not have changed

```
preservation : {e e' : Expr} {τ : Type}
  → (e ⟼ e') → WtExpr e τ → WtExpr e' τ
```

and there are two cases step⁺ and step[]; moreover we should expect to merely apply pres-* to each case, supplying the necessary lift functions and the induction hypothesis. This is indeed the case:

```
preservation (step⁺ ste) (lift-wt-sum wts)
  = pres-Sum lift-wt-nat lift-wt-sum
    preservation ste wts
preservation (step [] ste) (lift-wt-array wta)
  = pres-Array lift-wt-option lift-wt-array
    preservation ste wta
```

There is one caveat: Agda is unable to prove termination of this definition, but we hope to solve this problem soon.

Having shown type preservation, it is interesting to see the similarity between how terms are shown to be well-typed and to reduce and how the terms are expressed in μFExpr. Recall that each term in Expr is wrapped by a tag—given by inj_1 and inj_2—and the constructor inn plays the role of recursion. To reiterate consider the convenience functions,

```
nilE : Expr
nilE = inn (inj₂ (inj₁ (inj₂ tt)))
nat : ℕ → Expr
nat n = inn (inj₁ (inj₁ (inj₁ n)))
_ [_] =_ : Expr → Expr → Expr → Expr
a [n] = e = apply liftA (a [n] := e)
_!E_ : Expr → Expr → Expr
a !E n = apply liftA (a ! n)
_+E_ : Expr → Expr → Expr
e₁ +E e₂ = apply lift⁺ (e₁, e₂)
```

we may then ask: why is following term well-typed?

```
exp : Expr
exp = (nilE [nat 0] = nat 1) !E (nat 0 +E nat 1)
```

The answer given by WtExpr is

```
wt-exp : WtExpr exp TOption
wt-exp = lift-wt-array (ok-lookup wta wt+)
  where
    wta : WtExpr (nilE [nat 0] = nat 1) TArray
    wta = lift-wt-array
      (ok-ins (lift-wt-array ok-nil)
        (lift-wt-nat 1) (lift-wt-nat 0))
    wt+ : WtExpr (nat 0 +E nat 1) TNat
    wt+ = lift-wt-sum (ok-sum
      (lift-wt-nat 0) (lift-wt-nat 1))
```

The lift−wt−∗ functions play the same role in WtExpr as *inn* does in Expr; however rather than using the generalized

approach of a series of disjoint sums we bundle the tag and recursion into a single constructor for each language component. Evaluation displays a similar symmetry.

```
eval-expr : (nilE [nat 0] = nat 1) !E (nat 0 +E nat 1)
  ⟼ (nilE [nat 0] = nat 1) !E nat 1
eval-expr = step [] (stepi (step+ stepv))
```

What does the proof that (nilE [nat 0] = nat 1) $!E$ nat 1 is well-typed look like? We can compute it by invoking

```
preservation eval-expr wt-exp
```

which evaluates to

```
lift-wt-array
(ok-lookup
  (lift-wt-array
    (ok-ins (lift-wt-array ok-nil) (lift-wt-nat 1)
      (lift-wt-nat 0)))
  (lift-wt-nat 1))
```

8. Related Work

One of the more recent and important contributions to the area of modular interpreters is Data types à la carte [15] by Swierstra. It details solutions to a variety of similar problems including the boilerplate issue we have experienced. Unfortunately, the solutions are not easily translatable into Agda due to their reliance on language level support for type classes. For instance when using injection and projection functions within Agda, the subtyping class must be passed into the call explicitly. The approach also makes use of folds for performing evaluation. Akin to induction principles in Coq, these types of structured recursion schemes present possible solutions to the termination problems we've run into at the final stages of our proof.

Data types à la carte is among a number of papers that use the idea of structuring modular interpreters as a sum of algebras, including Monad transformers and modular interpreters [8] by Liang et al. and the earlier work by Duponcheel [5]. While this technique has been used in defining functional interpreters, we haven't seen the idea extended to modular proof construction in the presence of dependent types.

Mulhern's [13] work on *proof-weaving* is also related to ours, where small proofs are automatically merged into larger proofs. The idea is similar in spirit, relying on a uniform structure across language terms; however it relies on an external tool to perform the weaving and having its own logic for term inference and the extraction of subproofs for inclusion in a larger proof.

Concurrently with our work, Delaware, et al. [3] developed a solution to modular meta-theory in Coq. Both their approach and ours relies on the principle of representing data types as functors; however to avoid problems around recursion they have chosen to express inductive types using Church encodings and perform evaluation using Mendler algebras, which requires some extra sophistication. Here we avoid general recursion by restricting our universe, expressing types as data members of the family of polynomial functors and by expressing the dynamic semantics using small-step reduction. Our approach makes sense in Agda because of its strong pattern matching, however it could potentially present problems when moved directly into Coq. Another implementation detail that doesn't translate directly into Coq, due to its strict notion of positivity, is the definition of least fixed point over polynomial functors. Instead it might be possible to use Church encodings and Mendler algebras. Similarly, much of the Meta-Theory à la Carte infrastructure may be problematic when implemented naïvely in Agda due to its heavy reliance on Coq's type classes. Their work is further along than ours and has shown the important level of robustness required by most languages while there are more unanswered questions regarding the method presented here.

A similar vein of work by Axelsson [1] describes a modular functional interpreter in Haskell using the native type-system and ghc's type family extension to enforce the well-typing of terms. This is a powerful technique for embedding preservation in the type of the *eval* function though the question of how it would scale to other theorems such as progress is unclear.

9. Conclusion and Future Work

We should ask if we have accomplished the goal that we set out with. The language Expr was given componentwise and the boilerplate necessary to wrap each well-typing and step relation is minimal. The proof of type preservation was almost immediate, requiring only an invocation of previously defined proofs for each component. Moreover there is no copy and paste necessary and the repetitive components should be automatically producible given a sophisticated macro system where terms can be inspected by name—set equality is non-deterministic—rather than by value.

Using Agda as a proof language, although convenient, leaves the question of consistency open. We regard this as a minor problem and hope that our implementation would port to Coq. A more pertinent problem is the definition of preservation for Expr—Agda is unable to prove termination and we plan to address this soon. Though issues of termination can often be difficult, the broad range of work in manipulating descriptions of and generating induction principles for indexed datatypes [4, 6, 10, 11] gives us a great deal of hope.

The language presented is quite simple, unable to express even Euclid's algorithm, and the method of polynomial functor's used to express Expr precludes the possibility of first class function types which are critical for functional programming. Various solutions to this problem have been proposed [12] and the area of recursion schemes is rich [16]. A real world language calls for much heavier sophistication, but the ideas presented here are new and their reach is open to question and requires further exploration.

References

[1] Axelsson, E. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2012), ICFP '12, ACM, pp. 323–334.

[2] Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. Mechanized metatheory for the masses: the poplmark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2005), TPHOLs'05, Springer-Verlag, pp. 50–65.

[3] Benjamin Delaware, Bruno C. d. S. Oliveira, T. S. Meta-theory à la carte. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, January 2013.

[4] Chapman, J., Dagand, P.-E., McBride, C., and Morris, P. The gentle art of levitation. In *Proceedings of the*

15th ACM SIGPLAN International Conference on Functional programming (New York, NY, USA, 2010), ICFP '10, ACM, pp. 3–14.

[5] Duponcheel, L. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters., 1995.

[6] Ko, H.-S., and Gibbons, J. Modularising inductive families. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming* (New York, NY, USA, 2011), WGP '11, ACM, pp. 13–24.

[7] Mac Lane, S. *Categories for the Working Mathematician*. No. 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[8] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), POPL '95, ACM, pp. 333–343.

[9] Malcolm, M. *Algebraic Data Types and Program Transformation*. PhD thesis, Groningen University, Department of Computing Science, 1990.

[10] McBride, C. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*. To appear.

[11] McBride, C. The derivative of a regular type is its type of one-hole contexts. 2001.

[12] Meijer, E., and Hutton, G. Bananas in space: extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture* (New York, NY, USA, 1995), FPCA '95, ACM, pp. 324–333.

[13] Mulhern, A. Proof Weaving. In *Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory* (Portland, Oregon, September 2006), The Eleventh ACM SIGPLAN International Conference on Functional Programming.

[14] Norell, U. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2009), TLDI '09, ACM, pp. 1–2.

[15] Swierstra, W. Data types à la carte. *J. Funct. Program. 18*, 4 (July 2008), 423–436.

[16] Uustalu, T., Vene, V., and Pardo, A. Recursion schemes from comonads. *Nordic J. of Computing 8*, 3 (Sept. 2001), 366–390.