Modular Type-Safety Proofs using Dependant Types

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

Abstract
While methods of code abstraction and reuse are widespread and well researched, methods of proof abstraction and reuse are still emerging. We consider the use of dependent types for this purpose, introducing a completely mechanical approach to proof composition. We show that common techniques for abstracting algorithms over data structures naturally translate to abstractions over proofs. We first introduce a language composed of a series of smaller language components tied together by standard techniques from Malcom [2]. We proceed by giving proofs of type preservation for each language component and show that the basic ideas used in composing the syntactic data structures can be applied to their semantics as well.

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 response to this challenge, significant strides have been made in making it easier to mechanize the metatheory of programming languages, especially regarding variable binding [1]. 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 and then link the fragments together to obtain the definitions and meta-theory for a language composed of such fragments.

Dependent types have formed the foundation of a broad and rich range of type systems that allow values and types to be freely mixed. Programmers can express propositions as types viewed as sets, and proofs as objects viewed as inhabitants of those sets. This style of theorem proving suggests the use of familiar engineering abstractions as general solutions to questions about theorem proving. Rather than relying on semi-automated proof search such as Coq’s Ltac we propose a method of proof composition using simple abstractions whereby components are defined piecewise and “tied” together at the end using a wrapper datatype acting as a tagged union.

The method of language definition used is iterative. Components are defined separately from one another and are composable along with their proofs. Thus we would like for separate language designers to be able to reuse one another’s work without the need for sophisticated proof search algorithms or with effort spent copying and pasting terms.

The language we present is one of simple expressions using Agda as the implementation language and proof assistant. We begin by defining a series of language syntaxes for sums, options, and arrays. We chose to include arrays because they not only can result in runtime errors requiring the inclusion of 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 evaluation semantics and typing rules. The language is defined piecewise, each component is built in isolation alongside a proof of type preservation. We conclude with a presentation of how these components can be composed and a proof of type preservation for the combined language can be immediately derived from the component-wise proofs. The motivation for our technique is drawn from a solution to the expression problem where languages are defined as the disjoint sum of smaller languages by removing explicit recursion. We show that this idea can be recast from types and terms, to proofs.

2. A Review of the Expression Problem
When modeling a problem with a functional flavor often the natural solution emerges as several recursive cases handled by some helper functions. The expression problem states that this type of solution presents us with a choice: we may ordain our data structure forever unchanging, making it easy to add new functions without changing the program; or we may leave our data structure open, making it difficult to extend the original program with new functions.

While many solutions to the expression problem have been proposed over the years, here we make use of the method described by Malcom [2] which generalizes recursion operators such as fold, from lists to polynomial types. The problem we encounter 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 and the solution presented lies at the heart of our work. The idea is simply to remove immediate recursion and split a monolithic datatype into components to be later collected under the umbrella of a tagged union.

Throughout this paper we will work with a simple evaluator over natural numbers and basic arithmetic operators; in Agda we might first consider

\[
\text{data } \text{Expr}+: \text{Set where} \\
\text{atom } : \text{N} \rightarrow \text{Expr}+ \\
\_+\_ : \text{Expr}+ \rightarrow \text{Expr}+ \rightarrow \text{Expr}+
\]
This definition has the advantage of being direct and simple, however a problem lies within the explicit recursion; notice that when later extending expressions with arrays and option types we can make no reuse of Expr+ due to the closed nature of algebraic data types. To extend Expr+ we must define a whole new data type, as in the following definition of MonolithicExpr.

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

Suppose instead we begin with polymorphic definitions such as the following.

```
data Expr+2 (A : Set) : Set where
  _+_ : A \rightarrow A \rightarrow Expr+2 A
data Expr [] 2 (A : Set) : Set where
  nil [] : Expr [] 2 A
  _![]_ : Expr [] 2 A
  _[ ]_ := _[ ]_ : A \rightarrow A \rightarrow A \rightarrow Expr [] 2 A
data ExprOption (A : Set) : Set where
  esome : \N \rightarrow ExprOption A
  enone : \N \rightarrow ExprOption A
data Lit (A : Set) : Set where
  atom : \N \rightarrow Lit A
```

We then introduce recursion as follows, combining components as a disjoint sum, written \( \psi \) in Agda.

```
data RecExpr : Set where
  expr : Lit RecExpr
  \psi Expr+2 RecExpr
  \psi Expr [] 2 RecExpr
  \psi ExprOption RecExpr
  \rightarrow 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:

```
data _μ_ (F : Set \rightarrow Set) : Set where
  \inn : F (\mu F) \rightarrow \mu F
  Expr' = \lambda (A : Set) \rightarrow Lit A \psi Expr+2 A
  \psi Expr [] 2 A
  \psi ExprOption A
  Expr = \mu Expr'
```

It is easy to see that this new type is equivalent to MonolithicExpr up to isomorphism

\[
\begin{align*}
Expr &= \mu Expr' \\
&= Expr' (\mu Expr') \\
&= Expr' Expr \\
&\cong atom
\end{align*}
\]

2.1 Functors and Agda

The functor \( F \), passed into \( \mu \)– above, serves as the key abstraction allowing us to represent expressions as least fixed points. 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 \( FA \)
2. assigns to each function \( f : A \rightarrow B \), a function map \( f : FA \rightarrow FB \)

such that

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

One familiar example is the \( List \) functor mapping each type \( A \) to \( List A \) and each function \( f : A \rightarrow B \) to map \( f : List A \rightarrow List B \) which applies \( f \) to each element of a list. Here we define the least fixed point over a restricted class of functors called the \( polynomial \) functors. Polynomial functors are a subset roughly equivalent to the more familiar algebraic polynomials,

\[
\sum_{n \in \mathbb{N}} A_n X^n
\]

where addition is disjoint sum and multiplication is cartesian product. In Agda, Ulf Norell\cite{4} expresses this class as a datatype \( Functor \) along with an interpretation as a set \( [-] \)

```
infixl 6 \oplus_
infixr 7 \otimes_
data Functor : Set, where
  X : Functor
  A : Set \rightarrow Functor
  \oplus_ : Functor \rightarrow Functor \rightarrow Functor
  \otimes_ : Functor \rightarrow Functor \rightarrow Functor
  [\_] : Functor \rightarrow Set \rightarrow Set
  [X] B = B
  [A C] B = C
  [F \oplus G] B = [F] B \oplus [G] B
  [F \otimes G] B = [F] B \times [G] B
```

with least fixed point

```
data _μ_ (F : Functor) : Set where
  \inn : [F] (\mu F) \rightarrow \mu F
```

Then to reexpress \( Expr \) as a polynomial functor we use sum \( \oplus \) to define cases within a type, and product \( \otimes \) to represent arguments of a particular case

\[
\text{Option}_1 : \text{Functor}
\]

\[
\text{Option}_1 = X \oplus A \top
\]
Array₁ : Functor
Array₁ = X ⊕ X ⊕ X ⊕ A ⊕ X ⊕ X
Sum₁ : Functor
Sum₁ = X ⊕ X
F₁ = Functor
F₁ = A N ⊕ Option₁ ⊕ Sum₁ ⊕ Array₁
E₁ = Set
E₁ = µ F₁

Unfolding E₁ yields the same value calculated above—as we should hope!

E₁ = µ F₁
= [A N ⊕ Option₁ ⊕ Sum₁ ⊕ Array₁]
= [A N] (µ F₁)
∪ Option₁ (µ F₁)
∪ Sum₁ (µ F₁)
∪ Array₁ (µ F₁)
= N
∪ (µ F₁) × T
∪ (µ F₁) × (µ F₁)
∪ (µ F₁) × (µ F₁) ⊔ T ⊔ (µ F₁) ⊔ (µ F₁)

What does values in E₁ look like? Written directly they appear nonsensical, consider 6 + 7

the-sum : E₁
the-sum = inn (inn (inn₁ (inn₂ (inn (inn (inn₁ (inn₁ 7)))))))

Notice how the role that the injections and inn functions play. Traditionally, we would provide a unique name for each branch in an algebraic datatype, however here we only have two names inn₁ and inn₂ so we instead 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 this complexity provided the right abstractions

the-sum' : E₁
the-sum' = nat₁ 6 +₁ nat₁ 7
where nat₁ : N → E₁
nat₁ = inn ∘ inj₁ ∘ inj₁ ∘ inj₁
+₁ : E₁ → E₁ → E₁
e₁ +₁ e₂ = inn (inn₁ (inn₂ (e₁, e₂)))

3. Syntax and Evaluation Semantics

We are now ready to 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. In Agda, the unit type is written T and has only one member: tt. T is used to represent constructors that take no arguments such as nil, the empty list.

Option : Functor
Option = X ⊕ A ⊕ T
Array : Functor
Array = X ⊕ X ⊕ X ⊕ A ⊕ X ⊕ X ⊕ X
Sum : Functor
Sum = X ⊕ X
FExpr : Functor
FExpr = A N ⊕ Option ⊕ Sum ⊕ Array

Expr : Set
Expr = µ FExpr

What do each of these definitions mean? The maybe type has two constructors: some, which wraps a single expression; and none taking no arguments. We define more descriptive constructors for tagging these two types of values

none₁ : Expr
none₁ = inn (inn₁ (inn₁ (inn₂ (inn₂ tt))))
some₁ : Expr → Expr
some₁ = inn ∘ inj₁ ∘ inj₁ ∘ inj₁

Giving a convenient constructor for − − is similarly straightforward

enat : N → Expr
enat = inn ∘ inj₁ ∘ inj₁
−₁ : Expr → Expr → Expr
e₁ −₁ e₂ = inn (inn₁ (inn₂ (e₁, e₂)))

and to define arrays we have assignment taking 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]₁₁₁ : Expr → Expr → Expr → Expr
a [i] = inn (inn₂ (inn (a, i)))
nil₁ : Expr
nil₁ = inn (inn₂ (inn₁ (inn₂ tt)))
−₁₁₁ : Expr → Expr → Expr
a [i] = inn (inn₂ (inn (a, i)))

So far the definition of our syntax has used fairly standard techniques but we have failed to give 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 definition later in this section. Figure 1c defines a simple set of typing rules using metavariables e to range over expressions and n to range over values; Figure 1b gives a small step operational 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

and here is the definition of the monolithic type system and evaluation relation in Agda.

data Welltyped : Expr → Type → Set 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 n : Expr }
→ Welltyped a TArray
→ Welltyped e TNat
→ Welltyped (a [i] e) TOption
ok-ins : { a e n : Expr }
→ Welltyped a TArray
\[\begin{align*}
- \in E & \in E \rightarrow E \rightarrow E \\
\overline{[\cdot]} : - & \in E \rightarrow E \rightarrow E \\
\overline{!} - & \in E \rightarrow E \rightarrow E \\
n \in \mathbb{N} & \in E
\end{align*}\]

(a) Syntax

\[\begin{align*}
\text{(stepl)} & : e_{1} \rightarrow e'_{1} \\
& : e_{1} + e_{2} \rightarrow e'_{1} + e_{2} \\
\text{(stepr)} & : e_{2} \rightarrow e'_{2} \\
& : n_{1} + e_{2} \rightarrow n_{1} + e'_{2} \\
\text{(sum)} & : n_{1} + n_{2} \rightarrow n_{1} + n_{2} \\
\text{(stepi)} & : e \rightarrow e' \\
& : a! e \rightarrow a! e'
\end{align*}\]

(b) Evaluation Semantics

\[\begin{align*}
\text{(ok-value)} & : n : \mathbb{N} \\
\text{(ok-nil)} & : \text{nil} : \text{Array} \\
\text{(ok-lookup)} & : a : \text{Array} e : \mathbb{N} \\
\text{(ok-sum)} & : e_{1} : \mathbb{N} e_{2} : \mathbb{N} e_{1} + e_{2} : \mathbb{N} \\
\text{(ok-ins)} & : a : \text{Array} n : \mathbb{N} e : \mathbb{N} a[n] := e : \text{Array}
\end{align*}\]

(c) Value Typing

(d) Sum Typing

(e) Array Typing

\[\text{infix} 2 \_ 
\text{data} \_ \rightarrow \_ : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Set} \text{ where} \\
\text{stepl} : \{e_{1} e_{1'} e_{2} : \text{Expr}\} \\
& \rightarrow e_{1} \rightarrow E e_{1'} \\
& \rightarrow e_{1} + e_{2} \rightarrow E e_{1'} + e_{2} \\
\text{stepr} : \{n_{1} : \mathbb{N}\} \{e_{2} e_{2'} : \text{Expr}\} \\
& \rightarrow e_{2} \rightarrow E e_{2'} \\
& \rightarrow \text{enat} n_{1} + e_{2} \rightarrow E \text{enat} n_{1} + e_{2'} \\
\text{sum} : \{n_{1} n_{2} : \mathbb{N}\} \\
& \rightarrow \text{enat} n_{1} + \text{enat} n_{2} \rightarrow E \text{enat} (n_{1} + \mathbb{N} n_{2}) \\
\text{stepi} : \{e e' a : \text{Expr}\} \\
& \rightarrow e \rightarrow E e' \\
& \rightarrow a \overline{l}_{1} e \rightarrow E a \overline{l}_{1} e' \\
\text{lookup} : \{a n : \text{Expr}\} \\
& \rightarrow a \overline{l}_{1} n \rightarrow E L[a, n] \overline{l}_{1}
\]

The function \(L[a, n]\) is the lookup function that evaluates to some \(a_{n}\) when \(a_{n}\) has been defined and none otherwise.

Notice that we currently do not restrict the values of \(n\) enough in the \(ok\)-\text{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 \text{Value} to the \text{Expr} type and pattern match; here \text{Value} is called \([A N]\) and in a dependently typed context we might then define a predicate over \text{Value}. However because the sum type has only one type of value, a number, it is simpler to use \text{enat} directly.

This method for defining semantics is common with the advantage of being direct and concise, but similar to our first implementation of \text{Expr} and \text{MonolithicExpr} above: there is no simple mechanism for code reuse. The answer is again to delay recursion.

### 3.1 Dissecting the Step Relation

In order to modularize the evaluation rules we define a separate step relation for each functor making up our \text{Expr} type. First note that \(- + -\) doesn’t make use of \text{how} the step from \(e_{1}\) to \(e_{2}\) occurs so we can factor this top-level relation
data + : Expr → Expr → Set where
  stepl : {e₁ : e₂ : Expr}
  → e₁ → e₁' → e₁ + e₂ →+ e₁' + e₂
  stepr : {n₁ : N} {e₁ 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 parametrization by
two extra coercion functions, these are the +−− and enat
functions defined previously. The new names lift^+ and lift N
used here are meant to imply that a subtype is being “lifted”
into its supertype

data µE : Functor
  { _ → _ : µ E → µ E → Set } { _ →+ _ : E → Functor → µ E → Expr }
  { _ → _ : µ E → µ E → Set where
    stepl : {e₁ e₁' e₂ e₂ : µ E}
    → e₁ → e₁' → lift+ (e₁, e₂) →+ lift+ (e₁', e₂)
    stepr : {n₁ : N} {e₁ 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 this definition fails short too. When we lift
terms into the expression type µ E, Agda “forgets” the con-
stituents e₁ and e₂—in turn we lose the ability to reason
about these distinct components of the sums e₁ and e₂'.
This later becomes a problem when, for example, attempting
to abstract the welltyping relation.

An intelligent human can peel away lift^+ and see that the
terms e₁ and e₂ in − →− and − →+ are the same
because lift^+ is injective. However Agda is unconvincing,
and rightfully so, for it does not require particularly great
deal of ingenuity to find a counterexample, consider taking
E = FExpr so that µ E = Expr

forgetful-lift+ : [Sum] Expr → Expr
forgetful-lift+ (e₁ e₂) = enat 0

The problem is that our abstraction is too general. What
we require is a proof that [Sum](µ E) and N are subtypes
of the top-level expression datatype µE. The solution to the
problem is drawn from the notion of a categorical subobject.

We proceed by delaying application of injections and view
the objects as injectable, existential terms. The importance of
this approach is two-fold: firstly this allows us to take
inverses of lift functions while we are secondly able to retain
the perspective of operating on a single type µ E.

4. Lazy Coercions

A subobject of a type T is a left invertible function with
codomain T, lift : S → T. Being restricted to polynomial
functors, we know that all our subobjects lift : S → µ E will
be some composition of intr, inj₁ and inj₂ so a proof that T
is a subtype of µ E is merely a description of which direction
to move at each point in a disjoint

infix 3 _ Contains_
data _ Contains_ : Functor → Functor → Set₁ where
  refl : {F : Functor}

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

infix 3 _→_
data _→_ : Set → Set → Set₁ where
  inj₁ : {F : A B : Functor}
  → F Contains A → [A] (µ F) → (µ F)

with conversion functions defined as

upcast : ∀ {F A} → F Contains A → [A] (µ F) → µ F
upcast refl = intr
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 had in mind. We first wished to take
the inverse of a lift function to gain access to its arguments,
in the case of +−− these were e₁ and e₂. By representing
an injection as a delayed application of a subobject—because
the constructor’s arguments are stored as a part of the
coercion—finding left inverses will become a trivial case of
pattern matching. To delay function application allowing
Agda to effectively peel away the lift functions we define
a LazyCoercion datatype from type A to B representing the
intention of coercing an object a ∈ A while treating it at the
type-level as B. A lazy coercion is then an injection A → B
along with an object in A

data LazyCoercion : Set → Set₁ where
  inj₁ : {A B : Set} → (A → B) → A → LazyCoercion B
  coerce : {B : Set} → LazyCoercion B → B
  coerce {f e} = apply f e

Our second goal was to operate on objects of a single type.
Why is this the case? Recall that the type of our step relation
is indexed by two expressions: (e₁ : Expr) → e₁ (e₂ : Expr).
We should expect the same of the final abstraction over step
relations because it cannot easily name the underlying type
of its indexing expressions. Instead we have packaged
the indices as existential which are viewed as the type B.

We seem to be close to a modular step relation − →+ −,
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, evaluation is parametrized
and expression upcasts are delayed by viewing them as an
intention.

5. 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 → Expr₁ } { lift+ : [Sum] (µ E) → µ E }
  { liftN : N → µ E }
  : LazyCoercion (µ E) → LazyCoercion (µ E) → Set₁
where
  stepl : {e₁ e₁' e₂ : µ E} → F Contains F
  left : {A B F : Functor}
  → F Contains A ⊕ B → F Contains A
  right : {A B F : Functor}
  → F Contains A ⊕ B → F Contains B
but we've replaced our arbitrary arrows with objects

It appears we've littered an otherwise simple definition with

techniques we can modularize the welltyping relation over arrays that can be combined with the relations on sums.

few options available.

5.1 Arrays

We proceed by defining the step and welltypedness relations on arrays that can be combined with the relations on sums. The definitions for evaluation and welltypedness should look similar to those for sums—\( \rightarrow \rightarrow \rightarrow \).

data WtSum {E : Functor}
{ Wt : \mu E \rightarrow Type \rightarrow Set_1 }
{ lift^+ : [Sum] (\mu E) \rightarrow \mu E }
{ \text{LazyCoercion} (\mu E) \rightarrow Type \rightarrow Set_1 where
  ok-sum : \{ e_1 e_2 : \mu E \}
  \rightarrow Wt e_1 TNat \rightarrow Wt e_2 TNat
  \rightarrow WtSum (\text{inj lift}^+ (e_1, e_2)) 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. 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 evaluation

\[ e \rightarrow e' \land e : T \Rightarrow e' : T \] (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 proof is standard, proceeding by structural induction on the shape of the welltyping tree.

\begin{align*}
\text{preservation-MonolithicExpr} : \forall \{ e e' \} \{ \tau \}
& \rightarrow e \rightarrow C e' \\
& \rightarrow WtMonolithicExpr e \tau \\
& \rightarrow WtMonolithicExpr e' \tau \\
\text{preservation-MonolithicExpr} (\text{stepl ste}_1) (\text{ok-sum wte}_1 wte_2)
& = \text{ok-sum} (\text{preservation-MonolithicExpr ste}_1 \text{ wte}_1) wte_2 \\
& = \text{ok-sum-MonolithicExpr (step ste) (ok-sum wte wte)} \\
& = \text{ok-sum-MonolithicExpr (step ste) (ok-sum wte wte)} \\
& = \text{ok-sum-MonolithicExpr (step ste) (ok-sum wte wte)} \\
& = \text{ok-sum-MonolithicExpr (step ste) (ok-sum wte wte)} \\
& = \text{pro2} \text{LC} a n \\
\end{align*}

There are three items worth noting here: the first is the use of the function \(\text{LC}[-,-] : \text{MonolithicExpr} \rightarrow N \rightarrow \exists e. WtMonolithicExpr e \Rightarrow \text{TOption} \) which we have assumed produces a pair with first component an expression and second component a proof that the expression is a welltyped option; the second is that recursion acts as our induction hypothesis; and finally that Agda is smart enough to notice there is only a single possible welltyping constructor for each step constructor—in Agda all functions are 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 facts that subterms are well-typed but ignoring the reason they are welltyped. To show type preservation for sums we might start with

\begin{align*}
\text{preservation-Sum}_1 : \{ \tau : \text{Type} \} \{ E : \text{Functor} \}
& \rightarrow e \rightarrow^* e' \\
& \rightarrow WtSum e \tau \\
& \rightarrow WtSum e' \tau \\
\text{preservation-Sum}_1 (\text{stepl e}_1 e_1) (\text{ok-sum wte}_1 wte_2) = \\
\end{align*}

however recall that \( \rightarrow \rightarrow \rightarrow \rightarrow \) 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 welltyped 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.

\begin{align*}
\text{preservation-Sum}_2 : \{ \tau : \text{Type} \} \{ E : \text{Functor} \}
& \rightarrow e \rightarrow^* e' \\
& \rightarrow \rightarrow \rightarrow \rightarrow \tau \\
\end{align*}
We are pleased with how similar this is to the original.

The right way of expressing the induction hypothesis which states that
arrays is similarly natural:

factor out assumptions about the outside world similar to the previous
assumptions. Notice again that the solution was

a version of the induction hypothesis as an explicit assumption.

preservation-Sum:

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Nat}\} e, \text{Nat} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Option}\} e, \text{Option} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Array}\} e, \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{WtExpr}\} e, \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Expr}\} e, \mu \text{Expr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Array}\} e, \mu \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{WtExpr}\} e, \mu \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Opt}\} e, \text{Opt} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Array}\} e, \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{WtExpr}\} e, \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Expr}\} e, \mu \text{Expr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Array}\} e, \mu \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{WtExpr}\} e, \mu \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Opt}\} e, \text{Opt} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Array}\} e, \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{WtExpr}\} e, \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Expr}\} e, \mu \text{Expr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Array}\} e, \mu \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{WtExpr}\} e, \mu \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Opt}\} e, \text{Opt} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Array}\} e, \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{WtExpr}\} e, \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Expr}\} e, \mu \text{Expr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Array}\} e, \mu \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{WtExpr}\} e, \mu \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Opt}\} e, \text{Opt} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{Array}\} e, \text{Array} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\text{WtExpr}\} e, \text{WtExpr} n) \\
\]

\[
\text{preservation-Sum} \quad \vdash \quad (\text{Sum}\{\mu \text{Expr}\} e, \mu \text{Expr} n) \\
\]
supplying the necessary lift functions and the induction hypothesis. This is indeed the case:

\[
\begin{align*}
\text{preservation} & (\text{step}^+ \text{ ste}) (\text{lift-wt-sum} \text{ wts}) \\
& = \text{preservation-Sum lift-wt-nat lift-wt-sum} \\
& \text{preservation ste wts} \\
\text{preservation} & (\text{step }[\text{ ste}]) (\text{lift-wt-array} \text{ wta}) \\
& = \text{preservation-Array lift-wt-option lift-wt-array} \\
& \text{preservation ste wta}
\end{align*}
\]

Having shown type preservation it is interesting to see the similarity between how terms are shown to be welltyped and to evaluate and how the terms are expressed in \(\mu\text{Expr}\).

Recall that each term in \(\text{Expr}\) is wrapped by a tag—given by \(\text{inn}_1\) and \(\text{inn}_2\)—and the constructor \(\text{inn}\) plays the role of recursion. To reiterate consider the convenience functions, 

\[
\begin{align*}
\text{nili} & : \text{Expr} \\
\text{nili} & = \text{inn} (\text{inn}_2 (\text{inn}_1 (\text{inn}_2 (\text{tt})))) \\
\text{nat} & : \mathbb{N} \rightarrow \text{Expr} \\
\text{nat} \ n & = \text{inn} (\text{inn}_1 (\text{inn}_2 (\text{inn}_1 (\text{nili} \ n)))) \\
\text{a} \ [\text{n}] & = \text{e} = \text{apply} \text{liftA} (\text{a} \ [\text{n}] = \text{e}) \\
\text{IE} & : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \\
\text{IE} \ n & = \text{apply} \text{liftA} (\text{a} \ ! \ n) \\
\text{E} & : \text{Expr} \rightarrow \text{Expr} \\
\text{E} & = \text{apply} \text{lift}^+ (\text{e}_1, \text{e}_2)
\end{align*}
\]

we may then ask: why is the term 

\[
\begin{align*}
\text{exp} & \text{ : } \text{Expr} \\
\text{exp} & = (\text{nili} [\text{nat} 0] = \text{nat} 1 ! \text{e} (\text{nat} 0 + \text{E} \text{nat} 1))
\end{align*}
\]

welltyped? The answer given by \(\text{WtExpr}\) is 

\[
\begin{align*}
\text{wt-exp} & : \text{WtExpr} \text{ exp TOption} \\
\text{wt-exp} & = \text{lift-wt-array} (\text{ok-lookup wta} \text{ wt+}) \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{wta} & : \text{WtExpr} (\text{nili} [\text{nat} 0] = \text{nat} 1) \text{TArray} \\
\text{wta} & = \text{lift-wt-array} \\
& \text{(ok-ins (lift-wt-array ok-nil) (lift-wt-nat 1) TArray)} \\
\text{wt+} & : \text{WtExpr} (\text{nat} 0 + \text{E} \text{nat} 1) \text{TNat} \\
\text{wt+} & = \text{lift-wt-sum} (\text{ok-sum} (\text{lift-wt-nat 0} (\text{lift-wt-nat 1})))
\end{align*}
\]

The \(\text{lift-wt-}\) functions play the same role in \(\text{WtExpr}\) as \(\text{inn}\) does in \(\text{Expr}\); 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

\[
\begin{align*}
\text{eval-exp} & : (\text{nili} [\text{nat} 0] = \text{nat} 1) ! \text{E} (\text{nat} 0 + \text{E} \text{nat} 1) \\
\text{eval-exp} & \rightarrow (\text{nili} [\text{nat} 0] = \text{nat} 1) ! \text{E} \text{nat} 1 \\
\text{eval-exp} & = \text{step }[\text{ ste]} (\text{step}^+ (\text{step}^+ \text{ ste}))
\end{align*}
\]

What does the proof that \(\text{nili} [\text{nat} 0] = \text{nat} 1 ! \text{E} \text{nat} 1\) is welltyped look like? We can compute it by invoking 

\[
\begin{align*}
\text{preservation} \text{ eval-exp wt-exp}
\end{align*}
\]

which evaluates to 

\[
\begin{align*}
\text{lift-wt-array} \\
& \text{ok-lookup} \\
& (\text{lift-wt-array} \\
& \text{(ok-ins (lift-wt-array ok-nil) (lift-wt-nat 1) (lift-wt-nat 0)))} \\
& (\text{lift-wt-nat 1})
\end{align*}
\]

7. Related Work

Independent and concurrently with our work, Delaware, et al. [7] 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 they have chosen to express inductive types using Church encodings and recursive evaluation using Mendler algebras, which requires some extra sophistication. Here we express types as data members of the family of polynomial functors and apply recursive evaluation directly. Their approach presented is further along and has shown the important level of robustness required by most languages while there are more unanswered questions regarding the method presented here.

8. Conclusion and Future Work

We should ask if we have accomplished the goal that we set out with. The language \(\text{Expr}\) was given componentwise and the boiler-plate necessary to wrap each welltyping 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 producable given a sophisticated macro system where terms can be inspected by name—set equality is non-deterministic—rather than 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 \(\text{Expr}\)—Agda is unable to prove termination and we plan to address this soon.

The language presented is quite simple, unable to express even Euclid’s algorithm, and the method of polynomial functor’s used to express \(\text{Expr}\) precludes the possibility of first class function types which are critical for functional programming. Various solutions to this problem have been proposed [5] and the area of recursion schemes is rich [6]. 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] Aydemir, et al. Mechanized Metatheory for the Masses: The PopMark Challenge. In International Conference on Theorem Proving in Higher Order Logics (TPHOLs), August 2005.

[2] Grant Malcolm. Algebraic data types and program transformations. Ph.D. thesis, Department of Computing Science, Groningen University, 1990.

[3] Luc Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. Utrecht University, 1995.

[4] Ulf Norell. Dependently Typed Programming in Agda. In Proceedings of the 4th international workshop on Types in language design and implementation (TLDI ’09). ACM, New York, NY, USA, 1-2.

[5] Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In Proceedings of the seventh international conference on Functional programming languages and computer architecture (FPCA ’99). ACM, New York, NY, USA, 324-333.

[6] Tarma Uustalu, et al. Recursion Schemes from Comonads. Nordic J. of Computing 8, 3 (September 2001), 366-390.

[7] Benjamin Delaware, Bruno C. d. S. Oliveira, Tom Schrijvers. Meta-Theory à la Carte, unpublished manuscript, July, 2012.