Propagating Rose Trees with Dependent Types to Grow Shaped ASTs Generically

Jan de Muijnck-Hughes[1 [0000−1111−2222−3333]]

Department of Computer and Information Sciences, University of Strathclyde
Jan.de-Muijnck-Hughes@strath.ac.uk

Abstract Dependently typed languages supports the construction of intrinsically-typed (well-scoped) terms, ensuring that our core language representations are *correct-by-construction*. The datatypes we use to represent Abstract Syntax Trees (ASTs) are not well-typed nor well-scoped, nor should they be. Nonetheless, ASTs are often represented as a collection of bespoke datatypes that capture what it means for an AST to be 'well-structured'. Such *artisan* crafting of our ASTs comes at the price of generality. We must define the same traversals and operations time and time again over each AST.
Taking inspiration from intrinsically-typed datatypes and compact encoding of primitives, we show how rose trees can have their shape dictated by a description contained within a dependent type. We can thus, describe (type) the structure of our ASTs similar to how we describe the concrete syntax using eBNF and embed the description inside a pre-existing datatype. Through this novel combination, we now have a single generic datatype for ASTs and single operations that act on them.

Keywords: Dependent Types · Abstract Syntax Trees · Idris

1 Introduction

Dependent types provide an expressive environment in which to *reason* about our programs. By using interactive theorem proovers that support dependent types, such as Rocq/Coq [16] and Lean [8], we can construct formal models of our languages and reason mechanically about their correctness. Such reasoning is possible as dependent types enable the *Curry-Howard* correspondence: *Types as Propositions; Proofs as Programs* [17]. Rocq/Coq and Lean are geared towards interactive theorem proving. Programming languages that support dependent types, such as Agda [15,11] and Idris(2) [4] however, enable us to *program* with dependent types.

When reasoning about programs in a dependently-typed language, we use *Intrinsically-Typed* constructions that are well-typed and well-scoped by design [1,13,12]. Intrinsically-typed constructions tend to be the output of the elaboration process and corresponds to our program's *abstract syntax*. Unambiguous syntax that we know is well structured according to the language's given type system.

Concrete syntax, on the other hand, is the input to elaboration and is constructed during lexing/parsing. Also known as an Abstract Syntax Tree (AST), our

language's concrete syntax *may* contain ill-typed constructions but nonetheless has a regular structure. The structure of an AST enables one to differentiate, for example, between expressions, declarations, types, values, references *et cetera*. The elaboration process will inform us if a given AST instance is well-structured i.e. well-typed. Common engineering practice has established 'design patterns' in which separate data structures capture salient structure within our concrete syntax. Using separate data structures, however, comes at a cost of generality. We must construct the same operations, such as traversals and functors, over each data structure for each AST we define. In languages that support deriving construction of these operations is trivial. In languages that do not support deriving, we can coalesce the data structures representing a language's AST into a single data structure. Doing so means that we can no longer capture the inherent structure of an AST, as the resulting data structure is too unregulated. With dependent types we can change that!

1.1 Contribution

We use types to capture the computational structure of our programming languages. So why not use types to capture the inherent structure of our concrete syntax? To be clear: we are aiming for ASTs that are well-structured but may also be ill-typed. Elaboration will determine if the program is ill-typed.

Singular ASTs are, however, unregulated in their shape. If we view concrete syntax trees (i.e. ASTs) as rose trees we can already construct a generic datatype to capture our program's concrete syntax. Through introduction of a *meta* type system, that is embedded within the type of our AST, we can start to regulate the shape of the rose tree's structure as we do with intrinsically typed constructions. Through addition of a meta type system we can, in fact, align the structure with that of the language's eBNF grammar [6].

This pearl describes: how we designed our shapeable rose tree; how we use the data structure; and the limitations of our approach.

1.2 Reading Guide

We have based our designs within Idris2, a general purpose programming language with support for dependent types. Ostensibly, we will provide readers with enough information to understand the pearl, however and for those not familiar with dependently typed programming, more information is available [14,18,3].

1.3 Outline

We begin in Section 2 by motivating why ASTs should be generic and need shaping. Section 3 provides a primer on dependent types which can almost be skipped those already versed in dependently-typed programming, but we ask the same readers to make a not of the `DVect` data structure. The pearl itself is documented in Section 4 and describes how dependent types enable the

construction of generic yet precise ASTs using the `AST` datatype. We illustration generic programming using `AST` in Sections 5 and 6. Sections 7 and 8 concludes with a discussion of the pearl and concluding remarks.

2 Trees Need Shaping

We begin this pearl with a motivation on why shaping ASTs is important.

Consider the AST presented in Figure 1 for a simple functional language. The language is based on the Simply-Typed Lambda Calculus with natural numbers and booleans as base types, and binary addition and conjunction as primitive operations. The datatype `FC` represents file contexts, line spans within a file and whose definition we leave abstract.

```
data Ty = FUNC FC Ty Ty | NAT FC | BOOL FC              -- Types
data Expr                                               -- Expressions
   = BTrue FC | BFalse FC | NZero FC | NSucc FC
   | Ref FC String | Lam FC String Ty Expr
   | App FC Expr Expr
   | Add FC Expr Expr | And FC Expr Expr
data Func = F FC Ty Expr                                -- Named Functions
data Decl = DFunc FC String Func | DType FC String Ty   -- Declarations
data Prog = P (List Decl) Func                          -- Programs
```

Figure 1: Concrete Syntax for a Functional Lanugage based on the Simply-Typed Lambda Calulus extended with Natural Numbers and Booleans.

The definition of our language follows standard conventions in which datatypes capture various sub-structures of the AST itself. These datatypes represent: types (`Ty`); expressions (`Expr`); named/declared functions and types (`Decl`); and programmes (`Prog`) as list of declarations followed by a single 'main' function. We highlight that declared functions *must be* typed (`Func`), and that every node in the AST *must* have its file context stored in the node. Figures 2 and 3 show a simple program written in our language and the corresponding AST.

```
idBool : Bool -> Bool
idBool = (\x -> x)

main : Bool -> Bool
main = foo(False)
```

Figure 2: Example Program and its AST.

The structure of our AST is neat in several ways. We have stratified the various substructures of the AST, such that our view of the AST can be narrowed when

```
eg : Prog
eg
  = P [DFunc MkFC "idBool"                                  -- Function
        (F MkFC
           (FUNC MkFC (BOOL MkFC) (BOOL MkFC))              -- + Type sig
           (Lam MkFC "x" (BOOL MkFC) (Ref MkFC "x")))       -- + Body
      ]
      (F MkFC                                               -- Main
         (FUNC MkFC (BOOL MkFC) (BOOL MkFC))                -- + Type sig
         (App MkFC (Ref MkFC "idBool") (BFalse MkFC)))      -- + Body
```

Figure 3: Example Program and its AST.

operating over the AST. For example, during elaboration we know that when operating on expressions (`Expr`) we will not encounter types or declarations. That is, pattern matching need not have a general failing case. We can structure other similar operations (pretty printing and quasiquoting, for example) accordingly. Further, we highlight that the grouping of the datatypes, together with use of algebraic datatypes and lists, follows almost how one would present the concrete syntax as an (a/e)BNF grammar [6].

The structure of our AST is, however, combersome in several ways. First, the multiple datatypes means that simple traversal over the structure requires multiple functions. We cannot declare Functor instances for these datatypes. To do so we need to redefine our definitions so that they are indexed by some value, usually the file context itself—see Figure 4. Even then, we are nonetheless required to declare multiple separate functor instances.

```
data Ty a = FUNC a (Ty a) (Ty a) | NAT a | BOOL a             -- Types
data Expr a
   = BTrue a | BFalse a | NZero  a | NSucc  a                -- Exprs
   | Ref a String | Lam a String (Ty a) (Expr a)
   | App a (Expr a) (Expr a)
   | Add a (Expr a) (Expr a) | And a (Expr a) (Expr a)
data Func a = F a (Ty a) (Expr a)                             -- Funcs
data Decl a = DFunc a String (Func a) | DType a String (Ty a) -- Decls
data Prog a = P (List (Decl a)) (Func a)                      -- Programs
```

Figure 4: Concrete Syntax for a Simple Functional Language using Indexed Datatypes.

Generally speaking, any operation (pretty printing and quasiquoting, for example) on the AST requires one to operate over multiple datastructures. Depending on the complexity of the language definition, that is *if* our structures are mutually defined, this can affect the totality of operations over the AST. If, however, we consider that our ASTs are infact rose trees then we can coalesce our programs into a single definition that makes writing total operations easier–

Figure 5. Unfortunately, the resulting generality comes at the cost of precision. We no longer have control over what the AST's sub-structure should be. For instance, our example AST from Figure 3 (if written using the AST from Figure 5) can be malformed through types appearing in unexpected positions. That is, types and functional declarations are *not* expressions and we should not expect them to be. Thus, during elaboration we *need* to ensure that *all* failing cases are handled and their patterns caught.

```
data AST a
   = FUNC a (AST a) (AST a) | NAT a | BOOL a        -- Types
   | BTrue a | BFalse a | NZero a | NSucc a         -- Exprs
   | Ref a String | Lam a String (AST a) (AST a)
   | App a (AST a) (AST a)
   | Add a (AST a) (AST a) | And a (AST a) (AST a)
   | F a (AST a) (AST a)                            -- Funcs
   | DFunc a String (AST a)                         -- Decls
   | DType a String (AST a)
   | P (List (AST a)) (AST a)                       -- Progs
```

Figure 5: Concrete Syntax for a Simple Functional Language as a Single Indexed Datatype.

We can observe that when parsing raw text (regardless if one uses parser combinators or generators) we naturally avoid constructing generic parsers for our ASTs. We construct (sub-)parsers according to the shape of the resulting abstract syntax tree. To throw this information away for more generic operations on trees, and then reconstitute this information because of elaboration, is pointless. Through a novel application of dependent types we can get the best of both worlds. We explore this in the next section.

3 A Primer on Dependent Types.

Before we can consider a generic data structure for ASTs, let us first consider what dependent types give us: type-directed precision over our programs. The classic exemplar datatype that illustrates how dependent types offer such type-directed precision, which in turn is instrumental for our needs, are 'vectors'. Vectors, commonly known as *list with lengths*, are variants of the standard `List` datatype that keep track of the lists length directly within the type signature.

Within Idris2 (and Agda) we have access to standard algebraic datatypes. *Sums of Products*, through which one can define natural numbers.

```
data Nat = Z | S Nat
```

Where the data constructor `Z` represents the number zero, and `S` is the successor of a natural number. We can also define lists:

```
data List a = Nil | Cons a (List a)
```

The datatype `List` is an indexed datatype (i.e. is polymorphic) where the type variable `a` represents the type of elements. Much like natural numbers: lists are either empty (`Nil`); or the extension (`Cons`) of an element to an existing list.

Datatypes in Idris, however, are not algebraic datatypes. Datatypes are, instead, inductive families [7]. The previous definition for lists is short hand for a double declaration of a type constructor and a data constructor. The corresponding 'full' declaration is:

```
data List : (type : Type) -> Type where
  Nil  : List a
  Cons : (head : a) -> (tail : List a) -> List a
```

Here `List` is a type constructor whose arguments represent the indices (or parameters) of the family. In the case of `List`, there is only one index: the `type` of list elements. Thus our family is polymorphic. The data constructors, on the other hand, capture ways in which we can create inhabitants of this type family. `Nil` represents the empty list, and the implicit argument `a` is a type-level variable stating that the type is yet to be known. `Cons` captures how lists are extended. The argument `head` is the element to be added, which must be of the same type as the list. The argument `tail` is the remaining elements of the list.

In a dependently typed setting, however, our types can depend on more than *just* other types. Types can depend on values.

Consider the following standard definition for 'lists with length':

```
data Vect : (s : Nat) -> (type : Type) -> Type where
  Empty : Vect Z a
  Cons  : (head : a) -> (tail : Vect k  a) -> Vect (S k) a
```

The indexed family, (`Vect`), depends on the type of elements (`type`) *and* the current length of the list (`s`) itself. The type definition gives the length first as type-level partial application, and point free programming, means that we need not always give the type explicitly. Much like `List` the data constructors follow the same shape, the major difference is that now we keep track of the length of the list as the list grows. That is, empty lists have length zero and we increase the length by one when a single element is added.

With the extra type-level information we can start to provide more precise types for operations on vectors. For example, list append.

```
append : Vect a type -> Vect b type -> Vect (plus a b) type
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Here the type signature for appending is very precise and states that the resulting vector must, not only have the same element types, but that the length must be the sum of the lengths of the two input vectors.

We can take dependent types further and use type-level values to reason about our datatypes. Let us consider the following predicate on natural numbers (`NonZero`) that ensures a given natural number is non-zero.

```
data NonZero : (n : Nat) -> Type where
  IsNonZero : NonZero (S k)
```

`NonZero` is an inductive family whose type constructor operates on a single natural number. Inhabitants of `NonZero` are specified through a single data constructor (`IsNonZero`) that requires the type-level value to be an instance of the 'successor' constructor i.e. is non-zero. The shape of the implicit argument, `k`, need not be known. We can declare similar datatypes for `Vect`, and go further and reason about whole programs [18].

Instead let us look at a predicate that reasons about the elements within a vector. Figure 6 presents `DVect`, a datatype that not only encodes a vector within its type but also captures a second datatype that acts on the elements within the vector. `DVect` quantifies over the list `xs`, that contains elements of type `type`, with the predicate `typeE`. Predicates are datatypes that acts on values, encoding some 'truth'. Within `DVect`, however, the invariant `l` makes explicit the length of the quantified list. The standard libraries for both Idris and Agda will represent such quantifiers as `All` with the types of the list made implicit. For our needs later on (see Section 4) the explicit type-level declarations are important.

```
data DVect : (type  : Type)
          -> (typeE : type -> Type)
          -> (l     : Nat)
          -> (xs    : Vect l type)
          -> Type
  where
    Empty : DVect type typeE Z Nil
    Cons : (head :            typeE        x)
        -> (tail : DVect type typeE    k      xs)
                -> DVect type typeE (S k) (x::xs)
```

Figure 6: `DVect` a Vector Quantifier with explicit indices.

To further evidence the need for `DVect`, let us use `NonZero` to construct instances whose type level values can only contain non-zero naturals numbers.

```
threeOnes : DVect Nat NonZero
                  (S          (S          (S          Z)))
                  (Cons (S Z) (Cons (S Z) (Cons (S Z) Empty))
threeOnes
  = (Cons IsNonZero (Cons IsNonZero (Cons IsNonZero Empty)))
```

Idris' elaboration and syntax sugaring means we can write `threeOnes` in a more user friendly way. Natural numbers can be written using digits. Further, if

we use `(::)` for `Cons` and `Nil` for `Empty` as our data constructors for `Vect` *and* `DVect` (declared in separate namespaces) we also gain list syntax. So now our `threeOnes` can become:

```
threeOnes : DVect Nat NonZero 3 [1,1,1]
threeOnes = [IsNonZero, IsNonZero, IsNonZero]
```

Here we conclude our short primer on dependent types. We have taken the time to describe dependent types through 'lists with length' as we can use the length invariant to dictate the number of siblings at each node in our rose tree. Even more so, `DVect` enables us to maintain an invariant between the length of a type-level vector and a value level vector. In the next section we will see this in action to get our generic data structure for ASTs.

4 The Pearl

Rose trees are, *just*, trees with unbounded nodes at each level. For instance:

```
data RoseTree : (type : Type) -> Type where
  Branch : (value : type)
        -> (nodes : List (RoseTree type))
                  -> RoseTree type
```

Much like `List`, `RoseTree` is indexed by the type of elements. `RoseTree` has a single data constructor `Branch`, that takes as arguments a `value` to be stored at each branch and the `nodes` at each level. `RoseTree` is inductive in `nodes`, taking a list of `RoseTree`. With this construction leaf nodes are an instance of `RoseTree` with the empty list.

We can use `RoseTree` as the basis for how we can encode ASTs. The examples from Section 2 shows how a bespoke algebraic datatype can represent a bespoke rose tree for our language. To make this approach more generic, we can replace `value` with a unique label representing each concrete term. At each level can have a differing number of nodes and the shape of each node cannot be constrained. More specifically, each level of our AST contains a shape-label, a state (i.e. file context), and the node's children. First we give the `Shape` datatypes:

```
data Shape = FUNC | Nat | BOOL              -- Types
           | BTrue | BFalse | NZero | NSucc -- Exprs
           | Ref String | Lam | App
           | Add | And
           | F                              -- Funcs
           | DFunc String | DType String    -- Decls
           | P                              -- Prog
```

and then the AST itself:

```
data AST : (shape,state : Type) -> Type where
  Branch : (s     : shape)
        -> (st    : state)
        -> (nodes : List (AST shape state))
                 -> AST shape state
```

String names (i.e. references) are stored in `Shape` as they do not belong in the tree itself. With this approach we have presented a single definition for an AST in which we can label nodes, store state(s), and grow a (rose) tree. We must remember, however, that ASTs will have a different number of nodes at each level. Our approach does not support restriction on the number of children, nor the relationship between parent and child. To improve/constrain the relationship we need to revisit the type-level definition of `Shape` and how list of nodes are specified.

We can use a vector to control how many nodes there are at each level. Nonetheless, we need to link a shape with the allowed children *and* what those children can be. This is where smart use of our `DVect` and `Vect` datatypes enable this link to be made. To do so we must first introduce a kind system that categorises the different sub-structures of our AST. We will use kind, even though we are adding types, to not confuse with existing uses of `Ty`.

```
data Kind = TYPE | EXPRS | FUNC | DECL | DECLS | PROG
```

We can then use `Kind` to index the type of `Shape` with the ASTs typing rules, which we can encode using a type-level vector. We also make the arity of each shape explicit. Figure 7 shows these changes to `Shape`.

`Shape` now captures, at the type-level, the various (sub-)structures of our AST [1] and the relationships between each structure. We have also made smart use of Idris' implicits when defining declarations. With our approach we can specify a distinct substructure that enables declaratiosn to be a list of aribtrary length. At runtime the precise length will be determined. An alternative approach would be to nest declarations as one would let-bindings.

Similar to declarations, we can extend the definition of functions and types from being of fixed arity to being n-ary. By introduction of new data constructors for `Kind` we can describe these n-ary structures, and extend `Shape` were necessary. For example, if we wish to add n-ary function types we can extend `Kind` with `ARGS` and use `ARGS` to 'type' new shapes. The new data structures, to `Shape`, could be: `TyArgs`—to represent the inputs to n-ary function types; and `TyFunc`—that uses `Tys` to represent the function type itself. With the final constructors looking like:

```
TyArgs : Shape ARGS n (replicate n TYPE)
TyFunc : Shape TYPE 2 [ARGS,TYPE]
```

[1] within Idris (and Agda) we can 'double-up' data constructors that have the same type.

```
data Shape : (k     : Kind)
          -> (arity : Nat)
          -> (rules : Vect arity kind)
                   -> Type
  where
    TyFUNC : Shape TYPE 2 [TYPE, TYPE]       -- Types
    TyNAT, TyBOOL : Shape TYPE 0 Nil

    BFalse, BTrue, NZero : Shape EXPR 0 Nil  -- Exprs
    NSucc : Shape EXPR 1 [EXPR]

    Ref : String -> Shape EXPR 0 Nil
    Lam : Shape EXPR 2 [TYPE, EXPR]
    App : Shape EXPR 2 [EXPR, EXPR]

    Add, And : Shape EXPR 2 [EXPR, EXPR]

    F : Shape FUNC 2 [TYPE, EXPR]            -- Funcs
    DFunc : String -> Shape DECL 1 [FUNC]    -- Decl
    DType : String -> Shape DECL 1 [TYPE]

    Decls : Shape DECLS n (replicate n DECL) -- Decls
    Prog  : Shape PROG  2 [DECLS, FUNC]      -- Prog
```

Figure 7: Shape Data structure representing our langaguage's Concrete Syntax.

Whilst we have made the arities explicit, we have seen with declarations that they need not be. In fact, we can use dependent types to supply more (type-level) information (as an argument to the shape's data constructor) that can influence what, and how many, structures are listed. For instance, coalescing the construcors for individual declarations themselves.

We can now revisit the definition of `AST` and ensure that the newly provided type-level information enforces the described structure. We do so using `DVect`.

```
data AST : (k     : Kind)
        -> (index : Type)
                 -> Type
  where

  Branch : (shape : Shape k n meta)
        -> (annot : a)
        -> (nodes : DVect kind (\k => AST k a) n meta)
                 -> AST k a
```

Although expressive, the current definition of `AST` is still bespoke to our language. We can parameterise `AST` further with a description of what shapes should look like. Figure 8 presents our final datatype; our functional pearl. Within the definition of `AST`, the bound implicit `node`, must be bound for Idris'

elaboration to succeed. To ensure that the type is not used at runtime, we explicit mark the variable as erased. For more information about quantities in Idris2 is available [4,2].

```
data AST : (desc : (k     : kind)
                -> (arity : Nat)
                -> (meta  : Vect arity kind)
                          -> Type)
        -> (k     : kind)
        -> (index : Type)
                 -> Type
  where

  Branch : {0 node  : (k : kind) -> (n : Nat) -> Vect n kind -> Type}
        -> (  desc  : node k n meta)
        -> (  annot : a)
        -> (  nodes : DVect kind (\k => AST node k a) n meta)
                   -> AST node k a
```

Figure 8: A Generic Shapeable Datatype for ASTs.

Using `AST`, we can create a type-synonym that represents our simple functional language's AST:

```
ShapedAST : (k : Kind) -> Type
ShapedAST k = AST Shape k FC
```

Even though we can now show how we can build ASTs using `AST` (when paired with a shape) the *raw* data constructors can be unwieldly. Section 6 discusses how we can make ASTs easier to build through careful API construction. First, however, the next section details how, we can operate generically on `AST` instances.

5 Operating on Trees

With our shapeable datatype `AST`, we can now construct ASTs. We begin this section by looking at how `AST` enables generic operations over an AST. Section 6 shows how dependent types enables design of a generic API for constructing instances of `AST`.

Realising generic operations over `AST` is not complicated. As `AST` is a rose tree, we can construct generic operations that follows those designed for rose trees. We must remember, however, to ensure that type-level invariant hold. We show this by presenting functor and show instances.

Figure 9 presents *a* functor instance for `AST` in which the map acts on the annotation i.e. value stored at each branch. The design is standard, applying the function `f` first to the annotation and then to the child nodes. A helper function `mapV`, taken from the `DVect` library and whose type we present in Figure 9,

ensures that the type-level invariant for nodes holds as the function is recursively applied.

```
map : (f : a -> b) -> AST k ns a -> AST k ns b
map f (Branch kind annot nodes)
  = Branch kind (f annot) (mapV f nodes)

mapV : (f : a -> b) -> DVect d (\k => AST node k a) n ks
                    -> DVect d (\k => AST node k b) n ks
```

Figure 9: Functor Instance of `AST`.

As with functor, instances of `Show` are straightforward. Figure 10 presents a generic 'show' instance for `AST`. Whilst we can use interface constraints [2] to search for show instances for shapes and annotations, we instead present a completely generic function. As for functor, we rely on a pre-exiting `show` function (type signature also given in Figure 10) for `DVect` instances. We also use Idris2's string interpolation to cut down on the use of string combinators.

```
show : {0 shape : (k : kind) -> (n : Nat) -> Vect n kind -> Type}
    -> (  showS : forall k, n, ms . (shape k n ms) -> String)
    -> (  showA : a -> String)
    => (  ast   : AST shape k a)
                  -> String
show showS showA (Branch k a ns)
  = "(Branch \{showS k} \{showA a} \{showDVect showS showA nns})"

showDVect : (showFunc : forall a . elemTy a -> String)
         -> (l        : DVect aTy elemTy n as)
                     -> String
```

Figure 10: Generic Show Instance for `AST`.

6 Growing Trees

We move on from generic operations to examine how we can make node construction that little bit easier. We begin by providing a type-synonym to capture the type constructors that all shapes must conform to. This synonym is `SHAPE`, and follows the description given in Section 4:

```
SHAPE : (k : Type) -> Type
SHAPE k = (kind : k) -> (n : Nat) -> (desc : Vect n k) -> Type
```

[2] type-class constraints

We begin by looking at the construction of leaf nodes. Figure 11 shows the: type constructor `NULL`—which ensures that the node does not have any children; and the data constructor `null`—that constructs the `AST` instance.

```
NULL : (type : SHAPE k)
    -> (kind : k)
            -> Type
NULL ty kind
  =  ty kind 0 Nil
```

(a) Type Synonym

```
null : (shape : NULL type kind)
    -> (annot : a)
            -> AST type kind a
null shape annot
  = Branch shape annot []
```

(b) Data Constructor

Figure 11: API for Constructing Leaf Nodes.

The same style of 'type-data-factories' used for leaf nodes is used for unary and binary nodes. We illustrate the binary case in Figure 12; the unary case is much the same. We can even provide such factories for ternary nodes which we do not provide Although one could provide bespoke constructors for four or more items, going beyond that number implies that the raw API itself should be directly used.

```
BIN : (type    : SHAPE k)
   -> (kind    : k)
   -> (inA,inB : k)
             -> Type
BIN ty a b c
  = ty a 2 [b,c]
```

(a) Type Synonym

```
bin : {kindB, kindC : _}
   -> (shape : BIN type kindA
                        kindB
                        kindC)
   -> (annot : a)
   -> (inA   : AST type kindB a)
   -> (inB   : AST type kindC a)
            -> AST type kindA a
bin shape annot iA iB
  = Branch shape annot [iA,iB]
```

(b) Data Constructor

Figure 12: API for Constructing Binary Nodes.

With this API constructing `AST` instances is that little bit easier. We illustrate this in Figure 13 which shows how the program from Figure 2 is represented. If we compare both the ASTs in Figures 3 and 13 they are similar in structure, data factories aside. The core difference is that the inherent data structures are generic but have been constrained to offer the same structures.

```
prog : ShapedAST PROG
prog =
  bin Prog MkFC
      (Branch Decls MkFC
        [(un (DFunc "idBool") MkFC
              (bin F MkFC
                   (bin TYFUNC MkFC
                        (null TYBOOL MkFC) (null TYBOOL MkFC))
                   (bin (Lam "x") MkFC
                        (null TYBOOL MkFC) (null (Ref "x") MkFC))
                   ))
        ])
      (bin F MkFC
           (bin TYFUNC MkFC
                (null TYBOOL MkFC) (null TYBOOL MkFC))
           (bin App MkFC
                (null (Ref "idBool") MkFC) (null BFalse MkFC)))
```

Figure 13: Running Example as a ShapedAST Prog.

7 Discussion

Dependent types are not just for proving, and real (practical) systems can be built using them [4,5]. In our research [9,10] we have built many a mechanised language in Idris2 and made these mechanised implementations run. It was when building the lexers and parsers for these languages that we noticed the same boiler plate ASTs being written. Specifically, when amending file contexts (ensuring that they are all populated with the same file name) we were writing the same traversals time and time again. With `AST` we removed the need to *copy, paste, and amend* the same code.

Using a 'shape' datatype to hold structural information is not necessarily new. When designing Vélo [9], we encapsulated our primitive terms via a pointwise lifting to an algebraic specification. The resulting (type-level) specification presented arguments as a list and then the return type. We inverted the relation presenting the return type first and sub-arguments second. Our inversion highlights the link between the specification and grammar rules. Moreover, we have made the arity explicit to enable more precise modelling of n-ary structures and ensuring that, during elaboration, the length is found.

Need todo? – Related work...

8 Conclusion

Dependent types enable rich specification of type-level invariants and generic programming so that we can construct a generic shapeable/typeable data structure that captures parsing structure. There is a trade-off between how much structure

a datatype can encapsulate and how generic the resulting structure is. Using bespoke datatypes to capture the inherent structure of an AST is natural. Do so, however, looses the freedom to operate over a single structure and one must walk the line between many types. Much like intrinsically-typed terms, the provision of a 'type system' within the type enables us coalesce into a single datatype the different structures within an AST. Interestingly, the resulting type system rather somewhat mirrors the production rules one would write for the concrete syntax and how one would capture this structure using separate datatypes.

At the end of the day, types describe what values can do and datatypes describe how data can be structured. With dependent types we use values to create types that describe how data *should* be structured. How this can be applied to other typed data is worth investigating.

References

1. Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type and scope safe universe of syntaxes with binding: their semantics and proofs. Proc. ACM Program. Lang. **2**(ICFP), 90:1–90:30 (2018). https://doi.org/10.1145/3236785, https://doi.org/10.1145/3236785
2. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189, https://doi.org/10.1145/3209108.3209189
3. Brady, E.C.: Type-Driven Development with Idris. Manning (2017), https://www.manning.com/books/type-driven-development-with-idris
4. Brady, E.C.: Idris 2: Quantitative type theory in practice. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). pp. 9:1–9:26 (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.9, https://doi.org/10.4230/LIPIcs.ECOOP.2021.9
5. Chapman, J., Kireev, R., Nester, C., Wadler, P.: System F in agda, for fun and profit. In: Hutton, G. (ed.) Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11825, pp. 255–297. Springer (2019). https://doi.org/10.1007/978-3-030-33636-3_10, https://doi.org/10.1007/978-3-030-33636-3_10
6. Crocker, D., Overell, P.: Augmented BNF for Syntax Specifications: ABNF. RFC 5234 (Jan 2008). https://doi.org/10.17487/RFC5234, https://www.rfc-editor.org/info/rfc5234
7. Dybjer, P.: Inductive families. Formal Aspects Comput. **6**(4), 440–465 (1994). https://doi.org/10.1007/BF01211308, https://doi.org/10.1007/BF01211308
8. Moura, L.d., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction – CADE 28. pp. 625–635. Springer International Publishing, Cham (2021)
9. de Muijnck-Hughes, J., Allais, G., Brady, E.C.: Type theory as a language workbench. In: Lämmel, R., Mosses, P.D., Steimann, F. (eds.) Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands. OASIcs, vol. 109, pp. 9:1–9:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/OASIcs.EVCS.2023.9, https://doi.org/10.4230/OASIcs.EVCS.2023.9

10. de Muijnck-Hughes, J., Vanderbauwhede, W.: Wiring circuits is easy as {0, 1, ω}, or is it.. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.8, https://doi.org/10.4230/LIPIcs.ECOOP.2023.8
11. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
12. Poulsen, C.B., Rouvoet, A., Tolmach, A., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for imperative languages. Proc. ACM Program. Lang. **2**(POPL), 16:1–16:34 (2018). https://doi.org/10.1145/3158104, https://doi.org/10.1145/3158104
13. van der Rest, C., Poulsen, C.B., Rouvoet, A., Visser, E., Mosses, P.D.: Intrinsically-typed definitional interpreters à la carte. Proc. ACM Program. Lang. **6**(OOPSLA2), 1903–1932 (2022). https://doi.org/10.1145/3563355, https://doi.org/10.1145/3563355
14. Stump, A.: Verified Functional Programming in Agda, ACM Books, vol. 9. ACM (2016). https://doi.org/10.1145/2841316, https://doi.org/10.1145/2841316
15. Team, T.A.D.: Agda (2023), https://github.com/agda/agda
16. Team, T.C.P.A.D.: The Coq poof assistant (2023), https://coq.inria.fr/
17. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). https://doi.org/10.1145/2699407, https://doi.org/10.1145/2699407
18. Wadler, P., Kokke, W., Siek, J.G.: Programming Language Foundations in Agda (Aug 2022), https://plfa.inf.ed.ac.uk/22.08/