# Automating Code Generation for Higher Inductive Types

Paventhan Vivekanandan
School of Informatics, Computing and Engineering
Indiana University Bloomington
USA
pvivekan@indiana.edu

David Christiansen
School of Informatics, Computing and Engineering
Indiana University Bloomington
USA
davidchr@indiana.edu

## Abstract

This paper provides a sample of a LaTeX document which conforms, somewhat loosely, to the formatting guidelines for ACM SIG Proceedings.

***Keywords*** ACM proceedings, LaTeX, text tagging

## 1 Introduction

Type theory, a foundational language of mathematics, is a functional programming language and a theorem prover. We can view a type in type theory as a topological space in homotopy theory or a groupoid in a category theory. Elements of a type corresponds to points in a topological space and elements of identity type corresponds to paths in a topological space. Homotopy type theory extends Martin-Löf's intensional type theory by adding the notion of higher inductive type and univalence axiom. A higher inductive type is a general schema for defining new types using constructors for points and paths. The paths specified by a higher inductive type are realized by equivalences in the universe, and the mapping of paths to equivalences is made possible by univalence axiom [**?** ].

Proof assistants like Coq [**?** ] and Agda [**?** ] does not have built-in mechanisms to support the specification of higher inductive types. However, we can work with higher inductive types in Coq and Agda by postulating them using axioms. In this approach, the elimination rule acting on the higher inductive type does not compute. Dan Licata suggested an alternative method to work with higher inductive types using type abstraction [**?** ]. It involves defining a higher inductive type using an abstract type inside a module. The module consists of a boiler-plate code segment which defines the higher inductive type using a private base type. Inside the module, the recursion and the induction principles acts on the constructors of the private base type. The abstract type is then exported allowing the reduction rules for point constructors to hold definitionally.

Agda is a proof-assistant based on Martin-Löf's intensional type theory. In Agda, a metaprogram, which is a programs used to construct other programs, runs in a built-in type checking monad. Inspired by Idris's elaborator [**?** ], Agda's reflection library expose an interface to the type checker using the type checking monad which allows for the construction of more interesting metaprograms. The type checking monad provides the interface to the Agda type checker through a set of primitive operations which can be used to retrieve static type information regarding various code segments. The primitive operations can also be used to build code fragments using constructs of abstract syntax tree, and to convert an abstract syntax tree to its concrete syntax.

In this paper, we discuss a tool used to automate the generation of bolier-plate code segments of the higher inductive type and its reduction rules. This tool extensively uses the Agda reflection library and it builds the abstract syntax tree of the code segments with static type informations obtained using the reflection primitives. The generated code is then brought into scope by another top-level reflection primitive. More specifically, we discuss the following contributions.

- We introduce a tool to automate construction of the recursion and the induction principles for inductive types with constructors taking zero arguments, one or more arguments, and the type being defined itself as an argument.
- We discuss the automation of code generation of boiler-plate code for a higher inductive type defined inside a module. The boiler-plate code depends on a base type defined as private inside the module. The constructors of the private base type are not accessible outside the module.
- We discuss the automation of code generation of the elimination and the computation rules for a higher inductive type with point and path constructors. We demonstrate the code generation of the reduction rules, specified as postulates, for the path constructors of the higher inductive type.
- We discuss the automation of code generation for patch theory [**?** ] implementation enriched with patches of encryption.

We automated the code generation of approximately 1500 lines of code of the patch theory implementation with just

*TyDe'18, St. Louis, MO, USA*
2018. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

70 lines of automation code. The automation code extensively uses the static type information queried from the type checker using the reflection primitives.

## 2 Background

### 2.1 Higher Inductive Type

In homotopy type theory, an element of an *identity type* is used to model the notion of a path in a topological space or a morphism in a groupoid. An element of an identity type $a =_A b$ states that $a$ and $b$ are propositionally equal. In type theory, *propositional equality* is a proof-relevant notion of equality, internal to the theory, expressed by identity types. We also have a proof-irrelevant notion of equality, external to the theory, known as *judgmental equality* or *definitional equality*. Definitional equality is used to express equality by definition. For instance, when we have a function $f : Nat \to Nat$ defined as $f(x) = x^2$ then the expression $f(3)$ is definitionally equal to $3^2$.

A higher inductive type extends normal inductive type by providing constructors for generating paths in addition to providing constructors for generating points. For example, consider the following higher inductive type definition of circle.

```
data Circle : Set where
  base : Circle         -- point constructor
  loop : base ≡ base    -- path constructor
```

In Agda and Coq, we don't have built-in primitives to support the definition of `Circle`. However, the higher inductive types including `Circle` can be specified using Dan Licata's method [**?** ]. According to this method, a higher inductive type is defined using type abstraction. For example, `Circle` is defined using Dan Licata's method as follows.

```
module Circle where
  private
    data S* : Set where
      base* : S*

  S : Set
  S = S*

  base : S
  base = base*

  postulate
    loop : base ≡ base
```

Inside the module `Circle`, the type `S` is defined using a private type `S*`. The constructor `base` is defined using `base*` and the path `loop` is given as a propositional equality. The recursion and the induction principles are defined to pattern match on the constructor `base*` of the type `S*`. The clients of `Circle` will not have access to the constructor `base*` of the private type `S*` as it is not visible outside the module. The clients only way of access to the constructor is through the elimination rules `recS` and `indS` defined as follows.

```

  recS : {C : Set} →
    (cbase : C) →
    (cloop : cbase ≡ cbase) →
    S → C
  recS cbase cloop base* = cbase

  indS : {C : S → Set} →
    (cbase : C base) →
    (cloop : transport C loop cbase ≡ cbase) →
    (circle : S) → C circle
  indS cbase cloop base* = cbase

```

`recS` and `indS` ignores the path argument and simply computes to the appropriate answer for the point constructor. The computational behavior for the path constructor loop is postulated using reduction rules `βrecS` and `βindS` defined as follows.

```

  postulate
    βrecS : {C : Set} →
      (cbase : C) →
      (cloop : cbase ≡ cbase) →
      ap (recS cbase cloop) loop ≡ cloop

    βindS : {C : S → Set} →
      (cbase : C base) →
      (cloop : transport C loop cbase ≡ cbase) →
      apd (indS {C} cbase cloop) loop ≡ cloop

```

### 2.2 Agda Reflection

Agda has a reflection library that enables compile time metaprogramming. The reflection library provides an interface to the Agda type checker through a built-in type checking `TC` monad. Using the reflection library and its metaprogramming interface, we can convert a code fragment to its corresponding abstract syntax tree, and we can also convert the abstract representation back to its concrete agda code. Internally, agda translates the reflection code to a unique reflection syntax. The core library of Agda is enriched with functions supporting translations from reflection syntax to abstract syntax and from abstract syntax to concrete syntax and vice-versa. In this section, we will see usage examples for some of the primitive operations specified in the Agda reflection library which are used by the tool described in this paper. More information on the reflection library can be found in the Agda documentation [**?** ].

In Agda, macros are used to construct a metaprogram that executes in a `Term` position. Macros are functions of type `t1 → t2 → ... → Term → TC ⊤` defined inside a macro block. During a macro invocation, the last argument of type `Term` is provided by the type checker, and it represents the metavariable instantiated with the result of the macro invocation. Also, the arguments of type `Term`

and Name are automatically quoted. For example, consider the following macro invocation on an identity function.

```
macro
  mc1 : Term → Term → TC ⊤
  mc1 exp hole = bindTC (quoteTC exp)
                   (λ exp' → unify hole exp')

sampleTerm : Term
sampleTerm = mc1 (λ (n : Nat) → n)
```

The value of sampleTerm is as follows:

$$lam\ visible\ (abs\ "n"\ (var\ 0\ []))\qquad(1)$$

In (1), the constructor *lam* encodes the abstract representation of the identity function. The argument n is represented as explicit and the body of the lambda function refers to n using de-bruijn index.

Term is the central type that defines an abstract syntax tree representation for the Agda terms. It is mapped to the AGDATERM built-in. The surface syntax for reflection primitives is not fixed, and so they are mapped to their corresponding built-ins, using BUILTIN pragma, to inform the type checker about the reflection concept being defined. In the above macro, the quoteTC primitive converts a concrete agda syntax into its Term representation. unify performs the unification of two terms and attempts to solve any meta variables present during the unification process. bindTC unwraps the result of (quoteTC exp), which is captured in exp', and passes it to unify for the unification process.

The unquoteTC primitive converts the abstract syntax tree representation of Term into its concrete Agda syntax. Consider the following example:

```
macro
  mc2 : Term → Term → TC ⊤
  mc2 exp hole = bindTC (unquoteTC exp)
                   (λ exp' → unify hole exp')

sampleSyntax : Nat → Nat
sampleSyntax = mc2 (lam visible (abs "n" (var 0 [])))
```

Macro mc2 converts the abstract representation of the identity function given by (1) to its concrete syntax. We also have a type Name, an internal identifier mapped to a built-in type QName. In Agda, Name literals are created using quote primitive.

Metaprograms to create top-level definitions are brought into scope and executed using unquoteDecl or unquoteDef primitive. A function of a given type is declared using declareDef, and is later defined by defineFun. A postulate of a given type is defined using declarePostulate. Newly defined functions and postulates are brought into scope and executed by unquoteDecl. The usage of declareDef, defineFun and unquoteDecl is best illustrated with the following example:

```

plus : Nat → Nat → Nat
plus zero b = b
plus (suc n) b = suc (plus n b)

```

The above function computes addition of two natural numbers. It can be defined using Agda reflection primitives as follows:

```

pattern vArg x = arg (arg-info visible relevant) x
pattern _`⇒_ a b = pi (vArg a) (abs "_" b)
pattern `Nat = def (quote Nat) []

unquoteDecl plus =
  bindTC (declareDef (vArg plus) (`Nat `⇒ `Nat `⇒ `Nat)) λ _ →
      defineFun plus
        (clause (vArg (con (quote zero) []) ::
                       vArg (var "y") :: [])
           (var 0 []) ::
         clause (vArg (con (quote suc)
                       (vArg (var "x") :: [])) ::
                 vArg (var "y") :: [])
                 (con (quote suc)
                      (vArg
                          (def plus
                             (vArg (var 1 []) ::
                              vArg (var 0 []) :: [])) :: [])) :: [])

```

In the above code, declareDef declares the type of plus using the constructor pi, and defineFun uses the declared type information to define the clauses for plus using constructors con and def. Finally, the function plus is brought into scope by unquoteDecl. Terms use the de-bruijn index to refer to the declared variables. For example, the variable y declared at line 10 is referenced using a de-bruijn index 0 at line 11. vArg encodes the visibility and relevance informations of the arguments to the reflection constructs.

In the next section, we will see automation of code generation for normal inductive types. In section 4, we will see automation of code generation for higher inductive types, and in section 5, we will revisit patch theory [? ] application and discuss automation of code generation for the higher inductive types defined to model Darc's version control system.

## 3 Code Generation for Inductive Type

An inductive type X is a type that is freely generated by a finite collection of constructors. The constructors of X are functions with zero or more arguments and codomain X. The constructors can also take an element of type X itself as an argument, but only strictly positively. In Agda reflection library, data-type of type Definition stores the constructors of an inductive type as a list of Name. The type of a constructor can be retrieved by giving its Name as an input to the getType primitive. In the following subsections, we will discuss how to use the constructor informations to generate code for the elimination rules of an inductive type.

```
(pi (vArg
   (pi (vArg `A) (abs "_" `W)))     -- f1 : A → W
   (abs "_" (pi (vArg `B)           -- B
   (abs "_" (pi (vArg `W)           -- W
   (abs "_" `W))))))                -- W
```

**Figure 1.** Abstract syntax tree for constructor $g$

### 3.1 Non-dependent Eliminator

For an inductive type $W$, with a constructor $g$, the recursion principle says that to define a mapping $f : W \to P$, it suffices to define the action of $f$ on input $g$. For example, the recursion principle for `Nat` says that a mapping `f : Nat → P` can be given by defining the action of `f` on the constructors `zero` and `(suc x)` for `x : Nat`.

The constructor $g$ can take zero or more arguments including an element of the type $W$. Lets define the type of $g$ as follows.

$$g : (A \to W) \to B \to W \to W \tag{2}$$

To define the action of $f$ on input $g$, we need a function $d$ of the following type.

$$d : (A \to W) \to (A \to P) \to B \to W \to P \to P \tag{3}$$

The function $f$ will map the constructor $g$ to $d$. To construct the type of the recursion rule for $W$, we need to build the type of $d$. We can retrieve the static type information of $g$ using reflection primitives, and use that to construct the type of $d$. The constructor `pi` of type `Term` encodes the abstract syntax tree (AST) representation of $g$ (fig. 1). We can retrieve and traverse the AST of $g$, and add new type information into it to build a new type representing $d$.

During the traversal of the abstract syntax tree of $g$, when we identify a function `f1` with codomain $W$, we add a new function `f2`, with the same arguments as `f1` and codomain $P$, to the tree. For example, in figure 1, a new function is built from the first argument $A \to W$ by modifying the codomain $W$ to $P$. The new function of type $A \to P$ is added after the function $A \to W$ as in figure 2. Constant types require no modifications. Therefore, we copy $B$ into the new type without any changes. When the type $W$ occurs directly in a non-codomain position, we add the type $P$ next to it. Finally, we change the codomain $W$ of $g$ to $P$ resulting in an abstract syntax tree representation of $d$ (fig. 2). We repeat this process for all the constructors of a given type. In the case of `Nat`, we construct the output types for `zero` and `(suc x)` for `x : Nat` using the above method.

The parameter and the index of $W$, if present, should be encoded as part of the type of function $f$. We can retrieve information about the parameter and the index of $W$ from its type. The constructors refer to the parameter and the index using de-bruijn indices. During the construction of the output type $d$, we should update the de-bruijn indices

```
(pi (vArg
   (pi (vArg `A) (abs "_" `W)))      -- f1 : A → W
   (abs "_" (pi (vArg
   (pi (vArg `A) (abs "_" `P)))      -- f2 : A → P
   (abs "_" (pi (vArg `B)            -- B
   (abs "_" (pi (vArg `W)            -- W
   (abs "_" (pi (vArg `P)            -- P
   (abs "_" `P))))))))))            -- P
```

**Figure 2.** Abstract syntax tree for constructor $d$

accordingly. The constructor `data-type` contains the count $cp$ of parameters occurring in a defined type. It also encodes the constructors of the type as a list of `Name`. We can retrieve the index count by finding the difference between $cp$ and length of the constructor list. The parameters are common to all the constructors of a type. But the index values are different for each constructor. So we have to encode unique indices for each constructor. Also, some constructors might not take the same number of indices as the parent type. For example, in the case of `Vec`, the constructor `[]` excludes the index `Nat` from its type. We do not have any reflection primitive to retrieve the index count from a constructor name. A workaround is to pass the index count of each constructor explicitly to the automation tool.

Once we have the type of $d$, we can build the type of the recursion rule for $W$. To encode the mapping $W \to P$ in the recursion type we need to declare $P$. We can use the constructor `agda-sort` to introduce the type *(P : Set)*. The type of the recursion rule $f$ is given as follows.

$$\begin{aligned} f : (P : Set) \to \\ (d : (A \to W) \to (A \to P) \to B \to W \to P \to P) \to \\ W \to P \end{aligned} \tag{4}$$

The above type is declared using `declareDef`. We can build the computation rule representing the action of function $f$ on $d$ using `clause` (fig. 3). The first argument to `clause` encodes variables corresponding to the above type, and it also includes the abstract representation of $g$ on which the pattern matching should occur. The second argument to clause, which is of type `Term`, refers to the variables in the first argument using de-bruijn indices, and it encodes the output of the action of function $f$ on $d$. The constructor `var` in `Pattern` is used to introduce new variables in the `clause` definition. The type `Pattern` also has another constructor `con` used to represent the pattern matching term $g$. The type `Term` has similar constructors `var` and `con`, but with different types, used to encode the output of the recursion rule. Given $\alpha : A \to W$, $\beta : B$ and $\omega : W$, the computation rule corresponding to the above type is given as follows.

$$f(P, d, g(\alpha, \beta, \omega)) \equiv d(\alpha, f \circ \alpha, \beta, \omega, f(\omega)) \tag{5}$$

```
(clause
  (vArg (var "P") ::                          -- P
   vArg (var "d") ::                          -- d
   vArg (con (quote g)                        -- g
         (vArg (var "alpha") ::               -- α
          vArg (var "beta") ::                -- β
          vArg (var "omega") :: [])) :: [])   -- ω
  (var 3                                      -- d
    (vArg (var 2 []) ::                       -- α
     vArg (lam visible (abs "a"               -- a
       (def f                                 -- f ∘ α
         (vArg (var 3
         (vArg (var 0 []) :: [])) :: []))) ::
     vArg (var 1 []) ::                       -- β
     vArg (var 0 []) ::                       -- ω
     vArg (def f (vArg (var 0 []) :: [])))))  -- f(ω)
```

**Figure 3.** Clause definition for the computation rule of $W$

The second argument $(f \circ \alpha)$ to $d$ is the composite of function $f$ and $\alpha$. We can build a composite function inside a lambda using the constructor `lam`. The arguments to `lam` are referenced using de-bruijn indices inside the lambda body. So, the de-bruijn indices for referring variables outside the lambda body are updated accordingly. The clause definition, which evaluates to the above computation rule of $W$, is given in figure 3. The de-bruijn index reference increments right to left starting from the last argument $\omega$. Inside the lambda body, the reference `0` refers to the lambda argument `a`, and the index references to the variables outside the lambda body starts from 1 and increments towards the left. The above clause definition is defined using `defineFun` primitive, and the function $f$ is brought into scope by `unquotedecl`.

In the automation tool, `generateRec` interface is used to generate the resursion rule $f$ for the type $W$. The implementation of `generateRec` is given as follows.

```
1
2  generateRec : Arg Name → Name → List Nat → TC ⊤
3  generateRec (arg i f) t indLs =
4    bindTC (getConstructors t) λ cns →
5    bindTC (getLength cns) λ lcons →
6    bindTC (getClause lcons zero t f indLs cns) λ cls →
7    bindTC (getType t) λ RTy →
8    bindTC (getRtype t indLs zero RTy) λ funType →
9    bindTC (declareDef (arg i f) funType) λ _ →
10   (defineFun f cls)
11
```

`generateRec` uses `getClause` and `getRtype` to build the computation and elimination rules respectively. It takes three arguments which consists of the function name to be defined as an element of type `Arg Name`, the quoted `Name` of the type $W$ and a list containing the index count of the individual constructors. `generateRec` can be used to automate the generation of recursion rules for the inductive types with point constructors. It can be used with the constructors of inductive types that takes no arguments (eg. `Bool`), one or multiple arguments (eg. coproduct, cartesian product) and the constructors that takes argument from the inductive type being defined (eg. `Nat`, `List`, `Vec`). The recursion rule generated by `generateRec` is brought into scope using `unquoteDecl` as follows.

```
unquoteDecl f = generateRec (vArg f) (quote W) (0 :: [])
```

The third argument to `generateRec` is a list consisting of the index count for the constructors. Since the constructor $g$ does not have any index, the count is given as 0. It is required to pass the index count for each constructor explicitly as the Agda reflection library does not have built-in primitives to retrieve the index value.

### 3.2 Dependent Eliminator

The dependent eliminator or the induction principle is used to define the behavior of a mapping $f'$ when the output type is dependent on the input element to $f'$. For the inductive type $W$, the induction principle says that to define a mapping $f' : (w : W) \to P(w)$, it suffices to define the action of $f'$ on $g$. To define the action of $f'$ on $g$, we need an element of the following type.

$$d' : \prod_{\alpha:A\to W} (\prod_{a:A} P(\alpha(a))) \to \prod_{\beta:B} \prod_{\omega:W} P(\omega) \to P(g(\alpha,\beta,\omega)) \tag{6}$$

In the first argument of $d'$, the type $P$ depends on the action of the function $\alpha$ on input $a : A$. In the second argument, it depends on the input element $\omega : W$. The output of function $d'$ depends on the action of constructor $g$ on inputs $\alpha$, $\beta$, and $\omega$.

To build the dependent eliminator, we need the type of the function $d'$. We can construct the abstract syntax tree of $d'$ using the static type information obtained from $g$. To construct $d'$, during the traversal of the abstract syntax tree of $g$, when we identify a function $\alpha$ with codomain $W$, we add a new function with the same argument `A` as in $\alpha$ and codomain $P$, which depends on the action of $\alpha$ on `(a : A)`. We copy the constant $B$ directly without any changes as in the case of the non-dependant eliminator. When we see $(\omega : W)$ in a non-codomain position, we add the type $P$, which depends on the element $\omega$, next to it. Finally, the output type $W$ of the constructor $g$ is changed to $P$, which depends on the action of the constructor $g$ on inputs $\alpha$, $\beta$ and $\omega$ (fig. 4).

We can construct the type of the induction principle $f'$ using $d'$ . The type $P$ in the mapping $f'$ depends on the element of the input type $W$. The following equation gives the type of the induction principle $f'$.

$$\begin{aligned} f' : (P : W \to Set) \to \\ (d' : \prod_{\alpha:A\to W} (\prod_{a:A} P(\alpha(a))) \to \prod_{\beta:B} \prod_{\omega:W} P(\omega) \to \\ P(g(\alpha,\beta,\omega))) \to \\ (w : W) \to P(w) \end{aligned} \tag{7}$$

```
(pi (vArg
   (pi (vArg `A) (abs "_" `W)))                -- α : A → W
   (abs "_" (pi (vArg                          -- A → P(α(a))
   (pi (vArg `A)                               -- a : A
   (abs "_" (var 2                             -- P(α(a))
   (vArg (var 1                                -- α ref
   (vArg (var 0 []) :: [])) :: [])))))         -- a ref
   (abs "_" (pi (vArg `B)                      -- β : B
   (abs "_" (pi (vArg `W)                      -- ω : W
   (abs "_" (pi (vArg (var 4                   -- P(ω)
   (vArg (var 0 []) :: [])))                   -- ω ref
   (abs "_" (var 5                             -- P(g i β ω)
   (vArg (con (quote g)                        -- g ref
   (vArg (var 4 []) ::                         -- α ref
    vArg (var 2 []) ::                         -- β ref
    vArg (var 1 []) :: []) :: []))))))))))))  -- ω ref
```

**Figure 4.** Abstract syntax tree for constructor $d'$

The computation rule for the above type is the same as the computation rule of the recursion principle of $W$ except that the function $f$ and $d$ are changed to $f$' and $d'$ respectively. It is constructed using `clause` definitions using the same approach as the recursion principle. The computation rule corresponding to $f$' is given as follows.

$$f'(P, d', g(\alpha, \beta, \omega)) \equiv d'(\alpha, f' \circ \alpha, \beta, \omega, f'(\omega)) \qquad (8)$$

We can automate the generation of the induction rule $f'$ for the type $W$ using `generateInd` interface. The implementation of `generateInd` is given as follows:

```
1
2  generateInd : Arg Name → Name → List Nat → TC ⊤
3  generateInd (arg i f) t indLs =
4    bindTC (getIndex' t indLs) λ indLs' →
5    bindTC (getConstructors t) λ cns →
6    bindTC (getLength cns) λ lcons →
7    bindTC (getClauseDep lcons zero t f indLs' cns) λ cls →
8    bindTC (getType t) λ RTy →
9    bindTC (getRtypeInd t zero indLs' RTy) λ funType →
10   bindTC (declareDef (arg i f) funType) λ _ →
11   (defineFun f cls)
12
```

`generateInd` uses `getClauseDep` to generate the `clause` definitions representing the computation rule of the type $W$. The abstract representation of the type of $W$ is provided by `getRtypeInd`. $f'$ generated by `generateInd` is brought into scope by `unquoteDecl` as follows.

```
unquoteDecl f' = generateInd (vArg f') (quote W) []
```

We pass an empty list to `generateInd` as the type $W$ has no index. We can also pass an empty list if all the constructors of a type has the same number of index as the parent type. But if any one constructor has an index count different from the index count of the parent type, then we have to explicitly pass the index count of all the constructors.