Encoding Higher Inductive Types Without Boilerplate

A Study in Agda Metaprogramming

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

David Thrane Christiansen
Galois, Inc.
Portland, Oregon, USA
dtc@galois.com

Abstract

Higher inductive types are inductive types that include nontrivial higher-dimensional structure, represented as identifications at the type that are not reflexivity. While work proceeds on type theories with a computational interpretation of univalence and higher inductive types, it is convenient to encode these structures in more traditional type theories. However, these encodings involve a great deal of error-prone additional syntax. We present a library that uses Agda's metaprogramming facilities to automate this process, allowing higher inductive types to be specified with minimal additional syntax.

Keywords Higher inductive type, Elaboration, Elimination rules, Computation rules

1 Introduction

A metaprogram is a program used to create other programs. In dependently-typed programming languages like Agda and Coq, a reflection library provides metaprogramming interfaces to support code generation for proofs and programs. Elaborator reflection [8], a new paradigm for metaprogramming, provides the language users with a powerful set of interfaces for efficient code generation. An *elaborator* is a metaprogram that converts a high-level language syntax in its abstract representation to a core language which is then type checked independently. By exposing a primitive monad to elaboration framework in Idris, Christiansen [8] showed how to write termination proofs for general recursive functions by automating Bove-Capretta [5] transformation, and how to generate data types matching schemas of external data sources based on Idris's type providers [6].

Agda is a proof-assistant based on Martin-Löf's intensional type theory. Unlike Idris, which has a dedicated elaborator to convert the high-level syntax to its core language, Agda's elaboration mechanism is a part of its type checker. Inspired by Idris's elaborator reflection, Agda development team extended the reflection library of Agda by exposing its elaboration monad to its high-level metaprogramming instructions. The elaboration monad provides an 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 an abstract syntax tree, and to convert an abstract syntax tree to its concrete syntax.

In this paper, we discuss metaprogramming using elaborator reflection in Agda. By using the elaboration monad, we performed code generation for the elimination and computation rules of inductive types and higher inductive types. We also automated the code generation for the boiler-plate code segment used to implement the higher inductive type defined using Dan Licata's method [11]. We discuss the construction process in detail and introduce new metaprogramming interfaces which build the code segments using elaborator reflection. The interfaces extensively use Agda's reflection primitives and build the abstract syntax tree of the code segments with static type information obtained using the reflection primitives. The generated code is then brought into scope by another top-level reflection primitive.

Elaborator reflection is a powerful mechanism which allows us to create more interesting metaprograms. By performing the above automation, we abstracted the difficulties involved in implementing and using inductive types and higher inductive types by automating the construction of their corresponding elimination and computation rules. We also achieved an extensive reduction in code size, and we demonstrated how to extend a language without modifying the compiler or depending on the language implementers. More specifically, we discuss the following contributions in this paper.

Rewrite c
tions list
on actual
things - n
ple have a
eliminato
tion befor

- We demonstrate using the Agda's new metaprogramming library, extended with support for elaborator reflection, and show how to automate the construction of the recursion and the induction principles for inductive types using the library.
- 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.

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

- 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 [4] implementation enriched with patches of encryption. We also examine the usage of the automation tool in a cryptography application [18].

The automation tool abstracts the implementation difficulties of a higher inductive type and its recursion and induction principles. We automated the code generation of approximately 1500 lines of code of the patch theory implementation with just 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 Agda Reflection

Agda's reflection library enables compile-time metaprogramming. This reflection library directly exposes parts of the implementation of Agda's type checker and elaborator for use by metaprograms, in a manner that is similar to Idris's elaborator reflection [7, 8] and Lean's tactic metaprogramming [10]. The type checker's implementation is exposed as effects in a monad called `TC`.

Agda exposes a representation of its syntax to metaprograms, including datatypes for expressions (called `Term`) and definitions (called `Definition`). The primitives exposed in `TC` include declaring new metavariables, unifying two `Term`s, declaring new definitions, adding new postulates, computing the normal form or weak head normal form of a `Term`, inspecting the current context, and constructing fresh names. This section describes the primitives that are used in our code generation library; more information on the reflection library can be found in the Agda documentation [1].

`TC` computations can be invoked in three ways: by macros, which work in expression positions, using the `unquoteDecl` operator in a declaration position, which can bring new names into scope, and using the `unquoteDef` operator in a declaration position, which can automate constructions using names that are already in scope. This preserves the principle in Agda's design that the system never invents a name.

An Agda *macro* is a function of type $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow$ `Term → TC ⊤` that is defined inside a `macro` block. Macros are special: their last argument is automatically supplied by the type checker, and consists of a `Term` that represents the metavariable to be solved by the macro. If the remaining arguments are quoted names or `Term`s, then the type checker will automatically quote the arguments at the macro's use site. At some point, the macro is expected to unify the provided metavariable with some other term, thus solving it.

```
macro
  mc1 : Term → Term → TC ⊤
  mc1 exp hole =
    do exp' ← quoteTC exp
       unify hole exp'

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

Figure 1. A macro that quotes its argument

```
macro
  mc2 : Term → Term → TC ⊤
  mc2 exp hole =
    do exp' ← unquoteTC exp
       unify hole exp'

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

Figure 2. A macro that unquotes its argument

Figure 1 demonstrates a macro that quotes its argument. The first step is to quote the quoted expression argument again, using `quoteTC`, yielding a quotation of a quotation. The result of this double-quotation is passed, using Agda's new support for Haskell-style do-notation, into a function that unifies it with the hole. Because unification removes one layer of quotation, `unify` inserts the original quoted term into the hole. The value of `sampleTerm` is

```
lam visible (abs "n" (var 0 []))
```

The constructor `lam` represents a lambda, and its body is formed by the abstraction constructor `abs` that represents a scope in which a new name `"n"` is bound. The body of the abstraction is a reference back to the abstracted name using de Bruijn index `0`.

The `unquoteTC` primitive removes one level of quotation. Figure 2 demonstrates the use of `unquoteTC`. The macro `mc2` expects a quotation of a quotation, and substitutes its unquotation for the current metavariable.

The `unquoteDecl` and `unquoteDef` primitives, which run `TC` computations in a declaration context, will typically introduce new declarations by side effect. A function of a given type is declared using `declareDef`, and it can be given a definition using `defineFun`. Similarly, a postulate of a given type is defined using `declarePostulate`. Figure 3 shows an Agda implementation of addition on natural numbers, while figure 4 demonstrates an equivalent metaprogram that adds the same definition to the context.

In Figure 4, `declareDef` declares the type of `plus`. The constructor `pi` represents dependent function types, but a pattern synonym is used to make it shorter. Similarly, `def` constructs references to defined names, and the pattern

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

Figure 3. Addition on natural numbers

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

unquoteDecl plus =
  do declareDef (vArg plus) (`Nat `⇒ `Nat `⇒ `Nat)
     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 []) :: [])) ::
             [])) ::
        [])
```

Figure 4. Addition, defined by metaprogramming

synonym `Nat abbreviates references to the defined name Nat, and vArg represents the desired visibility and relevance settings of the arguments. Once declared, plus is defined using defineFun, which takes a name and a list of clauses, defining the function by side effect. Each clause consists of a pattern and a right-hand side. Patterns have their own datatype, while right-hand sides are Terms. The name con is overloaded: in patterns, it denotes a pattern that match a particular constructor, while in Terms, it denotes a reference to a constructor.

2.2 Higher Inductive Types

Homotopy type theory [17] is a research program that aims to develop univalent, higher-dimensional type theories. A type theory is *univalent* when equivalences between types are considered equivalent to equalities between types; it is *higher-dimensional* when we allow non-trivial identifications that every structure in the theory must nevertheless respect. Identifications between elements of a type are considered to be at the lowest dimension, while identifications between identifications at dimension n are at dimension $n+1$. Voevodsky added univalence to type theories as an axiom, asserting new identifications without providing a means to compute with them. While more recent work arranges the computational mechanisms of the type theory such that univalence can be derived, as is done in cubical type theories,we are concerned with modeling concepts from homotopy type theory in existing, mature implementations of type theory, so we follow Univalent Foundations Program [17] in modeling paths using Martin-Löf's identity type. Higher-dimensional structure can arise from univalence, but it can also be introduced by defining new type formers that introduce not only introduction and elimination principles, but also new non-trivial identifications.

and's
rtberg,
r's group

In homotopy type theories, one tends to think of types not as collections of distinct elements, but rather through the metaphor of topological spaces. The individual elements of the type correspond with points in the topological space, and identifications correspond to paths in this space.

While there is not yet a general schematic characterization of a broad class of higher inductive types along the lines of Dybjer's inductive families,it is convenient to syntactically represent the higher inductive types that we know are acceptable as if we had such a syntax. Thus, we sometimes specify a higher inductive type similarly to a traditional inductive type by providing its constructors (*i.e.* its points); we additionally specify the higher-dimensional structure by providing additional constructors for paths. For example, figure 5 describes Circle, which is a higher inductive type with one point constructor base and one non-trivial path constructor loop.

cite 1994

```
data Circle : Set where
  base : Circle
  loop : base ≡ base
```

Figure 5. A specification of a higher inductive type

Agda is a programming language that was originally an implementation of Luo's UTT extended with primitive dependent pattern matching, itself a derivative of the Calculus of Constructions and Martin-Löf's intensional type theory. Agda's type theory has since gained a number of new features, among them the ability to restrict pattern matching to that subset that does not imply Streicher's Axiom K , which is inconsistent with univalence. The convenience of programming in Agda, combined with the ability to avoid axiom K, makes it a good laboratory for experimenting with the idioms and techniques of univalent programming while more practical implementations of univalent type theories are under development.

Cite Ulf's
official wi

cite Luo's

Cite Cock

In Agda, we don't have built-in primitives to support the definition of higher inductive type such as Circle. One approach is to use Agda's rewrite rules mechanism to define higher inductive types. In this approach, we define the dependent and non-dependent eliminators of a higher inductive type as paramterized modules inside which we declare the computation rules for points as rewrite rules using {-# REWRITE , ...#-} pragma. However, Agda's reflection library do not have interfaces to support introducing new pragmas and defining new modules. Another approach to define higher inductive types is to use Dan Licata's method [11]. According to this method, a higher inductive type is defined using type abstraction inside a module. The module consists of a boiler-plate code segment which defines

cite Cock
Abel, exte
abstract a
2016

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

  S : Set
  S = S*

  base : S
  base = base*

  postulate
    loop : base ≡ base

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

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

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

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

Figure 6. An example of a higher inductive type using Licata's encoding

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. For example, Circle is defined using Dan Licata's method as follows.

Inside the module Circle, the type S is defined using a private datatype S*. The constructor base is defined using base* and the path loop is given as a postulated propositional equality. The recursion and induction principles are defined by pattern matching on the constructor base* of the type S*, and thus compute as expected. 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, which prevents them from writing functions that distinguish between multiple constructors of a higher inductive type that may be identified by additional path constructors. The client's *only* access to the constructor is through the provided elimination rules. The following code gives the non-dependent eliminator (sometimes called the *recursion rule*) recS.

recS 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 rule βrecS. The operator apPerhaps we should move the discussion of ap earlier, so that we don't need the digression is frequently referred to as cong, because it expresses that propositional equality is a congruence. However, when viewed through a homotopy type theory lens, it is often called ap, as it describes the action of a function on paths. In a higher inductive type, ap should compute new paths from old ones.

```
ap : {A B : Set} {x y : A}
     (f : A → B) (p : x ≡ y) → f x ≡ f y
```

The following code gives the dependent eliminator or the induction rule indS and its computational rules. The dependent eliminator relies on another operation on identifications, called transport, that coerces an inhabitant of a family of types at a particular index into an inhabitant at another index. Outside of homotopy type theory, transport is typically called subst or replace, because it also expresses that substituting equal elements for equal elements is acceptable.

```
transport : {A : Set} {x y : A} →
  (P : A → Set) → (p : x ≡ y) → P x → P y
```

In the postulated computation rule for indS, the function apd is the dependent version of ap: it expresses the action of dependent functions on paths.

```
apd : {A : Set} {B : A → Set} {x y : A} →
  (f : (a : A) → B a) →
  (p : x ≡ y) → (transport B p (f x) ≡ f y)
```

The next section introduces the necessary automation features by describing the automatic generation of eliminators for a variant on Dybjer's inductive families.Section 4 then generalizes this feature to automate the production of eliminators for higher inductive types using Licata's technique. Section 5 revisits Angiuli et al.'s encoding of Darcs's patch theory [4] and demonstrates that the higher inductive types employed in that paper can be generated succinctly using our library.

Check wh
is right -
be a varia
Luo's ind
types inst

3 Code Generation for Inductive Types

An inductive type X is a type that is freely generated by a finite collection of constructors. The constructors of X accept zero or more arguments, and result in an X. The constructors can also take an element of type X itself as an argument, but only *strictly positively*: any occurrences of the type constructor X in the type of an argument to a constructor of X must not be to the left of any arrows.

Type constructors can have a number of *parameters*, which may not vary between the constructors, as well as *indices*, which may.

In Agda, constructors are given a function type. In Agda's reflection library, the constructor `data-type` of the datatype `Definition` stores the constructors of an inductive type as a list of `Name`s. 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 list of constructors and their types to generate code for the elimination rules of an inductive type.

3.1 Non-dependent Eliminators

In Agda, we define an inductive type using `data` keyword. A definition of an inductive datatype declares its type and specifies its constructors. While Agda supports a variety of ways to define new datatypes, we will restrict our attention to the subset that correspond closely to Dyber's inductive families. In general, the definition of an inductive datatype D with constructors $c_1 \ldots c_n$ has the following form:

$$\begin{array}{l}\textbf{data}\ D\ (a_1 : A_1) \ldots (a_n : A_n) : (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \mathsf{Set}\ \textbf{where}\\ \quad c_1 : \Delta_1 \to D\ a_1 \ldots a_n\ e_{11} \ldots e_{1m}\\ \qquad\qquad\qquad\qquad\vdots\\ \quad c_r : \Delta_n \to D\ a_1 \ldots a_n\ e_{r1} \ldots e_{rm}\end{array}$$

where the index instantiations $e_{k1} \ldots e_{km}$ are expressions in the scope induced by the telescope Δ_k. Every expression in the definition must also be well-typed according to the provided declarations.

While inductive datatypes are defined by their constructors, it must also be possible to *eliminate* them. This section describes how to generate a non-dependent recursion principle for an inductive type; section 3.2 generalizes this technique to fully-dependent induction principles.

Based on the generic form of the inductive type given above, we can define the following schematic representation for the non-dependent eliminator.

$$\begin{array}{l}D_{rec} : (a_1 : A_1) \to \ldots \to (a_n : A_n) \to\\ \qquad (i_1 : I_1) \to \ldots \to (i_m : I_m) \to\\ \qquad (tgt : D\ a_1 \ldots a_n\ i_1\ \ldots\ i_n) \to\\ \qquad (C : \mathsf{Set}) \to\\ \qquad (f_1 : \Delta'_1 \to C) \to \ldots \to (f_r : \Delta'_r \to C) \to\\ \qquad C\end{array}$$

The type of f_i, which is the method for fulfilling the desired type C when eliminating the constructor c_i in D_{rec}, is determined by the type of c_i. The telescope Δ'_i is the same as Δ_i for non-recursive constructor arguments. However, Δ'_i binds additional variables when there are recursive occurrences of D in the arguments. If Δ_i has an argument $(y : B)$, where B is not an application of D or a function returning such an application, Δ'_i binds $(y : B)$ directly. If B is an application of D, then an additional binding $(y' : C)$ is inserted following y. Finally, if B is a function type $\Psi \to D$, the additional binding is $(y' : \Psi \to C)$.

```
data Vec (A : Set) : Nat → Set where
  []   : Vec A zero
  _::_ : {n : Nat} →
         (x : A) → (xs : Vec A n) →
         Vec A (suc n)
```

Figure 7. Length-indexed lists

When automating the production of D_{rec}, all the information that is needed to produce the type signature is available in the `TC` monad by looking up D's definition. 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`s. Metaprograms can retrieve the index count by finding the difference between cp and the length of the constructor list. The constructors of D refer to the parameter and the index using de Bruijn indices.

The general schema for the computation rules corresponding to D_{rec} and constructors $c_1, \ldots, c_n$ is given as follows.

$$\begin{array}{l}D_{rec}\ a_1\ \ldots\ a_n\ i_1\ \ldots\ i_m\ (c_1\ \overline{\Delta_1})\ C\ f_1 \ldots f_r\ =\\ \quad \mathsf{RHS}\ (f_1, \Delta'_1)\\ \vdots\\ D_{rec}\ a_1\ \ldots\ a_n\ i_1\ \ldots\ i_m\ (c_r\ \overline{\Delta_r})\ C\ f_1 \ldots f_r\ =\\ \quad \mathsf{RHS}\ (f_r, \Delta'_r)\end{array}$$

Here, $\overline{\Delta_j}$ is the sequence of variables bound in Δ_j. RHS constructs the application of the method f_j to the arguments of c_j, such that C is satisfied. It is defined by recursion on Δ_j. $\mathsf{RHS}\ (f_j, \cdot)$ is f_j, because all arguments have been accounted for. $\mathsf{RHS}\ (f_j, (y : B)\Delta_k)$ is $\mathsf{RHS}\ (f_j\ y, \Delta_k)$ when B does not mention D. $\mathsf{RHS}\ (f_j, (y : D)(y' : C)\Delta_k)$ is $\mathsf{RHS}\ (f_j\ y\ \ (D_{rec}\ \ldots y \ldots), \Delta_k)$, where the recursive use of D_{rec} is applied to the recursive constructor argument as well as the appropriate indices, and the parameters, result type, and methods remain constant. Higher-order recursive arguments are a generalization of first-order arguments. Finally,

$$\mathsf{RHS}\ (f_j, (y : \Psi \to D)(y' : \Psi \to C)\Delta_k)$$

is

$$\mathsf{RHS}\ \left(f_j\ y\ \ \left(\lambda\overline{\Psi}.D_{rec}\ \ldots \left(y\ \overline{\Psi}\right) \ldots\right), \Delta_k\right)$$

where the recursive use of D_{rec} is as before.

The Agda datatype `Vec` represents lists of a known length. It is defined in figure 7. For `Vec`, the recursion principle says that to define a mapping `f : Vec A n → C`, it suffices to define the action of `f` on inputs `[]` and `_::_`.

```
pattern _[_v]⇒_ a s b = pi (vArg a) (abs s b)
pattern _[_h]⇒_ a s b = pi (hArg a) (abs s b)

(agda-sort (lit 0) [ "A" h]⇒                -- A
 (def (quote Nat) [] [ "n" h]⇒              -- n
  (var 1 [] [ "x" v]⇒                       -- x
   (def (quote Vec)                         -- xs : Vec A n
        (vArg (var 2 []) ::
         vArg (var 1 []) :: [])
         [ "xs" v]⇒
    def (quote Vec)                         -- Vec A (suc n)
        (vArg (var 3 []) ::
         vArg (con (quote suc)
                (vArg (var 2 []) :: []))
         :: [])))))
```

Figure 8. Abstract syntax tree for constructor `_::_`

```
(agda-sort (lit 0) [ "A" h]⇒                      -- A
 (def (quote Nat) [] [ "n" h]⇒                    -- n
  (def (quote Vec) (vArg (var 1 []) ::            -- Vec A n
     vArg (var 0 []) :: []) [ "_" v]⇒
   (agda-sort (lit 0) [ "C" v]⇒                   -- C
    (var 0 [] [ "_" v]⇒                           -- c1
     ((def (quote Nat) [] [ "n" h]⇒               -- c2
       (var 5 [] [ "x" v]⇒                        -- x
        (def (quote Vec) (vArg (var 6 []) ::      -- Vec A n → C
           vArg (var 1 []) :: []) [ "xs" v]⇒
             (var 4 [] [ "_" v]⇒
              var 5 []))))                        -- C
    [ "_" v]⇒ var 2 []))))))                      -- C
```

Figure 9. Abstract syntax tree for recursor `f` on Vec

To define the action of `f` on inputs `[]` and `_::_`, we need elements `c1` and `c2` of the following type.

```
c1 : C
c2 : {m : Nat} → (x : A) → (xs : Vec A m) →
     C → C
```

The recursor `f` maps the constructor `[]`, which takes zero arguments, to `c1`. The constructor `_::_` takes a constant argument `x` and an argument `xs` of the inductive type `Vec`. `f` maps `(x :: xs)` to `(c2 x xs (f xs C c1 c2))`. To construct the type of the recursion rule for `Vec`, we need to build the type of `c1` and `c2`. Since `[]` is not a function type, we can map it directly to `c1 : C`. We can retrieve the static type information of `_::_` using reflection primitives, and use that to construct the type of `c2`. The constructor `pi` of type `Term` encodes the abstract syntax tree (AST) representation of `_::_` (fig. 8). We can retrieve and traverse the AST of `_::_`, and add new type information into it to build a new type representing `c2`.

During the traversal of abstract syntax tree of the type of `_::_`, when the type `Vec` occurs directly in a non-codomain position, we add the type `C` next to it. For example, in figure 8, a new function is built from the argument `(xs : Vec A n)` by modifying it to `(Vec A n) → C` (fig. 9). Constant types require no modifications. Therefore, we copy `(x : A)` into the new type without any changes. Finally, we change the codomain Vec A (suc n) of `_::_` to *C* resulting in an abstract syntax tree representation of the type `c2`. We repeat this process for each of the constructors.

The element `c2`, which represents the method for the constructor `_::_` in `Vec`, refers to the parameter and the index using de Bruijn indices. During the construction of the type of `c2`, the automation tool updates the de Bruijn indices accordingly. 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 AST of `c2`, we can build the type of the recursion rule `f` for `Vec` (fig.9). To encode the mapping `Vec A n → C` in the recursion type, we need to declare `C`. We can use the constructor `agda-sort` to introduce the type `(C : Set)`. The type of the recursion rule *f* is given as follows.

```
f : {A : Set} → {n : Nat} → Vec A n →
  (C : Set) →
  (c1 : C) →
  (c2 : {n : Nat} → (x : A) →
        (xs : Vec A n) → C → C) →
  C
```

The above type is declared using `declareDef`. We can build the computation rule representing the action of function *f* on `[]` and `_::_` using `clause` (fig. 10). The first argument to `clause` encodes variables corresponding to the above type, and it also includes the abstract representation of `[]`,`_::_` 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 `[]`,`_::_`. 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. The type `Term` has similar constructors `var` and `con`, but with different types, used to encode the output of the recursion rule. The computation rules corresponding to the above type is given as follows.

```
f [] C c1 c2 = c1
f (x :: xs) C c1 c2 = c2 x xs (f xs C c1 c2)
```

A clause definition, which evaluates to the above computation rule of `Vec` pattern matching on `_::_`, is given in figure 10. The de Bruijn index reference increments right to left starting from the last argument. The above clause definition is defined using `defineFun` primitive, and the function *f* is brought into scope by `unquotedecl`.

Lets consider another example, the `W`-type, to review the automation process for the recursion principle. `W`-type has a constructor `sup` given by the following type.

```
sup : (a : A) → (B a → W A B) → W A B
```

To define the action of a function `f : W A B → C` on input `sup`, we need a function `d` of the following type.

```
d : (a : A) → (B a → W A B) → (B a → C) → C
```

```
(clause
  (vArg (con (quote _::_)                              -- _::_
         (vArg (var "x") ::                            -- x
          vArg (var "xs") :: []))                      -- xs
   vArg (var "C") ::                                   -- C
   vArg (var "c1") ::                                  -- c1
   vArg (var "c2") :: [])                              -- c2
  (var 0
    (vArg (var 4 []) ::                                -- x
     vArg (var 3 []) ::                                -- xs
     vArg (def f                                       -- (f xs C c1 c2)
       (vArg (var 3 [])                                -- xs
        vArg (var 2 []) ::                             -- C
        vArg (var 1 []) ::                             -- c1
        vArg (var 0 []) :: [])))))                     -- c2
```

Figure 10. Clause definition for the computation rule of _::_

```
data W (A : Set) (B : A → Set) : Set where
  sup : (a : A) → (B a → W A B) → W A B
```

Figure 11. W-Type

The type of d is built by traversing the AST of sup. During the traversal of the AST of sup, the first argument to sup, which is a constant type A, is copied directly into the AST of d. The second argument (B a → W A B), which is a function with co-domain W A B, is modified to (B a → W A B) → (B a → C). Finally, the co-domain W A B of sup is replaced by C. The computation rule corresponding to sup is given as follows.

```
f (sup a b) C d = d a b (λ v → f (b v) C d)
```

In the above computation rule, the third argument to d is a composition of functions f and b. The automation tool composes functions inside a lambda directly, using the Term 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. For example, inside the lambda body, the reference 0 refers to the lambda argument v, and the index references to the variables outside the lambda body start from 1 and increment towards the left.

In the automation tool, generateRec interface is used to generate the recursion rule f. The implementation of generateRec is given as follows.

```
generateRec : Arg Name → Name →
  (indexList : List Nat) → TC ⊤
generateRec (arg i f) t indLs =
  do indLs' ← getIndex t indLs
     cns ← getConstructors t
     lcons ← getLength cns
     cls ← getClause lcons zero t f indLs cns
     RTy ← getType t
     funType ← getRtype t indLs' zero RTy
     declareDef (arg i f) funType
     defineFun f cls
```

generateRec uses getClause and getRtype to build the computation and elimination rules respectively. It takes three arguments: the name of the function to be defined (represented by an element of type Arg Name), the quoted Name of the type and a list containing the index count of the individual constructors. generateRec can be used to automate the generation of recursion rules for inductive types having the general schema given at the beginning of this section. The recursion rule generated by generateRec is brought into scope using unquoteDecl as follows.

```
unquoteDecl f = generateRec (vArg f)
                  (quote Vec)
                  (0 :: 1 :: [])
```

The third argument to generateRec is a list consisting of the index count for the constructors. 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 Eliminators

The dependent eliminator for a datatype, also known as the *induction principle*, is used to eliminate elements of a datatype when the type resulting from the elimination mentions the very element being eliminated. We can define the general schema for the induction principle as follows.

$$\begin{aligned}
D_{ind} :& (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\
& (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& (tgt : D\ a_1 \ldots a_n\ i_1\ \ldots\ i_n) \to \\
& (C : (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& \quad D\ a_1 \ldots a_n\ i_1\ \ldots\ i_n \to \\
& \quad \mathrm{Set}) \to \\
& (f_1 : \Delta_1 \to \Delta'_1 \to C\ j_{11} \ldots j_{1p}\ (c_1\ \overline{\Delta_1})) \to \\
& (f_r : \Delta_r \to \Delta'_r \to C\ j_{r1} \ldots j_{rp}\ (c_r\ \overline{\Delta_r})) \to \\
& C\ i_1\ \ldots\ i_n\ tgt
\end{aligned}$$

Similar to the recursion principle, the type of f_i depends on the type of constructor c_i of D. If c_i has an argument $(y : B)$, the automation tool copies $(y : B)$ directly to f_i. For an argument $(y : D)$ with inductive type D in c_i, the automation tool adds $(y : D) \to C\,y$ to f_i. For function type with co-domain D such as $(y : B_1 \to \ldots \to B_n \to D)$, the automation tool adds $(y : B_1 \to \ldots \to B_n \to D) \to ((b_1 : B_1) \to \ldots \to (b_n : B_n) \to C(y\,b_1 \ldots b_n))$ to the type of f_i. The computation rules corresponding to the induction principle follows the same pattern as for the recursion rule.

For the inductive type W, the induction principle says that to define a mapping f : (w : W A B) → C w, it suffices to define the action of f on the constructor sup. To define the action of f on sup, we need an element d of the following type.

```
(agda-sort (lit 0) [ "A" h]⇒                   -- A
 ((var 0 [] [ "_" v]⇒                          -- B : A → Set
    agda-sort (lit 0)) [ "B" h]⇒
  (def (quote W) (vArg (var 1 []) ::          -- c : W A B
    vArg (var 0 []) :: []) [ "c" v]⇒
   ((def (quote W) (vArg (var 2 []) ::        -- C : W A B → Set
      vArg (var 1 []) :: []) [ "_" v]⇒
    agda-sort (lit 0)) [ "C" v]⇒
    ((var 3 [] [ "a" v]⇒                       -- a : A
     ((var 3 (vArg (var 0 []) :: [])          -- y : B a → W A B
       [ "_" v]⇒
       def (quote W) (vArg (var 5 []) ::
         vArg (var 4 []) :: [])) [ "y" v]⇒
      ((var 4 (vArg (var 1 []) :: [])         -- z : B a → C (y v)
        [ "v" v]⇒
        var 3 (vArg (var 1 (vArg (var 0 [])
        :: [])) :: []))
       [ "z" v]⇒
       var 3                                   -- C (sup a y)
       (vArg
        (con (quote sup)
         (vArg (var 2 []) ::
          vArg (var 1 []) :: []))
        :: []))))
    [ "_" v]⇒ var 1 (vArg (var 2 []) :: []))))))  -- C c
```

Figure 12. Abstract syntax tree for the dependent eliminator of W

```
d : (a : A) → (y : B a → W A B) →
    (z : (v : B a) → C (y v)) →
    C (sup a y)
```

To build the dependent eliminator, we need the type of the function d. We can construct the AST of d using the static type information obtained from sup. To construct d, during the traversal of the AST of sup, we copy the constant (a : A) directly without any changes as in the case of the non-dependant eliminator. When we identify a function y : B a → W A B with codomain W A B, we add a new function (z : (v : B a) → C (y v)) with the same arguments as in y and codomain C (y v), which depends on the action of y on v. Finally, the co-domain *W A B* of the constructor sup is changed to C (sup a y), which depends on the action of the constructor sup on inputs a and y (fig. 12).

We can construct the type of the induction principle f using d. The type C in the mapping f depends on the element of the input type W A B. The type of the induction principle f (fig. 12) is given as follows.

```
f : {A : Set} {B : A → Set} →
    (c : W A B) → (C : W A B → Set) →
    ((a : A) → (y : B a → W A B) →
     (z : (v : B a) → C (y v)) →
     C (sup x y)) → C c
```

The computation rule corresponding to the above type is the same as the computation rule of the recursion principle of W. It is constructed using clause definitions following the same approach as the recursion principle. We can automate the generation of the induction rule f and its corresponding computation rules using generateInd interface. The following code gives the implementation of generateInd.

```
generateInd : Arg Name → Name →
  (indexList : List Nat) → TC ⊤
generateInd (arg i f) t indLs =
  do id' ← getIndex t indLs
     cns ← getConstructors t
     lcns ← getLength cns
     cls ← getClauseDep lcns zero t f id' cns
     RTy ← getType t
     funType ← getRtypeInd t zero id' RTy
     declareDef (arg i f) funType
     defineFun f cls
```

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

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

An empty list is passed to generateInd as W has no index. An empty list can also be passed 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 the index count of all the constructors should be passed explicitly.

4 Code Generation for Higher Inductive Types

In Agda, there are no built-in primitives to support the definition of higher inductive types. However, we can still define a higher inductive type with a base type using Dan Licata's [11] method as discussed in section 2.1. In this section, we discuss the automation of code generation for the boilerplate code segments defining the higher inductive type. We also describe how to automate the code generation for the elimination and the computation rules of the higher inductive type using static type information obtained from the base type.

4.1 Higher Inductive Type Definition

We define a higher inductive type G as a top-level definition using a base type D similar to the module Circle in section 2.1. The reflection type Definition provides us the type and the constructors of the base type D. We copy the type of D to G and for the constructors $g_1 \ldots g_n$ of G, we traverse the AST of the type of $c_1 \ldots c_n$ respectively replacing the occurrence of D to G in every strictly positive position. Consider a constructor c_i that has the following type.

$$c_i : (A \to D) \to (B \to D) \to C \to D \to D$$

The automation tool built the type of g_i by traversing the AST of c_i and replacing the base type D with the higher inductive type G. The AST of c_i incorporates the type of the parameters and the indices if present. We have to retain the parameters and the indices explicitly during the construction of g_i. The following represents the type of the constructor g_i.

$$g_i : (A \to G) \to (B \to G) \to C \to G \to G$$

We explicitly pass the types of the path constructors to the automation tool. The higher inductive type definition of `Circle` in section 2.1 represents the path constructors as propositional equalities. The automation tool takes the path types as input and declares them as propositional equalities using the reflection primitive `declarePostulate`. We introduce a new data type `ArgPath` to input the path types to the automation tool.

```
data ArgPath {ℓ₁} : Set (lsuc ℓ₁) where
  argPath : Set ℓ₁ → ArgPath
```

The constructor `argPath` takes the type of a path constructor as input. We define the generic form of a higher inductive type as follows.

```
$data\text{-}hit\ (quote\ D)\ G$
  $Gpoints\ \ (g_1 :: \ldots :: g_n :: [])$
  $Gpaths\ \ (p_1 :: \ldots :: p_n :: [])$
  $(argPath$
    $(\{x_1 : P_1\} \to \ldots \to \{x_n : P_n\} \to$
     $\{i_1 : Q_1\} \to \ldots \to \{i_n : Q_n\} \to Type_1 \to$
     $(g_i\{x_1\} \ldots \{x_n\}\{i_1\} \ldots \{i_n\} \ldots) \equiv (g_j \ldots)) ::$
                       $\vdots$
   $argPath$
    $(\{x_1 : P_1\} \to \ldots \to \{x_n : P_n\} \to$
     $\{j_1 : Q_1\} \to \ldots \to \{j_n : Q_n\} \to Type_n \to$
     $(g_i\{x_1\} \ldots \{x_n\}\{j_1\} \ldots \{j_n\} \ldots) \equiv (g_j \ldots)) :: [])$
```

We define holders *Gpoints* for point constructors and *Gpaths* for path constructors as part of the higher inductive type definition of G. We cannot retrieve the constructors of the higher inductive type G using `Definition`. Therefore, *Gpoints* and *Gpaths* act as the only references for the constructors of G. The elements of the `argPath` list represent the type of the path constructors $p_1 \ldots p_n$ respectively. We explicitly include the parameter references $\{x_1 : P_1\} \ldots \{x_n : P_n\}$ and the index references $\{k_1 : Q_1\} \ldots \{k_n : Q_n\}$ in the type of the arguments to `argPath`. The points $g_1 \ldots g_n$ are not in scope when used in the identity type passed to `argPath`. The automation tool uses the base type constructors $c_1 \ldots c_n$ as dummy arguments in the place of $g_1 \ldots g_n$ respectively. The automation tool implements the interface `data-hit` as follows.

```
data-hit : ∀{ℓ₁}
  (baseType : Name) → (indType : Name) →
  (pointHolder : Name) → (lcons : List Name) →
  (pathHolder : Name) → (lpaths : List Name) →
  (lpathTypes : (List (ArgPath {ℓ₁}))) → TC ⊤
data-hit base ind h1 lcons h2 lpaths pTy =
  do defineHindType base ind
     cns ← getConstructors base
     defineHitCons base ind cns lcons
     pTy' ← getPathTypes base ind cns lcons pTy
     defineHitPathCons lpaths pTy'
     definePointHolder h1 lcons
     definePathHolder h2 lpaths
```

The higher inductive type G, the points $g_1 \ldots g_n$, the paths $p_1 \ldots p_n$, and the holders *Gpoints* and *Gpaths* are brought into scope by `unquoteDecl`. In the above implementation of `data-hit`, `defineHindType` defines the higher inductive type as a top-level definition using the base type. `defineHitCons` specifies the point constructors of the higher inductive type using the type information obtained from the constructors of the base type, and `defineHitPathCons` builds the paths constructors of the higher inductive type using the `argPath` list. The following code automates the generation of the higher inductive type definition for `Circle` given in section 2.1.

```
unquoteDecl S Spoints base Spaths loop =
  data-hit (quote S*) S
    Spoints (base :: []) -- point constructors
    Spaths (loop :: [])  -- path constructors
    (argPath (base* ≡ base*) :: [])
    -- base replaces base*
```

The identity type input `(base* ≡ base*)` to argPath represents the type of the path `loop`, and it uses the inductive type constructor `base*` as a dummy argument in the place of the higher inductive type constructor `base`. The constructor `base` comes into scope only during the execution of `unquoteDecl`, and so cannot be used in the identity type reference in `argPath`. We use the constructor `base*` of type `S*` as dummy argument because the type of `base*` is similar to `base`, and has the same references for the common arguments. The automation tool traverses the abstract syntax tree of `loop` and replaces the occurrences of `base*` with `base`.

4.2 Non-dependent Eliminator

Non-dependent eliminator or the recursion principle of a higher inductive type G maps the points and paths of G to an output type C. We extend the general schema of the recursion principle given in section (3.1) by adding methods for path constructors as follows.

$$\begin{aligned}G_{rec} :&(a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\ &(i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\ &(tgt : G\, a_1 \ldots a_n\, i_1\ \ldots\ i_n) \to \\ &(C : \mathrm{Set}) \to \\ &(f_1 : \Delta_1 \to \Delta'_1 \to C) \to \\ &(f_r : \Delta_r \to \Delta'_r \to C) \to \\ &(k_1 : \Delta_1 \to \Delta'_1 \to (f_i \ldots) \equiv (f_j \ldots)) \to \\ &(k_r : \Delta_r \to \Delta'_r \to (f_i \ldots) \equiv (f_j \ldots)) \to \\ &C\end{aligned}$$

In the schema definition above, we have given only one-dimensional paths. The automation tool currently supports one-dimensional paths, and we are planning to improve the tool to support higher-dimensional paths in the future.

The type of f_i, method for the point constructor g_i in G_{rec}, is built the same way as for the normal inductive type D. The automation tool builds the type of k_i, method for path constructor p_i in G_{rec}, by traversing the AST of p_i. During the traversal, the references to point constructors g_i of G in the co-domain of p_i are replaced by f_i. The arguments of k_i are handled the same way as for f_i. The schema for the computation rules corresponding to points g_i is similar to the computation rules corresponding to constructors c_i of the inductive type D except that it has additional variables to represent paths. The schema for the computation rules corresponding to paths p_i is given as follows.

$$\begin{aligned}\beta G_{rec} :&(a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\ &(C : \mathrm{Set}) \to \\ &(f_1 : \Delta_1 \to \Delta'_1 \to C) \to \\ &(f_r : \Delta_r \to \Delta'_r \to C) \to \\ &(k_1 : \Delta_1 \to \Delta'_1 \to (f_i \ldots) \equiv (f_j \ldots)) \to \\ &(k_r : \Delta_r \to \Delta'_r \to (f_i \ldots) \equiv (f_j \ldots)) \to \\ &ap\,(\lambda\, x \to G_{rec}\, x\, C\, f_1\, f_r\, k_1\, k_r)\,(p_i \ldots) \equiv (k_i \ldots)\end{aligned}$$

The computation rule βG_{rec} exists only as propositional equality. The automation tool builds the type of βG_{rec} using the same approach as for the recursion rule G_{rec}. The type of G_{rec} and βG_{rec} is similar except for the mapping $G \to C$ in G_{rec} which is replaced by the term representing the action of function G_{rec} on the path $(p_i\ \ldots)$. The function ap (sec. 2.2) applies G_{rec}, which is nested inside a lambda function, on the path $(p_i \ldots)$. The tool uses the constructor lam of Term to introduce a lambda function.

Lets consider the higher inductive type S (sec 2.1), which represents the Circle. To define a mapping recS : S → C, we need a point cbase : C and a path cloop : C ≡ C in the space C. To construct the recursion principle recS, we need to build the type of point cbase and path cloop. The type of cbase is built from the AST of points base using the approach described in section (3.1). The automation tool builds the type of cloop by traversing the AST of loop. During the traversal, the tool replaces the point base, which forms the two arguments to the identity type, in the co-domain of the path loop by the point cbase.

```
(def (quote S) [] [ "_" v]⇒                              -- S
 (agda-sort (lit 0) [ "C" v]⇒                             -- C : Set
  (var 0 [] [ "cbase" v]⇒                                 -- cbase
   (def (quote _≡_)                                       -- cloop
    (vArg (var 0 []) ::
     vArg (var 0 []) :: [])
    [ "cloop" v]⇒ var 2 []))))
```

Figure 13. Abstract syntax tree for recS

The recursion rule recS corresponding to figure (13) is given as follows.

```
recS : S →
  (C : Set) →
  (cbase : C) →
  (cloop : cbase ≡ cbase) →
  C
```

The automation tool builds the computation rule for the point constructor base using the same approach as described in sec. 3.1. Additionally, it includes variables in the clause definition for the path constructor loop. The tool builds the computation rule for the path constructor loop using ap as follows.

```
βrecS : (C : Set) → (cbase : C) →
  (cloop : cbase ≡ cbase) →
  ap (λ x → recS x C cbase cloop) loop
  ≡ cloop
```

The application of function recS to the path loop substitutes the point base for the lambda argument x, and it evaluates to the path cloop in the output type C. The automation tool uses declarePostulate primitive to introduce βrecS as a postulate. We implement the generateRecHit interface as follows.

```
generateRecHit :
  Arg Name → List (Arg Name) →
  (baseType : Name) → (indexList : List Nat) →
  (baseRec : Name) → (indType : Name) →
  (points : List Name) →
  (paths : List Name) → TC ⊤
generateRecHit (arg i f) argD b il br i p1 p2 =
  do lcons ← getConstructors b
     lpoints ← getLength p1
     lpaths ← getLength p2
     clauses ← getPathClause lpoints lpaths br
     RTy ← getType baseType
     fTy ← getRtypePath b i br il p2 zero RTy
     declareDef (arg i f) fTy
     defineFun f clauses
     generateβRecHit argD b il br i f p1 p2
```

```
(agda-sort (lit 0) [ "C" v]⇒
 (var 0 [] [ "cbase" v]⇒
  (def (quote _≡_)
   (vArg (var 0 []) :: vArg (var 0 []) :: [])
   [ "cloop" v]⇒
   def (quote _≡_)
   (vArg
    (def (quote ap)
     (vArg
      (lam visible
       (abs "x"
        (def (quote recS)
         (vArg (var 0 []) ::
          vArg (var 3 []) :: vArg (var 2 []) ::
          vArg (var 1 []) :: []))))
      :: vArg (def (quote loop) []) :: []))
    :: vArg (var 0 []) :: []))))
```

Figure 14. AST representing the action of function recS on path loop

generateRecHit takes the base type recursion rule as input and uses that to eliminate the points in the AST of the path methods in the recursor G_{rec}. The second argument argD is a list of terms representing the computation rules for the path constructors. The generateβRecHit interface takes argD as input and builds the computation rule for the path constructors. Other inputs to generateRecHit are the point and path holders declared during the higher inductive type definition. The following automation code generates the recursion rule and the computation rules for Circle.

```
unquoteDecl recS₁* = generateRec (vArg recS₁*)
                       (quote S₁*) (0 :: [])

unquoteDecl recS₁ βrecS₁ =
    generateRecHit (vArg recS₁)
      ((vArg βrecS₁) :: [])
      (quote S₁*) (0 :: [])
      (quote recS₁*)
      (quote S₁) S₁points S₁paths
```

The term βrecS₁ represents the computation rule for the path constructor loop and is brought into scope by unquoteDecl.

4.3 Dependent Eliminator

Dependent eliminator or the induction principle of a higher inductive type G is a dependent function that maps an element g of G to an output type $C\,g$. The general schema for the induction principle of G is given as follows.

$$
\begin{aligned}
G_{ind} :& (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\
& (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& (tgt : G\, a_1 \ldots a_n\, i_1\, \ldots\, i_n) \to \\
& (C : (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& \qquad G\, a_1 \ldots a_n\, i_1\, \ldots\, i_n \to \\
& \qquad \mathrm{Set}) \to \\
& (f_1 : \Delta_1 \to \Delta_1' \to C\, j_{11} \ldots j_{1p}\, (c_1\, \overline{\Delta_1})) \to \\
& (f_r : \Delta_r \to \Delta_r' \to C\, j_{r1} \ldots j_{rp}\, (c_r\, \overline{\Delta_r})) \to \\
& (k_1 : \Delta_1 \to \Delta_1' \to transport\, C\, p_1\, (f_i \ldots) \equiv (f_j \ldots)) \to \\
& (k_r : \Delta_r \to \Delta_r' \to transport\, C\, p_r\, (f_i \ldots) \equiv (f_j \ldots)) \to \\
& C\, i_1\, \ldots\, i_n\, tgt
\end{aligned}
$$

Similar to G_{rec}, the type of f_i is built the same way as for the normal inductive type D. The automation tool builds the type of k_i, method for path constructor p_i in G_{ind}, by traversing the AST of p_i. During the traversal, the references to point constructors g_i of G in the co-domain of p_i are replaced by f_i. In the first argument to the identity type in the co-domain of k_i, the automation tool adds the quoted name of transport, reference to the motive C, and the path p_i. The arguments of k_i are handled the same way as for f_i. The schema for the computation rules corresponding to paths p_i is given as follows.

$$
\begin{aligned}
\beta G_i :& (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\
& (C : (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& \qquad G\, a_1 \ldots a_n\, i_1\, \ldots\, i_n \to \\
& \qquad \mathrm{Set}) \to \\
& (f_1 : \Delta_1 \to \Delta_1' \to C\, j_{11} \ldots j_{1p}\, (c_1\, \overline{\Delta_1})) \to \\
& (f_r : \Delta_r \to \Delta_r' \to C\, j_{r1} \ldots j_{rp}\, (c_r\, \overline{\Delta_r})) \to \\
& (k_1 : \Delta_1 \to \Delta_1' \to transport\, C\, p_1\, (f_i \ldots) \equiv (f_j \ldots)) \to \\
& (k_r : \Delta_r \to \Delta_r' \to transport\, C\, p_r\, (f_i \ldots) \equiv (f_j \ldots)) \to \\
& apd\, (\lambda\, x \to G_{ind}\, x\, C\, f_1\, f_r\, k_1\, k_r)\, (p_i \ldots) \equiv (k_i \ldots)
\end{aligned}
$$

The automation tool builds the type of βG_{ind} using the same approach as for the induction rule G_{ind}. The type of G_{ind} and βG_{ind} is similar except for the mapping $(g : G) \to Cg$ in G_{ind} which is replaced by the term representing the action of function G_{ind} on the path $(p_i\ \ldots)$. The function apd (sec. 2.2) applies G_{ind}, which is nested inside a lambda function, on the path $(p_i \ldots)$.

For the type S with point constructor base and path constructor loop, to define a mapping indS : (x : S) → C x, we need cbase : C and cloop : C, where cloop is a heterogeneous path transported over loop. The automation tool builds the type of cloop by traversing the abstract syntax tree of loop and adding relevant type information into it.

For the codomain of cloop, which is an identity type, we insert the quoted name of transport with arguments C, loop and cbase. The automation tool applies the base eliminator to the arguments of the path loop during the construction of the co-domain of cloop. The following declaration gives the type of indS.

```
indS : (circle : S) →
  (C : S → Set) →
  (cbase : C base) →
  (cloop : transport C loop cbase ≡ cbase) →
  C circle
```

The computation rule for base, which defines the action of indS on base, is built using the same approach as for the non-dependent eliminator recS. The computation rule βindS for the path loop is built using apd which gives the action of dependent function indS on the path loop.

```
βindS : (C : S → Set) →
  (cbase : C base) →
  (cloop : transport C loop cbase ≡ cbase) →
  apd (λ x → indS x C cbase cloop) loop ≡ cloop
```

The following code gives generateIndHit interface in the automation tool.

```
generateIndHit : Arg Name → List (Arg Name) →
  (baseType : Name) → (indLs : List Nat) →
  (baseElm : Name) → (indType : Name) →
  (points : List Name) →
  (paths : List Name) → TC ⊤
generateIndHit (arg i f) argD b il br i p1 p2 =
  do il' ← getIndex b il
     lcons ← getConstructors b
     lp1 ← getLength p1
     lp2 ← getLength p2
     clauses ← (getPathClauseDep lp1 lp2 b
                 br il' lcons)
     RTy ← getType b
     fTy ← (getRtypePathDep b i br p1 p2
              zero il' RTy)
     declareDef (arg i f) fTy
     defineFun f clauses
     generateβIndHit argD b il br i f p1 p2
```

generateβIndHit builds the computation rule for the path constructors. The following code builds the induction principle and the computation rules for Circle.

```
unquoteDecl indS₁* = generateInd (vArg indS₁*)
                       (quote S₁*) []

unquoteDecl indS₁ βindS₁ =
  generateIndHit (vArg indS₁)
    ((vArg βindS₁) :: [])
    (quote S₁*) []
    (quote indS₁*)
    (quote S₁) S₁points S₁paths
```

The primitive unquoteDecl declares βindS$_1$ as a postulate. βindS$_1$ gives the action of dependent function indS$_1$ on the path constructor loop.

5 Application

The field of homotopy type theory is less well-developed on the programming side. There are only few programming applications of homotopy type theory, and the role of computationally relevant equality proofs on programming is an area of active research. Applications such as homotopical patch theory [4] discuss the implementation of Darcs [16] version control system using patch theory [12] [9] in the context of homotopy type theory. Containers in homotopy type theory [3] [2] implement data structures such as multisets and cycles. The automation tool discussed in this paper abstracts away the difficulties involved in the implementation of a higher inductive type and its elimination rules. It introduces interfaces which simplify the intricacies of a higher inductive type definition and usage by automating the generation of the code segments defining the higher inductive type and its elimination rules. The automation tool is significant in reducing the development effort for existing applications, and it can also attract new programming applications in homotopy type theory.

In the next section, we discuss the implementation of patch theory application in Agda and exhibit a tremendous reduction in code size using the automation tool. In section 5.2, we present a cryptography application and discuss how to abstract the implementation difficulties of a higher inductive type making it more accessible to the cryptographers.

5.1 Patch Theory Revisited

A patch is a syntactic representation of a function that modifies a repository context when applied. For example, a patch $(s1 \leftrightarrow s2 \,@\, l)$, which replaces string s1 with s2 at line l, when applied to a repository context with string s1 at line l results in a repository context with string s2 at line l. In homotopical patch theory [4], the patches are modeled as paths in a higher inductive type. The higher inductive type representation of patches automatically satisfy groupoid laws such as the composition of patches is associative, and inverse composes to identity. Domain-specific laws related to the patches such as two swaps at independent lines commute are designed as higher dimensional paths. The computation content of the patches is extracted by mapping them to bijections in the universe with the help of univalence. Due to the functoriality of mappings in type theory, the functions preserve the path structures in their mapping to the universe.

We developed the patch theory application in Agda using Dan Licata's method [11]. We implemented basic patches like the insertion of a string as line l1 in a file and deletion of a line l2 from a file. The functions implementing insertion and deletion in the universe are not bijective. So, to map the paths representing the patches insert and delete into the universe, we used the patch history approach [4]. According to this approach, we developed a separate higher inductive type *History* which serves as the types of patches. In addition to basic patches, we also implemented patches of encryption using cryptosystems like rsa [15] and paillier [13].

We used the automation tool described in this paper to generate code for the higher inductive type definition representing *History* and the repository context *cryptR* for the patches. We also automated the code generation for the elimination and the computation rules for the higher inductive types *History* and *cryptR*. In addition to abstracting the implementation difficulties of higher inductive types, the automation tool helped us to achieve an extensive reduction in the code size of the original application. We were able to automate the generation of approximately 1500 lines of code with just about 70 lines of automation code. The automation massively reduced the code size of the application which is about 2500 lines resulting in 60% reduction in the original code size.

5.2 Cryptography

The work of [18] applies the tools of homotopy type theory for cryptographic protocol implementation. It introduces a new approach for the formal specification of cryptographic schemes using types. The work discusses modeling *cryptDB* [14] using a framework similar to patch theory. CryptDB employs layered encryption techniques and demonstrates computation on top of encrypted data. We can implement cryptDB by modeling the database queries as paths in a higher inductive type and mapping the paths to the universe using singleton types [4]. The automation tool can be applied to generate code for the higher inductive type representing cryptDB and its corresponding elimination and computation rules. By using the automation tool, we can abstract the convolutions of homotopy type theory thus making it more accessible to the broad community of cryptography.

A formal specification of a cryptographic construction promises correctness of properties related to security and implementation. The downside of formal specification is that it introduces a framework which requires expert knowledge on theorem proving and a strong mathematical background. By automating the code constructions for the mathematical part such as the higher inductive type implementation, we simplify theorem proving and formal specification to a considerable extent and make it more accessible to regular programmers without a strong mathematical background.

6 Conclusion and Future Work

We presented an automation tool developed using the new reflection library of Agda extended with support for elaborator reflection. Our automation tool handles code generation for inductive types with constructors taking zero arguments, one or more arguments, and type being defined itself as an argument. We simplified the syntax for defining higher inductive types through the mechanized construction of the boiler-plate code segments. By automating the generation of the elimination and the computation rules associated with a higher inductive type, we demonstrated an extensive reduction in code size and abstraction of difficulties involved in implementing and using the higher inductive type. Next, we intend to extend the support to include more categories of the inductive type such as the inductive-inductive type and the inductive-recursive type.

References

[1] 2017. *Agda's Documentation.* http://agda.readthedocs.io/en/latest/language/reflection.html.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: constructing strictly positive types. Theoretic Computer Science.

[3] Thorsten Altenkirch. 2014. Containers in homotopy type theory. (January 2014). Talk at Mathematical Structures of Computation, Lyon.

[4] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. 2014. Homotopical Patch Theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* ACM.

[5] Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. Mathematical Structures in Computer Science, 15(4):671–708.

[6] David Christiansen. 2005. Dependent type providers. In Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming, WGP '13, New York, USA.

[7] David Christiansen. 2016. *Practical Reflection and Metaprogramming for Dependent Types.* Ph.D. Dissertation. IT University of Copenhagen.

[8] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16).* ACM, Nara, Japan.

[9] Jason Dagit. 2009. Type-correct changes—a safe approach to version control implementation. (2009). MS Thesis.

[10] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 34 (August 2017), 29 pages.

[11] Daniel R. Licata. 2011. Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute. (April 2011). https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant.

[12] Samuel Mimram and Cinzia Di Giusto. 2013. A categorical theory of patches. Electronic Notes in Theoretic Computer Science, 298:283–307.

[13] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In: Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic

Techniques (EUROCRYPT), Prague, Czech Republic.

[14] Raluca Ada Popa, Catherine M.S. Redfield, and Hari Balakrishnan Nickolai Zeldovich. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal.

[15] Adi Shamir Ron Rivest and Leonard Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, 21 (2), pp. 120-126.

[16] David Roundy. 2005. Darcs: Distributed version management in haskell. In *ACM SIGPLAN Workshop on Haskell*. ACM.

[17] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[18] Paventhan Vivekanandan. 2018. A Homotopical Approach to Cryptography. (July 2018). To be presented at Workshop on Foundations of Computer Security (FCS 2018), University of Oxford, UK.