Metaprogramming using Elaborator Reflection

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

David Christiansen
Galois, Inc.
Portland, Oregon, USA
davidchr@indiana.edu

Abstract

We present an automation tool with metaprogramming interfaces built using Agda's reflection library enhanced with support for elaborator reflection. We discuss code generation for defining elimination and computation rules for inductive types and higher inductive types. The metaprogramming interfaces are built using static type information queried from the type checker using Agda reflection primitives. The abstract syntax tree of dependent and non-dependent eliminators are built by invoking the interfaces. Reduction rules for point and path constructors are generated and brought into scope along with the eliminators using a top-level reflection primitive. The tool handles constructors of inductive type taking zero arguments, one or more arguments and the inductive type being defined itself as an argument. The automation tool abstracts away the intricacies of defining and using higher inductive types and helps to achieve an extensive amount of reduction in code size.

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 [7], 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 [7] 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 define the higher inductive type. 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.

- We automate the 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

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

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 [4] implementation enriched with patches of encryption.

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 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 elaborator `TC` monad. Using the reflection library and its meta-programming 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 metaprogramming interfaces described in this paper. More information on the reflection library can be found in the Agda documentation [1].

In Agda, a macro is used to construct a metaprogram that executes in a `Term` position. A macro is a function 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.

```
1
2  macro
3    mc1 : Term → Term → TC ⊤
4    mc1 exp hole = bindTC (quoteTC exp)
5                     (λ exp' → unify hole exp')
6
7  sampleTerm : Term
8  sampleTerm = mc1 (λ (n : Nat) → n)
9
```

The value of `sampleTerm` is as follows:

$$lam\ visible\ (abs\ "n"\ (var\ 0\ [])) \tag{1}$$

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 a 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. The following code gives an example usage for the `unquoteTC` primitive.

```
1
2  macro
3    mc2 : Term → Term → TC ⊤
4    mc2 exp hole = bindTC (unquoteTC exp)
5                     (λ exp' → unify hole exp')
6
7  sampleSyntax : Nat → Nat
8  sampleSyntax = mc2 (lam visible (abs "n" (var 0 [])))
9
```

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:

```
1
2  plus : Nat → Nat → Nat
3  plus zero b = b
4  plus (suc n) b = suc (plus n b)
5
```

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

```
1
2  pattern vArg x = arg (arg-info visible relevant) x
3  pattern _`⇒_ a b = pi (vArg a) (abs "_" b)
4  pattern `Nat = def (quote Nat) []
5
6  unquoteDecl plus =
7    bindTC (declareDef (vArg plus) (`Nat `⇒ `Nat `⇒ `Nat)) λ _ →
8        defineFun plus
9          (clause (vArg (con (quote zero) []) ::
10                        vArg (var "y") :: [])
11             (var 0 []) ::
12           clause (vArg (con (quote suc)
13                        (vArg (var "x") :: [])) ::
14              vArg (var "y") :: [])
15              (con (quote suc)
16                   (vArg
17                       (def plus
18                          (vArg (var 1 []) ::
19                           vArg (var 0 []) :: [])) :: [])) :: [])
20
```

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 indices 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.

2.2 Higher Inductive Type

Type theory, a foundational language of mathematics, is a functional programming language and a theorem prover. We can view a type as a topological space in homotopy theory or a groupoid in a category theory. In type theory, elements of a type corresponds to points in a topological space and elements of an identity type corresponds to paths in a topological space. Homotopy type theory extends 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 [14].

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 $f : Nat \rightarrow Nat$ 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.

```
1
2  data Circle : Set where
3    base : Circle         -- point constructor
4    loop : base ≡ base    -- path constructor
5
```

Agda is an implementation of Martin-Löf's intensional type theory. 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* [1] 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 $\{-\#REWRITE \ldots\#-\}$ 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 [9]. 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 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.

```
1
2  module Circle where
3    private
4      data S* : Set where
5        base* : S*
6
7    S : Set
8    S = S*
9
10   base : S
11   base = base*
12
13   postulate
14     loop : base ≡ base
15
```

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 non-dependent and dependent elimination rules. The following code gives the non-dependent eliminator or the recursion rule recS.

```
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
```

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. In the above code, the function ap gives the action of a function f on paths. The following equation gives the type of ap.

$$ap_f : (x =_A y) \to (f(x) =_A f(y)) \tag{2}$$

The following code gives the dependent eliminator or the induction rule indS and its computational rules.

```
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
```

In the above code, the function apd gives the action of dependent functions of type $f : \prod_{(x:A)} B(x)$ on paths. The following equation gives the type of apd.

$$apd_f : \prod_{p:x=y} (p_*(f(x)) =_{B(y)} f(y)) \tag{3}$$

In (3), the point $p_*(f(x))$ lying in space $B(y)$ can be thought of as an endpoint of a path obtained by lifting the path p from f(x) to a path in the total space $\sum_{(x:A)} B(x) \to A$. The following equation gives the type of $p*$ which is also known as *transport* [14].

$$transport^B_p : B(x) \to B(y) \tag{4}$$

where $p : x = y$ for $x, y : A$.

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 [4] application and discuss automation of code generation for the higher inductive types defined to model Darc's version control system.

```
(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 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.

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.

$$g : (A \to W) \to B \to W \to W$$

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

$$d : (A \to W) \to (A \to P) \to B \to W \to P \to P$$

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

```
(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

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 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.

$$f : (P : Set) \to \\ (d : (A \to W) \to (A \to P) \to B \to W \to P \to P) \to \\ W \to P$$

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
  (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

`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.

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

The second argument $(f \circ \alpha)$ to d is the composite of function f and α. 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 ω. 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 →
```

```
    bindTC (getClause lcons zero t f indLs cns) λ cls →
    bindTC (getType t) λ RTy →
    bindTC (getRtype t indLs zero RTy) λ funType →
    bindTC (declareDef (arg i f) funType) λ _ →
    (defineFun f cls)

```

`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.

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

In the first argument of d', the type P depends on the action of the function α 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 α, β, and ω.

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 α with codomain W, we add a new function with the same argument `A` as in α and codomain P, which depends on the action of α on `(a : A)`. We copy the constant B directly without any changes as in the case of the non-dependant eliminator. When we see

```
(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'

$(\omega : W)$ in a non-codomain position, we add the type P, which depends on the element ω, 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 α, β and ω (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'.

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

The computation rule corresponding to 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 following the same approach as the recursion principle. The computation rule corresponding to f' is given as follows.

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

We can automate the generation of the induction rule f' and its corresponding computation rules for the type W using `generateInd` interface. The following code gives the implementation of `generateInd`.

```

generateInd : Arg Name → Name → List Nat → TC ⊤
generateInd (arg i f) t indLs =
  bindTC (getIndex t indLs) λ indLs' →
  bindTC (getConstructors t) λ cns →
  bindTC (getLength cns) λ lcons →
  bindTC (getClauseDep lcons zero t f indLs' cns) λ cls →
  bindTC (getType t) λ RTy →
  bindTC (getRtypeInd t zero indLs' RTy) λ funType →
  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.

4 Code Generation for Higher Inductive Type

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 [9] method as discussed in section 2.1. In this section, we discuss the automation of code generation for the boiler-plate 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

In Agda, we define an inductive type using `data` keyword. The `data` literal characterize a data type by declaring its type and specifying its constructors. Consider the following generic form for the definition of an inductive type W with constructors $g_1 \ldots g_n$.

$$
\begin{aligned}
&data\ \ W^*\ \ (x_1 : P_1) \ldots (x_n : P_n) : Q_1 \to \ldots \to Q_n \to Set\ \ell\ where\\
&\quad g_1^* : \{i_1 : Q_1\} \ldots \{i_n : Q_n\} \to Type_1 \to W^* x_1 \ldots x_n\ i_1 \ldots i_n\\
&\qquad\qquad\qquad\vdots\\
&\quad g_n^* : \{j_1 : Q_1\} \ldots \{j_n : Q_n\} \to Type_n \to W^* x_1 \ldots x_n\ j_1 \ldots j_n
\end{aligned}
$$

The parameters $(x_1 : P_1) \ldots (x_n : P_n)$ are common to all the constructors, and the type of each constructor implicitly encodes the parameter references. However, the indices are different for each constructor. So, the type of each constructor explicitly exhibits the index declaration. During the construction of a higher inductive type, we have to represent the parameters and the indices as explicit arguments in the type of the constructors. We define a higher inductive type W as a top-level definition using a base type W^* similar to the module `Circle` in section 2.1. The reflection type `Definition` provides us the type and the constructors of the base type W^*. We copy the type of W^* to W and for the constructors $g_1 \ldots g_n$ of W, we traverse the abstract representation of the type of $g_1^* \ldots g_n^*$ respectively replacing the occurrence of W^* to W in every strict positive position. Consider a constructor g_i^* having the following type.

$$g_i^* : (A \to W^*) \to (B \to W^*) \to C \to W^* \to W^*$$

We built the type of g_i by traversing the abstract syntax tree of g_i^* and replacing the base type W^* with the higher inductive type W. The abstract syntax tree of g_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 equation represents the type of the constructor g_i.

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

We explicitly pass the type 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.

$$
\begin{aligned}
&data\ ArgPath\ \{\ell_1\} : Set\ (lsuc\ \ell_1)\ where\\
&\quad argPath : Set\ \ell_1 \to ArgPath
\end{aligned}
$$

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

$$
\begin{aligned}
&data\text{-}hit\ (quote\ W^*)\ W\\
&\quad Wpoints\ \ (g_1 :: \ldots :: g_n :: [])\\
&\quad Wpaths\ \ (p_1 :: \ldots :: p_n :: [])\\
&\quad (argPath\\
&\qquad (\{x_1 : P_1\} \to \ldots \to \{x_n : P_n\} \to\\
&\qquad\ \{i_1 : Q_1\} \to \ldots \to \{i_n : Q_n\} \to Type_1 \to\\
&\qquad\ (g_i \{x_1\} \ldots \{x_n\} \{i_1\} \ldots \{i_n\} \ldots) \equiv (g_j \ldots)) ::\\
&\qquad\qquad\qquad\vdots\\
&\quad argPath\\
&\qquad (\{x_1 : P_1\} \to \ldots \to \{x_n : P_n\} \to\\
&\qquad\ \{j_1 : Q_1\} \to \ldots \to \{j_n : Q_n\} \to Type_n \to\\
&\qquad\ (g_i \{x_1\} \ldots \{x_n\} \{j_1\} \ldots \{j_n\} \ldots) \equiv (g_j \ldots)) :: [])
\end{aligned}
$$

We define holders *Wpoints* for point constructors and *Wpaths* for path constructors as part of the higher inductive type definition of W. We cannot retrieve the constructors of the higher inductive type W using `Definition`. Therefore, *Wpoints* and *Wpaths* act as the only references for the constructors of W. 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 $g_1^* \ldots g_n^*$

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 pointH lcons pathH lpaths lpathTypes =
  bindTC (defineHindType base ind) λ _ →
  bindTC (getConstructors base) λ lcons'  →
  bindTC (defineHitCons base ind lcons' lcons) λ _ →
  bindTC (getPathTypes base ind lcons' lcons lpathTypes) λ lp →
  bindTC (defineHitPathCons lpaths lp) λ _ →
  bindTC (definePointHolder pointHolder lcons) λ _ →
  definePathHolder pathHolder lpaths
```

The higher inductive type W, the points $g_1 \ldots g_n$, the paths $p_1 \ldots p_n$, and the holders $Wpoints$ and $Wpaths$ 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 W maps the points and paths of W to an output type P. Let $g_1 : A \to W$ and $g_2 : B \to W$ be the point constructors, and $l : (\alpha : A) \to (\beta : B) \to (g_1\, \alpha) \equiv (g_2\, \beta)$ be the path constructor of W. To define a mapping $f : W \to P$, we need points $d_1 : A \to P$ and $d_2 : B \to P$ and a path $h : (\alpha : A) \to (\beta : B) \to (d_1\, \alpha) \equiv (d_2\, \beta)$ in the space

```
(pi (vArg (agda-sort (lit 0)))                    -- P
   (abs "P"
(pi (vArg (pi (vArg (def (quote A) []))           -- d₁ : A → P
   (abs "_" (var 1 [])))))                        -- P ref
   (abs "d₁"
(pi (vArg (pi (vArg (def (quote B) []))           -- d₂ : B → P
   (abs "_" (var 2 [])))))                        -- P ref
   (abs "d₂"
(pi (vArg                                         -- h
   (pi (vArg (def (quote A) []))                  -- α : A
   (abs "α"
   (pi (vArg (def (quote B) []))                  -- β : B
   (abs "β"
   (def (quote _≡_)
      vArg (var 3                                 -- d₁ ref
         (vArg (var 1 []) :: [])) ::              -- α ref
      vArg (var 2                                 -- d₂ ref
         (vArg (var 0 []) :: [])) :: []))))))     -- β ref
   (abs "h"
(pi (vArg (def (quote W) []))                     -- W → P
   (abs "_" (var 4 []))))))))))))
```

Figure 5. Abstract syntax tree for constructor f

P. The path constructor l can take arguments of any type including the higher inductive type W but for simplicity we will consider only arguments of constant type.

To construct the recursion principle f, we need to build the type of points $d1, d2$ and path h. The type of $d1$ and $d2$ are built from the abstract syntax tree of points $g1$ and $g2$ respectively using the approach described in section (3.1). We build the type of h by traversing the abstract syntax tree of l. During the traversal, we replace the points g_1 and g_2, which are arguments to the identity type, in the co-domain of the path l by the points d_1 and d_2 respectively. In figure (5), the arguments of the identity type in the abstract syntax tree of h references the points d_1 and d_2 using de-bruijn index 3 and 2 respectively. We copy the other two arguments of l, which are of constant type A and B, without any changes into the type of h.

The recursion rule f corresponding to figure (5) is given by the following equation.

$$f : (P : Set) \to (d_1 : A \to P) \to (d_2 : B \to P) \to (h : (\alpha : A) \to (\beta : B) \to (d_1\, \alpha) \equiv (d_2\, \beta)) \to W \to P$$

The computation rules for constructors g_1 and g_2 give the action of eliminator f on input $g_1(\alpha)$ and $g_2(\beta)$ respectively. The computation rules say that the function f maps the points $g_1(\alpha)$ and $g_2(\beta)$ to the points $d_1(\alpha)$ and $d_2(\beta)$ respectively in the output type P. We specify the clause definition to build the computation rules for g_1 and g_2 in the same way as for a regular inductive type (sec. 3.1) except that the clause definition for the higher inductive type W has an additional reference variable to the path h. The following definitional

```
(def (quote _≡_)
   (vArg (def (quote ap)                          -- ap
      (vArg (lam visible (abs "x"                 -- x
         (def (quote f)                           -- f
            (vArg (var 6 []) ::                   -- P ref
             vArg (var 5 []) ::                   -- d₁ ref
             vArg (var 4 []) ::                   -- d₂ ref
             vArg (var 3 []) ::                   -- h ref
             vArg (var 0 []) :: [])))) ::         -- x ref
       vArg (def (quote l)                        -- l
          (vArg (var 1 []) ::                     -- α ref
           vArg (var 0 []) :: []) :: []) ::       -- β ref
       vArg (var 2                                -- h ref
          (vArg (var 1 []) ::                     -- α ref
           vArg (var 0 []) :: []) :: []))         -- β ref
```

Figure 6. Abstract syntax tree for the term representing the action of function f on path l

equalities give the computation rules for g_1 and g_2.

$$f(P, d_1, d_2, h, g_1(\alpha)) \equiv d_1(\alpha)$$
$$f(P, d_1, d_2, h, g_2(\beta)) \equiv d_2(\beta)$$

The computation rule for the path constructor l defines the action of function f on the path $(l\,\alpha\,\beta)$. In type theory, the functions are functorial, and they preserve the path structures in their mapping to the output type [14]. The function ap_f (2) defines the action of the non-dependent eliminator f on the path l. Using (2), we can build the computation rule for the path l as follows.

$$\begin{aligned}\beta_f : (P : Set) \to (d_1 : A \to P) \to (d_2 : B \to P) \to \\ (h : (\alpha : A) \to (\beta : B) \to (d_1\,\alpha) \equiv (d_2\,\beta)) \to \\ \{\alpha : A\} \to \{\beta : B\} \to \\ ap_{(\lambda x \to f(P, d_1, d_2, h, x))}(l\,\alpha\,\beta) \equiv (h\,\alpha\,\beta)\end{aligned}$$

The computation rule β_f exists only as propositional equality. We build the type of β_f using the same approach as for the recursion rule f. The type of f and β_f is identical except for the mapping $W \to P$ in f which is replaced by the term representing the action of function f on the path $(l\,\alpha\,\beta)$. The function *ap* applies f, which is nested inside a lambda function, on the arguments $(g_1\,\alpha)$ and $(g_2\,\beta)$ of the path $(l\,\alpha\,\beta)$. We use the constructor `lam` of `Term` to introduce the lambda function with argument `x` (fig. 6). Inside the lambda body, the argument `x` gets a de-bruijn index reference of zero and we update the remaining de-bruijn indices accordingly. The application of function f to the path $(l\,\alpha\,\beta)$ substitutes the points $(g_1\,\alpha)$ and $(g_2\,\beta)$ for the lambda argument `x`, and it evaluates to the path $(h\,\alpha\,\beta)$ in the output type P. We implement the `generateRecHit` interface as follows.

```
1
2  generateRecHit : Arg Name → List (Arg Name) →
3    (baseType : Name) → (indexList : List Nat) →
4    (baseRec : Name) → (indType : Name) →
5    (points : List Name) → (paths : List Name) → TC ⊤
6  generateRecHit (arg i f) argD base il rec ind points paths =
7    bindTC (getConstructors base) λ lcons →
8    bindTC (getLength points) λ lpoints →
9    bindTC (getLength paths) λ lpaths →
10   bindTC (getPathClause lpoints lpaths rec) λ clause →
11   bindTC (getType base) λ RTy →
12   bindTC (getRtypePath base ind rec il paths zero RTy) λ fty →
13   bindTC (declareDef (arg i f) fty) λ _ →
14   bindTC (defineFun f clause) λ _ →
15   (generateβRecHit argD base il rec ind f points paths)
16
```

`generateRecHit` takes the base type recursion rule as input and uses that to map the points $g_1(\alpha)$ and $g_2(\beta)$ to $d_1(\alpha)$ and $d_2(\beta)$ respectively in the abstract syntax tree of the path $(l\,\alpha\,\beta)$. 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 constructor l. Other inputs to `generateRecHit` are the point and path holders declared during the higher inductive type definition of W. 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 $\beta\texttt{recS}_1$ 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 W is a dependent function that maps an element ω of W to an output type $P\,\omega$. For the type W with point constructors $g_1 : A \to W$ and $g_2 : B \to W$ and path constructor $l : (\alpha \to A) \to (\beta \to B) \to (g_1\,\alpha) \equiv (g_2\,\beta)$, to define a mapping $f' : (\omega : W) \to P\,\omega$, we need $d_1 : (\alpha : A) \to P\,\alpha$, $d_2 : (\beta : B) \to P\,\beta$, and $h : (\alpha \to A) \to (\beta : B) \to transport\,P\,(l\,\alpha\,\beta)\,(d_1\,\alpha) \equiv (d_2\,\beta)$, where $(h\,\alpha\,\beta)$ is a heterogeneous path transported over $(l\,\alpha\,\beta)$ lying in the space $P\,\beta$. Equation (4) gives the type of *transport*.

In the abstract syntax tree of f' (fig. 7), the type of constructors d_1 and d_2 are built from the type of g_1 and g_2 respectively using the same approach as in section (3.2). When building the type of d_1, we copy the constant argument A directly into the abstract syntax tree of d_1, and for the codomain P, which depends on the action of constructor g_1 on the constant argument $\alpha : A$, we pass the de-bruijn index reference of α as an argument to the constructor g_1 built using the reflection construct con. Similarly, for d_2, we copy the type B directly into the abstract syntax tree of d_2 and pass the de-bruijn index of $\beta : B$ to the constructor g_2 in the

```
(pi (vArg (`W `⇒ agda-sort (lit 0)))
   (abs "P"
(pi (vArg
   (pi (vArg `A)
   (abs "α"
   (var 1 (vArg (con (quote g₁)
            (vArg (var 0 []) :: [])) :: [])))))
   (abs "d₁"
(pi (vArg
   (pi (vArg `B)
   (abs "β"
   (var 2 (vArg (con (quote g₂)
            (vArg (var 0 []) :: [])) :: [])))))
   (abs "d₂"
(pi (vArg
   (pi (vArg `A)
   (abs "α"
   (pi (vArg `B)
   (abs "β"
   (def (quote _≡_)
     (vArg (def (quote transport)
       (vArg (var 4 []) ::
        vArg (def (quote l) (vArg (var 1 []) ::
        vArg (var 0 []) :: [])) ::
        vArg (var 3 (vArg (var 1 []) :: [])) :: [])) ::
     vArg (var 2 (vArg (var 0 []) :: [])) :: [])))))))
   (abs "h"
(pi (vArg `W)
   (abs "ω" (var 4 (vArg (var 0 []) :: [])))))))))))
```

Figure 7. Abstract syntax tree of function f'

co-domain. We build the type of path h by traversing the abstract syntax tree of l and adding relevant type information into it. We copy the constant arguments of type A and B without modification. For the codomain of l, which is the identity type $(g_1\,\alpha) \equiv (g_2\,\beta)$, we insert the quoted name of function *transport* with arguments P, the path $(l\,\alpha\,\beta)$ and $(d_1\,\alpha)$. We apply the base eliminator to the arguments of the path $(l\,\alpha\,\beta)$ to form the elements $(d_1\,\alpha)$ and $(d_2\,\beta)$. The following declaration gives the type of f' corresponding to figure (7).

$$
\begin{aligned}
&f' : (P : W \to Set) \to \\
&\quad (d_1 : (\alpha : A) \to P(g_1\,\alpha)) \to (d_2 : (\beta : B) \to P(g_2\,\beta)) \to \\
&\quad (h : (\alpha : A) \to (\beta : B) \to \\
&\qquad transport\ P\ (l\,\alpha\,\beta)(d_1\,\alpha) \equiv (d_2\,\beta)) \to \\
&\quad (\omega : W) \to P(\omega)
\end{aligned}
$$

We build the computation rules corresponding to the mapping f' for the point constructors g_1 and g_2 using the same approach as for the non-dependent eliminator f. The following equations give the computation rules for the point constructors g_1 and g_2.

$$
\begin{aligned}
f'(P, d_1, d_2, h, g_1(\alpha)) &\equiv d_1(\alpha) \\
f'(P, d_1, d_2, h, g_2(\beta)) &\equiv d_2(\beta)
\end{aligned}
$$

```
(def (quote _≡_)
  (vArg (def (quote apd)                       -- apd
    (vArg (lam visible (abs "x"                -- x
      (def (quote f')                          -- f
        (vArg (var 6 []) ::                    -- P ref
         vArg (var 5 []) ::                    -- d₁ ref
         vArg (var 4 []) ::                    -- d₂ ref
         vArg (var 3 []) ::                    -- h ref
         vArg (var 0 []) :: [])))) ::          -- x ref
     vArg (def (quote l)                       -- l
       (vArg (var 1 []) ::                     -- α ref
        vArg (var 0 []) :: [])) :: [])) ::     -- β ref
     vArg (var 2                               -- h ref
       (vArg (var 1 []) ::                     -- α ref
        vArg (var 0 []) :: [])) :: []))        -- β ref
```

Figure 8. Abstract syntax tree for the term representing the action of dependent function f' on path l

The computation rule $\beta_{f'}$ for the heterogeneous path h depends on *apd* (3) which gives the action of dependent function f' on the path h. We build the type of $\beta_{f'}$ using (3) as follows.

$$
\begin{aligned}
&\beta_{f'} : (P : W \to Set) \to \\
&\quad (d_1 : (\alpha : A) \to P(g_1\,\alpha)) \to (d_2 : (\beta : B) \to P(g_2\,\beta)) \to \\
&\quad (h : (\alpha : A) \to (\beta : B) \to \\
&\qquad transport\ P\ (l\,\alpha\,\beta)(d_1\,\alpha) \equiv (d_2\,\beta)) \to \\
&\quad apd_{(\lambda x \to f'(P, d_1, d_2, h, x))}(l\,\alpha\,\beta) \equiv (h\,\alpha\,\beta)
\end{aligned}
$$

We build the abstract syntax tree for the computation rule $\beta_{f'}$ in the same way as for induction principle f'. In the codomain of $\beta_{f'}$ (fig. 8), which is an identity type, we insert the abstract representation of the function *apd* with arguments f', nested inside a lambda body, and the path $(l\,\alpha\,\beta)$. We pass the path $(h\,\alpha\,\beta)$, built using de-bruijn indices, as the second argument to the identity type. The following code gives the implementation of the `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 indLs elm ind p1 p2 =
  bindTC (getIndex base indLs) λ il →
  bindTC (getConstructors base) λ lcons →
  bindTC (getLength points) λ l1 →
  bindTC (getLength paths) λ l2 →
  bindTC (getPathClauseDep l1 l2 base elm il lcons) λ cl →
  bindTC (getType baseType) λ RTy →
  bindTC (getRtypePathDep base ind elm p1 p2 zero il RTy) λ fty →
  bindTC (declareDef (arg i f) fty) λ _ →
  bindTC (defineFun f cl) λ _ →
  (generateβIndHit argD base indLs elm ind f points paths)

```

The interface `generateIndHit` takes the base eliminator as an argument and uses that to map the point constructors

g_1 and g_2 to output d_1 and d_2 when building the codomain of the path constructor h. `generateβIndHit` builds the computation rule for the path constructor h. 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 $\beta\texttt{indS}_1$ as a postulate. $\beta\texttt{indS}_1$ gives the action of dependent function $\texttt{indS}_1$ 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 [13] version control system using patch theory [10] [8] 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 following section, we discuss the implementation of patch theory application in Agda using the automation tool.

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 $(s1 \leftrightarrow s2 \mathbin{@} l)$, 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 [9]. 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 [12] and paillier [11].

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.

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 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.

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

[9] 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.

[10] Samuel Mimram and Cinzia Di Giusto. 2013. A categorical theory of patches. Electronic Notes in Theoretic Computer Science, 298:283–307.
[11] 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.
[12] 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.
[13] David Roundy. 2005. Darcs: Distributed version management in haskell. In *ACM SIGPLAN Workshop on Haskell*. ACM.
[14] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.