Code Generation for Higher Inductive Types
A Study in Agda Metaprogramming

Paventhan Vivekanandan

Indiana University Bloomington

Abstract. Higher inductive types are inductive types that include non-trivial higher-dimensional structure, represented as identifications 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 with mature implementations. 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

Type theory unites programming and mathematics in a delightful synthesis, in which we can write programs and proofs in the same language. Work on higher-dimensional type theory has revealed a beautiful higher-dimensional structure, lurking just beyond reach. In particular, higher inductive types provide a natural encoding of many otherwise-difficult mathematical concepts, and univalence lets us work in our type theory the way we do on paper: up to isomorphism. Homotopy type theory, however, is not yet done. We do not yet have a mature theory or a mature implementation.

While work proceeds on prototype implementations of higher-dimensional type theories [**?**][**?**], much work remains before they will be as convenient for experimentation with new ideas as Coq, Agda, or Idris is today. In the meantime, it is useful to be able to experiment with ideas from higher-dimensional type theory in our existing systems. If one is willing to put up with some boilerplate code, it is possible to encode higher inductive types and univalence using a mixture of postulated identities and traditional datatypes [**?**]. We use rewrite rules to define higher inductive type in this paper.

Boilerplate postulates, however, are not just inconvenient, they are also an opportunity to make mistakes. Luckily, this boilerplate code can be mechanically generated using Agda's recent support for *elaborator reflection* [**?**], a paradigm for metaprogramming in an implementation of type theory. An elaborator is the part of the implementation that translates a convenient language designed for humans into a much simpler, more explicit, verbose language designed to be easy for a machine to process. Elaborator reflection directly exposes the primitive components of the elaborator to metaprograms written in the language being elaborated, allowing them to put these components to new uses.

Using Agda's elaborator reflection, we automatically generate the support code for many useful higher inductive types, specifically those that include additional paths between constructors, but not paths between paths. We automate the production of the recursion principles, induction principles, and their computational behavior. Angiuli et al.'s encoding of patch theory as a higher inductive type [?] requires approximately 1500 lines of code when represented using rewrite mechanism. Using our library, the encoding can be expressed in just 70 lines.

This paper makes the following contributions:

– We describe the design and implementation of a metaprogram that automates an encoding of higher inductive types with one path dimension using Agda's new metaprogramming system.
– We demonstrate applications of this metaprogram to examples from the literature, including both standard textbook examples of higher inductive types as well as larger systems, including both patch theory and specifying cryptographic schemes.
– This metaprogram serves as an example of the additional power available in Agda's elaborator reflection relative to earlier metaprogramming APIs.

2 Background

2.1 Higher Inductive Types

Homotopy type theory [?] 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 identifications 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 [?] 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.

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 **??** describes `Circle`, which is a higher inductive type with one point constructor `base` and one non-trivial path constructor `loop`. Agda [?] is a functional programming

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

Fig. 1: A specification of a higher inductive type

language with full dependent types and dependent pattern matching. Agda's type theory has gained a number of new features over the years, 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.

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. Another approach is to use Licata's method [?]. But Licata's method has inconsistencies [?], and so we use rewrite rules for defining higher inductive types as follows.

Inside the module `Circle`, the type `S` and the constructors `base` and `loop` and the recursion and induction principles are declared as postulates. The computation rules for point `base` are declared as rewrite rules using `{-# REWRITE , ...#-}` pragma. The computation rules for paths are then declared as postulates.

`recS` (fig **??**) 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 `βloop`. The operator `ap` 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
```

In addition to describing the constructors of the points and paths of `S`, figure **??** additionally demonstrates the dependent eliminator (that is, the induction

```
postulate
  _↦_ : ∀ {i} {A : Set i} → A → A → Set i
{-# BUILTIN REWRITE _↦_ #-}
module Circle where
  postulate
    S : Set
    base : S
    loop : base ≡ base
  postulate
    recS : S → (C : Set) → (cbase : C) → (cloop : cbase ≡ cbase) → C
    βbase : (C : Set) → (cbase : C) → (cloop : cbase ≡ cbase) →
      recS base C cbase cloop ↦ cbase
  {-# REWRITE βbase #-}
  postulate
    βloop : (C : Set) → (cbase : C) → (cloop : cbase ≡ cbase) →
      ap (λ x → recS x C cbase cloop) loop ≡ cloop
  postulate
    indS : (x : S) → (C : S → Set) →
      (cbase : C base) → (cloop : transport C loop cbase ≡ cbase) → C x
    iβbase : (C : S → Set) →
      (cbase : C base) → (cloop : transport C loop cbase ≡ cbase) →
      indS base C cbase cloop ↦ cbase
  {-# REWRITE iβbase #-}
  postulate
    iβloop : (C : S → Set) →
      (cbase : C base) → (cloop : transport C loop cbase ≡ cbase) →
      apd (λ x → indS x C cbase cloop) loop ≡ cloop
```

Fig. 2: A HIT encoded using rewrite rules

rule) `indS` and its computational meaning. 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 rewrite mechanism. Section 5 revisits Angiuli et al.'s encoding of Darcs's patch theory [?] and demonstrates that the higher inductive types employed in that paper can be generated succinctly using our library [1].

[1] Please see `https://github.com/pavenvivek/WFLP-18`

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

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

Fig. 3: 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 [])))
```

Fig. 4: A macro that unquotes its argument

2.2 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 [?,?] and Lean's tactic metaprogramming [?]. 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 [?].

`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 $t_1 \to t_2 \to \ldots \to$ `Term` $\to$ `TC` $\top$ 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.

Figure ?? 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. This doubly-quoted expression 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.

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

Fig. 5: 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 []) :: [])) ::
              [])) ::
        [])
```

Fig. 6: Addition, defined by metaprogramming

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 **??** 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 **??** shows an Agda implementation of addition on natural numbers, while figure **??** demonstrates an equivalent metaprogram that adds the same definition to the context.

In Figure **??**, `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 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 matches a particular constructor, while in `Terms`, it denotes a reference to a constructor.

3 Code Generation for Inductive Types

An inductive type D is a type that is freely generated by a finite collection of constructors. The constructors of D accept zero or more arguments, and result in an D. The constructors can also take an element of type D itself as an argument, but only *strictly positively*: any occurrences of the type constructor D in the type of an argument to a constructor of D 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 vary.

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 `Names`. The type of a constructor can be retrieved by giving its `Name` as an input to the `getType` primitive. In this section, we discuss how to use the 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:

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

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.

As an example, the datatype `Vec` represents lists of a known length. It is defined as follows.

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

There is one parameter, namely `(A : Set)`, and one index, namely `Nat`. The second constructor, `_::_`, has a recursive instance of `Vec` as an argument.

While inductive datatypes are essentially characterized by their constructors, it must also be possible to eliminate their inhabitants, exposing the information in the constructors. This section describes an Agda metaprogram that generates a non-dependent recursion principle for an inductive type; section **??** generalizes this technique to fully dependent induction principles.

For `Vec`, the recursion principle says that, in order to eliminate a `Vec A n`, one must provide a result for the empty `Vec` and a means for transforming the head and tail of a non-empty `Vec` combined with the result of recursion onto a tail into the desired answer for the entire `Vec`. Concretely, the type of the recursor `recVec` is given in figure ??.

The recursor `recVec` maps the constructor `[]`, which takes zero arguments, to `base`. It maps `(x :: xs)` to `(step x xs (recVec xs C base step))`. Because `step` is applied to a recursive call to the recursor, it takes one more argument than the constructor `_::_`.

Based on the schematic presentation of inductive types D earlier in this section, we can define a schematic representation for their non-dependent eliminators D_{rec}.

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

The type of f_i, which is the method for fulfilling the desired type C when eliminating the constructor c_i, 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 $(y' : \Psi \to C)$.

To construct the type of `recVec`, we need to build the types of `base` and `step`. These are derived from the corresponding types of `[]` and `_::_`, which can be discovered using reflection primitives. Since `[]` requires no arguments, its corresponding method is `(base : C)`. The constructor `pi` of type `Term` encodes the abstract syntax tree (AST) representation of `_::_` (figure ??). We can retrieve and traverse the AST of `_::_`, and add new type information into it to build a new type representing `step`.

During the traversal of abstract syntax tree of the type of `_::_`, when the type `Vec` occurs directly as an argument, the result type `C` is added next to it. For example, in figure ??, a new function is built from the argument `(xs : Vec A n)` by modifying it to `(Vec A n) → C` (figure ??). Arguments other than `Vec` require no modifications. Therefore, `(x : A)` is copied into the new type without any changes. Finally, the codomain `Vec A (suc n)` of `_::_`'s type is replaced with `C`, resulting in an AST for the type of `step`.

In general, 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 number of parameters occurring in a defined type. It also encodes the constructors of the type as a list of `Names`. Metaprograms can retrieve the index count by finding the difference

```
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 []) :: []))
         :: [])))))
```

Fig. 7: AST for the type of _::_

```
recVec : (A : Set) →
         {n : Nat} → Vec A n →
         (C : Set) →
         (base : C) →
         (step : {n : Nat} →
                 (x : A) →
                 (xs : Vec A n) → C →
                 C) → C
```

Fig. 8: Non-dependent eliminator of Vec

between the number of parameters and the length of the constructor list. The constructors of D refer to the parameter and the index using de Bruijn indices.

The method step for the constructor _::_ in Vec, refers to the parameter and the index using de Bruijn indices. During the construction of the type of step, the recursor generator updates the de Bruijn indices accordingly. Note that not all indices occur as arguments to a constructor: in Vec, the constructor [] instantiates the index with a constant.

```
(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]⇒                   -- base
     ((def (quote Nat) [] [ "n" h]⇒       -- step
       (var 5 [] [ "x" v]⇒                -- x
        (def (quote Vec)
          (vArg (var 6 []) ::
           vArg (var 1 []) ::
           [])
          [ "xs" v]⇒                      -- xs
          (var 4 [] [ "_" v]⇒             -- → C
           var 5 []))))                   -- C
      [ "_" v]⇒ var 2 []))))))            -- C
```

Fig. 9: Abstract syntax tree of recVec's type

Once the AST for step's type has been found, it is possible to build the type of recVec in figure ??. To quantify over the return type (C : Set), the Term constructor agda-sort refers to Set.

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

```
recVec []        C base step =
  base
recVec (x :: xs) C base step =
  step x xs (f xs C base step)
```

Fig. 10: Computation rules for `recVec`

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

Fig. 11: W-Type

$$
\begin{aligned}
&D_{rec}\; a_1 \;\dots\; a_n \; i_1 \;\dots\; i_m \; (c_1 \; \Delta_1) \, C \, f_1 \dots f_r = \\
&\quad \mathsf{RHS}\,(f_1, \Delta'_1) \\
&\vdots \\
&D_{rec}\; a_1 \;\dots\; a_n \; i_1 \;\dots\; i_m \; (c_r \; \Delta_r) \, C \, f_1 \dots f_r = \\
&\quad \mathsf{RHS}\,(f_r, \Delta'_r)
\end{aligned}
$$

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 $\mathsf{RHS}\,(f_j \; y \;\; (D_{rec} \;\; \dots y \dots) \, , \Delta_k)$, 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

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

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

After declaring `recVec`'s type using `declareDef`, it is time to define its computational meaning. The computation rule representing the action of function `recVec` on `[]` and `_::_` is defined using `clause` (figure **??**). The first argument to `clause` encodes variables corresponding to the above type, and it also includes the abstract representation of the constructors `[]` and `_::_` 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 `recVec` when the pattern matches. The constructor `var` of `Pattern` is used to introduce new variables in the `clause` definition. The type `Pattern` also has another constructor `con` that represents patterns that match specific constructors. The type `Term` has similar constructors `var` and `con`, that respectively represent variable references and constructor invocations. The computation rules for `recVec` are given in fig **??**.

Figure **??** presents the AST for the clause that matches `_::_`. The de Bruijn index reference increments right-to-left, starting from the last argument. Definitions by pattern matching are added to the global context using the `defineFun` primitive.

```
(clause
  (vArg (con (quote _::_)           -- _::_
          (vArg (var "x") ::         -- x
           vArg (var "xs") :: []))  -- xs
   vArg (var "C") ::                 -- C
   vArg (var "base") ::              -- base
   vArg (var "step") :: [])          -- step
  (var 0
    (vArg (var 4 []) ::              -- x
     vArg (var 3 []) ::              -- xs
     vArg
      (def recVec                    -- recursion
       (vArg (var 3 [])              -- xs
        vArg (var 2 []) ::           -- C
        vArg (var 1 []) ::           -- base
        vArg (var 0 []) :: [])))))  -- step
```

Fig. 12: Clause definition for the computation rule of `_::_`

Figure **??** presents an Agda encoding of Martin-Löf's well-orderings, the so-called *W-type*. `W` is interesting because its constructor exhibits a *higher-order* recursive instance of the type being defined. Following the recipe yields the recursor in figure **??**, in which the recursive call occurs under a function representing arbitrary choices of tag.

```
recW : ∀ {A B}
       (tgt : W A B) →
       (C : Set) →
       (step : (x : A) →
               (f : B x → W A B) →
               (f' : B x → C) →
               C) →
       C
recW (sup x f) C step =
  step x f (λ b → recW (f b) C step)
```

Fig. 13: The non-dependent eliminator for `W`

```
indW : {A : Set} → {B : A → Set} →
       (tgt : W A B) →
              (mot : W A B → Set) →
       (step : (x : A) →
               (f : B x → W A B) →
               (f' : (b : B x) → mot (f b)) →
               mot (sup x f)) →
       mot tgt
indW (sup x f) mot step =
  step x f (λ b -> indW (f b) mot step)
```

Fig. 14: The induction principle for `W`

The type of `step` is built by traversing the AST of `sup`'s type. The first argument to `sup`, which is a constant type `A`, is copied directly into `step`'s type. The second argument is a `(B x → W A B)`, which is a function whose codomain is a recursive instance of `W`. The resulting arguments must account for the recursion and are thus `(B x → W A B)` and `(B x → C)`. Finally, the codomain `W A B` of `sup` is replaced by `C`.

In the above computation rule (fig. **??**), the third argument to `step` is a function that works for any choice of tag `b`. The arguments to `lam` are referenced using de Bruijn indices inside the lambda body. Thus, the de Bruijn indices for referring variables outside the lambda body must be updated accordingly.

`generateRec` build the computation and elimination rules respectively. The recursion rule generated by `generateRec` is brought into scope using `unquoteDecl`.

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. The type of the induction principle for D is:

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

Unlike the non-dependent recursion principle D_{rec}, the result type is now computed from the target and its indices. Because it expresses the reason that the target must be eliminated, the function C is often referred to as the *motive*. Similarly to D_{rec}, the type of each method f_i is derived from the type of the constructor c_i—the method argument telescope Δ'_k is similar, except the arguments that represent the result of recursion now apply the motive C to appropriate 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 still binds $(y : B)$ directly. If B is an application of D to parameters $a \ldots$ and indices $e \ldots$, then an additional binding $(y' : C\ e \ldots\ y)$ is inserted following y. Finally, if B is a function type $\Psi \to D\ a \ldots\ e \ldots$, the additional binding is $(y' : \Psi \to C\ e \ldots (y\ \overline{\Psi}))$.

The computation rules for the induction principle are the same as for the recursion principle. Following these rules, the induction principle for `W` can be seen in figure **??**.

Automating the production of the dependent eliminator is an extension of the procedure for automating the production of the non-dependent eliminator.

We can construct the AST of `step` using the static type information obtained from `sup`. To construct `indW`, during the traversal of the AST of `sup`'s type, the argument `(a : A)` is copied without any changes, just as it is in the case of the non-dependent eliminator. The next argument `f`, however, is a function that returns a `W`. An additional argument, representing the induction hypothesis, is needed. The induction hypothesis `(f' : (b : B x) → mot (f b))` takes the same arguments as `f`, but it returns the motive instantiated at the application of `f`. The final return type is found by applying the motive to the target and its indices.

We can construct the type of the induction principle `indW` using the AST of `step`. The type `C` in the mapping `indW` depends on the element of the input type `W A B`. Operationally, the induction principle computes just like the recursion principle. It is constructed using `clause` definitions following the same approach. The generation of induction principles is carried out using `generateInd`.

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

Fig. 15: Generic schema for recursor

$$\begin{aligned}\beta G_{rec} : {} & (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\ & (C : \mathsf{Set}) \to \\ & (f_1 : \Delta'_1 \to C) \ldots (f_r : \Delta'_r \to C) \to \\ & (k_1 : \Delta'_1 \to (f_i \ldots) \equiv (f_j \ldots)) \to \\ & \vdots \\ & (k_q : \Delta'_q \to (f_i \ldots) \equiv (f_j \ldots)) \to \\ & ap\ (\lambda\ x.G_{rec}\ x\ C\ f_1\ \ldots\ f_r\ k_1\ \ldots\ k_q) \\ & \quad (p_i \ldots) \equiv (k_i \ldots)\end{aligned}$$

Fig. 16: Generic schema for computation rule corresponding to Grec

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 using rewrite rules, as described in section **??**. In this section, we discuss the automation of code generation for the elimination and the computation rules of higher inductive types. The schema given in this section follows a pattern similar to Basold et al.'s [**?**] general rules for higher inductive types. However, we do not know how to generalize the formulation of the elimination rules for higher inductive types.

4.1 Non-dependent Eliminators for HITs

The recursion principle of a higher inductive type G maps the points and paths of G to points and paths in an output type C. We extend the general schema of the recursion principle given in section **??** by adding methods for path constructors (fig. **??**).

The schematic definition of G_{rec} supports only one-dimensional paths. Our metaprogram currently supports only one-dimensional paths, but we are planning to improve the tool to support higher-dimensional paths in the future.

The type of the method f_i for a point constructor g_i in G_{rec} is built the same way as for the normal inductive type D, as described in section **??**. The code generator builds the type of k_i, method for path constructor p_i in G_{rec}, by traversing the AST of p_i. The arguments of k_i are handled the same way as for the point constructor's method f_i. During the traversal, the code generator uses the base type recursor D_{rec} to map the point constructors g_i of G in the codomain of p_i to f_i. Determining 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 there are additional methods to handle paths. Paths compute new paths; the computation rules that govern the interaction of recursors and paths p_i are named and postulated. They identify the action of the recursor on the path with the corresponding method. As an example, if code for the circle HIT from section **??** has been generated, and the type is called `S`, then the recursor needs a method for `base` and one for `loop`. The method for `base` should be an inhabitant of `C`. If it is called `cbase`, then the method for `loop` should be a path `cbase ≡ cbase`. The types of the path methods depend on the values of the point methods. The code generator builds the type of `loop`'s method by traversing the AST of `loop`'s type, replacing references to point constructors with the result of applying the base type's recursor to the point methods.

The recursion rule `recS` follows this pattern.

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

The code generator builds the computation rule for the point constructor `base` using the same approach as described in section **??**, as if it were for the base type. Additionally, it includes variables in the `clause` definition for the path constructor `loop`. The code generator postulates the following computation rule `βloop` for the path constructor `loop`:

```
βloop : (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 argument `x`, and it evaluates to the path `cloop` in the output type `C`. In the tool, `generateRecHit` is used to build the elimination rule and the computation rules for points, and `generateβRecHitPath` is used to build the computation rules for paths.

4.2 Dependent Eliminators for HITs

The dependent eliminator (fig. **??**) for 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 in figure **??**.

Similar to G_{rec}, the type of f_i is built the same way as for the normal inductive type D. The code generator builds the type of the method for path constructor p_i, called k_i, in G_{ind}, by traversing the AST of p_i. During the traversal, the code generator uses the base eliminator D_{ind} to map the point constructors g_i of G in the codomain of p_i to f_i. In the first argument to the identity type in the codomain of k_i, the code generator adds an application of `transport` to the motive `C` and the path p_i. The arguments of k_i are handled the same way as for f_i. The computation rules corresponding to paths p_i are postulated as given in figure **??**.

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 base` and `cloop : transport C loop cbase ≡ cbase`, where `cloop` is a heterogeneous path transported over `loop`. The code generator builds the type of `cloop` by adding relevant type information to the type of `loop`. The type of the method for the path constructor `cloop` is derived by inserting a call to `transport` with arguments `C`, `loop` and `cbase`. The code generator applies the base eliminator to

$$
\begin{aligned}
G_{ind} :\ & (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\
& (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& (tgt : G\ a_1 \ldots a_n\ i_1\ \ldots\ i_n) \to \\
& (C : (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& \quad G\ a_1 \ldots a_n\ i_1\ \ldots\ i_n \to \\
& \quad \mathsf{Set}) \to \\
& (f_1 : \Delta'_1 \to C\ j_{11} \ldots j_{1p}\ (c_1\ \overline{\Delta_1})) \to \\
& \vdots \\
& (f_r : \Delta'_r \to C\ j_{r1} \ldots j_{rp}\ (c_r\ \overline{\Delta_r})) \to \\
& (k_1 : \Delta'_1 \to \texttt{transport}\ C\ p_1\ (f_i \ldots) \equiv (f_j \ldots)) \to \\
& \vdots \\
& (k_q : \Delta'_q \to \texttt{transport}\ C\ p_q\ (f_i \ldots) \equiv (f_j \ldots)) \to \\
& C\ i_1\ \ldots\ i_n\ tgt
\end{aligned}
$$

Fig. 17: Generic schema for induction principle

$$
\begin{aligned}
\beta G_i :\ & (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \\
& (C : (i_1 : I_1) \to \ldots \to (i_m : I_m) \to \\
& \quad G\ a_1 \ldots a_n\ i_1\ \ldots\ i_n \to \\
& \quad \mathsf{Set}) \to \\
& (f_1 : \Delta'_1 \to C\ j_{11} \ldots j_{1p}\ (c_1\ \overline{\Delta_1})) \to \\
& (f_r : \Delta'_r \to C\ j_{r1} \ldots j_{rp}\ (c_r\ \overline{\Delta_r})) \to \\
& (k_1 : \Delta'_1 \to \texttt{transport}\ C\ p_1\ (f_i \ldots) \equiv (f_j \ldots)) \to \\
& (k_r : \Delta'_r \to \texttt{transport}\ C\ p_r\ (f_i \ldots) \equiv (f_j \ldots)) \to \\
& \texttt{apd}\ (\lambda\ x\ .\ G_{ind}\ x\ C\ f_1\ \ldots\ f_r\ k_1\ \ldots\ k_r) \\
& \quad (p_i \ldots) \equiv \\
& (k_i \ldots)
\end{aligned}
$$

Fig. 18: Generic schema for computation rule corresponding to Gind

map the point `base` to `cbase` during the construction of the codomain 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 postulated computation rule `iβloop` for the path `loop` uses `apd` which gives the action of dependent function `indS` on the path `loop`.

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

`generateIndHit` is used to build the elimination rule and the computation rules for points, and `generateβIndHitPath` is used to build the computation rules for paths.

5 Applications

Homotopy type theory has thus far primarily been applied to the encoding of mathematics, rather than to programming. Nevertheless, there are a few applications of homotopy type theory to programming. Applications such as homotopical patch theory [?] discuss a model of the core of the of Darcs [?] version control system using patch theory [?] encoded as a HIT. Containers in homotopy type theory [?,?] implement data structures such as multisets and cycles. Automating the HIT boilerplate code allows more programmers to begin experimenting with programming with HITs.

5.1 Patch Theory Revisited

We reimplemented Angiuli et al.'s patch theory in Agda. We implemented basic patches such as the insertion of a string as line l_1 in a file and deletion of a line l_2 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 Angiuli et al.'s patch history approach. 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 involving encryption or decryption with cryptosystems like RSA and Paillier.

Having implemented patch theory, we then reimplemented `History` and the encrypted repository type `cryptR` using our code generator. We also automated the code generation for the elimination and the computation rules for the higher inductive types `History` and `cryptR`. In addition to easing the implementation difficulties of higher inductive types, the code generator greatly reduced the code size. The type definitions shrank from around 1500 to around 70 lines, resulting in a 60% decrease in the overall number of lines of code in the development.

5.2 Cryptographic Protocols

Vivekanandan [?] models certain cryptographic protocols using homotopy type theory, introducing a new approach to formally specificying cryptographic schemes using types. The work discusses modeling cryptDB [?] using a framework similar to Angiuli et al.'s patch theory. CryptDB employs layered encryption techniques and homomorphic encryption. 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 [?]. The code generator can be applied to generate code for the higher inductive type representing cryptDB and its corresponding elimination and computation rules. By using the code generator, we can decrease the length and increase the readability of the definitions, hopefully making it more accessible to the broad cryptographic community.

6 Related Work

Kokke and Swierstra [?] implemented a library for proof search using Agda's old reflection primitives, from before it had elaborator reflection. They describe a Prolog interpreter in the style of Stutterheim et al. [?]. It employs a hint database and a customizable depth-first traversal, with lemmas to assist in the proof search.

van der Walt and Swierstra [?] and van der Walt [?] discuss automating specific categories of proofs using proof by reflection. A key component of this proof technique is a means for converting an expression into a quoted representation. They automate this process, giving a user-defined datatype. van Der Walt gives an overview of Agda's old metaprogramming tools.

Ongoing work on cubical type theories [?][?][?] provides a computational interpretation of univalence and HITs. We strenuously hope that these systems quickly reach maturity, rendering our code generator obsolete. In the meantime, however, these systems are not yet as mature as Agda.

7 Conclusion and Future Work

We presented a code generator that generates the encodings of higher inductive types, developed using Agda's new support for Idris-style elaborator reflection. In particular, the tool generates the dependent and non-dependent elimination rules and the computational rules for 1-dimensional higher inductive types. This syntax is greatly simplified with respect to writing the encoding by hand. We demonstrated an extensive reduction in code size by employing our tool. Next, we intend to extend the tool to support higher-dimensional paths in the definition of HITs, bringing its benefits to a wider class of problems.

References

1. The Univalent Foundations Program, Institute for Advanced Study. Homotopy Type Theory: Univalent Foundations Of Mathematics (2013). `homotopytypetheory.org`
2. Anguili, C., Morehouse, E., Licata, D., Harper, R.: Homotopical Patch Theory. In: International Conference on Functional Programming (ICFP), Sweden (2014)
3. Popa, R.A., Redfield, C.M.S, Zeldovich, N., Hari Balakrishnan, H. : CryptDB: Protecting Confidentiality with Encrypted Query Processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), Portugal (2011)
4. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology, Sweden (2007)
5. Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.2. INRIA (2009). `http://coq.inria.fr/`
6. Licata, D.: Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute. `http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant` (2011)
7. Kokke, P., Swierstra, W.: Auto in Agda. In: Hinze R., Voigtländer J. Mathematics of Program Construction. Lecture Notes in Computer Science, vol 9129, Cham (2015).
8. Agda's Documentation (2017). `https://agda.readthedocs.io/en/latest/`
9. Christiansen, D. and Brady, E.: Elaborator Reflection: Extending Idris in Idris. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16). Nara, Japan (2016).
10. Roundy, D.: Darcs: Distributed version management in haskell. ACM SIGPLAN Workshop on Haskell (2005).
11. Bove, A. and Capretta V.: Modelling general recursion in type theory. Mathematical Structures in Computer Science, 15(4):671–708 (2005).
12. Christiansen, D.: Dependent type providers. In: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming, WGP '13. New York, USA (2005).
13. Altenkirch, T.: Containers in homotopy type theory. Talk at Mathematical Structures of Computation, Lyon (2014).

14. Abbott M., Altenkirch, T. and Ghani, N.: Containers: constructing strictly positive types. Theoretic Computer Science (2005).
15. Mimram, S. and Giusto D. C.: A categorical theory of patches. Electronic Notes in Theoretic Computer Science, 298:283–307 (2013).
16. Stutterheim, J., Swierstra, W. and Swierstra, D.: Forty hours of declarative programming: Teaching Prolog at the Junior College Utrecht. Electronic Proceedings in Theoretical Computer Science, volume 106, pages 50–62, 2013.
17. van der Walt, P. and Swierstra, W.: Engineering proof by reflection in Agda. In: In Ralf Hinze, (ed) Implementation and Application of Functional Languages. Lecture Notes in Computer Science, pages 157–173. Springer Berlin Heidelberg, 2013.
18. van der Walt, P.: Reflection in Agda. Master's thesis, Department of Computer Science, Utrecht University, Utrecht, Netherlands (2012).
19. Vivekanandan, P.: A Homotopical Approach to Cryptography. Workshop on Foundations of Computer Security (FCS'18), University of Oxford, UK (2018).
20. Christiansen, D.: Practical Reflection and Metaprogramming for Dependent Types. IT University of Copenhagen (2016).
21. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and de Moura, L.: A Metaprogramming Framework for Formal Verification. Proceedings of the ACM on Programming Languages. ICFP New York, NY, USA (2017).
22. Flatt, M., Culpepper, R., Darais, D., Findler R. B.: Macros that Work Together: Compile-time bindings, partial expansion, and definition contexts. Journal of Functional Programming (2012).
23. Coquand, T., Huber, S., and Mörtberg, A.: On Higher Inductive Types in Cubical Type Theory. arXiv:1802.01170 (2018).
24. Angiuli, C., Harper, R., and Wilson, T.: Computational Higher-dimensional Type Theory. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17), Paris, France (2017).
25. Cohen, C., Coquand, T., Huber, S., and Mörtberg, A.: Cubical Type Theory: a constructive interpretation of the univalence axiom. 21st International Conference on Types for Proofs and Programs (2015).
26. Norell, U.: Towards a practical programming language based on dependent type theory. Chalmers University of Technology, Sweden (2007).
27. Cockx, J., Devriese, D., and Piessens, F.: Pattern Matching Without K. Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP'14), Gothenburg, Sweden (2014).
28. Cockx, J. and Abel, A.: Sprinkles of Extensionality for Your Vanilla Type Theory. 22nd International Conference on Types for Proofs and Programs (TYPES 2016).
29. Dybjer, P.: Inductive families. Formal Aspects of Computing (1994). `https://doi.org/10.1007/BF01211308`
30. Basold, H., Geuvers, H. and van der Weide, N.: Higher Inductive Types in Programming. Journal of Universal Computer Science, vol. 23, no. 1 (2017).