Tactic for Inductive Proofs

Part II of the Tools and Tactics for Idris project

Ahmad Salim Al-Sibahi (asal@itu.dk)

IT University of Copenhagen
Supervisors:
David Raymond Christiansen (drc@itu.dk)
Dr. Peter Sestoft (sestoft@itu.dk)

December 15, 2013

Idris is a dependently-typed programming language with support for tactics-based theorem proving. This report outlines my work on understanding what dependently typed languages are, how it is possible to do proves using tactics and finally implement a tactic that allows proofs by induction myself. The induction tactic is based on eliminators, which represent a logic-based approach to computation using an inductive data-type with some similarities to pattern matching in the programming world.

The work resulted in an algorithm for generating eliminators which supported all various kinds of inductive data-structures, except inductive-recursive ones. An induction tactic based on the eliminators was created successfully, albeit with minor limitations that should be easily solvable in the future.

The following pull requests signifies my work on eliminators and the induction tactic: `https://github.com/idris-lang/Idris-dev/pull/673` and `https://github.com/idris-lang/Idris-dev/pull/713`.

1 Introduction

Today, functional languages are becoming increasingly popular for use in the enterprise world [1], and some traditional languages like C# and Java are getting more and more functional features [2][3]. Functional programming languages allow writing elegant and concise programs using concepts such as higher-order functions, inductive data-types and pattern matching. Additionally, features such as purity and parametric polymorphism enable the type system to provide free theorems about functions, and allow the programmer to reason more logically about a given program [4].

Yet, there are still limitations of what these type systems provide in terms of correctness and tool support. For example, the programmer cannot ensure that two lists have equal lengths before doing a zip operation.

Dependently typed programming [5] tries to mitigate these limitations by allowing types to be indexed by values, and provides no distinction between terms and types. Furthermore, because of the way many dependently typed languages are designed, the programmer can prove theorems using intuitionistic logic in accordance with the Curry-Howard isomorphism[6].

Idris is a general-purpose programming language designed by Edwin Brady with support for dependent types [7]. In addition to a dependent type system, Idris features so-called tactics which allow an interactive style of theorem proving similar to Coq [8].

This report will be structured as follows. In Section 2 I will investigate and explain what dependent types are. Section 3 will discuss what tactics are, and how to use them. My work on the induction tactic will be outlined in Section 4 and evaluated in Section 5. Finally, I will conclude in Section 6.

2 Programming with Dependent Types

2.1 Dependent Types in Idris

As mentioned in Section 1, dependent types allow types to not only dependent on other types but also on ordinary values *e.g.* natural numbers. The power is apparent when it is possible to define data structures and functions that ensures, at compile-time, that only specific input can be provided and can provide programmers with interesting theorems about the given programs. In that way, the types represent proofs that particular properties hold for the written program.

```
module DependentTypes

data Balanced1 : Nat -> Nat -> Type where
  RightLeaning : {n : Nat} -> Balanced1 n    (S n)
  Balance      : {n : Nat} -> Balanced1 n     n
  LeftLeaning  : {n : Nat} -> Balanced1 (S n) n

data AVL : Nat -> Type -> Type where
  Leaf : {a : Type} -> AVL Z a
  Node : {a : Type} -> {nl : Nat} -> {nr : Nat}
         -> AVL nl a -> a -> AVL nr a -> Balanced1 nl nr
         -> AVL (S (max nl nr)) a
```

Figure 1: AVL Tree

Figure 1 shows a definition of an AVL tree in Idris. AVL trees [9] are restructured on insertion and removal when the difference between the two child trees is larger than one. Thus, it is ensured that many operations stay in logarithmic time by limiting the depth of the final tree. Using the power of dependent types the definition of AVL trees in the figure ensures that only balanced trees can be constructed. This is mainly done by the following two properties:

- The AVL tree is indexed by its depth which is defined as '`0`' for leafs and '`1+max(`$\mathtt{n}_l$`, `$\mathtt{n}_r$`)`' for intermediate nodes, where '$\mathtt{n}_l$' and '$\mathtt{n}_r$' are the depth of the left and right sub-trees.
- When trying to construct a new tree-node with two sub-trees it requires a *proof object*, '$\mathtt{Balanced}_1$', which ensures that the difference in depth is at most 1.

The way '$\mathtt{Balanced}_1$' ensures there is at most a difference of 1 between its two arguments is by providing no constructors that can create a value of '$\mathtt{Balanced}_1$' with a larger difference. So even if a type like '$\mathtt{Balanced}_1$ `3 5`' is referred to, a value of this type cannot be constructed and the program relying on this value wouldn't compile.

```
testAVLCorrect : AVL 2 Nat
testAVLCorrect = Node (Node Leaf 2 Leaf Balance) 1 Leaf LeftLeaning

testAVLIncorrect : AVL 3 Nat
testAVLIncorrect = Node
                     (Node (Node Leaf 3 Leaf Balance) 2 Leaf LeftLeaning)
                     1 Leaf ?no_value_can_be_here
```

Figure 2: AVL Tree Instances

Figure 2 shows a correct and an incorrect (non-balanced) instance of the AVL tree. The '`testAVLCorrect`' value is accepted by the compiler since the depth of the left sub-tree is only 1 larger than the depth of the right sub-tree. In contrast '`testAVLIncorrect`' is rejected by the compiler, because the difference is 2 and it is not possible to create a value of the '$\mathtt{Balanced}_1$' proof object.

2.2 Propositional Equality

Since Idris permits restricting how objects are constructed by predicating types on values, it opens up the possibility of defining a type for reasoning about propositional equality (see Figure 3) [10].

```
data (=) : a -> b -> Type where
  refl : {x : a} -> x = x
```

Figure 3: Heterogeneous Propositional Equality

While the propositional equality type permits the programmer to refer to objects of different types as being equal, the only way to actually construct a value of the type is using '`refl`' which only accepts a value of a single type. Therefore, propositional equality allows the programmer to state interesting theorems about the program which then must be proved later.

```
cong : {a : Type} -> {b : Type} -> {x : a} -> {y : a} -> {p : a -> b}
        -> x = y -> p x = p y
cong refl = refl

npluso : {n : Nat} -> n + Z = n
npluso {n = Z}    = refl
npluso {n = S n}  = cong (npluso {n})
```

Figure 4: Proof of n + 0 = n

Figure 4 shows a proof of that adding a zero on the left-hand side of a natural number yields the same natural number. While this property might seem obvious for a human reader, if the addition was defined by recursion on the first argument, the type checker would not use this equality while comparing types if no further proof was provided.

The proof itself is straightforward induction on '`n`', here represented as a recursive function. In the base case '`0`', it is obvious for the type checker than it holds because everything can be reduced to the same form (it transforms '`0 + 0 = 0`' to '`0 = 0`'). The inductive step is to simply apply the congruence relation on the inductive hypothesis to transform '`n + 0 = n`' to '`S n + 0 = S n`', which is the expected type. The congruence relation utilises the purity and injectivity of functions and specify that given two values which are proven equal in a domain, they are also equal in the input function's co-domain. Propositional equality is a key concept in dependently typed languages, since it allows the programmer to prove theorems about types that wouldn't simply have been accepted by the type checker.

2.3 Formation, Introduction, Elimination

Type declarations, constructors and pattern matching are in many ways programming-oriented terms. A more logic-oriented way of thinking of types and constructors is to talk about formation, introduction and elimination rules.

$$\frac{}{Nat : *}$$

Figure 5: Formation rule for natural numbers

Formation rules are used to define what valid ways of constructing a type there are, i.e. what type parameters and indices are required in order to create such type, analogous to the signature of a data declaration in Idris. Figure 5 shows the formation rule for natural numbers, where it is declared as an ordinary type without any type arguments ('$*$' is the type of types).

$$\frac{}{\mathbf{Z} : Nat} \qquad\qquad \frac{n : Nat}{\mathbf{S}\, n : Nat}$$

Figure 6: Introduction rules for natural numbers

Introduction rules are analogous to constructor declarations, in that they define how it is possible to create a type in the current context. Figure 6 shows an inductive definition for natural numbers, where '`0`' is defined to be a natural

number and then given any number the successor of that number is defined to be a natural number.

$$\frac{P : \ Nat \to * \qquad P_{\mathbf{Z}} : P\,\mathbf{Z} \qquad P_{\mathbf{S}} : n : Nat\,, P\,n \vdash P\ (\mathbf{S}\,n) \qquad m : Nat}{ind_{Nat}\,(P, P_{\mathbf{Z}}, P_{\mathbf{S}}, m) : P\,m}$$

Figure 7: Elimination rule for natural numbers

The elimination rule is probably the most important rule in this collection of rule, since they define how to use the types to prove theorems. While eliminators exist in the programming world, the most common way of performing computation using a data type is by pattern matching. It will however become apparent that if it is necessary to prove something about all possible values of a particular type, eliminators provide a convenient way of doing that using induction.

Generally, the elimination rule of a type works by requiring the motive to proof as an input, in addition to "messages" which represent case analyses on each possible constructor and a "scrutinee" argument of the type which is used to determine the result. The messages are structured such that they match the arguments of a particular constructor. If there are any inductive arguments in a constructor, the inductive hypothesis for each of the arguments must additionally be provided to the relevant message. Figure 7 shows the elimination rule for natural numbers where '`P`' is the motive to be proven, '$\texttt{P}_Z$' and '$\texttt{P}_S$' are the messages for the zero and successor case and '`m`' is the scrutinee.

There are exception on how the elimination look for types with indices and a types like '$\bot$' and propositional equality. The rules for these types are the following:

Types with indices The indices must be a part of the motive to proof, since they can vary depending on what constructor is used (see Figure 8 for elimination rule of length-indexed lists).

Types without constructors For '$\bot$' and similar types, the elimination rule is simplified since it is not possible to construct a value of such type. Therefore, if these types are in a premise of a proof one could proof anything (principle of explosion).

Type for propositional equality While the type presented for propositional equality can relate values of different types at type level, it was only possible to construct an object if both sides of the equality had the same type. Similarly, the eliminator for propositional equality should only eliminate over values of the same type; otherwisem the rule wouldn't allow substitution of two values of the same type and be significantly less useful in practice [10].

$$\frac{\begin{array}{c}\alpha : * \qquad P : (n : Nat) \to Vect\, n\, \alpha \to * \qquad P_{\mathbf{Nil}} : P\, 0\, \mathbf{Nil} \\ P_{::} : n : Nat, x : \alpha, xs : Vect\, n\, \alpha, P\, n\, xs \vdash P\ (\mathbf{S}\, n)\ (x :: xs) \\ m : Nat \\ ys : Vect\, m\, \alpha\end{array}}{ind_{Vect}\, (P, P_{\mathbf{Nil}}, P_{::}, m, ys) : P\, m\, ys}$$

Figure 8: Elimination rule for length-indexed lists

2.4 Elaboration and Unification

When talking about the compilation and typing processes in Idris, I am usually referring to the relevant part of the "elaboration" process [7].

Elaboration is the process of translating the high-level code of Idris to a core type theory called TT. The main idea of elaboration is to simplify constructs like type-classes, implicit arguments and various pattern matching clauses (like where-blocks) into dependent records, explicit arguments and case-trees respectively.

The elaboration process is done using a combination of tactics, and tries to achieve it goal by three means: type checking, normalisation and unification.

1. Type checking ensures that values belong correctly to the types specified, *e.g.* that '`S Z`' is of type '`Nat`'.

2. Normalisation tries do reduce a term to its simplified form, and is especially useful in Idris because terms can be index types. This is done to ensure that *e.g.* '`0+n`' is reduced to '`n`', so that the programmer does not need to prove things that are definitionally equal.

3. Unification is the process of resolving what values are valid in unfilled parts of a program *i.e.* implicit arguments and arguments denoted with '`_`', in order to get a complete term. For example, given the value '`[1,2]`' of a type '`Vect n a`', the type parameter '`a`' must be unified to '`Integer`' and the length index '`n`' must be unified to 2.

3 Tactics-based Theorem Proving

In order to aid the programmer when proving properties about a particular program that is written, Idris support tactic-scripts. A tactic is a structured readily-proven instruction that changes the context (the structure of goals and premises) of a proof, and can be combined in order to achieve the desired goal.

```
reflexiveEq : {n : Nat} -> n = n
reflexiveEq = ?reflexiveEqProof

symmetricEq : {n, m : Nat} -> (H : n = m) -> m = n
symmetricEq = ?symmetricEqProof

transitiveEq : {n, m, o : Nat} -> (H1 : n = m) -> (H2 : m = o) -> n = o
transitiveEq = ?transitiveEqProof
```

Figure 9: Theorems to Proof

Figure 9 states three example theorems about equality of natural numbers: reflexivity, symmetry and transitivity. The theorems are represented at the type level using propositional equality, and at the value level each term is assigned a meta-variable. Meta-variables specify that the proofs will be done separately, possibly using the tactic support of Idris.

```
reflexiveEqProof = proof
  intro n
  trivial

symmetricEqProof = proof
  intro n, m, H
  rewrite H
  trivial

transitiveEqProof = proof
  intro n, m, o, H1, H2
  rewrite (sym H1)
  exact H2
```

Figure 10: Tactic-based proofs for Theorems

The proofs for the theorems stated is found in Figure 10. Reflexivity is simple to proof: first, the '`intro`' tactic is used such that the parameter '`n`' is added to the premises and then the tactic '`trivial`' is used. '`trivial`' tries to solve the goal by either using definitional equality or by matching the goal against the premises. As such, '`trivial`' can be seen as the simplest tactic that can actually finish a proof, since it refines using things already known.

The proof for symmetry is marginally more complicated as it uses the '`rewrite`' tactic. '`rewrite`' accepts an equation, and replaces all instances of the expression on the right hand side of the equation in the goal with the expression on the left hand side. In the case for symmetry it had the replaced the '`m = n`' goal with '`n = n`'. After rewriting, the goal is true by definitional equality and '`trivial`' can simply be used.

The final example of proving transitivity, uses yet another two tactics '`sym`' and '`exact`'. '`sym`' is a tactic that returns the symmetric version of an equation, and is often useful in combination with '`rewrite`', e.g. if the programmer wants to rewrite in the other direction. In the example the '`rewrite (sym H1)`' part replaces '`n = o`' with '`m = o`'. Since '`m = o`' exist in the context the '`exact`' tactic finishes the proof by specifying that the goal is already proven as '`H1`'. While it is possible to use '`trivial`' instead of '`exact`' in this case, '`exact`' provides more information on how the goal was reached.

The tactics shown was just a small subset of the tactics available in Idris, and many more exist such as '`compute`' which normalises current term in the context, and '`try`' which iterates through a tactic sequence until one succeeds.

4 Solution

The primary objective of this project is to create an induction tactic for Idris. This is done in two steps: generating the eliminator for a given data-type, and using that eliminator on the goal in a proof context to do inductive proofs.

4.1 Generating Eliminators

```
1  procedure ElaborateEliminator(name, type, constructors)
2      (parameters, indices) ← TypeArguments(type)
3      motiveType ← ConstructMotiveType(name, parameters, indicies)
4      messageTypes ← {}
5      for all constructor ∈ constructors do
6          messageTypes ↞ ConstructMessageType(constructor, parameters)
7      scrutineeIndicies ← ConstructScrutineeIndicies(indicies)
8      scrutineeType ← ConstructScrutineeType(parameters, scrutineeIndicies)
9      eliminatorType ← ConstructEliminatorType(parameters,
   motiveType, messageTypes, scrutineeIndicies, scrutineeType)
10     eliminatorName ← EliminatorName(type)
11     ElaborateTypeDeclaration(eliminatorName, eliminatorType)
12     if messageTypes ≠ {} then
13         clauses ← ConstructEliminatorClauses(constructors, eliminatorType)
14         ElaborateClauses(eliminatorName, clauses)
```

Figure 11: Algorithm for Elaboration of Eliminators

A simplified pseudo-code version of the algorithm for generating the eliminators of a type is shown in Figure 11 (the original code is written in imperative-style Haskell with do-notation, see Appendix B). The algorithm takes three arguments which are the name, type and constructors of the type which needs the generation of eliminators.

The first step of the algorithm is to differentiate parameters from indices in the type arguments, because Idris doesn't have a built-in syntactic distinction. The distinction between parameters and indices is important, since parameters must be generalised across the eliminator type, while indicies can vary in the motive and messages, and as a consequence must be passed along as input alongside to the term to prove. The algorithm then tries to construct the correct type for the motive, which requires all the indices as input in addition to the variable which must be proven, *e.g.* for length-indexed lists the motive type would be '`(n : Nat) → (xs : Vect n a) → Type`' (note that the type parameter '`a`' is parametrised over the whole type).

Following the creation of the motive, the algorithm tries to construct a fitting message type for each constructor. For a particular constructor, the message type contains all its arguments, and must additionally contain a proof of the motive for each recursive argument (induction hypothesis). The result type for a message is the application of the motive to the constructor given its input arguments. For types with indicies the indices must be retrieved for each proof of the motive, *i.e.* in the types for inductive arguments for recursive applications of the motive and the return type of the constructor for the final result. For example, the motive for '`::`' is '`(n : Nat) → (x : a) → (xs : Vect n a) → P n xs → P (S n) (x :: xs)`'.

```
elim_vect : (a : Type) -> (P : (n : Nat) -> (xs : Vect n a) -> Type)
          -> P Z Nil
          -> ((n : Nat) -> (x : a) -> (xs : Vect n a) -> P n xs -> P (S n) (x :: xs))
          -> (m : Nat) -> (ys : Vect m a)
          -> (P m ys)
elim_vect a p pnil pcons Z     Nil     = pnil
elim_vect a p pnil pcons (S n) (x::xs) = pcons n x xs
                                         (elim_vect a p pnil pcons n xs)
```

Figure 12: Generated Eliminator for Length-indexed Lists

Finally, the complete eliminator type must be assembled, by generalising the parameters, the motive type, the type of messages and the scrutinee indices and scrutinee type over the motive application on the scrutinee, *e.g.* lines 1–5 of Figure 12 shows the final type for the eliminator of length-indexed lists. After constructing the eliminator type, it is elaborated using the Idris built-in method for elaborating type declarations.

After finding the correct type for the eliminator, constructing the clauses is a relatively simple task. If there are no constructors for a type nothing is further done, and if there are constructors then a clause is generated for each constructor.

The left hand side of the clause for each constructor differ only by the patterns the scrutinee and related indices are matched to. The corresponding right hand side for a constructor contains a call to the relevant message with the bound arguments, and for each induction hypothesis required, a recursive application of the eliminator is used. The eliminator clauses are then collectively elaborated using Idris built-in method for elaborating function declarations. Figure 12 line 6–8, shows the eliminator clauses for length-indexed lists.

4.1.1 Limitations of the Generation Algorithm

A current limitation of the algorithm for generating eliminators is that it does not support inductive-recursive data structures [11]. The eliminator of an inductive-recursive data structure is dependent on finding a fix-point of the type definition and the dependent recursive function. This requires a separate non-trivial analysis on the data declaration, which can be considered future work. When these fix-point definitions are found, simple rules like the ones presented in this algorithm can be applied and the eliminator can be generated.

4.1.2 The Eliminator Type Class

Because eliminators could be useful in other circumstances than in the induction tactic, it could be useful to allow access to them using a standardized interface. One way this could be allowed is to define a general type class for eliminators such as the one shown in Figure 13.

```
class Elim e where
  elimTy : Type
  elim   : elimTy
```

Figure 13: Type class for Eliminators

It seems however that this kind of type class definitions, where the class

arguments are left unused, is currently not supported in Idris. Until this get supported, a special keyword '`elim_for`' is provided to access the generated eliminators.

4.2 The Induction Tactic

The induction tactic takes a variable name as input and proceeds in two phases: extracting the motive from the current goal, and applying the eliminator on the motive possibly creating new goals. Figure 14 shows the pseudocode for the induction tactic algorithm (see Appendix C for actual implementation).

```
1  procedure INDUCTION(name)
2      goal ← GOAL()
3      (termValue, termType) ← CHECK(name)
4      NORMALISE(termType)
5      (typeName, typeArguments) ← UNAPPLY(termType)
6      eliminatorName ← ELIMINATORNAME(typeName)
7      eliminatorType ← LOOKUPTYPE(eliminatorName)
8      parameters ← PARAMETERS(typeArguments)
9      indicies ← INDICIES(typeArguments)
10     (parameterTypes, motiveType, messageTypes, scrutineeIndices, scrutineeType) ←
   ARGUMENTS(eliminatorType)
11     motive ← ABSTRACTMOTIVE(motiveType, termValue, indicies, goal)
12     holes ← MAKEHOLES(messageTypes)
13     REMOVEHOLE(goal)
14     res ← ELIMINATEMOTIVE(parameters, motive, holes, indicies, termValue)
15     return SPECIALISE(res)
```

Figure 14: Induction Tactic

The algorithm starts by retrieving the relevant term value and type from the context, and then normalising the type. The next steps are to retrieve the eliminator from context and decomposing it into its various parts (parameters, motive type, etc.). The most important part of the tactic is the abstraction of the motive from the current goal. This is done by abstracting the term to proof and its indices from its goal as arguments to an anonymous function that conforms to the motive type in the eliminator. For example, given the goal '`n = length xs`' where '`xs : Vect n a`', the algorithm abstract all instances of '`xs`' and '`n`' such that the motive is 'λ `m, ys` $\Rightarrow$ `m = length ys`'. The algorithm proceeds by creating new holes for each of the messages, in order to create the goals necessary to proof the motive (*i.e.* the inductive steps) and removes the current goal from the context. Finally the eliminator is called with the parameters, the motive, the generated holes and the extracted indices and value from the old goal. In order to aid proving, the result is specialised before being returned, *i.e.* lambda arguments are normalised where possible.

4.2.1 Current State of the Induction Tactic

Due to time constraints, the induction tactic was only implemented for types without arguments (it is however still possible to do inductive proofs by directly

using the eliminators). The limitation is mainly due to the current lack of distinction between parameters and indices at TT level, and some additional work is needed to propagate that information from Idris. Nonetheless, the algorithm presented above is generalised for all cases and when the type arguments can be differentiated it should be relatively easy to make it work for all data types with eliminators.

5 Evaluation

To evaluate the generation of eliminators and the induction tactic, a test file (see Appendix A) was created with various interesting data-structures and some inductive proofs. Additionally, the elimination generation algorithm was tried on all inductive data-types in the standard library.

5.1 Evaluating Elimination Generation

In the general case, it seems that the elimination generation works correctly for various kinds of inductive data structures. This has been confirmed by the following test cases:

1. Basic non-inductive data types such as unit, boolean and pairs
2. Simple inductive types such as natural numbers
3. Inductive types with both simple and complex type arguments, such as lists and binary trees
4. Polymorphic recursive types like finger trees
5. Mutually inductive data types such as odd-even proof objects

However, during the test on the standard library, issues were discovered regarding type declarations with implicit arguments that were not handled correctly. Since implicit arguments do not differ semantically from explicit arguments in the eliminator, it should be relatively simple to correct this issue. However, due to time constraints, this was left as future work.

5.2 Inductive Proofs

To test if the eliminators and induction tactic worked as indented various inductive proofs were performed. For types supported by the induction tactics like natural numbers, the proofs was carried exclusively using tactic scripts.

For other data types such as lists and binary trees, the eliminators were written manually and each of the inductive steps was left as a hole. In other words, the induction tactic was simulated manually using the eliminators.

```
ELIMTest.npluso_proof = proof
  intros
  induction n
  trivial
  intros
  compute
  rewrite iht__0
  trivial
```

Figure 15: Proof for $n + 0 = n$ using the induction tactic

Figure 15 shows the proof for '`n + 0 = 0`' again, this time done using the induction tactic. Because the induction tactic creates two branches for natural numbers, it can be observed that the '`trivial`' tactic is applied twice in the proof script, when normally it would finish the proof script. What can also be observed in the proof script is that it was possible to rewrite using the induction hypothesis, which is necessary if the step case has to be proven.

The induction tactic was used for proving common properties on natural numbers, like plus commutativity and associativity, with good results. Similarly, the eliminators worked perfectly where the induction tactic was not yet implemented.

6 Conclusion

In this report I had presented an algorithm for generating eliminators for inductive data types in Idris. I conclude that this was generally achieved for both inductive and inductive-inductive data types, regarding of complexity. Additionally, with simple modification to how implicits are handled, it should be possible to correctly generate eliminators for most types in the standard library except for inductive-recursive cases.

I also conclude that the generated eliminators were successfully used to create a tactic for inductive proofs. While the current implementation was lacking support for types with arguments, only small modifications were necessary to support all data structures which had eliminators.

References

[1] Neil Ford: Functional thinking: Why functional programming is on the rise. `http://www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf` (2013)

[2] Oracle: Jsr 335 project lambda. `http://openjdk.java.net/projects/lambda/` Acessed 2013-12-15.

[3] Microsoft: Linq query expressions. `http://msdn.microsoft.com/en-us/library/bb397676.aspx` Acessed 2013-12-15.

[4] Wadler, P.: Theorems for free! In: Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM (1989) 347–359

[5] The Univalent Foundations Program: Chapter 1 Type Theory. In: Homotopy Type Theory. The Univalent Foundations Program (2013)

[6] McKinna, J.: Why dependent types matter. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '06, New York, NY, USA, ACM (2006) 1–1

[7] Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23** (9 2013) 552–593

[8] The Coq Development Team: The Coq Proof Assistant Reference Manual. (2013)

[9] Goodrich, M.T., Tamassia, R.: Chapter 3.2 AVL Trees. In: Algorithm Design. John Wiley & Sons, Inc. (2012)

[10] McBride, C.: Dependently Typed Functional Programs and their Proofs. PhD thesis, University of Edinburgh (1999) Chapter 5.1.3 John Major Equality. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

[11] Dybjer, P., Setzer, A.: Induction–recursion and initial algebras. Annals of Pure and Applied Logic **124**(1) (2003) 1–47

A Induction Test File

```
module ELIMTest

%elim data  Unit : Type where
  MkUnit : Unit

%elim data  Boolean : Type where
  False : Boolean
  True  : Boolean

%elim data Pair : Type -> Type -> Type where
  MkPair : a -> b -> Pair a b

%elim data  TNat : Type where
  TO : TNat
  TS : TNat -> TNat

infixl 8 +++

(+++) : TNat -> TNat -> TNat
TO +++ m     = m
(TS n) +++ m = TS (n +++ m)

%elim data TList : Type -> Type where
  TNil : TList a
  TCons : a -> TList a -> TList a

infixl 8 @@@
(@@@) : TList a -> TList a -> TList a
TNil          @@@ ys = ys
(TCons x xs)  @@@ ys = TCons x (xs @@@ ys)

%elim data TVect : TNat -> Type -> Type where
  VNil : TVect TO a
  VCons : a -> TVect n a -> TVect (TS n) a

length : (TVect n a) -> TNat
length VNil         = TO
length (VCons x xs) = TS (length xs)

length_proof : (n : TNat) -> (vs : TVect n a) -> (n = length vs)
length_proof {a} n vs = elim_for ELIMTest.TVect a
                         (\m, xs => m = length xs)
                         ?nilCase
                         ?consCase
                         n
                         vs

n_plus_O : (n : TNat) -> (n +++ TO = n)
n_plus_O n = ?npluso_proof

n_plus_S_m : (n : TNat) -> (m : TNat) -> (n +++ TS m = TS (n +++ m))
n_plus_S_m n m = ?nplusSm_proof

plus_commutes : (n : TNat) -> (m : TNat) -> (n +++ m = m +++ n)
plus_commutes n m = ?pluscommutes_proof

%elim data BinaryTree : Type -> Type where
  BLeaf : BinaryTree a
  BNode : BinaryTree a -> a -> BinaryTree a -> BinaryTree a

%elim data WeightedBinaryTree : TNat -> Type -> Type where
  WLeaf : WeightedBinaryTree TO a
  WNode : WeightedBinaryTree n a -> a -> WeightedBinaryTree m a
          -> WeightedBinaryTree (n +++ m +++ TS TO) a

%elim data Node : Type -> Type where
  Node2 : a -> a -> Node a
  Node3 : a -> a -> a -> Node a
```

Figure 16: Various Data-types and Properties for Testing Eliminator Generation and Induction Tactic

```
%elim data FingerTree : Type -> Type where
  Empty : FingerTree a
  Single : a -> FingerTree a
  Deep : List a -> FingerTree (Node a) -> List a -> FingerTree a

mutual
  %elim data Even : TNat -> Type where
    evenZero : Even TO
    evenSuc  : Odd n -> Even (TS n)

  %elim data Odd : TNat -> Type where
    oddSuc   : Even n -> Odd (TS n)

mutual
  data FList : Type -> Type where
    FNil  : FList a
    FCons : (x : a) -> (l : FList a) -> {isFresh : fresh x l} -> FList a

  fresh : a -> (FList a) -> Type
  fresh x FNil           = ()
  fresh x (FCons y xs) = ((x = y) -> _|_, fresh x xs)

mirror : BinaryTree a -> BinaryTree a
mirror BLeaf = BLeaf
mirror (BNode l v r) = BNode (mirror r) v (mirror l)

mirror_proof : {a : Type} -> (b : BinaryTree a) -> (b = mirror (mirror b))
mirror_proof {a} b = elim_for ELIMTest.BinaryTree a
                        (\bb => bb = mirror (mirror bb))
                        ?mirror_leaf_case ?mirror_node_case b
```

Figure 17: Various Data-types and Properties for Testing Eliminator Generation and Induction Tactic, cont.

```
ELIMTest.pluscommutes_proof = proof
  intros
  induction n
  compute
  rewrite (n_plus_O m)
  trivial
  intros
  compute
  rewrite (sym iht__0)
  rewrite (n_plus_S_m m t__0)
  trivial


ELIMTest.nplusSm_proof = proof
  intros
  induction n
  compute
  trivial
  intros
  compute
  rewrite iht__0
  trivial


ELIMTest.npluso_proof = proof
  intros
  induction n
  trivial
  intros
  compute
  rewrite iht__0
  trivial


ELIMTest.mirror_node_case = proof
  compute
  intros
  rewrite ihb__0
  rewrite ihb__1
  trivial


ELIMTest.mirror_leaf_case = proof
  compute
  intros
  trivial

ELIMTest.consCase = proof
  compute
  intros
  rewrite iht__2
  trivial


ELIMTest.nilCase = proof
  compute
  intros
  trivial
```

Figure 18: Various Data-types and Properties for Testing Eliminator Generation and Induction Tactic, cont., cont.

B Eliminator Elaboration

```
elabEliminator :: [Int] -> Name -> PTerm -> [(String, Name, PTerm, FC)] -> ElabInfo -> EliminatorState ()
elabEliminator paramPos n ty cons info = do
  elimLog $ "Elaborating␣eliminator"
  let (cnstrs, _) = splitPi ty
  let (splittedTy@(pms, idxs)) = splitPms cnstrs
  generalParams <- namePis False pms
  motiveIdxs    <- namePis False idxs
  let motive = mkMotive n paramPos generalParams motiveIdxs
  consTerms <- mapM (\(c@(_,cnm,_,_)) -> do
                              name <- freshName $ "elim_" ++ simpleName cnm
                              consTerm <- extractConsTerm c generalParams
                              return (name, expl, consTerm)) cons
  scrutineeIdxs <- namePis False idxs
  let motiveConstr = [(motiveName, expl, motive)]
  let scrutinee = (scrutineeName, expl, applyCons n (interlievePos paramPos generalParams scrutineeIdxs 0))
  let eliminatorTy = piConstr (generalParams ++ motiveConstr ++ consTerms ++ scrutineeIdxs ++ [scrutinee])
        (applyMotive (map (\(n,_,_) -> PRef elimFC n) scrutineeIdxs) (PRef elimFC scrutineeName))
  let eliminatorTyDecl = PTy (show n) defaultSyntax elimFC [TotalFn] elimDeclName eliminatorTy
  let clauseConsElimArgs = map getPiName consTerms
  let clauseGeneralArgs' = map getPiName generalParams ++ [motiveName] ++ clauseConsElimArgs
  let clauseGeneralArgs  = map (\arg -> pexp (PRef elimFC arg)) clauseGeneralArgs'
  let elimSig = "——␣eliminator␣signature:␣" ++ showImp Nothing True True eliminatorTy
  eliminatorClauses <- mapM (\(cns, cnsElim) -> generateEliminatorClauses cns cnsElim clauseGeneralArgs generalParams)
    (zip cons clauseConsElimArgs)
  let eliminatorDef = PClauses emptyFC [TotalFn] elimDeclName eliminatorClauses
  elimLog $ "——␣eliminator␣definition:␣" ++ showDeclImp True eliminatorDef
  Control.Monad.State.lift $ idrisCatch (elabDecl EAll info eliminatorTyDecl) (\err -> return ())
  -- Do not elaborate clauses if there aren't any
  case eliminatorClauses of
    [] -> return ()
    _  -> Control.Monad.State.lift $ idrisCatch (elabDecl EAll info eliminatorDef) (\err -> return ())
  where elimLog :: String -> EliminatorState ()
        elimLog s = Control.Monad.State.lift (logLvl 2 s)

        elimFC :: FC
        elimFC = fileFC "(eliminator)"

        elimDeclName :: Name
        elimDeclName = SN $ ElimN n

        applyNS :: Name -> [String] -> Name
        applyNS n []  = n
        applyNS n ns  = NS n ns

        splitPi :: PTerm -> ([(Name, Plicity, PTerm)], PTerm)
        splitPi = splitPi' []
          where splitPi' :: [(Name, Plicity, PTerm)] -> PTerm -> ([(Name, Plicity, PTerm)], PTerm)
                splitPi' acc (PPi pl n tyl tyr) = splitPi' ((n, pl, tyl):acc) tyr
                splitPi' acc t                  = (reverse acc, t)

        splitPms :: [(Name, Plicity, PTerm)] -> ([(Name, Plicity, PTerm)], [(Name, Plicity, PTerm)])
        splitPms cnstrs = (map fst pms, map fst idxs)
           where (pms, idxs) = partition (\c -> snd c `elem` paramPos) (zip cnstrs [0..])

        isMachineGenerated :: Name -> Bool
        isMachineGenerated (MN _ _) = True
        isMachineGenerated _        = False

        namePis :: Bool -> [(Name, Plicity, PTerm)] -> EliminatorState [(Name, Plicity, PTerm)]
        namePis keepOld pms = do names <- mapM (mkPiName keepOld) pms
                                 let oldNames = map fst names
                                 let params   = map snd names
                                 return $ map (\(n, pl, ty) -> (n, pl, removeParamPis oldNames params ty)) params

        mkPiName :: Bool -> (Name, Plicity, PTerm) -> EliminatorState (Name, (Name, Plicity, PTerm))
        mkPiName keepOld (n, pl, piarg) | not (isMachineGenerated n) && keepOld = do return (n, (n, pl, piarg))
        mkPiName _ (oldName, pl, piarg) =  do name <- freshName $ keyOf piarg
                                              return (oldName, (name, pl, piarg))
```


Figure 19: Implementation of Eliminator Elaboration in Haskell

```
                                        return (oldName, (name, pl, piarg))
  where keyOf :: PTerm -> String
        keyOf (PRef _ name) | isLetter (nameStart name) = (toLower $ nameStart name):"__"
        keyOf (PApp _ tyf _) = keyOf tyf
        keyOf PType = "ty__"
        keyOf _     = "carg__"
        nameStart :: Name -> Char
        nameStart n = nameStart' (simpleName n)
          where nameStart' :: String -> Char
                nameStart' "" = ' '
                nameStart' ns = head ns

simpleName :: Name -> String
simpleName (NS n _) = simpleName n
simpleName (MN i n) = n ++ show i
simpleName n        = show n

nameSpaces :: Name -> [String]
nameSpaces (NS _ ns) = ns
nameSpaces _         = []

freshName :: String -> EliminatorState Name
freshName key = do
  nameMap <- get
  let i = fromMaybe 0 (Map.lookup key nameMap)
  let name = key ++ show i
  put $ Map.insert key (i+1) nameMap
  return (UN name)

scrutineeName :: Name
scrutineeName = UN "scrutinee"

scrutineeArgName :: Name
scrutineeArgName = UN "scrutineeArg"

motiveName :: Name
motiveName = UN "prop"

mkMotive :: Name -> [Int] -> [(Name, Plicity, PTerm)] -> [(Name, Plicity, PTerm)] -> PTerm
mkMotive n paramPos params indicies =
  let scrutineeTy = (scrutineeArgName, expl, applyCons n (interlievePos paramPos params indicies 0))
  in piConstr (indicies ++ [scrutineeTy]) PType

piConstr :: [(Name, Plicity, PTerm)] -> PTerm -> PTerm
piConstr [] ty = ty
piConstr ((n, pl, tyb):tyr) ty = PPi pl n tyb (piConstr tyr ty)

interlievePos :: [Int] -> [a] -> [a] -> Int -> [a]
interlievePos idxs []     l2     i = l2
interlievePos idxs l1     []     i = l1
```

Figure 20: Implementation of Eliminator Elaboration in Haskell, cont.

```
interlievePos idxs l1     (y:ys) i = y:(interlievePos idxs l1 ys (i+1))

replaceParams :: [Int] -> [(Name, Plicity, PTerm)] -> PTerm -> PTerm
replaceParams paramPos params cns =
  let (_, cnsResTy) = splitPi cns
  in case cnsResTy of
       PApp _ _ args ->
        let oldParams = paramNamesOf 0 paramPos args
        in removeParamPis oldParams params cns
       _ -> cns

removeParamPis :: [Name] -> [(Name, Plicity, PTerm)] -> PTerm -> PTerm
removeParamPis oldParams params (PPi pl n tyb tyr) =
  case findIndex (== n) oldParams of
    Nothing -> (PPi pl n (removeParamPis oldParams params tyb) (removeParamPis oldParams params tyr))
    Just i  -> (removeParamPis oldParams params tyr)
removeParamPis oldParams params (PRef _ n) =
  case findIndex (== n) oldParams of
       Nothing -> (PRef elimFC n)
       Just i  -> let (newname,_,_) = params !! i in (PRef elimFC (newname))
removeParamPis oldParams params (PApp _ cns args) =
  PApp elimFC (removeParamPis oldParams params cns) $ replaceParamArgs args
    where replaceParamArgs :: [PArg] -> [PArg]
          replaceParamArgs []          = []
          replaceParamArgs (arg:args)  =
            case extractName (getTm arg) of
              []   -> arg:replaceParamArgs args
              [n]  ->
                case findIndex (== n) oldParams of
                  Nothing -> arg:replaceParamArgs args
                  Just i  -> let (newname,_,_) = params !! i
                    in arg {getTm = PRef elimFC newname}:replaceParamArgs args
removeParamPis oldParams params t = t

paramNamesOf :: Int -> [Int] -> [PArg] -> [Name]
paramNamesOf i paramPos []          = []
paramNamesOf i paramPos (arg:args)  =
  (if i `elem` paramPos then extractName (getTm arg) else [])
    ++ paramNamesOf (i+1) paramPos args

extractName :: PTerm -> [Name]
extractName (PRef _ n) = [n]
extractName _          = []

splitArgPms :: PTerm -> ([PTerm], [PTerm])
splitArgPms (PApp _ f args) = splitArgPms' args
  where splitArgPms' :: [PArg] -> ([PTerm], [PTerm])
        splitArgPms' cnstrs = (map (getTm . fst) pms, map (getTm . fst) idxs)
            where (pms, idxs) = partition (\c -> snd c `elem` paramPos) (zip cnstrs [0..])
splitArgPms _                = ([],[])


implicitIndexes :: (String, Name, PTerm, FC) -> EliminatorState [(Name, Plicity, PTerm)]
implicitIndexes (cns@(doc, cnm, ty, fc)) = do
  i <-  Control.Monad.State.lift getIState
  implargs' <- case lookupCtxt cnm (idris_implicits i) of
    [] -> do fail $ "Error while showing implicits for " ++ show cnm
    [args] -> do return args
    _ -> do fail $ "Ambigous name for " ++ show cnm
  let implargs = mapMaybe convertImplPi implargs'
  let (_, cnsResTy) = splitPi ty
```

Figure 21: Implementation of Eliminator Elaboration in Haskell, cont., cont.

```
          let (_, cnsResTy) = splitPi ty
          case cnsResTy of
             PApp _ _ args ->
              let oldParams = paramNamesOf 0 paramPos args
              in return $ filter (\(n,_,_) -> not (n `elem` oldParams))implargs
            _ -> return implargs

        extractConsTerm :: (String, Name, PTerm, FC) -> [(Name, Plicity, PTerm)] -> EliminatorState PTerm
        extractConsTerm (doc, cnm, ty, fc) generalParameters = do
          let cons' = replaceParams paramPos generalParameters ty
          let (args, resTy) = splitPi cons'
          implidxs <- implicitIndexes (doc, cnm, ty, fc)
          consArgs <- namePis True args
          let recArgs = findRecArgs consArgs
          let recMotives = map applyRecMotive recArgs
          let (_, consIdxs) = splitArgPms resTy
          return $ piConstr (implidxs ++ consArgs ++ recMotives) (applyMotive consIdxs (applyCons cnm consArgs))
            where applyRecMotive :: (Name, Plicity, PTerm) -> (Name, Plicity, PTerm)
                  applyRecMotive (n,_,ty)  = (UN $ "ih" ++ simpleName n, expl, applyMotive idxs (PRef elimFC n))
                      where (_, idxs) = splitArgPms ty

        findRecArgs :: [(Name, Plicity, PTerm)] -> [(Name, Plicity, PTerm)]
        findRecArgs []                            = []
        findRecArgs (ty@(_,_,PRef _ tn):rs)
    | simpleName tn == simpleName n = ty:findRecArgs rs
        findRecArgs (ty@(_,_,PApp _ (PRef _ tn) _):rs) | simpleName tn == simpleName n = ty:findRecArgs rs
        findRecArgs (ty:rs)
    = findRecArgs rs

        applyCons :: Name -> [(Name, Plicity, PTerm)] -> PTerm
        applyCons tn targs = PApp elimFC (PRef elimFC tn) (map convertArg targs)

        convertArg :: (Name, Plicity, PTerm) -> PArg
        convertArg (n, _, _)      = pexp (PRef elimFC n)

        applyMotive :: [PTerm] -> PTerm -> PTerm
        applyMotive idxs t = PApp elimFC (PRef elimFC motiveName) (map pexp idxs ++ [pexp t])

        getPiName :: (Name, Plicity, PTerm) -> Name
        getPiName (name,_,_) = name

        convertImplPi :: PArg -> Maybe (Name, Plicity, PTerm)
        convertImplPi (PImp {getTm = t, pname = n}) = Just (n, expl, t)
        convertImplPi _                             = Nothing

        generateEliminatorClauses :: (String, Name, PTerm, FC) -> Name ->
                  [PArg] -> [(Name, Plicity, PTerm)] -> EliminatorState PClause
        generateEliminatorClauses (doc, cnm, ty, fc)
            cnsElim generalArgs generalParameters = do
          let cons' = replaceParams paramPos generalParameters ty
          let (args, resTy) = splitPi cons'
          i <- Control.Monad.State.lift getIState
          implidxs <- implicitIndexes (doc, cnm, ty, fc)
          let (_, generalIdxs') = splitArgPms resTy
          let generalIdxs = map pexp generalIdxs'
          consArgs <- namePis True args
          let lhsPattern = PApp elimFC (PRef elimFC elimDeclName)
              (generalArgs ++ generalIdxs ++ [pexp $ applyCons cnm consArgs])
          let recArgs = findRecArgs consArgs
          let recElims = map applyRecElim recArgs
          let rhsExpr    = PApp elimFC (PRef elimFC cnsElim)
              (map convertArg implidxs ++ map convertArg consArgs ++ recElims)
          return $ PClause elimFC elimDeclName lhsPattern [] rhsExpr []
            where applyRecElim :: (Name, Plicity, PTerm) -> PArg
                  applyRecElim (constr@(recCnm,_,recTy)) = pexp $ PApp elimFC (PRef elimFC elimDeclName)
                              (generalArgs ++ map pexp idxs ++ [pexp $ PRef elimFC recCnm])
                    where (_, idxs) = splitArgPms recTy
```

Figure 22: Implementation of Eliminator Elaboration in Haskell, cont., cont., cont.

C Induction Tactic

```
induction :: Name -> RunTactic
induction nm ctxt env (Bind x (Hole t) (P _ x' _))  |x == x' = do
  (tmv, tmt) <- lift $ check ctxt env (Var nm)
  let tmt' = normalise ctxt env tmt
  case unApply tmt' of
    (P _ tnm _, pms@[]) -> do
        case lookupTy (SN (ElimN tnm)) ctxt of
          [elimTy] -> do
             let args     = getArgTys elimTy
             let pmargs   = take (length pms) args
             let args'    = drop (length pms) args
             let propTy   = head args'
             let consargs = init $ tail args'
             let scr      = last $ tail args'
             let prop = Bind nm (Lam tmt') t
             let res = substV prop $ bindConsArgs consargs
                  (mkApp (P Ref (SN (ElimN tnm)) (TType (UVal 0))) ([prop] ++ map makeConsArg consargs ++ [tmv]))
             action (\ps -> ps {holes = holes ps \\ [x]})
             mapM_ addConsHole (reverse consargs)
             let res' = forget $ res
             (scv, sct) <- lift $ check ctxt env res'
             let scv' = specialise ctxt env [] scv
             return scv'
          [] -> fail $ "Induction needs an eliminator for " ++ show nm
          xs -> fail $ "Multiple definitions found when searching for the eliminator of " ++ show nm
    (P _ nm _, _) -> fail "Induction not yet supported for types with arguments"
    _ -> fail "Unkown type for induction"
    where scname = MN 0 "scarg"
          makeConsArg (nm, ty) = P Bound nm ty
          bindConsArgs ((nm, ty):args) v = Bind nm (Hole ty) $ bindConsArgs args v
          bindConsArgs [] v = v
          addConsHole (nm, ty) =
            action (\ps -> ps { holes = nm : holes ps })
induction tm ctxt env _ = do fail "Can't do induction here"
```

Figure 23: Implementation of Induction Tactic in Haskell