Tactic for Inductive Proofs

A part 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

1 Introduction

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

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

Dependently-typed programming [3] allow the programmer try to mitigate these limitations by allowing types to be indexed by values, and provide no distinction between types and terms. Furthermore, because of the way many dependently-typed languages are designed, the programmer can do proving of theorems within the realm of intuitionistic logic in accordance with the Curry-Howard isomorphism[4].

Idris is a general-purpose programming language designed by Edwin Brady with support for dependent types [5]. In addition to a dependent type system, Idris features so-called tactics which allow an interactive style of doing theorem proving similar to Coq [6].

The 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 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 we can define data structures and functions which ensure at compile-time that only specific input can be provided at compile time 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 [7] 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 are 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(n$_l$, n$_r$)’ for intermediate nodes where ‘n$_l$’ and ‘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* which ensures that the difference in depth is at most 1 (see ‘Balanced$_1$’).

The data type ‘Balanced$_1$’ ensures there is at most a difference of 1 between its two arguments by providing no constructors that can create a value of ‘Balanced$_1$’ with a larger difference. So even if a type like ‘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 ‘Balanced$_1$’ proof object.

2.2 Propositional Equality

Since Idris permits restricting how objects are constructed bye predicating types on values, it opens up the possibility of defining a type for reasoning about propositional equality (see Figure 3) [8].

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

Figure 3: Propositional Equality

While the propositional equality type permits the programmer to refer to say that objects of different types might be 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 separately 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 theory might seem a bit simple for a human reader, if the addition was defined by recursion on the first argument the type checker would reject the theorem if no further proof was provided.

The proof itself is simple in essence, as it is done by recursion. In the base case (0), it is obvious for the type checker than it holds because everything is a constant and can be reduced to the same form (this it transforms '`0 + 0 = 0`' to '`0 = 0`'). The inductive step is simply apply the congruence relation on the inductive hypothesis to transform 'n+0=n' to 'S n + 0 = S n', which is the expected resulting 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 functions co-domain. Propositional equality is a key concept in Dependently-typed languages, since it allows the programmer to proof theorems about types that wouldn't simply have been accepted by the type checker.

2.3 Formation, Introduction, Elimination

Type declaration, 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 that it is an ordinary type without any parameters ('$*$' 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 integers, where 0 is defined to be a natural number and then given any number the successor of that number is 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 set, since they define how to actually use the types to prove theorems. While eliminators exist in the programming world, the most common way of doing computation using a data type is by pattern matching. However, it will become apparent that if it is necessary to prove something about all possible values of a particular type, eliminators provide a convenient way to do induction proofs.

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 the arguments for the particular constructor are given as arguments and must return a proof of the property on that constructor. If there are any inductive arguments in a constructor, the inductive hypothesis (recursive proof) must additionally be given as argument to the relevant message. Figure 7 shows the elimination rule for natural numbers where '`P`' is the motive to be proven, '`P`$_Z$' and '`P`$_S$' are the messages for the zero and successor case and '`m`' is the scrutinee.

There is however variance on how the elimination look for type with indices and a couple of special types like '$\perp$' and propositional equality. The rules for these types are the following:

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

Types without constructors For '$\perp$' 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 equal 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; otherwise the rule wouldn't allow substitution of two values of the same type [8].

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

Figure 8: Elimination rule for length-indexed lists

2.4 Elaboration and Unification

When talking about the compiling and typing proceses in Idris, I am usually referring to the "elaboration" and "unification" processes in Idris (although there are various other actions in the pipeline) [5].

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 a simplified form, and is especially useful in Idris because terms can be index types. This is done to ensure that *e.g.* '`Z+n`' is reduced to '`n`', so that the programmer does not need to explicitly proof things that are equal by definition (assuming that '+' reduces on the left argument as is done in Idris).

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`' (as that is the type of elements) and the length index '`n`' must be unified to 2 (size of the list).

3 Tactics-based Theorem Proving

In order to aid the programmer when doing proving properties about a particular programmer that is written, Idris support tactic-scripts. A tactic is a structured proven instruction that changes the context (either the goals or premises) of a proof, and that is available in such way that a combination of tactics represent the particular program necessary 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 theorems about equality of natural numbers that needs to be proven: 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. The proof for reflexivity is simple, the '`intro`' tactic is used such that the parameter '`n`' is added to the premises and then the tactic '`trivial`' is used. The '`trivial`' tries to solve the goal by either using definitional equality or by finding a matching expression among 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. The '`rewrite`' accepts an equation, and replaces all instances of the equation right hand side with the equation left hand side in the goal. 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 constructs '`sym`' and '`exact`'. '`sym`' is a construct that exploits symmetry and returns the symmetric version of an equation, and is useful in combination with '`rewrite`' if the programmer wants to replace with the right hand side of an equation instead. 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 my solution 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 in a proof setting by abstracting the current goal as a motive to permit inductive proves.

4.1 Generating Eliminators

```
procedure ElaborateEliminator($name, type, constructors$)
    $(parameters, indices) \leftarrow$ TypeArguments($type$)
    $motiveType \leftarrow$ ConstructMotiveType($name, parameters, indicies$)
    $messageTypes \leftarrow \{\}$
    for all $constructor \in constructors$ do
        $messageTypes \twoheadleftarrow$ ConstructMessageType($constructor, parameters$)
    $scrutineeIndicies \leftarrow$ ConstructScrutineeIndicies($indicies$)
    $scrutineeType \leftarrow$ ConstructScrutineeType($parameters, scrutineeIndicies$)
    $eliminatorType \leftarrow ConstructEliminatorType(parameters, motiveType, messageTypes, scrutineeIndicies, scrutineeType)$
    $eliminatorName \leftarrow$ EliminatorName($type$)
    ElaborateTypeDeclaration($eliminatorName, eliminatorType$)
    if $messageTypes \neq \{\}$ then
        $clauses \leftarrow$ ConstructEliminatorClauses($constructors, eliminatorType$)
        ElaborateClauses($eliminatorName, clauses$)
```

Figure 11: Algorithm for Elaboration of Eliminators

A simplified pseudo-code version of the algorithm for generating the eliminators for a type is shown in Figure 11 (the original code is written in imperative-style Haskell with do-notation). The algorithm takes three arguments, namely 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 unlike other dependently-typed languages doesn't have a built-in 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 also be given as input along with the scrutinee. 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. The message type for a constructor contains all the constructor arguments, and for each inductive argument the recursive application of the motive on that argument must additionally be given as argument. The result type for a message is the application of the motive to the constructor given its input arguments. For types with indicies it should be noted that indices must be retrieved from the relevant parts, *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
3             -> ((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)
6 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 eliminator type must be constructed, 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.* The 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 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.

All left hand sides of clauses for constructors share the first part, namely the parameter, motive and message arguments and the only thing that changes is the pattern matching on the scrutinee and related indicies to check if they match the correct constructor. On the right hand sides the matching messages are called with the constructor arguments, and if there are any recursive application of the motive in the type of message, the eliminator is applied recursively on the matching arguments. 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.

NOTE: Write about limitations regarding induction-recursion **NOTE:** Write something about having an eliminator type class

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 13 shows the pseudocode for the induction tactic algorithm.

```
procedure $\textsc{Induction}(name, goal)$
    $(termValue, termType) \leftarrow \textsc{Check}(name)$
    $\textsc{Normalise}(termType)$
    $(typeName, typeArguments) \leftarrow \textsc{UnApply}(termType)$
    $eliminatorName \leftarrow \textsc{EliminatorName}(typeName)$
    $eliminatorType \leftarrow \textsc{LookupType}(eliminatorName)$
    $parameters \leftarrow \textsc{Parameters}(typeArguments)$
    $indicies \leftarrow \textsc{Indicies}(typeArguments)$
    $(parameterTypes, motiveType, messageTypes, scrutineeIndices, scrutineeType) \leftarrow \textsc{Arguments}(eliminatorType)$
    $motive \leftarrow \textsc{AbstractMotive}(motiveType, termValue, indicies, goal)$
    $holes \leftarrow \textsc{MakeHoles}(messageTypes)$
    $\textsc{RemoveHole}(goal)$
    $res \leftarrow \textsc{EliminateMotive}(parameters, motive, holes, indicies, termValue)$
    return $\textsc{Specialise}(res)$
```

Figure 13: Induction Tactic

The algorithm starts by first retrieving the relevant term value and type from the context, and then normalising the type. The next steps is 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 creating lambda function for the indices and term in the goal that fits the motive type, and replacing all usages of the indices and term to use the names bound in the lambda value. 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 ⇒ m = length ys`'. The algorithm proceeds by creating new holes for each of the messages, in order to create the new goals necessary to proof the motive (*i.e.* the induction steps) and removes the current goal from the context. Finally the eliminator is called with parameters, the motive, the generated holes and the old goals extracted indices and value. In order to aid proving, the result is specialised before being returned, *i.e.* lambda arguments are normalised where possible.

NOTE: Write about limitations of induction tactic

5 Evaluation

6 Conclusion

References

[1] Neil Ford: Functional thinking: Why functional programming is on the rise. IBM DeveloperWorks (2013)

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

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

[4] 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

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

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

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

[8] 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/.