Structural Induction Principles for Functional Programmers

James Caldwell

Department of Computer Science
University of Wyoming
Laramie, WY 82071

User defined recursive types are a fundamental feature of modern functional programming languages like Haskell and the ML family of languages. Properties of programs defined by recursion on the structure of a recursive types are generally proved by structural induction on the type. It is well known in the theorem proving community how to generate a structural induction principle from the type signature. These methods deserve to be better know in the functional programming community. Existing functional programming textbooks gloss over this material. And yet, if functional programmers do not know how to write down the structural induction principle for a new type - how are they supposed to reason about it? In this paper we describe an algorithm to generate structural induction principles from type signatures. We also discuss how these methods are taught in the Functional Programming course at the University of Wyoming. A Haskell implementation of the algorithm is included in an appendix.

1 Introduction

A fundamental claim made for functional programs is that they are easier to reason about. This is largely true because:

i.) the evaluation mechanism is substitution based (following ordinary mathematical practice), and

ii.) structural induction provides a straightforward mechanism for reasoning about programs defined by recursion on recursive data types.

A recursive type definition naturally gives rise to a structural induction principle for the type. For functions defined by recursion on the structure of a type, structural induction is the natural mechanism for reasoning about those functions. A functional programming course is the obvious place to make the relationship between induction and recursion explicit, and yet, we know of no standard text suitable for undergraduates that does so. In fact, there is no other point in the undergraduate curriculum where the concrete relationship between induction and recursion can be made as explicit as it can be in a course on Functional Programming.

In this paper we introduce the functional programming course taught at the University of Wyoming, briefly discuss well-founded induction, the justification for structural induction and then describe an algorithm for generating a structural induction schema from a data type declaration. We also discuss the pedagogical approach we use at the University of Wyoming to teach this material and give a few examples.

2 Functional Programming in Wyoming

The University of Wyoming offers an ABET accredited Bachelor of Science in Computer Science. Functional Programming (COSC 3015) is a required third year course for undergraduate students in the Computer Science degree. This is a somewhat unusual requirement for an undergraduate Computer Science program in the U.S., the University of Wyoming has had the requirement for at least fifteen years.

Functional Programming is taught once a year and is a prerequisite for the senior level Principles of Programming Languages (COSC 4870).

The functional programming course has been Haskell based since at least 2006; in earlier carnations it has been Scheme and LISP based. Students who take the course have already taken Discrete Structures (COSC 2300) which, as taught in Wyoming, emphasizes mathematical proofs. Students taking functional programming will (in theory) already know predicate logic, and will have done proofs by mathematical induction and possibly by complete induction. There is often a gap between the time a student takes Discrete Structure and Functional Programming and so proof methods are reviewed for the first week or two of the functional programming course. This is done by reviewing the the mathematical definition of a function and extensional equality for functions. This provides a nice segue into higher order functions and students apply the definitions to reason about *curry* and *uncurry*.

Though the course does not follow any one text, over the years it has been taught with the assistance of a variety of texts [2, 15, 6, 7, 8]. Bird's text [2] was the first one used to teach the course when it transitioned to Haskell and is still the book that is philosophically closest in spirit to the course as taught today. An objective for the functional programming course is for students to do learn to do proofs in concert with program development[1] Of the texts mentioned, only Bird's book carries through the proof theme from the beginning of the book to the end. Bird introduces a number of induction principles and, by example, expects students to be able to derive new structural induction principles from type declarations [2, Exercise 6.2.1, pp. 191]. In the functional programming class at Wyoming we make the relationship more explicit and expect students to be able to write down the structural induction principle for an arbitrary Haskell type and to use it to prove some simple properties about programs defined by recursion on the type.

Among the other texts references in the course, Thompson's book [15, pp. 141] introduces structural induction for finite lists and touches on induction for infinite lists but does not discuss induction principles for other structures. This first introduction is well after the lists have first been introduced. Similarly, Hudak's book introduces list induction [6, pp. 131] and natural number induction [6, pp. 141], other forms of induction are not discussed. Hutton's text relegates reasoning about programs to the last chapter [7, Chap.13] where he introduces both natural number induction and list induction. Lipovaca [8] never mentions induction at all. All of the Haskell texts mentioned have their strengths; but regarding the goal of teaching programming together with the methods for reasoning about programs, Bird's book does it best.

Among the texts that have not been used in the course (though some have been on the recommended readings list) Reade's book [13, pp.185] contains the only elementary explanation we know of regarding how structural induction principles can be derived from the declaration of a new type. Like Bird, Cousineau and Mauny [5] present a series of examples which serve to indicate how the induction principles can be gleaned from the type declaration. Of course more advanced programming language texts explain these topics in some detail [16, 9, 12].

The derivation of structural induction principles has been best covered in the theorem proving literature. The classic paper is Burstall's [3]. Paulson covers it in detail [11, pp. 77-135]. The implementations in Isabelle HOL prover [10] is excellent as are the accounts of structural induction principles automatically generated by the Coq prover [1, 4].

Perhaps the point we'd most like to make here is that, if programming and proving are to go hand-in-hand, understanding how to generate structural induction principles from data-type declarations is

---

[1]Students undoubtedly do not carry this practice with them to program developments beyond the functional programming course, but they learn a disciplined and formal way to think about and reason about programs.

essential knowledge. It is not hard to do nor is it hard to teach, and students who learn this are not hamstrung when it comes to proving things about a user defined type they've never seen before.

3 Background

3.1 Recursive Type Declarations

Modern functional programming languages all support some convenient form declaring new recursive types. In Haskell [14, Section 4.2.1] user defined recursive datatypes are given by *Algebraic Datatype Declarations*.

The declarations of new recursive types are given by specifying, in some way, constructor names and their signatures. In dialects of ML and in Haskell, such type declarations may also be polymorphically parameterized and appear as follows.

$$type\ T\ [tVars] = C_0\,[typeNames]\mid C_1\,[typeNames]\mid \cdots \mid C_m\,[typeNames]$$

where T is the name of the new type being defined. The type name T is followed by a list $[TVar]$ of polymorphic type variable names, say $[V_1, \cdots, V_k]$. If the type is not parameterized then the list is empty. The length of the list is the arity of T. On the right side of the declaration there are m constructor names $C_i, 0 \le i < m, m > 0$. Each constructor is followed by a list $[typeNames]$ of the names of the types of its parameters. The list may contain previously declared types, polymorphic type variables, and completely parameterized recursive instances of the type T. The *arity* of the constructor C_i is the length of the specified parameter list. A constructor name with no parameters is a constant of type T.

Recursive type declarations of this kind can be interpreted as many-sorted Σ-algebras and the finite instances of an inductively defined datatype are denoted by the freely generated terms of the algebra.

If a datatype T has arity 0 then it denotes a type of kind $*$. If T has arity 1 then it is a type constructor and has kind $* \to *$. If T has arity k it is a type constructor of k arguments and has kind $\overbrace{* \to \cdots \to *}^{k\ arrows}$. Note that by convention the function type constructor "$\to$" associates to the right so $* \to * \to *$ means $* \to (* \to *)$. If S is a type (possibly parameterized) of the form $(T\ T_1 \cdots T_k))$ we write $TyCon(S)$ for T, *i.e.* $TyCon$ denotes the type constructor of the type S. If S is not parameterized then $TyCon(S) = S$.

Note that the constructors provide the only means for building instances of a recursive data-type of the kind described here. The implication is that every instance is the application of a constructor C_i to appropriately typed arguments.

3.2 Well-founded Induction

Well-founded induction is a powerful and flexible form of induction and based on an ordering given by a well-founded relation over a type T. To show that a recursive function defined over a recursive type T terminates it is enough to show that there is some well-founded relation $\prec$ for which every recursive call is on a smaller instance of T with respect to $\prec$. Excellent accounts of well-founded induction can be found in [16, 9]

Definition 3.1 (Well-founded Relation) A binary relation on a set A is *well-founded* iff there are no infinitely descending chains $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$, *i.e.* there is no function $a : \mathbb{N} \to A$ such that for all $i \in \mathbb{N}, a(i+1) \prec a(i)$.

Note that a well-founded relation need not be transitive [9]. Thus, for example, the relation $i \prec j \stackrel{\text{def}}{=} j = i + 1$ is well-founded but $1 \prec 2$ and $2 \prec 3$ but $1 \not\prec 3$.

Theorem 3.1 (Well-founded Induction) Let $\prec$ be a well-founded binary relation on a set A and let P be a property of A, then

$$(\forall x{:}A. (\forall y{:}A. \, y \prec x \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x{:}A. P(x)$$

If Γ is a context, well-founded induction can be written in the form of a derived proof rule as follows:

$$\frac{\Gamma, x{:}A, \forall y{:}A. \, y \prec x \Rightarrow P(y) \vdash P(x)}{\Gamma \vdash \forall x{:}A. P(x)}$$

Pedagogically, well-founded induction is a bit more difficult to justify than structural induction which is more concrete.

3.3   Structural Induction for $\mathbb{N}$

The principle of mathematical induction (to show that a property P hold for all naturals) is often presented as follows.

$$(P(0) \wedge \forall k : \mathbb{N}.P(k) \Rightarrow P(k+1)) \Rightarrow \forall j : \mathbb{N}.P(j)$$

The natural numbers can be defined as a datatype with two constructors, a constant $Z : N$ and the successor function $S : N \to N$.

```
data Nat = Z | S Nat
```

Replacing Z for 0 and S for $(+1)$ yields the following:

$$(P(Z) \wedge \forall k : \mathbb{N}.P(k) \Rightarrow P(Sk)) \Rightarrow \forall j : \mathbb{N}.P(j)$$

This is simply the structural induction principle for $\mathbb{N}$. As a proof rule, this appears as follows.

$$(\text{MInd}) \quad \frac{\Gamma \vdash P(Z) \qquad \Gamma, k : \mathbb{N}, P(k) \vdash P(Sk)}{\Gamma \vdash \forall j{:}\mathbb{N}.P(j)}$$

To show that a property P of natural numbers hold for all natural numbers, show $P(Z)$ holds and then, assuming $P(j)$ holds for some arbitrary $j \in \mathbb{N}$ show $P(Sj)$ holds as well. Note that this is an instance of well-founded induction using the well-founded relation for natural numbers restated using the successor function in place of adding one: $i \prec_{\mathbb{N}} j \stackrel{\text{def}}{=} j = Si$.

How did the base case arise? Look at the rule for well-founded induction where the type A is specialized to $\mathbb{N}$ and the relation is the immediate successor relation ($\prec_{\mathbb{N}}$).

$$\frac{\Gamma, j{:}\mathbb{N}, \forall k{:}\mathbb{N}. \, k \prec_{\mathbb{N}} j \Rightarrow P(k) \vdash P(j)}{\Gamma \vdash \forall j{:}\mathbb{N}.P(j)}$$

Since $j \in \mathbb{N}$ we know $j = Z$ or $j = Si$ for some $i \in \mathbb{N}$. Do a case split on j giving two subgoals:

$$i.) \quad \Gamma, Z : \mathbb{N}, \forall k : \mathbb{N}.k \prec_{\mathbb{N}} Z \Rightarrow P(k) \vdash P(Z)$$
$$ii.) \quad \Gamma, i : \mathbb{N}, \forall k : \mathbb{N}.k \prec_{\mathbb{N}} (Si) \Rightarrow P(k) \vdash P(Si)$$

For (i) note that the antecedent in the induction hypothesis ($k \prec_\mathbb{N} Z$) is always false and so the implication is vacuously true. Thus $\forall k : \mathbb{N}.\, k \prec_\mathbb{N} Z \Rightarrow P(k)$ is trivially true and adds no information to our assumptions. Also, we already know $Z : \mathbb{N}$ so (i) simplifies to the following:

$$i.) \quad \Gamma \vdash P(Z)$$

For (*ii.*) note that by the definition of $\prec_\mathbb{N}$, if $k \prec_\mathbb{N} Si$ then $k = i$ and so $Si = Sk$. We do not need i at all, nor do we need the quantifier because the predecessor of Sk is just k itself. Using these facts we can simplify (*ii.*) to the following:

$$ii.) \quad \Gamma, k : \mathbb{N}, P(k) \vdash P(Sk)$$

This yields the ordinary rule for proof by mathematical induction (MInd).

3.4 Structural Induction in General

Structural induction is an instance of well-founded induction where the well-founded relation on pairs of terms of type T, $s \prec_T t$ is interpreted to mean that s is an immediate subterm of t (s is a child of t). The immediate subterm relation is not transitive, but as noted above, well-founded relations need not be. It is easy to see that for finite instances of a recursive datatype this definition yields a well-founded relation. Also note that $\prec_\mathbb{N}$ is the immediate subterm relation for the type $\mathbb{N}$.

For a recursive type T, the fact that instances of T must have been generated by one of the constructors yields together with simplifications based on the definition of $\prec_T$ can be used to justify the structural induction principles we now describe how to generate.

3.5 Generating the Induction Principle for a Recursive Type

We build the formula expressing the induction principle for a type T directly from its datatype declaration. In Appendix A there is Haskell code which does this.

Consider a parameterized type declaration of the following form:

$$T\,[V_1, \cdots, V_k] = C_0\,[typeNames] \mid \cdots \mid C_m\,[typeNames]$$

We build the structural induction principle in steps. The polymorphic type parameters (V_i) may denote any type. Recall that $*$ is the kind denoting types. Thus, the induction principle is a universally quantified formula of of the form[2].

$$\forall V_1 : *.\, \cdots \forall V_k : *.\, \square$$

A property of the type is a predicate over instances of the type. This yields the following:

$$\forall V_1 : *.\, \cdots \forall V_k : *.\forall P : (T\,V_1\,\cdots\,V_k) \rightarrow \mathbb{B}.\, \square$$

The goal we intend to prove is that the property holds for *all* instances of T so we can fill in the following bit:

$$\forall V_1 : *.\, \cdots \forall V_k : *.\forall P : (T\,V_1\,\cdots\,V_k) \rightarrow \mathbb{B}.$$
$$(\square \Rightarrow \forall t : (T\,V_1\,\cdots\,V_k).\,P(t))$$

---

[2]Here $\square$ denotes a hole in the formula yet to be defined.

Now, since every instance of the type is of the form C_i applied to the appropriate number and types of arguments, if we can show that, no matter which constructor was used, the property holds, then we've show that it holds for all instances, no matter how the instance was constructed.

Consider a constructor declaration of the form $C_i[T_1, \cdots, T_j]$. This constructor has the following type:

$$C_i : T_1 \rightarrow \cdots \rightarrow T_j \rightarrow (T\ V_1\ \cdots\ V_k)$$

Note that some of the T_i may be instances of the type T (the one being defined) itself. These references to T are the recursive parts of the declaration and by the well-foundedness of the immediate subterm relation, we may assume the property holds for these instances. The clause for C_i is given as follows:

$$\mathscr{F}(C_i) \stackrel{\text{def}}{=} \forall x_1 : T_1. \cdots \forall x_j : T_j. \left(\bigwedge_{\substack{i \in \{1..j\} \\ TyCon(T_i) = T}} P(x_i) \right) \Rightarrow P(C_i x_1 \cdots x_j)$$

For each type T_i that is a recursive instance of T, we assume P holds for that instance. Note that the constraint on inductive hypotheses is not that $T_i = (T V_1 \cdots V_k)$ but simply that T is the type constructor for the type T_i. This allows for types where the recursive instances in the type declaration do not have the same arguments in every call *i.e.* see the *SwapTree* example included below.

Putting it all together we get the following structural induction principle.

$$\forall V_1 : *. \cdots \forall V_k : *.\forall P : (T\ V_1\ \cdots\ V_k) \rightarrow \mathbb{B}.$$
$$\left(\bigwedge_{i \in \{1..j\}} \mathscr{F}(C_i) \right) \Rightarrow \forall t : (T\ V_1\ \cdots\ V_k).\ P(t)$$

The algorithm described here is implemented by the Haskell code in Appendix A. The function stind takes a Haskell datatype representing the abstract syntax of a Haskell data declaration and returns an instance of a formula type encoding the structural induction principle. Within the body of stind. the locally defined function mkConstructorClause implements the formula transformation defined above as $\mathscr{F}$.

4   In the classroom

Students in the functional programming course at the University of Wyoming are required to be able to write down the structural induction principle for user defined types and to use them to prove properties of functions defined by recursion on the type. The justification for the method presented above has not been part of the course. The program, written for this paper, to generate induction principles will be used when the course is next offered in the Fall 2013 semester. As Bird [2] and Cousineau and Mauny [5] have noted, examples suffice to show the pattern.

Consider the following Haskell data types:

```
data Nat = Z | S Nat
data List a = Nil | Cons a (List a)
data Tsil a = Snoc (Tsil a) a | Lin
data Btree a = Leaf a | Fork (Btree a) (Btree a)
data SwapTree a b = Leaf | Node a (SwapTree b a) (SwapTree b a)
```

The following are the structural induction principles generated by the Haskell code in Appendix A for the types just given. The show function was specialized to "show" the LaTeX output.

Nat :
$\forall P : Nat \rightarrow Bool.$
$\quad ((P\,Z) \wedge \forall n_1 : Nat.((P\,n_1) \Rightarrow (P(S\,n_1)))) \Rightarrow \forall n : Nat.(P\,n)$

List a :
$\forall a : *. \forall P : (List\,a) \rightarrow Bool.$
$\quad ((P\,Nil) \wedge \forall x_1 : a. \forall l_2 : (List\,a).$
$\qquad\quad ((P\,l_2) \Rightarrow (P\,(Cons\,x_1\,l_2))))$
$\quad\quad \Rightarrow \forall l : (List\,a).\,(P\,l)$

Tsil a :
$\forall a : *. \forall P : (Tisl\,a) \rightarrow Bool.$
$\quad (\forall x_1 : a. \forall t_2 : (Tisl\,a).$
$\qquad\quad ((P\,t_2) \Rightarrow (P\,(Snoc\,x_1\,t_2))) \wedge (P\,Lin))$
$\quad\quad \Rightarrow \forall t : (Tisl\,a).\,(Pt)$

BTree a :
$\forall a : *. \forall P : (Tree\,a) \rightarrow Bool.$
$\quad (\forall x_1 : a.\,(P\,(Leaf\,x_1))$
$\qquad \wedge\,(\forall t_1 : (Tree\,a). \forall t_2 : (Tree\,a).$
$\qquad\quad (((P\,t_1) \wedge (Pt_2)) \Rightarrow (P\,(Node\,t_1\,t_2)))))$
$\quad\quad \Rightarrow \forall t : (Tree\,a).\,(P\,t)$

SwapTree a b :
$\forall a : *. \forall b : *. \forall P : (SwapTree\,a\,b) \rightarrow Bool.$
$\quad ((P\,Leaf)$
$\qquad \wedge\,(\forall x_1 : a. \forall s_2 : (SwapTree\,b\,a). \forall s_3 : (SwapTree\,b\,a).$
$\qquad\quad (((P\,s_2) \wedge (P\,s_3)) \Rightarrow (P\,(Node\,x_1\,s_2\,s_3)))))$
$\quad\quad \Rightarrow \forall s : (SwapTree\,a\,b).\,(P\,s)$

Now consider some seemingly pathological examples; these datatypes are not recursive.

data Bool = T | F
data Maybe a = Nothing | Just a

The induction principles generated by the method described above yield the following formulas.

Bool :
$\forall P : Bool \rightarrow Bool.\,((P\,T) \wedge (P\,F)) \Rightarrow \forall b : Bool.\,(P\,b)$

Maybe a :
$\forall a : *. \forall P : (Maybe\,a) \rightarrow Bool.$
$\quad ((P\,Nothing) \wedge \forall x_1 : a.\,P(Just\,x_1)) \Rightarrow \forall m : (Maybe\,a).\,(P\,m)$

The structural induction principles are generated by case analysis (on the constructors) and by including the appropriate induction hypotheses for each case. If there is no recursion in the type definition,

the induction principle reduces to case analysis. The resulting formulas are theorems whether there is recursion or not.

In lazy languages like Haskell, all types T are inhabited by the undefined value $\bot$. $\bot$ is a value that polymorphically inhabits every type. With regards to well-founded relations on terms of type T, for all finite terms s, $\bot \prec_T s$ and $\bot \not\prec_T \bot$. Case analysis on the natural numbers yielded two cases, one for numbers constructed from the constant Z and the other for numbers constructed by the successor function S. To prove a property of lazy natural numbers an additional case is added to show that $(P \bot)$ holds. This extra case arises naturally from the schema of well-founded induction in the same way the simplified case for Z did when the case analysis splits to include the possibility of $\bot$. The structural induction principles can be extended to work on these *pointed* types [**?**, pp.310] simply by adding a clause $P(\bot)$ which must be shown to hold in addition to the others.

5 Conclusion

Students learning functional programming are in a unique position to be able to prove properties about the programs they write during the development process. It is often the case that putative properties of the functions serve as specifications for the functions and can be used to verify their correctness. Fundamental properties about recursive types can be verified in the context of a functional programming course that are virtually impossible to do in the imperative setting. However, if they do not have the ability define structural induction principles for newly defined types, students are left wanting in their skills.

At the University of Wyoming we have been presenting these methods for years in the functional programming course (COSC 3015). Students practice the methods in homework assignments and there are often questions on the final examination requiring them to write structural induction principles for types they have not seen before. On the most recent exam they were required to write induction principles for a type representing lambda terms and for a tree structure with three kinds of nodes.

The methods described here have been widely implemented in the theorem proving community and deserve to be better known in the functional programming community.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, second edition, 1998.

[3] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[4] Adam Chlipala. Certified programming with dependent types. Feb. 12, 2013, availabe online http://adam.chlipala.net/cpdt/.

[5] Guy Couineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1995.

[6] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming theough Multimedia*. Cambridge University Press, 2000.

[7] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[8] Miran Lipovaca. *Learn You a Haskell for Great Good: A Beginner's Guide*. No Starch Press, 2011.

[9] John Mitchell. *Foundations for Programming Langauges*. MIT Press, 1996.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[11] Lawrence C Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1990.

[12] Benjamin Pierce. *Types and Programming Langauges*. MIT Press, 2002.

[13] Chris Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[14] et. al. Simon Peyton Jones. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):1–255, Jan 2003.

[15] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, second edition, 1999.

[16] Glynn Winskel. *The Formal Semantics of Programming Langauges: An Introduction*. MIT Press, 1993.

Appendix A Haskell code to generate a structural induction principle from a data declaration (stind).
Pretty printing code not included.

```
import Data.Char
data Type = Star | Simple String [Type] | Tuple [Type] | Arrow Type Type
   deriving (Eq, Show)
type CName = String
type TName = String
data Data = Data TName [TName] [(CName, [Type])] deriving (Eq,Show)
data Formula = FTrue
      | Pred String [Formula]
      | And Formula Formula
      | Implies Formula Formula
      | Forall String Type Formula
   deriving (Eq,Show)
conjoin = foldr1 (\f fs → And f fs)
forall = foldr (\(v,ty) more → Forall v ty more)
mkFormulaVars = map (\v → Pred v [])
numberedVars vars = map (\(x,i) → x ++ (show i)) (zip vars [1..])
stind (Data tname targs constructors) =
  let indVarName = map toLower (take 1 tname) in
  let varName = "x" in
  let newType = Simple tname (map (\t → Simple t []) targs) in
  let prefix body =
        forall (Forall "P" (Arrow newType (Simple "Bool" []))) body)
             (zip targs (repeat Star)) in
  let mkConstructorClause (c, types) =
        if null types then
          Pred "P" [Pred c []]
        else
          let arity = length types in
          let vars = numberedVars
               (map (\(Simple name _) →
                       if name == tname then indVarName else varName) types) in
          let varsXTypes = zip vars types in
          let indVars = mkFormulaVars $
               map fst (filter (\(_, Simple t _) → t == tname) varsXTypes) in
          let antecedents = conjoin (map (\t → Pred "P" [t]) indVars) in
          let concl =  Pred "P" [Pred c (mkFormulaVars vars)] in
          let universal body = forall body varsXTypes  in
            case indVars of
              [] →  universal concl
              _ →  universal (Implies antecedents concl) in
  let antecedent = conjoin (map mkConstructorClause constructors) in
  let concl = Forall indVarName newType (Pred "P" [Pred indVarName []]) in
     prefix (Implies antecedent concl)
```