Producing Certified Functional Code from Inductive Specifications

Pierre-Nicolas Tollitte[1], David Delahaye[2], and Catherine Dubois[3]

[1] CEDRIC/ENSIIE, Évry, France,
tollitte@ensiie.fr
[2] CEDRIC/CNAM, Paris, France,
David.Delahaye@cnam.fr
[3] CEDRIC/ENSIIE/INRIA, Paris, France,
dubois@ensiie.fr

Abstract. Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). Both styles have pros and cons. Relational style may be preferred because it allows the user to describe only what is true, discard momentarily the termination question, and stick to a rule-based description. However, a relational specification is usually not executable. This paper proposes to turn an inductive specification into a functional one, in the logical setting itself, more precisely Coq in this work. We define for a certain class of inductive specifications a way to extract functions from them and automatically produce the proof of soundness of the extracted function w.r.t. its inductive specification. In addition, using user-defined modes which label inputs and outputs, we are able to extract several computational contents from a single inductive type.

Keywords: Executable Specifications, Inductive Relations, Functional Code Generation, Soundness Proof Generation, Coq

1 Introduction

Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). The choice between the two styles may be guided by different requirements, but it is also a matter of taste. Using inductive types or relational style may be preferred because it allows the user to specify only what is true, discard termination issues, and stick to usual rule-based presentations. A typical example, which illustrates these points, concerns the definition of an operational semantics including a while loop. While it is rather tricky to define an interpretation function [4], it is easier and more natural to define its operational semantics by means of an inductive relation. The former approach has to deal with general recursion, partiality, and termination. The latter approach provides the user with a straightforward implementation, where the inference system is formalized as an inductive type whose constructors are a direct rewording of the inference rules. However, in some systems like Coq

or Isabelle, these definitions are not directly executable. Simulating the execution of a program therefore requires proving a judgment using the constructors of the inductive type with more or less automation, and this kind of process does not scale up to complex specifications. Another argument in favor of relational style is that an inductive specification may describe several computational behaviors according to the arguments of the inductive relation which have been selected as inputs. For example, from the predicate *add* where *add* n m p specifies that p is the sum of n and m, it is possible to extract not only a function realizing the addition, but also a function realizing the subtraction.

A feature offered by some systems such as Coq, Isabelle, or HOL, consists of the possibility to extract code from functional specifications. In Isabelle [3, 2] and Coq (see the previous work of some of the authors [6], as well as the plugin distributed with Coq), it is even possible to do so from inductive specifications. However, in the Coq framework, an ML function extracted from an inductive type, even if it terminates, cannot be used in the Coq environment itself.

In this paper, we propose an approach which turns an inductive specification into a functional one within the logical setting itself, i.e. Coq [10] in particular. For a class of inductive specifications, we define a way to extract functions from these specifications and produce the proof of soundness of any extracted function w.r.t. its inductive specification. This allows us to not only use the generated functions within the proof assistant, but also reason over them. This approach is fully automatic if the extracted function follows a structural recursion scheme. Otherwise, if the recursion is general, the user must provide termination information, such as a measure or a well-founded order. Our approach is limited to inductive specifications from which we can extract structurally recursive functions. Our contribution consists in automating a common but tedious practice, as illustrated by this quotation by Blazy and Leroy in [5]: "*The recommended approach to execute a Coq specification by inductive predicates, therefore, is to define a reference interpreter as a Coq function, prove its equivalence with the inductive specification, and evaluate applications of the function.*".

To produce the Coq functional code, we follow the translation scheme given in [6] for extracting ML code from inductive specifications. Compared to [6], we are able to deal with a larger family of inductive specifications, involving in particular some specific cases of non-deterministic inductive relations, and as said previously, we are also able to deal with proofs of soundness.

The closest approach to the work described in this paper concerns the compilation of inductive relations in Isabelle/HOL into executable programs [3, 2]. In this approach, the authors rely on a mode consistency analysis, in the same way as in [6] and the work presented here. The notion of mode comes from logic programming and helps us perform various analyses and optimizations. In [2], the inductive definition is translated into a set of equations equivalent to the initial moded definition, and then exported to a functional programming language. The equivalence is proved by means of a sound and complete procedure. The main difference between this approach and our work is that the authors are able to compile non-deterministic specifications, while we reject some of them. As a con-

sequence, this approach relies on an infrastructure of sequences in order to have all the possible results even if the specification is deterministic, resulting in less readable programs in some cases. In another context, inductive specifications encoded in Twelf [9] can be executed using a higher-order logic programming language, but it does not export any code within or outside of the logic.

The paper is structured as follows: in Section 2, we illustrate our approach on a basic example; we then introduce, in Section 3, our notion of inductive specification, and present our code generation algorithm; next, in Section 4, we describe the generation of proofs of soundness for the extracted functions w.r.t. their corresponding inductive specifications; finally, in Section 5, we provide information regarding the implementation which has been realized in the framework of Coq.

2 An Example

In this section, we present how our functional extraction should work on an example of inductive specification within the Coq framework [10]. This functional extraction is performed in several steps. First, the user annotates his/her inductive relation with a mode specifying which arguments are inputs, the others being considered as outputs. A mode consistency analysis is then performed to determine if the extraction is possible w.r.t. the provided mode. If the previous analysis is successful, the inductive relation is translated into a functional program. Finally, if the previous translation is successful, a proof of soundness is produced, ensuring that the generated function verifies the initial inductive relation. Compared to [6], the translation into a functional program is performed within the logical framework (i.e. Coq), which requires the extracted function to terminate and allows us to generate a proof of soundness.

As an example of extraction, let us consider the specification consisting in searching for a value in a binary search tree, which can be formalized in Coq as follows (let us note that we introduce two inductive relations, one for comparing two values of the tree, and another one for searching a value in the tree):

```
Inductive bst : Set :=
  | Empty : bst
  | Node : bst → nat → bst → bst.

Inductive comp_nat : Set := | Inf | Sup | Eq.

Inductive path: Set :=
 | Not_found | End_path
 | Left : path → path | Right : path → path.

Inductive compare : nat → nat → comp_nat → Prop :=
  | Compare_eq : compare 0 0 Eq
  | Compare_inf : forall n, compare 0 (S n) Inf
  | Compare_sup : forall n, compare (S n) 0 Sup
  | Compare_rec : forall n m c, compare n m c →
      compare (S n) (S m) c.
```

```
Inductive search : bst → nat → path → Prop :=
  | Search_empty : forall n, search Empty n Not_found
  | Search_found : forall n m t1 t2, compare n m Eq →
      search (Node t1 m t2) n End_path
  | Search_inf : forall n m t1 t2 b, search t1 n b →
      compare n m Inf → search (Node t1 m t2) n (Left b)
  | Search_sup : forall n m t1 t2 b, search t2 n b →
      compare n m Sup → search (Node t1 m t2) n (Right b).
```

Using the mode $\{1, 2\}$ both for the *compare* and *search* relations (which means that we use the two first arguments of *compare* and *search* as inputs), the following function can be automatically extracted from the relation *search*:

```
Fixpoint search12 (p1 : bst) (p2 : nat) : path :=
  match p1 with
  | Empty ⇒ Not_found
  | Node t1 m t2 ⇒
    match compare12 p2 m with
    | Inf ⇒ let b := search12 t1 p2 in Left b
    | Sup ⇒ let b := search12 t2 p2 in Right b
    | Eq ⇒ End_path
    end
  end.
```

where *compare12* is the function extracted from the relation *compare*.

It should be noted that using the mode $\{1, 2\}$, the relation *search* appears as non-deterministic in the sense that several constructors overlap (in this case, *Search_found*, *Search_inf*, and *Search_sup*). This requires a specific analysis of the premises of the corresponding constructors to realize that the relation is actually deterministic using the result of a given call to distinguish them (here, the result of the application of *compare12*).

Once the previous function has been generated, it is possible to produce a proof of soundness for this function, i.e. to prove that it verifies the relation from which it has been extracted. To do so, the idea is to use the functional induction scheme of the extracted function generated by Coq, which is the following (due to space restrictions, we only describe the cases of *Search_empty* and *Search_inf*):

```
Lemma search12_ind :
  forall P : bst → nat → path → Prop,
    (forall (p1 : bst) (p2 : nat), p1 = Empty →
       P Empty p2 Not_found) →
    (forall (p1 : bst) (p2 : nat) (t1 : bst) (m : nat)
       (t2 : bst), p1 = Node t1 m t2 →
       compare12 p2 m = Inf → P t1 p2 (search12 t1 p2) →
       let p := search12 t1 p2 in
       P (Node t1 m t2) p2 (Left p)) → ...
    forall (p1 : bst) (p2 : nat), P p1 p2 (search12 p1 p2).
```

Using this induction scheme, it is possible to automatically complete the proof of soundness for the function previously extracted as follows (we still focus on the cases corresponding to the constructors *Search_empty* and *Search_inf*):

```
Lemma search12_sound :
  forall (p1 : bst) (p2 : nat) (p : path),
    search12 p1 p2 = p → search p1 p2 p.
Proof.
  intros until 0; intro H; subst p; apply search12_ind.
  (* Search_empty *)
  intros until 0; intro H; apply Search_empty.
  (* Search_inf *)
  intros until 0; intros H1 H2 H3; simpl.
  apply Search_inf;
    [assumption | apply compare12_sound; assumption].
  ... (* Search_sup and Search_found *)
Save.
```

where *compare12_sound* is the soundness lemma for the *compare12* function.

3 Code Generation

Our code generation algorithm consists in producing a functional program from an inductive relation and an extraction mode. In the following, we will borrow some definitions and notations from [6], and in particular, an inductive relation will be called logical inductive type. If the extraction is performed from a logical inductive type d, the definition of d may use other logical inductive types named d_i. In this case, extraction modes must be provided for all these types. We will not deal with mutually recursive definitions, and we will assume that each dependency w.r.t. d_i has already been extracted with its extraction mode.

3.1 Logical Inductive Types

The Coq proof assistant relies on the Calculus of Inductive Constructions (CIC for short) type theory, for which a description can be found in the Coq documentation [10]. This theory is actually too extensive for the purpose of this paper, and we will use a subset of CIC to describe logical inductive types. The subset of CIC that we will consider is very similar to the one used in [6], and we will only add some restrictions on the form of the terms. An inductive definition is noted $\mathsf{Ind}(d : \tau, \Gamma)$, where d is the name of the inductive definition, τ its type, and Γ the context representing the constructors (their names together with their respective types). In this notation, two restrictions have been made: we do not deal with parameters (i.e. the additional arguments which are shared by the type τ of the inductive definition and the types of constructors defined in Γ) and mutual inductive definitions. In addition, dependent types, higher order and propositional arguments are not allowed in the type of an inductive definition; more precisely, this means that τ has the following form

$\tau_1 \rightarrow \ldots \tau_i \rightarrow \ldots \tau_n \rightarrow \mathsf{Prop}$ where τ_i, with $i = 1 \ldots n$, is of type Set or Type, and does not contain any product or dependent inductive type. Moreover, we suppose that the types of constructors are in prenex from, with no dependency between the bounded variables and no higher order; thus, the type of a constructor is $\forall x_1 : X_1, \ldots, x_n : X_n . T_1 \rightarrow \ldots T_j \rightarrow \ldots T_m \rightarrow d\ t_1\ \ldots\ t_p$ where $x_i \notin X_l$, with $l > i$, X_i is of type Set or Type, T_j is of type Prop and does not contain any product or dependent inductive type, and t_k, with $k = 1 \ldots p$, are terms. In the following, the terms T_j are called the premises of the constructor, whereas the term $d\ t_1\ \ldots\ t_p$ is called the conclusion of the constructor. We impose the additional constraint that T_j is a fully applied logical inductive type, i.e. T_j is of the form $d_j\ t_{j1}\ \ldots\ t_{jp_j}$, where d_j is a logical inductive type (possibly different from d), t_{jk}, with $k = 1 \ldots p_j$, are terms, and p_j is the arity of d_j. Additionally, we put some restrictions on the form of a term t_i, which is either a variable or a fully applied constructor $c_i\ t_{i1}\ \ldots\ t_{ip_i}$, where p_i is the arity of c_i. An inductive type verifying the conditions above is called a logical inductive type. We aim to propose an extraction method for this kind of inductive types.

In the general case, we aim to extract only deterministic specifications. We actually distinguish two kinds of determinism. The basic notion of determinism is when for a given extraction mode, the inputs of the conclusions of constructors are pairwise non-unifiable. However, there also exists another kind of determinism, where the logical inductive type seems non-deterministic but actually remains deterministic, i.e. where there are overlapping conclusions of constructors, but where a function can still be extracted (see the example of Section 2). In contrary to [6], we propose to also deal with this other kind of determinism in some specific cases, where using a premise, we can distinguish between the constructors whose conclusions overlap. This requires the use of a specific representation of logical inductive types, which is introduced in the next subsection.

In the following, we will refer to the constructors using their names. In order to denote the constructor named C, we will use $\Gamma(C)$. We will also add the notation $P(C)$ to denote the set of premises of a constructor named C, and the notation $P(C, i)$ to denote the i^{th} premise of the constructor C.

3.2 Intermediate Representation for Merging Constructors

As said previously, the work developed in this paper also proposes to relax some restrictions imposed in [6]. One of them is that constructors do not overlap, which means that their respective conclusions are pairwise non-unifiable. This restriction is too strong as it prevents from handling quite common specifications like the example of Section 2. However, when some conclusions overlap, we can still generate some code in some cases. The first pattern matching of the generated function is usually used to distinguish between constructors. To extract specifications with overlapping conclusions, the idea is to merge them and use premises to distinguish between constructors. In the example of Section 2, we have to merge the conclusions of the *Search_ inf*, *Search_ sup*, and *Search_ found* constructors. These three conclusions will be compiled as the same pattern in the extracted function. In some cases, it may be also necessary to merge premises.

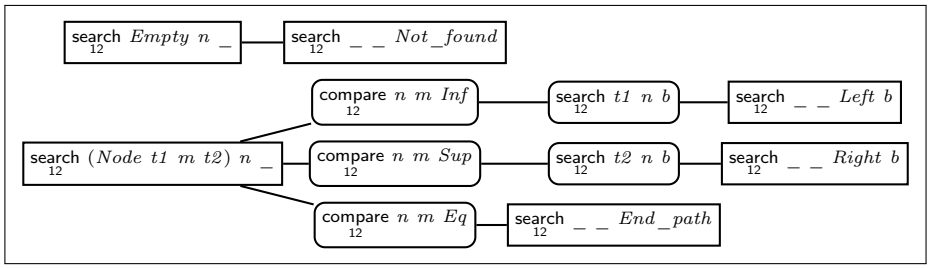

Fig. 1. Rel-Tree Representation for the Binary Search Tree Example

Relation-Tree Definition In order to represent the merging of constructors, we introduce a new intermediate data structure to represent logical inductive types where constructors can be merged. This new representation is based on trees, a data structure which eases both the verification of some properties over the specification and the code generation. Logical inductive types are actually represented as a forest called a relation-tree, which is defined as follows:

Definition 1 (Relation-Tree). *Given a logical inductive type d, it can be represented by the following relation-tree (or rel-tree for short):*

$$\mathsf{Rel{-}Tree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{k1}\ \ldots\ t_{kp}, \mathsf{Nodes}_k)\})$$

where Nodes_i *is either* $\{(T_{i1}, \mathsf{Nodes}_{i1}),\ \ldots,\ (T_{ik}, \mathsf{Nodes}_{ik})\}$ *or* $d\ t_{i1}\ \ldots\ t_{ip}$.

It should be noted that in this definition, k is always smaller than or equal to the number of constructors in the logical inductive type d, because there is at most one node by constructor, and less if there are merged constructors.

Considering the example of the binary search tree of Section 2, it is possible to represent the inductive relation *search* by the rel-tree of Figure 1. In this representation, we use several conventions. The leaves are represented at the right-hand side. The nodes annotated by a conclusion of the specification are represented by the boxes with sharp corners, whereas the nodes related to premises are in the boxes with rounded corners. In these nodes, some arguments may be hidden with underscores when they are not relevant; for instance, we hide the output in the root nodes and the inputs in the leaf nodes, because they are not involved when extracting the code. In addition, in each node, the extraction mode is mentioned under the considered relation name.

Rel-Tree Properties The main task of the code generation is to build a rel-tree verifying the three properties described below (the code generation itself is in turn quite straightforward). In order to describe these three properties, we need a function to get a path from a rel-tree. In the following, the treepaths function will return the set of paths that can be built from a rel-tree. For example, if *bst_tree* denotes the rel-tree of Figure 1, we have:

$\mathsf{treepaths}(bst\_tree) =$
$\{\ [\ \mathsf{search}\ Empty\ n\ \_;\ \mathsf{search}\ \_\ \_\ Not\_found\ ],$
$[\ \mathsf{search}\ (Node\ t1\ m\ t2)\ n\ \_;\ \mathsf{compare}\ n\ m\ Inf;\ \mathsf{search}\ t1\ n\ b;$
$\mathsf{search}\ \_\ \_\ Left\ b\ ],\ \ldots\ \}$

We also need functions to compute input and output terms according to a mode, where a mode is a set of indexes which correspond to the arguments of the logical inductive type used as inputs:

Definition 2 (Functions for Inputs/Outputs). *Given a logical inductive type d, some terms $t_1 \ldots t_{p_d}$, and a mode m, we define the following functions:*

$$\mathsf{in}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq (t_{i_1},\ \ldots,\ t_{i_m}),\ \textit{where } m \textit{ is } \{i_1,\ \ldots,\ i_m\}$$
$$\mathsf{invars}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq \mathsf{variables}(\mathsf{in}(d\ t_1\ \ldots\ t_{p_d}, m))$$
$$\mathsf{out}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq \begin{cases} \textit{if } \exists j \in 1 \ldots p_d, j \notin \{i_1,\ \ldots,\ i_m\} \textit{ then } t_j \\ \textit{else true} \end{cases}$$
$$\textit{where } m \textit{ is } \{i_1,\ \ldots,\ i_m\}$$
$$\mathsf{outvars}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq \mathsf{variables}(\mathsf{out}(d\ t_1\ \ldots\ t_{p_d}, m))$$

where $\mathsf{variables}(t)$ *returns the set of variables occurring in the term* t*.*

We also define the global environment $\mathcal{M}$, which contains the extraction modes for all the logical inductive types used in the logical inductive type being extracted. We can get the extraction mode for the logical inductive type d using the notation $\mathcal{M}(d)$. This notation is extended for the premises as follows: if T_i is of the form $d_i\ t_{i1}\ \ldots\ t_{ip_i}$, then $\mathcal{M}(T_i) = \mathcal{M}(d_i)$. In the following, d and m will refer to the logical inductive type being extracted and its extraction mode.

The first property describes the relationship between the logical inductive type and its rel-tree. We must ensure that we find all constructors with their conclusions and their premises in the rel-tree (up to renaming).

Property 1 (Specification Compliance). The rel-tree r is said to comply with its logical inductive type d iff it verifies the following property:

$$\mathrm{SC}(r) \triangleq \forall C \in \Gamma, \exists! p \in \mathsf{treepaths}(r), \mathrm{SC}'(p, C) \wedge$$
$$\forall p \in \mathsf{treepaths}(r), \exists! C \in \Gamma, \mathrm{SC}'(p, C)$$

where SC' is the compliance of a path $[T_0;\ T_1;\ \ldots;\ T_n;\ T_{n+1}]$ for a given constructor named C defined as follows:

$$\mathrm{SC}'([T_0;\ T_1;\ \ldots;\ T_n;\ T_{n+1}], C) \triangleq$$
$$\exists \sigma, \exists \Pi, n = card(P(C)) \wedge \forall i \in 1 \ldots n, T_j = \sigma(P(C, \Pi i)) \wedge$$
$$\mathsf{out}(T_{n+1}, m) = \mathsf{out}(\sigma(\mathsf{concl}(\Gamma(C))), m) \wedge$$
$$\mathsf{in}(T_0, m) = \sigma(\mathsf{in}(\mathsf{concl}(\Gamma(C)), m))$$

where σ is a variable renaming, Π a permutation of $1 \ldots n$, and concl a function which returns the conclusion term of a constructor.

The second property is similar to the mode consistency analysis of [6], but is performed on the rel-tree instead of the logical inductive type. It verifies that variables are not used before they are defined in the generated function.

Property 2 (Mode Consistency Analysis). Given a mode m and a rel-tree r of the form $\mathsf{Rel{-}Tree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1);\ \ldots;\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\})$, m is said to be consistent w.r.t. r iff the following property is verified:

$$\begin{array}{l}\mathrm{MCA}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1);\ \ldots;\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\}) \triangleq \\ \quad \forall i, i \in 1 \ldots n, \mathrm{MCA}'(\mathsf{Nodes}_i, \mathsf{invars}(d\ t_{i1}\ \ldots\ t_{ip}))\end{array}$$

where MCA' is defined as follows:

$$\mathrm{MCA}'(nodes, S) \triangleq \begin{cases} \text{if } nodes \text{ is } d\ t_1\ \ldots\ t_p \text{ then} \\ \quad \mathsf{outvars}(d\ t_1\ \ldots\ t_p, m) \subseteq S \\ \text{if } nodes \text{ is } \{(T_1, \mathsf{Nodes}_1);\ \ldots;\ (T_n, \mathsf{Nodes}_n)\} \text{ then} \\ \quad \forall i, i \in 1 \ldots n, \mathsf{invars}(T_i, \mathcal{M}(T_i)) \subseteq S \wedge \\ \qquad \mathrm{MCA}'(\mathsf{Nodes}_i, S \cup \mathsf{outvars}(T_i, \mathcal{M}(T_i))) \end{cases}$$

The third property ensures that we build valid pattern matchings from the rel-tree, i.e. with exclusive clauses (involving non-overlapping patterns). The patterns of the same pattern matching will be the outputs of the premise nodes which are the children of the same parent node.

Property 3 (Non-Overlapping Patterns). Given a mode m and a rel-tree r of the form $\mathsf{Rel{-}Tree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1);\ \ldots;\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\})$, the function extracted from r in mode m will only involve non-overlapping patterns iff the following property is verified:

$$\begin{array}{l}\mathrm{NO}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1);\ \ldots;\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\}) \triangleq \\ \quad \forall i, i \in 1 \ldots n, \mathrm{NO}'(\mathsf{Nodes}_i) \wedge \forall j, j \in 1 \ldots n, j \neq i \Rightarrow \\ \qquad \mathsf{in}(d\ t_{i1}\ \ldots\ t_{ip}, m) \text{ and } \mathsf{in}(d\ t_{j1}\ \ldots\ t_{jp}, m) \text{ are not unifiable}\end{array}$$

where the NO' property is defined as follows:

$$\begin{array}{l}\mathrm{NO}'(nodes) \triangleq \\ \quad \begin{cases} \text{if } nodes \text{ is } d\ t_1\ \ldots\ t_p \text{ then } true \\ \text{if } nodes \text{ is } [(T_1, \mathsf{Nodes}_1);\ \ldots;\ (T_n, \mathsf{Nodes}_n)] \text{ then} \\ \quad \forall i, i \in 1 \ldots n, \mathrm{NO}'(\mathsf{Nodes}_i) \wedge \forall j, j \in 1 \ldots n, j \neq i \Rightarrow \\ \qquad \mathsf{in}(T_i, \mathcal{M}(T_i)) = \mathsf{in}(T_j, \mathcal{M}(T_j)) \wedge \\ \qquad \mathsf{out}(T_i, \mathcal{M}(T_i)) \text{ and } \mathsf{out}(T_j, \mathcal{M}(T_j)) \text{ are not unifiable} \end{cases}\end{array}$$

Due to space restrictions, we do not present the rel-tree generation algorithm in details here, and we only provide a short description of this algorithm. The complexity of this algorithm mainly comes from the possible permutations of premises. The basic idea is to generate all the possible rel-trees from the specification and find if there is any rel-tree verifying the three properties introduced

above. However, in order to generate fewer rel-trees (because there are many permutations of premises), we therefore add some heuristics using the three properties described above. From the first property, we can deduce a general form of rel-trees. Each rel-tree must contain one path for each constructor of the specification. Each path must begin and end with the conclusion. Except these two nodes, there is one node for each premise. The second property can be verified independently for each path. Only the third property needs all the constructors to be present in the rel-tree to be verified, but it can be verified after each insertion. As a result, we can insert the constructors one by one, and verify the three properties at each step.

3.3 Partial Mode Extraction of Complete Specifications

As said in the introduction, we only consider structurally recursive functions in this paper. As a consequence, the extracted functions are generated as regular fixpoints of CIC, which consists of our target language. We actually use the following subset of CIC (we use the notations of the Coq documentation [10]):

$$
\begin{array}{l}
t ::= \ fix\ f\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n)\ :\ \tau\ :=\ t \\
\quad |\ \ f\ t1\ \ldots\ t_{p_f} \mid c\ t1\ \ldots\ t_{p_c} \mid let\ x\ :=\ t_1\ in\ t_2 \mid x \\
\quad |\ \ (match\ t_m\ \ with \\
\qquad \mid c_1\ x_{11}\ \ldots\ x_{1p_1}\ \Rightarrow\ f_1 \mid \ldots \mid c_n\ x_{n1}\ \ldots\ x_{np_n}\ \Rightarrow\ f_n)
\end{array}
$$

In this language, there is no complex pattern matching. We can only match one term of type τ and there is one pattern per constructor of the type τ. However, to simplify the presentation of the code generation, we use more complex pattern matching expressions (with nested patterns) as follows:

$$
\begin{array}{l}
match\ (t_{m1},\ \ldots,\ t_{mn})\ with \\
\mid p_{11},\ \ldots,\ p_{1n}\ \Rightarrow\ t_1 \mid \ldots \mid p_{k1},\ \ldots,\ p_{kn}\ \Rightarrow\ t_k
\end{array}
$$

where "$p ::= \ c\ p_1\ \ldots\ p_{p_c} \mid x \mid \_$".

These more complex pattern matching expressions are then compiled into simpler pattern matching expressions, which conform to the initial language of the CIC subset considered for the extraction. This compilation is performed using a specialized version of the algorithm described in [7].

In the following, we describe the code generation for partial modes and complete specifications. A mode is partial iff there is one output (otherwise, when there is no output, the mode is full). A specification is complete for a given mode iff its extraction produces a complete function, in which all the pattern matchings are exhaustive. The extraction for full modes and incomplete specifications will be explained later, as evolutions of the algorithm described below.

The code generation of a rel-tree r built from the inductive definition d, of the form $\mathsf{Rel{-}Tree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{k1}\ \ldots\ t_{kp}, \mathsf{Nodes}_k)\})$, and extracted with the mode $m = \{i_1,\ \ldots,\ i_m\}$, is denoted by $[\![r]\!]_{\mathcal{M}}$ and performed in the following way:

$$
\begin{array}{l}
[\![\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{k1}\ \ldots\ t_{kp}, \mathsf{Nodes}_k)\}]\!]_{\mathcal{M}} \triangleq \\
\quad fix\ f_d\ (x_1\ :\ \tau_{i_1})\ \ldots\ (x_m\ :\ \tau_{i_m})\ :\ \tau_o\ := \\
\qquad match\ (x_1,\ \ldots,\ x_m)\ with \\
\qquad |\ \mathsf{in}(d\ t_{11}\ \ldots\ t_{1p})\ \Rightarrow\ [\![\mathsf{Nodes}_1]\!]_{\mathcal{M}} \\
\qquad |\ \ldots \\
\qquad |\ \mathsf{in}(d\ t_{k1}\ \ldots\ t_{kp})\ \Rightarrow\ [\![\mathsf{Nodes}_k]\!]_{\mathcal{M}}
\end{array}
$$

This function generates the outermost pattern matching of the extracted function, and the generated code for each node is produced as follows:

$$
\begin{array}{l}
[\![\mathsf{Nodes}]\!]_{\mathcal{M}} \triangleq \\
\quad \left\{ \begin{array}{l}
\text{if } \mathsf{Nodes} \text{ is } d\ t_1\ \ldots\ t_p \text{ then } \mathsf{out}(d\ t_1\ \ldots\ t_p, m) \\
\text{if } \mathsf{Nodes} \text{ is } \{(T_{i1}, \mathsf{Nodes}_{i1}),\ \ldots,\ (T_{ik}, \mathsf{Nodes}_{ik})\} \text{ then} \\
\qquad match\ [\![T_{i1}]\!]_{\mathcal{M}}\ with \\
\qquad |\ \mathsf{out}(T_{i1}, \mathcal{M}(T_{i1}))\ \Rightarrow\ [\![\mathsf{Nodes}_{i1}]\!]_{\mathcal{M}} \\
\qquad |\ \ldots \\
\qquad |\ \mathsf{out}(T_{ik}, \mathcal{M}(T_{ik}))\ \Rightarrow\ [\![\mathsf{Nodes}_{ik}]\!]_{\mathcal{M}}
\end{array} \right.
\end{array}
$$

where $[\![T_{i1}]\!]_{\mathcal{M}}$ involves the function extracted from the logical inductive type upon which T_{i1} is built. If $T_{i1} = d_j\ t_1\ \ldots\ t_{p_j}$ then $[\![T_{i1}]\!]_{\mathcal{M}} = f_{d_j}\ \mathsf{in}(T_{i1}, \mathcal{M}(T_{i1}))$.

Using the code generation algorithm described above, we obtain the following function from the binary search tree example introduced in Section 2:

$$
\begin{array}{l}
fix\ search12\ (p1 : bst)\ (p2 : nat)\ :\ path\ := \\
\quad match\ (p1,\ p2) \\
\quad |\ (Empty,\ \_)\ \Rightarrow\ Not\_found \\
\quad |\ (Node\ t1\ m\ t2,\ n)\ \Rightarrow \\
\qquad (match\ compare12\ n\ m\ with \\
\qquad\ |\ Inf\ \Rightarrow\ (match\ search12\ t1\ p2\ with\ b\ \Rightarrow\ Left\ b) \\
\qquad\ |\ Sup\ \Rightarrow\ (match\ search12\ t2\ p2\ with\ b\ \Rightarrow\ Right\ b) \\
\qquad\ |\ Eq\ \Rightarrow\ End\_path)
\end{array}
$$

3.4 Extensions of the Code Generation

This section proposes two extensions of the code generation algorithm. The first one concerns a larger family of non-deterministic specifications, while the second one aims to deal with full modes and incomplete specifications.

Non-Deterministic Specifications It is possible to accept specifications with overlapping conclusions where the premises cannot help distinguish between them, but can be ordered using an order over the patterns defined as follows:

Definition 3 (Pattern Order). *Given two patterns t_1 and t_2, t_1 is more general than t_2, denoted by $t_1 > t_2$, iff the following property is verified:*

$$t_1 > t_2 \Leftrightarrow (t_1 = v \wedge t_2 = c_l\ t_1\ \ldots\ t_{p_l}) \vee$$
$$(t_1 = \_ \wedge t_2 = c_l\ t_1\ \ldots\ t_{p_l}) \vee$$
$$(t_1 = c_l\ t'_1\ \ldots\ t'_{p_l} \wedge t_2 = c_l\ t_1\ \ldots\ t_{p_l} \wedge$$
$$\exists i \in 1 \ldots p_l, t'_i > t_i \wedge\ \forall i \in 1 \ldots p_l, t'_i > t_i \vee t'_i = t_i)$$

It should be noted that in the generated code, some decisions are made according to this order, and completeness may therefore be lost, i.e. some possible outputs w.r.t. the specification cannot be computed by the extracted function.

To illustrate this extension of the code generation algorithm, let us consider an improvement of the binary search tree example introduced in Section 2, which consists in adding two constructors (in bold font) in the *search* logical inductive type in order to correctly propagate the *Not_found* value as follows:

```
Inductive search : bst → nat → path → Prop :=
  | Search_empty : forall n, search Empty n Not_found
  | Search_found : forall n m t1 t2, compare n m Eq →
      search (Node t1 m t2) n End_path
  | Search_inf : forall n m t1 t2 b, search t1 n b →
      compare n m Inf → search (Node t1 m t2) n (Left b)
  | Searchinf_nf : forall n m t1 t2, search t1 n Not_found →
      compare n m Inf → search (Node t1 m t2) n Not_found
  | Search_sup : forall n m t1 t2 b, search t2 n b →
      compare n m Sup → search (Node t1 m t2) n (Right b)
  | Searchsup_nf : forall n m t1 t2, search t2 n Not_found →
      compare n m Sup → search (Node t1 m t2) n Not_found.
```

As can be observed, the conclusions of *Search_inf* and *Searchinf_nf* overlap, and the two premises *search t1 n b* and *search t1 n Not_found* as well, but these premises can be ordered: *b* is more general than *Not_found*.

Considering this new kind of specifications requires some modifications in the rel-tree representation and consequently in the three related properties. The rel-tree representation is adapted to allow this ordering, and a rel-tree is defined by using lists of nodes instead of sets of nodes.

Definition 4 (Rel-Tree with Lists). *Given a logical inductive type d, it can be represented using the new following definition of rel-tree:*

$$\mathsf{Rel{-}Tree}([(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{k1}\ \ldots\ t_{kp}, \mathsf{Nodes}_k)])$$

where Nodes_i *is* $[(T_{i1}, \mathsf{Nodes}_{i1}),\ \ldots,\ (T_{ik}, \mathsf{Nodes}_{ik})]$ *or* $d\ t_{i1}\ \ldots\ t_{ip}$.

The previous properties SC, MCA, and NO must be adapted according to this new definition of rel-trees. In particular, in the property NO, we have to change the "not unifiable" statements by a new relation defined as follows:

Definition 5 (Pattern Relation). *Given two patterns t_1 and t_2, t_1 is more general than t_2 or non-unifiable with t_2, denoted by $t_1 \succ t_2$, iff the following property is verified:*

$$t_1 \succ t_2 \Leftrightarrow t_1 > t_2 \text{ or } t_1 \text{ and } t_2 \text{ are not unifiable}$$

As for the code generation algorithm from a rel-tree, it remains unchanged. Regarding the new specification of the binary search tree, the extracted function is then obtained as follows:

```
fix search12 (p1 : bst) (p2 : nat) : path :=
  match (p1, p2)
  | (Empty, _) ⇒ Not_found
  | (Node t1 m t2, n) ⇒
    (match compare12 n m with
     | Inf ⇒
       (match search12 t1 n with
        | Not_found ⇒ Not_found
        | b ⇒ Left b)
     | Sup ⇒
       (match search12 t2 n with
        | Not_found ⇒ Not_found
        | b ⇒ Right b)
     | Eq ⇒ End_path)
```

Full Modes and Incomplete Specifications In the code generation described previously, we only consider partial modes and complete specifications. We are also able to deal with full modes and incomplete specifications. In the case of a full mode, the output of the extracted function is a boolean: *true* when the relation between the arguments is verified, *false* otherwise. The code generation follows the same algorithm than previously. In addition, for each case where a constructor is not defined, we add to the generated function the default case "| _ → *false*". Regarding incomplete specifications, if we follow the previous code generation algorithm, it produces a partially defined function, which is not supported by the CIC type theory. The algorithm has therefore to be adapted to extract a function whose result type is an option type of T, where T is the type of the output. The code generation then follows the same algorithm than previously, but for each case where a constructor is not defined, we add to the generated function the default case "| _ → *None*", where *None* is the empty constructor of the option type.

4 Soundness Proof Generation

In the previous section, we have explained how to extract functions from logical inductive types. In addition, we want to automatically provide proofs of soundness for these functions. In the following, we will only consider proofs of soundness for extractions of complete specifications with partial modes. However, the principle of soundness proof generation can be generalized to the other cases. The theorem of soundness has the following form:

$$\forall p_1,\ \ldots,\ p_n, f_d\ p_1\ \ldots\ p_{n-1} = p_n \rightarrow d\ p_1\ \ldots\ p_n$$

where f_d is the name of the extracted function from the logical inductive type d with the mode $\{1,\ \ldots,\ n-1\}$.

We prove the previous theorem by automatically providing a Coq proof script, which performs a functional induction using the extracted function [1]. Actually, for any function, Coq generates a functional induction scheme, which follows precisely the execution paths of the function (see the scheme *search12_ind* generated for the function *search12* in Section 2). When applying the induction scheme to the goal representing the theorem of soundness to be proved, we get a subgoal for each execution path of the extracted function. It should be noted that in the code generation described previously, we only use a high-level pattern matching, which is automatically compiled into a low-level pattern matching. This compilation may introduce some code duplication, and some "*let-in*" constructs are introduced to avoid the duplication of recursive calls. In the following, we will consider extracted functions where this compilation will have been performed, and which will be allowed to involve "*let-in*" expressions.

4.1 Annotated Execution Paths

Before generating the proof script, we compute, from the generated code, the annotated execution paths, which correspond to the different cases of the functional induction scheme. An execution path is very similar to the target language used for the code generation, but it contains only one branch for each pattern matching. In the following, C will refer to the name of a constructor of a logical inductive type, while c will refer to a constructor of an inductive data type.

Definition 6 (Annotated Execution Path). *An annotated execution path is defined as follows:*

$$\begin{array}{l} b ::= \ t \mid let_l\ x\ :=\ t\ in\ b \\ \quad\ \mid\ match\ t\ with\ c\ x_1\ \ldots\ x_p\ \Rightarrow_l\ b \end{array}$$

where t is a term and l is an annotation which is either a set of constructor names $\{C_1,\ \ldots,\ C_n\}$, or a set of premise positions $\{(C_1, i_1),\ \ldots,\ (C_n, i_n)\}$, in which (C_i, i_k) denotes the i_k^{th} premise of constructor C_i.

This representation will help us generate the proof script because it contains information on both the generated code (and therefore the subgoal) and the specification (through the annotations). An annotation indicates the parts of the specification from whence the generated code comes. Thus, if (C, j) appears in the annotation l of a matching clause "$c\ x_1\ \ldots\ x_p\ \Rightarrow_l\ b$", then the constructor c appears in the premise $P(C, j)$. These annotated execution paths are generated from the extracted code, which is also annotated. The code generation algorithm is adapted to produce the annotations, which are initially computed from the specification and embedded in the rel-tree representation.

4.2 Proof Script of the Soundness Proof

As seen previously, we know that applying the functional induction scheme generates one subgoal per execution path of the extracted function. Each execution path is associated with one constructor in the specification, that we call C. One or more execution paths may be associated with the same constructor (due to the compilation of pattern matchings). Finally, it should be noted that we have an association between a constructor, an execution path, a case in the induction scheme, and a subgoal of the proof (once the induction scheme has been applied).

Each subgoal has the following form:

$$\forall \overrightarrow{v}, \forall \overrightarrow{v_1}, \overrightarrow{a_1} \rightarrow H_1 \rightarrow \ \ldots \forall \overrightarrow{v_k}, \overrightarrow{a_k} \rightarrow H_k \rightarrow \ \ldots \forall \overrightarrow{v_j}, \overrightarrow{a_j} \rightarrow H_j \rightarrow d\ t_1 \ldots t_p$$

where $\overrightarrow{v}$, $\overrightarrow{v_k}$, with k in $1 \ldots j$, are lists of variables, j the number of premises of the associated constructor C, $\overrightarrow{a_k}$ a list of equalities or "*let-in*" expressions corresponding to the associated annotated execution path, and H_k the soundness hypothesis if the logical inductive type involved in the premise, d_k, is d or an equality of the form $f_k\ w_1 \ldots w_u = r$, where f_k is the extracted function for d_k. We assume that the extraction has been already performed from d_k, and as a consequence, the theorem of soundness related to f_k and d_k is available.

Each subgoal is transformed by applying successive introductions and rewritings using the several a_{ki}, leading to a goal which is the conclusion of the constructor C (up to renaming), and where each premise is present in the context. When a_{ki} is a "*let-in*" expression, it is transformed into an equality, and a rewriting is performed. The annotated execution paths are used to determine from which premise the equalities and "*let-in*" expressions of the subgoal come from. Thanks to this information, we know how to rewrite the goal. Finally, we apply the constructor C and this nearly finishes the proof of the subgoal: the arguments of the constructor are either present in the hypothesis context, or must be proved using the theorems of soundness related to the logical inductive types (other than d) used in the constructor (like in the example of Section 2).

5 Implementation

We have implemented the extraction of logical inductive types within the Coq proof assistant as a plugin (not yet distributed, but available on demand by sending a mail to the authors). For information, another plugin (distributed since Coq version 8.4) allows the user to extract ML code from logical inductive types. In the short term, we plan to merge these two plugins.

With the current implementation, it is possible to extract specifications involving several logical inductive types, but there are some restrictions. First, the definitions must not be mutually recursive, and the extracted functions must rely on structural recursion. Moreover, logical inductive types must contain neither logical connectives ($\wedge$, $\vee$, or $\neg$), nor equality symbols. Finally, regarding proofs of soundness, we are only able to generate them for complete functions extracted with partial modes. These restrictions should be relaxed in the near future.

6 Conclusion

We have presented an operational approach allowing the extraction of computational content written as a Coq function from a Coq inductive specification. This extracted function is accompanied by a proof of soundness establishing that the result of the function complies with the specification. Future work will consist in completing the proof generation: generating soundness proof for the other kinds of modes and specifications, and also (when it is relevant) producing completeness proofs. The former is just an adaptation of the approach presented here (i.e. a functional induction exploiting annotations produced during the code generation), while the latter requires a different proof generation scheme. The next step will be to address inductive specifications embedding a general recursion. Finally, we could also try to extract functions from non-terminating specifications (e.g. the semantics of a language featuring a while loop), expressed as mixed inductive-coinductive definitions (see [8] for some examples). A simple approach would consist in adding to the extracted function a non-negative integer counter which bounds the depth of the computation (as done in CompCert [5]).

References

1. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2410 of *LNCS*, pages 31–46, Hampton (VA, USA), Aug. 2002. Springer.
2. S. Berghofer, L. Bulwahn, and F. Haftmann. Turning Inductive into Equational Specifications. In *Theorem Proving in Higher Order Logic (TPHOLs)*, volume 5674 of *LNCS*, pages 131–146, Munich (Germany), Aug. 2009. Springer.
3. S. Berghofer and T. Nipkow. Executing Higher Order Logic. In *Types for Proofs and Programs (TYPES)*, volume 2277 of *LNCS*, pages 24–40, Durham (UK), Dec. 2000. Springer.
4. Y. Bertot, V. Capretta, and K. Das Barman. Type-Theoretic Functional Semantics. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2410 of *LNCS*, pages 83–98, Hampton (VA, USA), Aug. 2002. Springer.
5. S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning (JAR)*, 43(3):263–288, Oct. 2009.
6. D. Delahaye, C. Dubois, and J.-F. Étienne. Extracting Purely Functional Contents from Logical Inductive Types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 70–85, Kaiserslautern (Germany), Sept. 2007. Springer.
7. F. Le Fessant and L. Maranget. Optimizing Pattern-Matching. In *International Conference on Functional Programming (ICFP)*, SIGPLAN, pages 26–37, Florence (Italy), Sept. 2001. ACM.
8. X. Leroy and H. Grall. Coinductive Big-Step Operational Semantics. *Information and Computation (IC)*, 207(2):284–304, Feb. 2009.
9. F. Pfenning and C. Schürmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Conference on Automated Deduction (CADE)*, volume 1632 of *LNCS*, pages 202–206, Trento (Italy), jul 1999. Springer.
10. The Coq Development Team. *Coq, version 8.4*. INRIA, Aug. 2012. http://coq.inria.fr/.