Extracting Purely Functional Contents from Logical Inductive Types

David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne

CEDRIC/CNAM-ENSIIE, Paris, France
David.Delahaye@cnam.fr, dubois@ensiie.fr, etien_je@auditeur.cnam.fr

Abstract. We propose a method to extract purely functional contents from logical inductive types in the context of the Calculus of Inductive Constructions. This method is based on a mode consistency analysis, which verifies if a computation is possible w.r.t. the selected inputs/outputs, and the code generation itself. We prove that this extraction is sound w.r.t. the Calculus of Inductive Constructions. Finally, we present some optimizations, as well as the implementation designed in the Coq proof assistant framework.

1 Introduction

The main idea underlying extraction in proof assistants like Isabelle or Coq is to automatically produce certified programs in a correct by construction manner. More formally it means that the extracted program realizes its specification. Programs are extracted from types, functions and proofs. Roughly speaking, the extracted program only contains the computational parts of the initial specification, whereas the logical parts are skipped. In Coq, this is done by analyzing the types: types in the sort *Set* (or *Type*) are computationally relevant while types in sort *Prop* are not. Consequently, inductively defined relations, implemented as Coq logical inductive types, are not considered by the extraction process because they are exclusively dedicated to logical aspects. However such constructs are widely used to describe algorithms. For example, when defining the semantics of a programming language, the evaluation relation embeds the definition of an interpreter.

Although inductive relations are not executable, they are often preferred because it is often easier to define a relational specification than its corresponding functional counterpart involving pattern-matching and recursion. For example, in Coq, it is easier to define the relation "the terms t and u unify with s as a most general unifier" than defining the function that computes the most general unifier of t and u if it exists. In this case, the difficulty is to prove the termination of the function while simultaneously defining it. Moreover, proof assistants offer many tools to reason about relational specifications (e.g. elimination, inversion tactics) and the developer may prefer the relational style rather than the functional style, even though recent work (e.g. in Coq, functional induction or recursive definition) provide a better support for defining functions and reasoning about them.

Based on these observations, our aim is to translate logical inductive specifications into functional code, with an intention close to the one found in the Centaur [3] project, that is extracting tools from specifications. Another motivation for extracting code from logical inductive specifications is to get means to execute these specifications (if executable), in order to validate or test them. Better still, in a formal testing framework, the extracted code could be used as an oracle to test a program independently written from the specification.

In this paper, we propose a mechanism to extract functional programs from logical inductive types. Related work has been done on this subject around semantics of programming languages, for example [3,8,1,4,12] and [2] in a more general setting. Like [2], our work goes beyond semantic applications, even though it is a typical domain of applications for such a work. In addition, our extraction is intended to only deal with logical inductive types that can be turned into purely functional programs. This is mainly motivated by the fact that proof assistants providing an extraction mechanism generally produce code in a functional framework. Thus, we do not want to manage logical inductive specifications that would require backtracking, that is more a Prolog-like paradigm. In that sense, our work separates from Centaur [3], Petterson's RML translator [8] and Berghofer and Nipkow's approach [2]. The first one translates Typol specifications, which are inductively defined semantic relations, into Prolog programs. RML is also a formalism to describe natural semantics of programming languages, and the corresponding compiler produces C programs that may backtrack if necessary. Finally, Berghofer and Nipkow's tool produces code that can compute more than one solution, if any. We can also mention the use of Maude [12] to animate executable operational semantic specifications, where these specifications are turned into rewriting systems.

To turn an inductive relation into a function that can compute some results, we need additional information. In particular, we need to know which arguments are inputs and which arguments are outputs. This information is provided by the user using the notion of modes. Furthermore, these modes are used to determine if a functional computation is possible, in which case we say that the mode is consistent. Otherwise, the functional extraction is not possible and is rejected by our method. In order to make functional computations possible, some premises in the types of constructors may have to be reordered. The notion of mode, going back actually to attribute grammars [1], is fairly standard, especially in the logical programming community. For example, the logical and functional language Mercury [7] requires mode declarations to produce efficient code. Similar mode systems have already been described in [4,2,9].

The paper is organized as follows: first, we informally present our extraction mechanism with the help of some examples; next, we formalize the extraction method itself (in particular, the mode consistency analysis and the code generation) and prove its soundness; finally, we describe our prototype developed in the Coq proof assistant framework and discuss some optimizations.

2 Informal Presentation

In this section, we present how our functional extraction must work on some examples. For these examples, we use the Coq framework with, in particular, its syntax and OCaml [11] as one of its target languages for extraction. Our general approach is the following:

1. the user annotates his/her logical inductive type with a mode that specifies which arguments are inputs, the others being considered as outputs;
2. a mode consistency analysis is performed to determine if the extraction is possible w.r.t. the provided mode;
3. if the previous analysis is successful, the logical definition is translated into a functional program.

This process may be recursive and may call the regular extraction mechanism to extract code from functions or proofs.

A mode can be seen as the computational behavior of a logical inductive type. It is defined as a set of indices denoting the inputs of the relation. For example, let us consider the predicate add that specifies the addition of two natural numbers, i.e. given three natural numbers n, m and p, (add n m p) defines that p is the result of the addition of n and m. This predicate is defined as follows:

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

The mode $\{1, 2\}$ indicates that we consider n and m as inputs and we would like to compute p. The extracted function is the expected one, defined by pattern-matching on both arguments (actually only the second one is significant):

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m → let p = add n m in S p
  | _ → assert false
```

We can also propose to extract a function with the mode $\{2, 3\}$. Thus, we obtain a function that performs subtraction:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | S p, S m → add p m
  | _ → assert false
```

Finally the mode $\{1, 2, 3\}$ means that the three arguments are known and that we want to produce a function that checks if the triple constitutes a possible computation or not (as a boolean result):

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m  when n = m→ true
  | n, S m, S p→ add n m p
  | _→ false
```

However, with the mode $\{1, 3\}$, the extraction is refused, not because of the mode analysis (which succeeds) but because it would produce a function with two overlapping patterns, (n,n) (obtained from the type of the first constructor addO) and (n,p) (obtained from the type of the second constructor addS). With such a configuration, more than one result might be computed and therefore the function would not be deterministic, which is incompatible with a proper notion of function. Extraction with modes involving only one input are refused for the same reason.

As a last example, let us consider a case where constraints are put on the results, e.g. in the eval_plus constructor the evaluation of a1 and a2 must map to values built from the N constructor:

```
Inductive Val  : Set := N : Z→ Val | ...
Inductive Expr: Set := V : Var→Expr | Plus : Expr→Expr→Expr.

Inductive eval : Sigma→ Expr→ Val→ Prop :=
  | eval_v : forall (s : Sigma) (v : Var), eval s (V v) (valof s v)
  | eval_plus : forall (s : Sigma) (a1 a2 : Expr) (v w : Z),
      eval s a1 (N v)→ eval s a2 (N w)→ eval s (Plus a1 a2)(N (v + w)).
```

where Sigma is an evaluation context and valof is a function looking for the value of a variable in an evaluation context.

With the mode $\{1, 2\}$, the extracted function is the following:

```
let rec eval s e = match s, e with
  | s, V v→ valof s v
  | s, Plus (a1, a2)→
    (match eval s a1 with
       | N v→
         (match eval s a2 with
            | N w→ N (zplus v w)
            | _→ assert false)
       |_→ assert false)
```

where valof and zplus are respectively the extracted functions from the definitions of valof and the addition over Z.

Regarding mode consistency analysis, detailed examples will be given in the next section.

3 Extraction of Logical Inductive Types

The extraction is made in two steps: first, the mode consistency analysis tries to find a permutation of the premises of each inductive clause, which is compatible w.r.t. the annotated mode; second, if the mode consistency analysis has been successful, the code generation produces the executable functional program. Before describing the extraction method itself, we have to specify which inductive types we consider and in particular, what we mean exactly by logical inductive types. We must also precise which restrictions we impose, either to ensure a purely functional and meaningful extraction, or to simplify the presentation of this formalization, while our implementation relaxes some of these restrictions (see Section 5).

3.1 Logical Inductive Types

The type theory we consider is the Calculus of Inductive Constructions (CIC for short; see the documentation of Coq [10] to get some references regarding the CIC), i.e. the Calculus of Constructions with inductive definitions. This theory is probably too strong for what we want to show in this paper, but it is the underlying theory of the Coq proof assistant, in which we chose to develop the corresponding implementation. An inductive definition is noted as follows (inspired by the notation used in the documentation of Coq):

$$\mathsf{Ind}(d : \tau, \Gamma_c)$$

where d is the name of the inductive definition, τ a type and Γ_c 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[1] and mutual inductive definitions. Actually, these features do not involve specific technical difficulties. Omitting them allows us to greatly simplify the presentation of the extraction, as well as the soundness proof in particular. Also for simplification reasons, dependencies, 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:

$$\tau_1 \to \ldots \tau_n \to \mathsf{Prop}$$

where τ_i is of type Set or Type, and does not contain any product or dependent inductive type. In addition, we suppose that the types of constructors are in prenex form, with no dependency between the bounded variables and no higher order; thus, the type of a constructor is as follows:

$$\textstyle\prod_{i=1}^{n} x_i : X_i.T_1 \to \ \ldots \ \to T_m \to (d\ t_1\ \ldots\ t_p)$$

where $x_i \notin X_j$, X_i is of type Set or Type, T_i is of type Prop and does not contain any product or dependent inductive type, and t_i are terms. In the following, the terms T_i will be called the premises of the constructor, whereas the term $(d\ t_1\ \ldots\ t_p)$ will be called the conclusion of the constructor. We impose the additional constraint that T_i is a fully applied logical inductive type, i.e. T_i has the following form:

[1] In CIC, parameters are additional arguments, which are shared by the type of the inductive definition (type τ) and the types of constructors (defined in Γ_c).

$d_i\ t_{i1}\ \ldots\ t_{ip_i}$

where d_i is a logical inductive type, t_{ij} are terms, and p_i is the arity of d_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.

3.2 Mode Consistency Analysis

The purpose of the mode consistency analysis is to check whether a functional execution is possible. It is a very simple data-flow analysis of the logical inductive type. We require the user to provide a mode for the considered logical inductive type, and recursively for each logical inductive type occurring in this type.

Given a logical inductive type I, a mode md is defined as a set of indices denoting the inputs of I. The remaining arguments are the output arguments. Thus, $md \subseteq \{1, \ldots, a_I\}$, where a_I is the arity of I. Although a mode is defined as a set, the order of the inputs in the logical inductive type is relevant. They will appear in the functional translation in the same order.

In practice, it is often the case to use the extraction with, either a mode corresponding to all the arguments except one, called the computational mode, or a mode indicating that all the arguments are inputs, called the fully instantiated mode. The formalization below will essentially deal with these two modes with no loss of generality (if more than one output is necessary, we can consider that the outputs are gathered in a tuple).

In order to make functional computations possible, some premises in a constructor may have to be reordered. It will be the case when a variable appears first in a premise as an input and as an output in another premise written afterwards. For a same logical inductive type, some modes may be possible whereas some others may be considered inconsistent. Different mode declarations give different extracted functions.

Given $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, a mode md is consistent for I w.r.t. $\mathcal{M}$ iff it is consistent for Γ_c (i.e. all the constructors of I) w.r.t. $\mathcal{M}$. A mode md is consistent for the constructor c of type $\Pi_{i=1}^{n} x_i : X_i.T_1 \to \ldots \to T_n \to T$ w.r.t. $\mathcal{M}$, where $T = d\,t_1\ \ldots\ t_p$, iff there exist a permutation π and the sets of variables S_i, with $i = 0 \ldots m$, s.t.:

1. $S_0 = \mathsf{in}(md, T)$;
2. $\mathsf{in}(m_{\pi j}, T_{\pi j}) \subseteq S_{j-1}$, with $1 \leq j \leq m$ and $m_{\pi j} = \mathcal{M}(\mathsf{name}(T_{\pi j}))$;
3. $S_j = S_{j-1} \cup \mathsf{out}(m_{\pi j}, T_{\pi j})$, with $1 \leq j \leq m$ and $m_{\pi j} = \mathcal{M}(\mathsf{name}(T_{\pi j}))$;
4. $\mathsf{out}(md, T) \subseteq S_m$.

where $\mathsf{name}(t)$ is the name of the logical inductive type applied in the term t (e.g. $\mathsf{name}(add\ n\ (S\ m)\ (S\ k)) = add$), $\mathsf{in}(m, t)$ the set of variables occurring in the terms designated as inputs by the mode m in the term t (e.g. $\mathsf{in}(\{1, 2\}, (add\ n\ (S\ m)\ (S\ k))) = \{n, m\}$), and $\mathsf{out}(m, t)$ the set of variables occurring in the terms designated as outputs by the mode m in the term t (e.g. $\mathsf{out}(\{1, 2\}, (add\ n\ (S\ m)\ (S\ k))) = \{k\}$).

The permutation π denotes a suitable execution order for the premises. The set S_0 denotes the initial set of known variables and S_j the set of known variables after the execution of the πj^{th} premise (when $T_{\pi 1}$, $T_{\pi 2}$, $\ldots$, $T_{\pi j}$ have been executed in this order). The first condition states that during the execution of T (for the constructor c) with the mode md, the values of all the variables used in the terms designated as inputs have to be known. The second condition requires that the execution of a premise T_i with a mode m_i will be performed only if the input arguments designated by the mode are totally computable. It also requires that m_i is a consistent mode for the logical inductive type related to T_i (we have constrained T_i in the previous section to be a fully applied logical inductive type). According to the third condition, all the arguments of a given premise T_i are known after its execution. Finally, a mode md is said to be consistent for c w.r.t. $\mathcal{M}$ if all the arguments in the conclusion of c (i.e. T) are known after the execution of all the premises.

We have imposed some restrictions on the presence of functions in the terms appearing in the type of a constructor. To relax these conditions, such as accepting functions in the output of the premises (see Section 5), in the step j, we should verify that the function calls are computable, that is to say their arguments only involve known variables (belonging to S_{j-1}).

To illustrate the mode consistency analysis, let us consider the logical inductive type that specifies the big step semantics of a small imperative language. The evaluation of commands is represented by the relation $s \vdash_c i : s'$, which means that the execution of the command i in the store s leads to the final store s'. This relation has the type *store* $\rightarrow$ *command* $\rightarrow$ *store* $\rightarrow$ *Prop*, where *store* and *command* are the corresponding types for stores and imperative commands. For example, the types of the constructors for the *while* loop are the following:

$$\begin{aligned}
&\mathit{while}_1 : (s \vdash_e b : \mathit{true}) \rightarrow (s \vdash_c i : s') \rightarrow (s' \vdash_c \mathit{while}\ b\ \mathit{do}\ i : s'') \rightarrow \\
&\qquad\qquad (s \vdash_c \mathit{while}\ b\ \mathit{do}\ i : s'') \\
&\mathit{while}_2 : (s \vdash_e b : \mathit{false}) \rightarrow (s \vdash_c \mathit{while}\ b\ \mathit{do}\ i : s)
\end{aligned}$$

where given a store s, an expression t and a value v, $s \vdash_e t : v$ represents the evaluation of expressions, i.e. the expression t evaluates to v in the store s. For clarity reasons, we do not indicate the universally quantified variables, which are the stores s, s' and s'', the expression b and the command i.

If the mode $\{1, 2\}$ is consistent for $\vdash_e$ then the mode $\{1, 2\}$ is consistent for both *while* constructors of $\vdash_c$. But considering the typing relation of the simply typed λ-calculus $\Gamma \vdash t : \tau$, denoting that t is of type τ in context Γ and where Γ is a typing context, t a term and τ a type, the mode $\{1, 2\}$ is not consistent for this relation. Actually, this mode is not consistent for the typing of the abstraction; the type τ_1 in the premise, considered here as an input, is not known at this step ($\tau_1 \notin S_0$):

$$\mathit{abs} : (\Gamma, (x : \tau_1) \vdash e : \tau_2) \rightarrow (\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2)$$

In the following, we will ignore the permutation of the premises and will assume that all the premises are correctly ordered w.r.t. to the provided mode.

3.3 Code Generation

The functional language we consider as target for our extraction is defined as follows (mainly a functional core with recursion and pattern-matching):

$$\begin{array}{lll} e & ::= & x \mid c^n \mid C^n \mid \mathsf{fail} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1\ e_2 \mid \mathsf{fun}\ x \rightarrow e \\ & \mid & \mathsf{rec}\ f\ x \rightarrow e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e_1, \dots, e_n) \\ & \mid & \mathsf{match}\ e\ \mathsf{with} \mid gpat_1 \rightarrow e_1\ \dots \mid gpat_n \rightarrow e_n \end{array}$$

$$gpat ::= pat \mid pat\ \mathsf{when}\ e$$

$$pat\ ::= x \mid C^n \mid C^n\ pat \mid (pat_1, \dots, pat_n) \mid \_$$

where c^n is a constant of arity n, to be distinguished from C^n, a constructor of arity n. Both constants and constructors are uncurrified and fully applied. Moreover, we use the notations $e_1\ e_2\ \dots\ e_n$ for $((e_1\ e_2) \dots e_n)$, and $\mathsf{fun}\ x_1\ \dots\ x_n \rightarrow e$ for $\mathsf{fun}\ x_1 \rightarrow\ \dots \mathsf{fun}\ x_n \rightarrow e$ (as well as for rec functions).

Given $I = \mathsf{Ind}(d : \tau, \Gamma_c)$, a logical inductive type, well-typed in a context Γ, and $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, we consider that each logical inductive type is ordered according to its corresponding mode (with the output, if required, at the last position), i.e. if the mode of a logical inductive type J is $\mathcal{M}(J) = \{n_1, \dots, n_J\}$, then the n_1^{th} argument of J becomes the first one and so on until the n_J^{th} argument, which becomes the c_J^{th} one, with $c_J = \mathsf{card}(\mathcal{M}(J))$. The code generation for I, denoted by $[\![I]\!]_{\Gamma,\mathcal{M}}$, begins as follows (we do not translate the type τ of I in Γ since this information is simply skipped in the natural semantics we propose for the target functional language in Section 4):

$$[\![I]\!]_{\Gamma,\mathcal{M}} = \begin{cases} \mathsf{fun}\ p_1\ \dots\ p_{c_I} \rightarrow [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, & \text{if } d \notin \Gamma_c, \\ \mathsf{rec}\ d\ p_1\ \dots\ p_{c_I} \rightarrow [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, & \text{otherwise} \end{cases}$$

where $c_I = \mathsf{card}(\mathcal{M}(I))$ and $P = \{p_1, \dots, p_{c_I}\}$.

Considering $\Gamma_c = \{c_1, \dots, c_n\}$, the body of the function is the following:

$$\begin{array}{rl} [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P} = & \mathsf{match}\ (p_1, \dots, p_{c_I})\ \mathsf{with} \\ & \mid [\![c_1]\!]_{\Gamma,\mathcal{M}} \\ & \mid \dots \\ & \mid [\![c_n]\!]_{\Gamma,\mathcal{M}} \\ & \mid \_ \ \rightarrow\ \mathsf{default}_{I,\mathcal{M}} \end{array}$$

where $\mathsf{default}_{I,\mathcal{M}}$ is defined as follows:

$$\mathsf{default}_{I,\mathcal{M}} = \begin{cases} \mathsf{false}, & \text{if } c_I = a_I, \\ \mathsf{fail}, & \text{if } c_I = a_I - 1 \end{cases}$$

where $c_I = \mathsf{card}(\mathcal{M}(I))$ and a_I is the arity of I.

The translation of a constructor c_i is the following (the name of c_i is skipped and only its type is used in this translation):

$$\llbracket c_i \rrbracket_{\Gamma,\mathcal{M}} = \llbracket \prod_{j=1}^{n_i} x_{ij} : X_{ij}.T_{i1} \to \ldots \to T_{im_i} \to (d\ t_{i1}\ \ldots\ t_{ip_i}) \rrbracket_{\Gamma,\mathcal{M}}$$
$$= \begin{cases} (\llbracket t_{i1} \rrbracket, \ldots, \llbracket t_{ic_I} \rrbracket) \to \llbracket T_{i1} \to \ldots \to T_{im_i} \rrbracket_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}}, \\ \quad \text{if } t_{i1}, \ldots, t_{ic_I} \text{ are linear,} \\ (\llbracket \sigma_i(t_{i1}) \rrbracket, \ldots, \llbracket \sigma_i(t_{ic_I}) \rrbracket) \ \mathsf{when}\ \mathsf{guard}(\sigma_i) \to \\ \quad \llbracket T_{i1} \to \ldots \to T_{im_i} \rrbracket_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}}, \text{ otherwise} \end{cases}$$

where σ_i is a renaming s.t. $\sigma_i(t_{i1}), \ldots, \sigma_i(t_{ic_I})$ are linear, $\mathsf{guard}(\sigma_i)$ is the corresponding guard, of the form $x_{ij_1} = \sigma_i(x_{ij_1})$ and $\ldots$ and $x_{ij_k} = \sigma_i(x_{ij_k})$, with $\mathsf{dom}(\sigma_i) = \{x_{ij_1}, \ldots, x_{ij_k}\}$ and $1 \le j_l \le n_i$, $l = 1 \ldots k$, and $\mathsf{cont}_{I,\mathcal{M}}$ is defined as follows:

$$\mathsf{cont}_{I,\mathcal{M}} = \begin{cases} \mathsf{true}, \text{ if } c_I = a_I, \\ \llbracket t_{ip_i} \rrbracket, \text{ if } c_I = a_I - 1 \end{cases}$$

The terms $t_{i1}, \ldots, t_{ic_I}$ must only contain variables and constructors, while t_{ip_i} (if $c_I = a_I - 1$) can additionally contain symbols of functions. The corresponding translation is completely isomorphic to the structure of these terms and uses the regular extraction, presented in [6]. Moreover, we consider that the terms t_{ik} and t_{jk} are not unifiable, for $i, j = 1 \ldots n$ and $k = 1 \ldots c_I$ (otherwise, a functional extraction may not be possible, i.e. backtracking is necessary or there are several results).

The right-hand side of the pattern rules is generated with the following scheme:

$$\llbracket T_{ij} \to \ldots \to T_{im_i} \rrbracket_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} =$$
$$\begin{cases} \mathsf{cont}_{I,\mathcal{M}}, \text{ if } j > m_i, \\ \mathsf{if}\ \llbracket d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}} \rrbracket\ \mathsf{then}\ \llbracket T_{i(j+1)} \to \ldots \to T_{im_i} \rrbracket_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mathsf{else}\ \mathsf{default}_{I,\mathcal{M}}, \text{ if } j \le m_i \text{ and } c_{ij} = a_{ij}, \\ \mathsf{match}\ \llbracket d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}} \rrbracket\ \mathsf{with} \\ \mid \llbracket t_{ija_{ij}} \rrbracket \to \llbracket T_{i(j+1)} \to \ldots \to T_{im_i} \rrbracket_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mid \_ \to\ \mathsf{default}_{I,\mathcal{M}}, \text{ if } j \le m_i \text{ and } c_{ij} = a_{ij} - 1 \end{cases}$$

where $T_{ij} = d_{ij}\ t_{ij1} \ldots t_{ija_{ij}}$, a_{ij} is the arity of d_{ij} and $c_{ij} = \mathsf{card}(\mathcal{M}(\Gamma(d_{ij})))$. We consider that the term $t_{ija_{ij}}$ is linear and does not contain computed variables or symbols of functions when $c_{ij} = a_{ij} - 1$ (in this way, no guard is required in the produced pattern).

4 Soundness of the Extraction

Before proving the soundness of the extraction of logical inductive types, we need to specify the semantics of the functional language we chose as a target for our extraction and presented in Section 3. To simplify, we adopt a pure Kahn style big step natural semantics (with call by value) and we introduce the following notion of value:

$$v ::= c^0 \mid C^0 \mid \Delta_C \mid \mathsf{fail} \mid <x \leadsto e, \Delta> \mid <x \leadsto e, \Delta>_{\mathsf{rec}(f)} \mid (v_1, \ldots, v_n)$$

where Δ is an evaluation context, i.e. a list of pairs (x, v), and Δ_C is a set of values of the form $C^n\ (v_1, \ldots, v_n)$. In addition, we introduce a set Δ_c of tuples of the form $(c^n, v_1, \ldots, v_n, v)$, with $n > 0$.

An expression e evaluates to v in an environment Δ, denoted by the judgment $\Delta \vdash e \rhd v$, if and only if there exists a derivation of this judgment in the system of inference rules described in Appendix 6 (to save space, we do not include the corresponding error rules returning fail).

Given I, a logical inductive type, well-typed in a context Γ, and $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, we introduce the translation (and evaluation) of the context Γ w.r.t. the logical inductive type I and the set of modes $\mathcal{M}$. A context is a set of assumptions $(x : \tau)$, definitions $(x : \tau := t)$ and inductive definitions $\mathsf{Ind}(d : \tau, \Gamma_c)$, where x and d are names, τ a type, t a term and Γ_c the set of constructors. The translation of Γ w.r.t. I and $\mathcal{M}$, denoted by $[\![\Gamma]\!]_{I,\mathcal{M}}$, consists in extracting and evaluating recursively each assumption, definition or inductive definition occurring in I (thus, this translation provides an evaluation context). For assumptions, definitions and inductive definitions which are not logical inductive types, we use the regular extraction, presented in [6]. Regarding logical inductive types, we apply the extraction described previously in Section 3.

The soundness of our extraction is expressed by the following theorem:

Theorem (Soundness). *Given $I = \mathsf{Ind}(d : \tau, \Gamma_c)$, a logical inductive type, well-typed in a context Γ, and $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, we have the two following cases:*

- $c_I = a_I$: *if* $[\![\Gamma]\!]_{I,\mathcal{M}} \vdash [\![I]\!]_{\Gamma,\mathcal{M}}\ [\![t_1]\!] \ldots [\![t_{c_I}]\!] \rhd \mathsf{true}$ *then the statement* $\Gamma \vdash d\ t_1 \ldots t_{c_I}$ *is provable;*
- $c_I = a_I - 1$: *if* $[\![\Gamma]\!]_{I,\mathcal{M}} \vdash [\![I]\!]_{\Gamma,\mathcal{M}}\ [\![t_1]\!] \ldots [\![t_{c_I}]\!] \rhd v \neq \mathsf{fail}$ *then there exists* t *s.t.* $[\![t]\!] = v$ *and the statement* $\Gamma \vdash d\ t_1 \ldots t_{c_I}\ t$ *is provable.*

where $c_I = \mathsf{card}(\mathcal{M}(I))$, a_I *is the arity of* I, *and* $t_1 \ldots t_{c_I}$, t *are terms.*

Proof. The theorem is proved by induction over the extraction. We suppose that $\Delta \vdash [\![I]\!]_{\Gamma,\mathcal{M}}\ [\![t_1]\!] \ldots [\![t_{c_I}]\!] \rhd v$, with $\Delta = [\![\Gamma]\!]_{I,\mathcal{M}}$ and either $v = \mathsf{true}$ if $c_I = a_I$, or $v \neq \mathsf{fail}$ if $c_I = a_I - 1$. Using the definition of $[\![I]\!]_{\Gamma,\mathcal{M}}$ given in Section 3 and the rules of Appendix 6, this expression is evaluated as follows:

$$\Delta \vdash [\![I]\!]_{\Gamma,\mathcal{M}} \rhd \begin{cases} <p_1\ \ldots\ p_{c_I} \leadsto [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, \Delta>, \text{ if } d \notin \Gamma_c, \\ <p_1\ \ldots\ p_{c_I} \leadsto [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, \Delta>_{\mathsf{rec}(d)} = c, \text{ otherwise} \end{cases}$$

where $P = \{p_1, \ldots, p_{c_I}\}$.

The arguments are also evaluated: $\Delta \vdash [\![t_1]\!] \rhd v_1, \ldots,\ \Delta \vdash [\![t_{c_I}]\!] \rhd v_{c_I}$. Using the definition of $[\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}$, we have the following evaluation:

$$\Delta_b \vdash \begin{pmatrix} \mathsf{match}\ (v_1, \ldots, v_{c_I})\ \mathsf{with} \\ \mid [\![c_1]\!]_{\Gamma,\mathcal{M}} \\ \mid \ldots \\ \mid [\![c_n]\!]_{\Gamma,\mathcal{M}} \\ \mid \_ \ \rightarrow\ \mathsf{default}_{I,\mathcal{M}} \end{pmatrix} \rhd v$$

where Δ_b is defined as follows:

$$\Delta_b = \begin{cases} \Delta, (p_1, v_1), \ldots, (p_{c_I}, v_{c_I}), \text{ if } d \notin \Gamma_c, \\ \Delta, (d, c), (p_1, v_1), \ldots, (p_{c_I}, v_{c_I}), \text{ otherwise} \end{cases}$$

We know that either $v = \mathsf{true}$ or $v \neq \mathsf{fail}$ (according to c_I); this means that there exists i s.t. the pattern of $[\![c_i]\!]_{\Gamma,\mathcal{M}}$ matches the value $(v_1, \ldots, v_{c_I})$. Using the definition of $[\![c_i]\!]_{\Gamma,\mathcal{M}}$, we have to evaluate:

$$\Delta_p \vdash [\![T_{i1} \rightarrow \ \ldots \ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v \tag{1}$$

with $\Delta_p = \Delta_b, \Delta_i$, where Δ_i has the following form (by definition of $\mathsf{filter}_{\Delta_b}$):

$$\Delta_i = \begin{cases} \mathsf{mgu}_{\Delta_b}(v_t, ([\![t_{i1}]\!], \ldots, [\![t_{ic_I}]\!])), \text{ if } t_{i1}, \ldots, t_{ic_I} \text{ are linear} \\ \mathsf{mgu}_{\Delta_b}(v_t, ([\![\sigma_i(t_{i1})]\!], \ldots, [\![\sigma_i(t_{ic_I}))), \text{ otherwise} \end{cases}$$

where $v_t = (v_1, \ldots, v_{c_I})$. In addition, we have $\Delta_p \vdash \mathsf{guard}(\sigma_i) \triangleright \mathsf{true}$ if $t_{i1}, \ldots, t_{ic_I}$ are not linear.

The reduction of our extraction language is weaker than the one defined for CIC; in particular, this means that we have: given $\Delta = [\![\Gamma]\!]_{I,\mathcal{M}}$, a term t and a value $v \neq \mathsf{fail}$, if $\Delta \vdash [\![t]\!] \triangleright v$ then there exists a term t' s.t. $[\![t']\!] = v$ and $\Gamma \vdash t \equiv t'$, where $\equiv$ is the convertibility relation for CIC. Moreover, considering $\Delta = [\![\Gamma]\!]_{I,\mathcal{M}}$, a term t and a value $v \neq \mathsf{fail}$, if $\sigma = \mathsf{mgu}_{\Delta}(v, [\![t]\!])$ then there exist t' and $\bar{\sigma}$ s.t. $[\![t']\!] = v$, $\mathsf{dom}(\bar{\sigma}) = \mathsf{dom}(\sigma)$, $[\![\bar{\sigma}(x)]\!] = \sigma(x)$ for all $x \in \mathsf{dom}(\bar{\sigma})$, and $\bar{\sigma} = \mathsf{mgu}_{\Gamma}(t', t)$. Using these two remarks, there exists $\bar{\Delta}_i$ as described above s.t.:

$$\Gamma \vdash \bar{\Delta}_i(d \ t_{i1} \ldots t_{ic_I}) \equiv (d \ t_1 \ldots t_{c_I}) \tag{2}$$

Note that we can consider $\Delta_b = \Delta$, since the variables p_i, $i = 1 \ldots c_I$, do not occur in I; actually, these variables are just used for the curryfication of $[\![I]\!]_{\Gamma,\mathcal{M}}$. Note also that this unification between $(d \ t_{i1} \ldots t_{ic_I})$ and $(d \ t_1 \ldots t_{c_I})$ may be total or partial according to c_I (if $c_I = a_I$ or $c_I = a_I - 1$).

Regarding the arguments of c_i, we have to consider another property: given a context Δ_a, if $\Delta_p, \Delta_a \vdash [\![T_{i1} \rightarrow \ \ldots \ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v$ then there exists a context $\Delta'_a \supseteq \Delta_a$ s.t. $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ij}$ is provable for $j = 1 \ldots m_i$, and $\Delta_p, \Delta'_a \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$. This property is proved by induction over the product type. Using the definition of $[\![T_{i1} \rightarrow \ \ldots \ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M}}$, we have three cases:

- $j > m_i$: $\Delta_p, \Delta_a \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$ and $\Delta'_a = \Delta$.
- $j \leq m_i$, $c_{ij} = a_{ij}$:

$$\Delta_p, \Delta_a \vdash \begin{pmatrix} \mathsf{if}\ [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!]\ \mathsf{then} \\ \quad [\![T_{i(j+1)} \rightarrow \ \ldots \ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mathsf{else\ default}_{I,\mathcal{M}} \end{pmatrix} \triangleright v$$

  Since either $v = \mathsf{true}$ or $v \neq \mathsf{fail}$, we have $\Delta_p, \Delta_a \vdash [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!] \triangleright \mathsf{true}$ and the then branch is selected. By hypothesis of induction (over the soundness theorem), this means that $\Gamma \vdash \bar{\Delta}_a \bar{\Delta}_i T_{ij}$ is provable. Next, we have the evaluation $\Delta_p, \Delta_a \vdash [\![T_{i(j+1)} \rightarrow \ \ldots \ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v$ and by hypothesis of induction, there exists $\Delta'_a \supseteq \Delta_a$ s.t. $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ik}$ is provable for

$k = j+1 \dots m_i$, and $\Delta_p, \Delta'_a \vdash \mathsf{cont}_{I,\mathcal{M}} \rhd v$. As $\Delta'_a \supseteq \Delta_a$, $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ij}$ is also provable.

- $j \leq m_i$, $c_{ij} = a_{ij} - 1$:

$$\Delta_p, \Delta_a \vdash \left(\begin{array}{l}\mathsf{match}\ [\![d_{ij}\ t_{ij1} \dots t_{ijc_{ij}}]\!]\ \mathsf{with}\\ \mid [\![t_{ija_{ij}}]\!] \to [\![T_{i(j+1)} \to \dots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}}\\ \mid \_ \ \to\ \mathsf{default}_{I,\mathcal{M}}\end{array}\right) \rhd v$$

Since either $v = \mathsf{true}$ or $v \neq \mathsf{fail}$, we have $\Delta_p, \Delta_a \vdash [\![d_{ij}\ t_{ij1} \dots t_{ijc_{ij}}]\!] \rhd v' \neq \mathsf{fail}$ and the pattern $[\![t_{ija_{ij}}]\!]$ matches v'. By hypothesis of induction (over the soundness theorem), this means that $\Gamma \vdash \bar{\Delta}_a \bar{\Delta}_i T_{ij}$ is provable. Next, we have the evaluation $\Delta_p, \Delta'_p \vdash [\![T_{i(j+1)} \to \dots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \rhd v$, with $\Delta'_p = \Delta_a, \Delta_m$ and $\Delta_m = \mathsf{mgu}_{\Delta_p,\Delta_a}(v', [\![t_{ija_{ij}}]\!])$. By hypothesis of induction, there exists $\Delta'_a \supseteq \Delta'_p$ s.t. $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ik}$ is provable for $k = j+1 \dots m_i$, and $\Delta_p, \Delta'_a \vdash \mathsf{cont}_{I,\mathcal{M}} \rhd v$. As $\Delta'_a \supseteq \Delta'_p$, $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ij}$ is also provable.

Using (1), (2) and the above property (with the empty context for Δ_a), there exists a context Δ'_i s.t. $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i T_{ij}$ is provable for $j = 1 \dots m_i$, $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i (d\ t_{i1} \dots t_{ic_I}) \equiv (d\ t_1 \dots t_{c_I})$ and $\Delta_p, \Delta'_i \vdash \mathsf{cont}_{I,\mathcal{M}} \rhd v$. We distinguish two cases:

- $c_I = a_I$: $\mathsf{cont}_{I,\mathcal{M}} = \mathsf{true}$ and the constructor c_i can be directly applied to prove $\Gamma \vdash d\ t_1 \dots t_{c_I}$.
- $c_I = a_I - 1$: we have $\Delta_p, \Delta'_i \vdash [\![t_{ip_i}]\!] \rhd v$. There exists a term t'_{ip_i} s.t. $[\![t'_{ip_i}]\!] = v$ and $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i t_{ip_i} \equiv t'_{ip_i}$. Thus, $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i (d\, t_{i1} \dots t_{ic_I}\ t_{ip_i}) \equiv (d\, t_1 \dots t_{c_I}\ t'_{ip_i})$ and the constructor c_i can be applied to prove $\Gamma \vdash d\ t_1 \dots t_{c_I}\ t'_{ip_i}$.

5 Implementation and Optimizations

5.1 Implementation

A prototype has been implemented and integrated to the Coq [10] proof assistant. This implementation follows what has been described in Section 3 and is plugged into the regular extraction mechanism of Coq so that it can produce code for the same (functional) languages, i.e. OCaml, Haskell and Scheme. As an example, let us consider the addition predicate presented in Section 2 with an extraction to the OCaml language [11]:

```
Coq < Inductive add : nat → nat → nat → Prop :=
Coq <    | addO : forall n, add n O n
Coq <    | addS : forall n m p, add n m p → add n (S m) (S p).

Coq < Extraction Logical add [1 2].
(** val add : nat → nat → nat **)

let rec add p0 p1 =
  match (p0, p1) with
  | (n, O) → n
  | (n, S m) → let p = add n m in S p
  | _ → assert false
```

This implementation should be part of the forthcoming version of Coq, and currently, the development in progress is available on demand (sending a mail to the authors).

5.2 Optimizations

Conclusion inputs. In Section 3, we described the translation of a logical inductive type when rules do not overlap, that is when the types of the conclusions do not unify. However, we can implement some heuristics to overcome some of these cases. Let us consider the example of the while loop, seen in Section 3:

```
Inductive exec : store → command → store → Prop := ...
  | while1 : forall (s s1 s2 : Sigma) (b : expr) (c : command),
      (eval s b true) → (exec s c s1) → (exec s1 (while b do c) s2) →
      (exec s (while b do c) s2)
  | while2 : forall (s : Sigma) (b : expr) (c : command), (eval s b false) →
      (exec s (while b do c) s).
```

These two constructors overlap: the types of their conclusion are identical up to renaming. A Prolog-like execution would try to apply the first rule by computing the evaluation of the boolean expression b and matching it with the value true. If the matching fails, the execution would backtrack to the second rule. However, in this case, the execution is completely deterministic and no backtracking is necessary. In fact, we can discriminate the choice between both constructors thanks to their first premise. We introduce a heuristic to handle such cases efficiently. It requires the ability to detect common premises between both overlapping rules and to discriminate w.r.t. syntactic exclusive premises (p and $\neg p$, values constructed with different constructors of an inductive type, for example).

For the while loop example, the extracted function with the mode $\{1, 2\}$ involves only one case in the global pattern-matching to be able to handle correctly the execution:

```
let rec exec s c = match s, c with ...
  | s, while(b,c) →
    (match (eval s b) with
        | true → s
        | false →
          let s1 = exec s c in
          let s2 = exec s1 (while (b, c)) in s2)
```

Premise outputs. In the formalization, we also assumed that the outputs of the premises do not contain computed variables. Consequently, we cannot translate rules where constraints exist on these outputs, which is the case for the following constructor that describes the typing of a conditional expression in a logical inductive type named typecheck, when the mode $\{1, 2\}$ is specified:

```
Inductive typecheck : env → expr → type → Prop := ...
  | if : forall (g : env) (b, e1, e2 : expr) (t : type),
      (typecheck g b bool) → (typecheck g e1 t) → (typecheck g e2 t) →
      (typecheck g (if b then e1 else e2) t).
```

There is no difficulty to adapt the translation for such cases. Once the nonlinearity between premise outputs has been detected, we use fresh variables and guards as follows:

```
let rec typecheck g e = match g, e with ...
  | g, if (b, e1, e2) →
    (match typecheck g b with
        | bool → let t = typecheck g e1 in
          (match typecheck g e2 with
           | t' when t' = t → t
           | _ → assert false)
        | _ → assert false)
```

In the same way, it is also possible to deal with nonlinearity or symbols of functions in the output of a premise.

6 Conclusion

In this paper, we have presented an extraction mechanism in the context of CIC, which allows us to derive purely functional code from relational specifications implemented as logical inductive types. The main contributions are the formalization of the extraction itself (as a translation function) and the proof of its soundness. In addition, a prototype has been implemented and integrated to the Coq proof assistant, whereas some optimizations (relaxing some limitations) are under development.

Regarding future work, we have several perspectives. First, we aim to prove the completeness of our extraction (the mode consistency analysis should be used in this proof). Concerning our implementation, the next step is to manage large scale specifications, with, for example, the extraction of an interpreter from a development of the semantics of a programming language (in Coq, there are many developments in this domain). Another perspective is to adapt our mechanism to produce Coq functions, taking benefit from the new facilities offered by Coq to define general recursive functions [10]. These new features rely on the fact that the user provides a well-founded order establishing the termination of the described function. Provided the mode and this additional information, we could extract a Coq function from a logical inductive type, at least for a large class of logical inductive types (e.g. first order unification, strongly normalizable calculi, etc). Finally, the mode consistency analysis should be completed by other analyses like determinism or termination. The logical programming community has investigated abstract interpretation to check this kind of operational properties [5]. Similar analyses could be reproduced in our case. We could also benefit from results coming from term rewriting system tools.

References

1. Attali, I., Parigot, D.: Integrating Natural Semantics and Attribute Grammars: the Minotaur System. Technical Report 2339, INRIA (1994)
2. Berghofer, S., Nipkow, T.: Executing Higher Order Logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 24–40. Springer, Heidelberg (2002)
3. Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the System. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE) 24(2), 14–24 (1988)
4. Dubois, C., Gayraud, R.: Compilation de la sémantique naturelle vers ML. In: Weis, P. (ed.) Journées Francophones des Langages Applicatifs (JFLA), Morzine-Avoriaz (France) (February 1999)
5. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization using Abstract Interpretation (and the Ciao System Preprocessor. Science of Computer Programming 58(1-2), 115–140 (2005)
6. Letouzey, P.: A New Extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
7. Overton, D., Somogyi, Z., Stuckey, P.J.: Constraint-based Mode Analysis of Mercury. In: Principles and Practice of Declarative Programming (PPDP), Pittsburgh (PA, USA), October 2002, pp. 109–120. ACM Press, New York (2002)
8. Pettersson, M.: A Compiler for Natural Semantics. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 177–191. Springer, Heidelberg (1996)
9. Stärk, R.F.: Input/Output Dependencies of Normal Logic Programs. Journal of Logic and Computation 4(3), 249–262 (1994)
10. The Coq Development Team: Coq, version 8.1. INRIA (November 2006), available at: `http://coq.inria.fr/`
11. The Cristal Team: Objective Caml, version 3.09.3. INRIA (September 2006), available at: `http://caml.inria.fr/`
12. Verdejo, A., Martí-Oliet, N.: Executable Structural Operational Semantics in Maude. Journal of Logic and Algebraic Programming 67(1-2), 226–293 (2006)

Appendix A: Semantic Rules of the Extraction Language

$$\frac{(x,v)\in\Delta}{\Delta\vdash x\rhd v}\ \mathsf{Var} \qquad \frac{}{\Delta\vdash \mathsf{fail}\rhd\mathsf{fail}}\ \mathsf{fail}$$

$$\frac{}{\Delta\vdash c^0\rhd c^0}\ \mathsf{const}_0 \qquad \frac{}{\Delta\vdash C^0\rhd C^0}\ \mathsf{constr}_0$$

$$\frac{\Delta\vdash e_1\rhd v_1 \quad \dots \quad \Delta\vdash e_n\rhd v_n \quad (c^n,v_1,\dots,v_n,v)\in\Delta_c}{\Delta\vdash c^n\ (e_1,\dots,e_n)\rhd v}\ \mathsf{const}_n$$

$$\frac{\Delta\vdash e_1\rhd v_1 \quad \dots \quad \Delta\vdash e_n\rhd v_n \quad C^n\ (v_1,\dots,v_n)\in\Delta_C}{\Delta\vdash C^n\ (e_1,\dots,e_n)\rhd C^n\ (v_1,\dots,v_n)}\ \mathsf{constr}_n$$

$$\frac{\Delta\vdash e_1\rhd\mathsf{true} \quad \Delta\vdash e_2\rhd v_2}{\Delta\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rhd v_2}\ \mathsf{if}_{\mathsf{true}} \qquad \frac{\Delta\vdash e_1\rhd\mathsf{false} \quad \Delta\vdash e_3\rhd v_3}{\Delta\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rhd v_3}\ \mathsf{if}_{\mathsf{false}}$$

$$\frac{}{\Delta\vdash \mathsf{fun}\ x\rightarrow e\rhd <x\rightsquigarrow e,\Delta>}\ \mathsf{fun} \qquad \frac{}{\Delta\vdash \mathsf{rec}\ f\ x\rightarrow e\rhd <x\rightsquigarrow e,\Delta>_{\mathsf{rec}(f)}}\ \mathsf{rec}$$

$$\frac{\Delta\vdash e_1\rhd v_1 \quad \Delta,(x,v_1)\vdash e_2\rhd v_2}{\Delta\vdash \mathsf{let}\ x=e_1\ \mathsf{in}\ e_2\rhd v_2}\ \mathsf{let} \qquad \frac{\Delta\vdash e_1\rhd v_1 \quad \dots \quad \Delta\vdash e_n\rhd v_n}{\Delta\vdash (e_1,\dots,e_n)\rhd (v_1,\dots,v_n)}\ \mathsf{Tuple}$$

$$\frac{\Delta\vdash e_1\rhd <x\rightsquigarrow e_3,\Delta> \quad \Delta\vdash e_2\rhd v_2 \quad \Delta,(x,v_2)\vdash e_3\rhd v_3}{\Delta\vdash e_1\ e_2\rhd v_3}\ \mathsf{App}$$

$$\frac{\begin{array}{c}\Delta\vdash e_1\rhd <x\rightsquigarrow e_3,\Delta>_{\mathsf{rec}(f)}=c \quad \Delta\vdash e_2\rhd v_2\\ \Delta,(f,c),(x,v_2)\vdash e_3\rhd v_3\end{array}}{\Delta\vdash e_1\ e_2\rhd v_3}\ \mathsf{App}_{\mathsf{rec}}$$

$$\frac{\begin{array}{c}\Delta\vdash e\rhd v \quad \mathsf{filter}_\Delta(v,gpat_i)=\Delta_i\\ \mathsf{filter}_\Delta(v,gpat_j)=\mathsf{fail},\ 1\le j<i \quad \Delta,\Delta_i\vdash e_i\rhd v_i\end{array}}{\Delta\vdash \mathsf{match}\ e\ \mathsf{with}\ \mid gpat_1\rightarrow e_1\ \dots\ \mid gpat_n\rightarrow e_n\rhd v_i}\ \mathsf{match}$$

$$\text{with } \mathsf{filter}_\Delta(v,gpat)=\begin{cases}\Delta_p, & \text{if } gpat=pat \text{ and } \mathsf{mgu}_\Delta(v,pat)=\Delta_p,\\ \Delta_p, & \text{if } gpat=pat\ \mathsf{when}\ e,\ \mathsf{mgu}_\Delta(v,pat)=\Delta_p,\\ & \text{and } \Delta,\Delta_p\vdash e\rhd\mathsf{true},\\ \mathsf{fail}, & \text{otherwise}\end{cases}$$