



A Minimalist Approach to Trustworthy Programming with Precise Types using Small Inversions

Pierre Corbineau^{1,2}, Basile Gros^{1,3}, and Jean-François Monin^{1,4}

¹ Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France

² pierre.corbineau@univ-grenoble-alpes.fr

³ basile.gros@univ-grenoble-alpes.fr

⁴ jean-francois.monin@univ-grenoble-alpes.fr

Abstract

Pattern matching is the primary programming construct used to analyze the contents of algebraic data types in statically typed functional languages. Additionally, dependent types allow us to benefit from very precise information about data being processed. However, pattern matching on dependent types turned out to raise surprisingly complex challenges, that have required more than two decades of intensive research to produce current implementations in AGDA, EPIGRAM and the EQUATIONS plugin of ROCQ. Another popular tool among ROCQ users is the historical *inversion* tactic which solves many use cases at the cost of a loss of control over the form of the programs obtained, unpredictable proof-term size and a loss of source-level coherence. Both approaches in ROCQ rely on intermediate equalities to produce a well-typed term in the underlying type theory, many of which turn out to be unnecessary.

Alongside those tools, we believe that there is room for a complementary minimalist approach based on recent improvements of *small inversions* that generates explicit terms that are more readable and explainable. We present here this approach and its current automation implemented in METAROCQ.

1 Introduction

ROCQ (formerly known as COQ) [28] is an interactive proof assistant, used in particular to prove the correction of software and to verify mathematical proofs. It implements the Calculus of Inductive Constructions (CIC) [31] by allowing the definition of algebraic types and then function and properties on those types [10, 14]. Famous achievements realized in ROCQ include the verification of the proof of the Four Color Theorem [17], its use as the basis of the certified C compiler CompCert [27], and more recently it was used to prove the number of steps of the fifth Busy Beaver [4].

Following the Curry-Howard isomorphism, CIC is both an expressive logic and a statically typed functional programming language featuring dependent types that allow us to impose

*Institute of Engineering Univ. Grenoble Alpes

very precise constraints on data being processed. The ability to correctly deal with functional programs and logic in an unified framework is of great importance not only in applications to computer science but also to mathematics, as witnessed by the Four Color Theorem.

1.1 Trust and Trap

The confidence that can be placed in proofs and programs developed in systems in the tradition of AUTOMATH, such as ROCQ, rests on a criterion coined the *de Bruijn criterion* by Barendregt and Geuvers [3]: they “generate ‘proof objects’ (of some form) that can be checked by an ‘easy’ algorithm”. This criterion is of paramount importance for the development of software systems subject to very high security requirements. In the case of ROCQ, the proof checking algorithm is called the *kernel*. The big advantage is that we get full freedom in the process of creation of proof objects: anything goes provided that, at the end, the proof object you get is accepted by the kernel. In particular, such proof assistants offer various means allowing the user to incrementally and interactively build a proof, for instance commands called *tactics* in ROCQ, that can be combined using a scripting language, in combination with automated mechanisms such as decision procedures. Then, by design, there is no need to worry about providing guarantees on these additional tools, only their result matters. Tactics and scripting languages do not require the precise semantics required for the kernel-verified terms language. This makes it possible to state and prove theorems with a very high degree of confidence, even though all sorts of workarounds or heuristics are used in the process of developing the proofs. In the Curry-Howard-de Bruijn setting, where proofs are programs and their statements are types, the proof checker is essentially a type checker: checking that a proof object p is indeed a proof for a statement S , is the same as checking that p , as a program, has the type S .

With this in mind, we get for the same price an environment that helps us to define typed functional programs in an interactive way. This approach becomes particularly powerful and convenient when developing programs that process data with complex dependent types.

However, a caveat is in order here: in the case of programs, it is their content itself that matters, and not just their type, as it affects the meaning of logical statements where it is used. For a basic example, consider the formula $\forall u\ v, \text{len}(u ++ v) = \text{len } u + \text{len } v$. This statement involves two lists u and v and three functions: len for the length of lists, $++$ for the concatenation and $+$ for the addition. Each of these three functions is defined by a program, and the intended meaning of the above formula relies entirely on our understanding of the code of those programs. In general, although it is legitimate to use automated tools in the design of programs, the thing to be exposed at the end is a piece of code expressed in a principled language that should be as short, clear, and readable as possible, so that we can be certain it will be interpreted in the same way by human readers and by computers. As such, using tactics and a tactic language can be helpful but the resulting source script cannot be accepted as a definite, explicit formulation of the desired program.

An additional important *source-level coherence criterion* is the following: when calculations are performed, the link with the source code should be kept, so that the user can relate the output traces to the source code. This coherence criterion fails to be satisfied when using a tactic language, since reduction and conversion rules are expressed on terms, not on source scripts.

In contrast, the type theory checked by the kernel (CIC, in the case of of ROCQ) is – by design – a trustworthy language. All of this argues in favor of sticking to CIC for writing ROCQ programs, or at least adhering to CIC as closely as possible: for instance, automatically derived inferred sub-terms and implicit arguments are acceptable and even useful for code readability,

as far as parsing and displaying remains consistent according to *Pollack's criterion* [23].

1.2 CIC as a Programming Language

As far as advanced aspects are left aside, CIC is a statically typed functional language similar to languages of the ML family, where algebraic data structures such as lists and trees can be defined. The main programming construct used to analyze the contents of an arbitrary algebraic data structure is *pattern matching*, an extended case analysis that opens a new typed scope providing the components specified by a given pattern, for each pattern that matches a given piece of data D – the different patterns correspond to possible cases for D . CIC includes two kinds of recursive algebraic data structures: *inductive* structures and *co-inductive* structures. The latter are similar to the former but allow data defined by an infinity of constructors.

An important feature that distinguishes languages in the tradition of Type Theory like CIC from common programming languages is that all programs are total, i.e., computations never diverge. In the case of programs on inductive data structures, this is achieved by a restriction called *positivity* on the shape of inductive types and a *guard condition*; there is a dual condition for CIC co-inductive structures.

Since the work by Giménez in the 1990s [15, 16], we recovered for CIC a separation of concerns which is quite common in programming: issues on termination related to recursive calls are dissociated from other issues, in particular those related to typing and pattern matching¹. For instance, the core of the well-known `map` function is the same for inductive lists and for co-inductive lists. The only difference is that the former is defined using a *fixpoint* construct whereas the latter uses a *cofixpoint* construct; the appropriate guard condition is checked accordingly, either way both of them are satisfied here. However, type checking is performed in exactly the same way: in particular, if f is the functional argument of `map`, any sub-expression $f\ x$ occurring in the program is well-typed and any recursive call to `map` is itself well-typed. Details on the definition of CIC can be found in [29].

1.3 Programming with Dependent Types

Things get more interesting when we look at dependent types. As a simple example, consider the `map2` function, that is similar to `map` but takes *two* lists of the *same* length as input instead of just one. Given three types A, B, C , `map2` takes a function f of type $A \rightarrow B \rightarrow C$, two lists $u : \text{list } A$ and $v : \text{list } B$ as input, and returns a list of type `list C` such that the following equations are satisfied:

$$\text{map2 } f \ [] \ [] = [] \tag{1a}$$

$$\text{map2 } f \ (x :: u') \ (y :: v') = f\ x\ y :: \text{map2 } f\ u' \ v' \tag{1b}$$

At this point, let us introduce the dependent data type `vect A n` (for “vectors”) as a refinement of `list A`: in addition to the *parameter* A , which is used for specifying the type of elements, we have an *index* n . The type `vect A n` is similar to `list A` but it forces its inhabitants to have the length n . Its ROCQ definition is:

¹However, pattern matching is the construct that provides structurally smaller terms to be recognized by the guard condition. This consideration can be ignored to a great extent in this work (more on this in 2.3). Note that the guard condition for co-recursive programs is unrelated to pattern matching: it is a productivity requirement, stating that co-recursive calls must be protected by at least a constructor. For instance, we have the following co-fixpoint, where $\bar{S}$ is the successor in $\bar{\mathbb{N}}$, the co-inductive version of $\mathbb{N}$: `cofix  $\omega : \bar{\mathbb{N}} := \bar{S}\ \omega$` . There is no pattern matching in the body of ω .

```

Inductive vect (A : Type) :  $\mathbb{N} \rightarrow$  Type :=
| nil : vect A 0
| cons : A  $\rightarrow \forall n$ , vect A n  $\rightarrow$  vect A (S n).

```

According to a general rule of Type Theory, the parameter is used uniformly whereas the shape of the index can be chosen freely in $\mathbb{N}$: here, the type of `(nil ...)` is `(vect A 0)` whereas the type of `(cons ...)` is `(vect A (S n))`. The previous definition can also be stated in terms of co-inductive types: just replace `Inductive` by `CoInductive` and $(\mathbb{N}, 0, S)$ by co-inductive natural numbers. We no longer need lists thereafter and then use the notation `[]` for `(nil A)` and `(x :: u)` for `(cons A x n u)` – note that A and n are inferred from the type of u . When we need to refer to the length n of u , we write $(x : n : u)$.

A pleasant advantage of dependent data structures is that we expect irrelevant situations to be discarded, e.g., `map2 f (x :: u') []`. Can we ask pattern matching to do this job for us? Why not, but this requires more careful work than it might seem at first glance. Intuitively, in an expression `map2 f (x :: u') v`, the type of v is forced to be `vect B (S n')`, where n' is the size of u' . Taking `[]` for v is then discarded because it boils down to accept that its actual index `0` could be a possible instance of the pattern `(S n')`. Note that, in this context, we are concerned only with indices whose types are algebraic, and that in the reasoning above, it is not the algebraic structure of vectors that is relevant, but rather that of the *indices* of this family.

Finding a general extension of pattern matching to dependent types turned out to be more complicated than anticipated. Solutions based on first-order unification have been developed, notably in the Agda community, raising unexpected and subtle issues related to the necessity to rely (or not) on additional axioms. The reader is referred to [7] for details. As a result, the trust criteria mentioned in Section 1.1 are potentially compromised. We come back to those issues in Section 5 devoted to related work.

1.4 Pattern Matching in CIC

The primitive syntactic pattern matching construct of CIC is less ambitious but much simpler and therefore presumably more trustworthy than dependent pattern matching as mentioned at the end of Section 1.3, though its complete syntax is somewhat frightening at first sight. It can be illustrated as follows on a single vector v , where v is a term. The scheme on the right is a special case of the scheme on the left. As it is more often used, it deserves special attention and we name it *index-dependent pattern matching*.

<pre> match v as u in vect _ n return P n u with [] $\Rightarrow$ (term₀ : P 0 []) x : n' : u' $\Rightarrow$ (term₁ : P (S n') (x :: u')) end </pre>	<pre> match v in vect _ n return Q n with [] $\Rightarrow$ (term₀ : Q 0) x : n' : u' $\Rightarrow$ (term₁ : Q (S n')) end </pre>
--	--

This is very close to usual basic pattern matching: the two possible cases for v are considered, and an expression is required in each case: $term_0$ when v is `[]` and $term_1$ when v is `x :: u'`. The novelty is that $term_0$ and $term_1$ are allowed to have different types, according to the *return clause* specified by “`in vect _ n return P n u`”: the formula (or type) $P n u$, called the *motive* in the terminology of [18], depends on n , the actual index of the type of u , and sometimes on u as well. As parameters are not free variables of the motive, they are represented by a wildcard “`_`”. Fortunately, the `return` clause can actually be inferred from the context in simple enough situations, making the corresponding piece of code similar to what would be written in a language of the ML family.

This conceptual simplicity has a cost: it is more limited than the more ambitious version

of pattern matching mentioned above. In particular, in contrast with parameters (like A), *indices* (like n) are *generalized over* in this process. In [7], Cockx remarks that thanks to this crucial generalization, the need for first-order unification disappears. The price to pay is that information on the shape of actual indices is lost unless precautions are taken in the motive (e.g., using equalities), and even then, taking these precautions into account later in the code complicates the latter.

For instance, let us try to define the `map2` function using two successive index-dependent pattern matchings, first on u and then on v . The shape of our program could be:

```

Fixpoint map2 {A B C} (f : A → B → C) {n} (u : vect A n) : vect B n → vect C n :=
  match u with
  | []      ⇒ λ v, []
  | x :: u' ⇒ λ v, match v in vect _ m return motivem with cases end
end.

```

The return clause for “`match u`” is “`in vect _ n return vect B n → vect C n`”. It is not an issue because the actual index of u is n , a variable. It can then be inferred and it is omitted above. Next, we need to find $motive_m$ (and *cases*) given $(x : A)$, $(u' : vect A n')$ for some n' and $(v : vect A (S n'))$. But here, the actual index m of v is $S n'$, so that the corresponding $motive_m$ and the subsequent *cases* become challenging because of the generalization over m . Known solutions often involve additional equality manipulations that obscure the code [22, 21, 12].

1.5 Contributions

This paper explores a minimalist approach based on a direct use of the primitive pattern matching of CIC, in the line of previous work by the third author and Shi on a proof technique named *small inversion* [22, 21], that deduces information about the premises of a hypothesis H from its conclusion when the type of H is a (co-)inductively indexed relation and some indices in H follow a specific pattern. The explicit terms provided by small inversions are much smaller and readable than those produced by the popular *inversion* tactic.

This becomes even more true with the *proxy-based small inversions* (PBSI) developed below, that move the aforementioned “suitable motive” to a single place that we call a *proxy*. The basic form of PBSI corresponds to indexed-dependent pattern matching – it improves the latter and can be used in `map2`, for instance. We also derive slightly more sophisticated proxies named *dependent proxies*. They are useful in more complex programming tasks, where not only expressions returned in each case *but also their type* depend on the constructor of R considered – this is similar to full CIC pattern matching but taking the specific shape of actual indices into account. Dependent inversions are also very useful for reasoning on all those programs. Altogether, our approach results in source code that is reasonably satisfactory regarding the requirements stated in 1.1.

An automated implementation of PBSI, illustrated by relevant use cases, is available at <https://github.com/BasileGros/proxy-based-small-inversions>. Some examples initially developed in Epigram [19] or in Agda [24] are also included.

The remainder of this paper is organized as follows. Section 2 gives an intuitive presentation of PBSI that is suitable for a handcrafted usage of this approach. Section 3 formalizes a systematic and modular way to derive the auxiliary definitions of PBSI, and then provides a foundation for the implementation of PBSI that is briefly discussed in Section 4. Finally, Section 5 discusses related work and we conclude in Section 6.

2 Proxy-Based Small Inversions

As our approach is agnostic to the inductive or co-inductive nature of data types, we will use the term *algebraic type* to cover both cases.

2.1 Basic PBSI

In basic proxy-based small inversions, we systematically derive an auxiliary algebraic type R_ψ — called *partial algebraic type* — for each pattern ψ of *indices* occurring in conclusions of the type family R to be inverted; the constructors of R_ψ mimic the corresponding ones of R . For example, for `vect` we have one index, of type $\mathbb{N}$ and two constructors, `nil` and `cons`. The patterns for the index are respectively `0` and `(S n)`. The corresponding partial algebraic types are:

```
Variant vect_0 A : Type := nil_0 : vect_0 A.
Variant vect_S A n : Type := cons_S : A → vect A n → vect_S A n.
```

Remark that `vect_0 A` and `vect_S A n` correspond to a syntactic partition of `vect` characterized by the patterns `0` and `(S n)` of the index of `vect`. This allows us to define the *proxy type* of `vect` as a *function* `vect_proxy_type` by pattern matching on the index as follows:

```
Definition vect_proxy_type A n : Type :=
  match n with 0 ⇒ vect_0 A | S n ⇒ vect_S A n end.
```

This proxy type provides the return clause of the following *proxy function* `vect_proxy`:

```
Definition vect_proxy {A n} (u : vect A n) : vect_proxy_type A n :=
  match u in vect _ n return vect_proxy_type A n with
  nil ⇒ nil_0 | cons x u ⇒ cons_S x u end.
```

Using curly brackets around `A` and `n` makes those arguments implicit. The above definitions are written once and for all and can be automatically produced. We can then define `map2`.

```
Fixpoint map2 {A B C} (f : A → B → C) {n} (u : vect A n) : vect B n → vect C n :=
  match u with
  | [] ⇒ λ v, []
  | x :: u' ⇒ λ v, let '(cons_S y v') := vect_proxy v in f x y :: map2 f u' v'
  end.
```

Remind that in the second case, the type of `v` is `S n'` where `n'` is the (unnamed) index for the length of `u'`, so that the *type* of `vect_proxy v` is known to be convertible with `vect_S A n'` at compile time; as this type has only one constructor, we can use a deconstructing `let` and the returned expression is well-typed. Thanks to the proxy, there is no need to forge a suitable return clause in the code of `map2`. It is also possible to use a more compact notation for the deconstructing `let`, as follows:

$$\lambda v, \text{let } (y, v') := \text{vect_proxy } v \text{ in } f \ x \ y :: \text{map2 } f \ u' \ v'$$

Note that `vect_0 A` and `vect_S A n` are isomorphic to, respectively, `unit` and `A * vect A n` that could be used as well, but this lucky opportunity does not scale to constructors of arbitrary complexity.

In the general case of a type family R , we first build a syntactic partition of R into partial algebraic types R_ψ depending on the syntactically decidable patterns ψ of one or more of the (co-)inductively typed indices of R , called *pilot indices*. The partial algebraic types R_ψ are systematically derived from R as follows:

- The components of the constructors of R are left unchanged in their counterpart in R_ψ , including recursive occurrences of R ; therefore, partial algebraic types are *not* recursive; in the corresponding ROCQ code, we use the keyword **Variant** to emphasize this.
- Information in the indices that was to be preserved for pattern matching is stored as *parameters* of R_ψ which the motive does not generalize over, so that the limitation mentioned in Section 1.4 is circumvented. For example in **vect**, the **n** component of **cons** becomes a parameter of **vect_S**; it is then no longer a component of **cons_S** and the corresponding deconstructing **let** used in **map2** provides exactly the two main components **y** and **v'**, as expected.

Note that sometimes, the ψ occurring in R do not exhaustively cover the types of the pilot indices. Lacking patterns are then added: a lacking pattern ψ_- simply corresponds to the special case where R_{ψ_-} has no constructor – it happens to be an empty type.

We then define the *proxy type* of R as a function R_proxy_type mapping targeted index patterns to the corresponding R_ψ . It provides the return clause of the *proxy function* R_proxy to be applied to terms of type R : the proxy function maps each constructor C of R to the corresponding constructor C'_ψ of the respective R_ψ . The desired inversion on $r : R$ is obtained by a CIC pattern matching on $R_proxy\ r$: instead of

```
match r with ... | C x y ... => E | ... end,
```

we write

```
match R_proxy r with ... | C'_psi x y ... => E | ... end
```

or, using a deconstructing **let**, when R_ψ has exactly one constructor:

```
let (...) := R_proxy r in E
```

In contrast with other available technologies, *no additional equality is involved in this process*.

The partitioning of an algebraic type into partial algebraic types — called *specialization* — can be carried out recursively on previously defined partial algebraic types, to allow for small inversion on multiple indices and with nested patterns.

Basic PBSI actually corresponds to index-dependent pattern matching presented in 1.4. As such, it is sufficient for a number of programs. For instance, functions on vectors and bounded natural numbers of ROCQ's standard library no longer present any difficulties, as can be seen on **map2**. This is illustrated in the accompanying material with a definition of the transposition of matrices, implemented as vectors of vectors. We also provide a formalization of intrinsically typed expressions and a definitional interpreter for simply typed lambda-calculus (STLC) written in Agda from [24] – the latter used Agda, pointing out that “*Coq's limited support for dependent pattern matching makes it [...] unattractive for writing dependently-typed programs*”.

2.2 Dependent PBSI

However, correctness proofs on programs defined with basic PBSI require the equivalent of the full power of CIC pattern matching. Even in programming with dependent types, there are subtle situations that are beyond the power of basic PBSI, like in our ROCQ encoding of the extended example given in [19] – a type checker for STLC.

For illustration purposes only, consider a program **remap2** similar to **map2**, where the type of the result remembers the two input vectors according to the following inductive definition.

```
Inductive Remap2 {C A B : Type} : ∀ {n}, vect A n → vect B n → Set :=
| Rmnil : Remap2 [] []
| Rmcons {a b n} {u : vect A n} {v : vect B n} : C → Remap2 u v → Remap2 (a :: u) (b :: v).
```

We cannot write, as expected at first sight:

```

Fixpoint remap2 {A B C : Type} (f : A → B → C) {n} (u : vect A n) :
  ∀ v : vect B n, Remap2 C u v :=
  match u with
  | []      ⇒ λ v, Rmnil
  | x :: u' ⇒ λ v, let '(cons_S y v') := vect_proxy v in
    Rmcons (f x y) (remap2 f u' v')
  end.

```

The reason for this is that the *type* of `Rmcons (f x y) (remap2 f u' v')` itself depends on `y` as well as on `v'`. More generally, in the usage scheme provided at the end of 2.1, E can contain occurrences of the components $x, y \dots$ encapsulated by C'_ψ but the *type* of E cannot. This is annoying because the original construct `match r with ... | C ⇒ E | ... end` does not have this restriction.

One of the first trick that comes to mind consists in writing a left inverse of R_proxy , say $back_R$, then proving $\forall r, r = back_R (R_proxy r)$ so that this equation can be convoyed in the motive of `match R_proxy r with ... end`. A drawback of this approach is that even when it works, it obscures code and proofs with additional equalities that can be hard to handle.

We propose instead to use *dependent* PBSI, of which basic PBSI is a special case in the same way that index-dependent pattern matching is a special case of full CIC pattern matching. To this effect, we only need to add to R_ψ an index of type R , suitably instantiated with the indices specified in ψ , and correspondingly an additional argument to the type of the proxy that is naturally generalized over the indices of R . For example, the dependent partial algebraic type of `vect` are:

```

Variant vect_0_dep A : vect A 0 → Type := nil_0_dep : vect_0_dep A [].
Variant vect_S_dep A n : vect A (S n) → Type :=
  cons_S_dep x u' : vect_S_dep A n (x :: u').

```

The dependent proxy is defined following the same method as for basic PBSI.

```

Definition vect_dproxy_type A n : vect A n → Type :=
  match n with 0 ⇒ vect_0_dep A | S n ⇒ vect_S_dep A n end.

```

```

Definition vect_dproxy {A n} (u : vect A n) : vect_dproxy_type A n u :=
  match u with nil ⇒ nil_0_dep | x :: u' ⇒ cons_S_dep x u' end.

```

Using the above definitions, the code for `remap2` is essentially the same as for `map2`. The main difference is that in the `[]` case, we have also an inversion on `v`: it is necessary for the returned expression `Rmnil` to be well-typed.

```

Fixpoint remap2 {A B C : Type} (f : A → B → C) {n} (u : vect A n) :
  ∀ v : vect B n, Remap2 C u v :=
  match u with
  | []      ⇒ λ v, let 'nil_0_dep := vect_dproxy v in Rmnil
  | x :: u' ⇒ λ v, let '(cons_S_dep y v') := vect_dproxy v in
    Rmcons (f x y) (remap2 f u' v')
  end.

```

Intuitively, the information captured by the `as` clause in the original CIC pattern matching is moved to the `in` clause, that is,

```

match r as s in R x y ... return P x y ... s with ... end

```

becomes

```
match R_dproxy r in R_dproxy_type x y ... s return P x y ... s with ... end.
```

2.3 PBSI vs. the Guard Condition

A bit of explanation is in order about the interaction between PBSI and the guard condition related to structural recursion. Syntactically, a pattern matching on $g\ b$ provides sub-terms of $(g\ b)$ but not of b itself. This holds when g is some PBSI proxy function, say `proxy`, even though in this case, $(\text{proxy } b)$ is isomorphic to b . But, as observed on `map2` and `remap2` above, a common reason why a pattern matching on $(\text{proxy } b)$ is needed is that b depends on an index x that is already constrained by a previous pattern matching on some a that precisely also depends on x . The typical recursive programming pattern that entails an inversion on b is actually the following:

```
Fixpoint f ... {x} ... a ... b ... :=
  match a with
  ...
  | ... a1 a2 ... => match proxy b with
                        ...
                        | ... b1 b2 ... => ... (f ... ai ... bj ...) ...
                        end
  end
```

where f is the structurally recursive function to be defined, a_i is a sub-term of a coming from the pattern “... $a_1\ a_2\ \dots$ ” and similarly, b_j is an *intended* sub-term of b . However, there is clearly no need for b_j to be an actual sub-term of b since the guard condition is already satisfied on a_i .

This programming pattern occurs naturally in all programs specified by dependent recursive equations such as (1): in the above definitions of `map2` and `remap2`, v' is *not* recognized as structurally smaller than v ; however, the recursive calls `map2 f u' v'` and `remap2 f u' v'` are correct because the pattern matching on u already provides a component u' which is structurally smaller than u , so that the guard condition is satisfied.

3 Transformations and their Combination

This section is dedicated to the systematic construction of proxy functions such as `vect_proxy` or `vect_dproxy`. They are obtained in a compositional manner, using steps called *transformations*. This decomposition is relevant to a mechanized derivation of PBSI as sketched in Section 4, not to handcrafted PBSI.

We assume source algebraic type family R with parameters $(\vec{p} : \vec{P})$, indices $(\vec{a} : \vec{A})$ and sort $\mathcal{S}$. For example, when R is `vect`, we have $(\vec{p} : \vec{P}) = (A : \text{Type})$, $(\vec{a} : \vec{A}) = (n : \mathbb{N})$ and $\mathcal{S} = \text{Type}$. A transformation operating on R is defined by:

- a positive number of auxiliary algebraic types $R_1, \dots$;
- as a first approximation, an elementary proxy type `R_proxy_type` which is a function returning one of the algebraic types $R_1, \dots$; `R_proxy_type` has the same arity as R , it is actually meant to be isomorphic to R ;

- and finally, a number of *proxy substitution functions* that can be seen as building blocs of the final proxy function.

Actually, some flexibility is needed in order to compose transformations. R_proxy_type is not directly defined, instead, we define an abstract version of it called $R_abproxy_type$, where $R_1, \dots$ are abstracted. The above R_proxy_type is recovered by simple application of $R_abproxy_type$ to $R_1, \dots$. In this way, we gain the opportunity to apply $R_abproxy_type$ to other actual arguments having the same type, in particular proxied versions of $R_1, \dots$, themselves obtained by another transformation. Altogether, the desired final proxy type is expressed by a functional composition of abstract proxy types, applied to abstract proxy types, $\dots$, applied to suitable auxiliary algebraic types. This functional expression can then be simplified by CIC reductions, and we get a readable version of the desired final proxy type. Proxy functions are also defined using a similar approach. A point to keep in mind is that here, R can be the initial algebraic type R_0 to be inverted, or a partial algebraic type R_1 of R_0 , or a partial algebraic type R_2 of R_1 , etc.

To simplify the presentation of transformations below, we consider the “concrete” elementary proxy types and functions, without the abstractions we have just mentioned, loosely denoting by $[E]$ a placeholder for a variable to be substituted by E itself or by a term having the same type and meant to be isomorphic to E .

We assume that the constructors of R are C_i , with arguments $(\vec{x}_i : \vec{X}_i)$, and the $\vec{\alpha}_i$ are terms instantiating the indices $\vec{a}$ in the conclusion of C_i .

(Co)Inductive $R(\vec{p} : \vec{P}) : \forall (\vec{a} : \vec{A}), \mathcal{S} := \dots \mid C_i : \forall (\vec{x}_i : \vec{X}_i), R \vec{p} \vec{\alpha}_i$

Taking for R the initial R_0 and using the $[.]$ convention, we aim at a final proxy for R_0 with the following shape:

Definition $R_0_proxy(\vec{p} : \vec{P})(\vec{a} : \vec{A})(r : R_0 \vec{p} \vec{a}) :=$
`match r with`
`...`
`| C_i _ \vec{x}_i => [C_i] \vec{p} \vec{x}_i`
`end.`

where the $[C_i]$ are ultimately obtained by the composition of proxy substitution functions coming from transformations. The initial proxy is obtained by *derecursion* or by *reflection* of R_0 , yielding R_0^d for basic PBSI or R_0^r for dependent PBSI. Then a number of *specializations* are applied to the initial proxy considered and finally, a last transformation called *parameterization*.

3.1 Initial Proxy, by Derecursion or Reflection

Given R as presented above, we define its *derecursed* version R^d as the non-recursive algebraic type that has the same structure as R .

Variant $R^d(\vec{p} : \vec{P}) : \forall (\vec{a} : \vec{A}), \mathcal{S} := \dots \mid C_i^d : \forall (\vec{x}_i : \vec{X}_i), R^d \vec{p} \vec{\alpha}_i$. Note that C_i^d has the same arguments as C_i . In particular, arguments of type $(R \dots)$ have the same type in C_i^d , so that the latter is indeed non-recursive. The *reflection* R^r of R is defined similarly to R^d , with an additional final index that reflects the construction used in R .

Variant $R^r(\vec{p} : \vec{P}) : \forall (\vec{a} : \vec{A})(r : R \vec{p} \vec{a}), \mathcal{S} := \dots \mid C_i^r : \forall (\vec{x}_i : \vec{X}_i), R^r \vec{p} \vec{\alpha}_i (C_i \vec{p} \vec{x}_i)$.

In the spirit of transformations presented above, we have two possible initial proxies, allowing us to get either basic PBSI or dependent PBSI. They are respectively given by:

Definition $R_0_proxy_type^d (\vec{p} : \vec{P}) (\vec{a} : \vec{A}) := [R_0^d] \vec{p} \vec{a}$.

Definition $R_0_proxy^d (\vec{p} : \vec{P}) (\vec{a} : \vec{A}) (r : R_0 \vec{p} \vec{a}) : R_0^d \vec{p} \vec{a} :=$

```
match r with
...
| Ci _  $\vec{x}_i$   $\Rightarrow$  [Cid]  $\vec{p}$   $\vec{x}_i$ 
end.
```

and

Definition $R_0_proxy_type^r (\vec{p} : \vec{P}) (\vec{a} : \vec{A}) (r : R \vec{p} \vec{a}) := [R_0^r] \vec{p} \vec{a} r$.

Definition $R_0_proxy^r (\vec{p} : \vec{P}) (\vec{a} : \vec{A}) (r : R_0 \vec{p} \vec{a}) : R_0^r \vec{p} \vec{a} r :=$

```
match r with
...
| Ci _  $\vec{x}_i$   $\Rightarrow$  [Cir]  $\vec{p}$   $\vec{x}_i$ 
end.
```

3.2 Core Transformation : Specialization

The proxy function maps an object r of type $R_0^d \vec{p} \vec{a}$ or $R_0^r \vec{p} \vec{a}$ to its mirror element in the corresponding partial algebraic type, using a pattern matching expression on r with, in each case, an expression containing placeholders $[C_i]$, where C_i is a constructor of some algebraic type R . The specialization of R provides a proxy substitution function for each of those $[C_i]$, based on the definition of partial algebraic types for R , that can be seen as a partition of R . The partial algebraic types, are specialized versions of R for relevant patterns of a single pilot index a_κ , where κ is the position of this index in the arity of R . This transformation is iterated if we want an inversion piloted by several indices, or using a deep pattern on at least one index. The type of the proxy type is the arity of the algebraic type R given as input. The proxy type is defined by a case analysis (using a large elimination) on a_κ that returns the partial algebraic type corresponding to the pattern considered. In turn, the proxy maps a constructor of R to the corresponding constructor in one or several partial algebraic types; in the second case, an additional case analysis on a_κ is performed.

In the rest of this section, the type of a_κ is denoted here by *Pilot*. Its parameters are denoted by $(\vec{q} : \vec{Q})$, its indices by $(\vec{b} : \vec{B})$, its constructors by D_j with arguments $(\vec{y}_j : \vec{Y}_j)$ and the $\vec{\beta}_j$ are terms instantiating the indices $\vec{b}$ in the conclusion of D_j . Its sort $\mathcal{T}$ must be informative: *Set* or *Type*. The arity of *Pilot* is then $\forall (\vec{q} : \vec{Q}) (\vec{b} : \vec{B}), \mathcal{T}$. In summary *Pilot* is defined as:

(Co)Inductive *Pilot* $(\vec{q} : \vec{Q}) : \forall (\vec{b} : \vec{B}), \mathcal{T} := \dots \mid D_j : \forall (\vec{y}_j : \vec{Y}_j), \text{Pilot } \vec{q} \vec{\beta}_j$.

3.2.1 Partial Algebraic Types

To partition the algebraic type, the shapes considered here follow the elimination scheme of *Pilot*: they are the primitive patterns $\psi_j = D_j \vec{y}_j$ formed by the application of a constructor of the pilot type D_j to as many fresh variables $\vec{y}_j$ as needed to form an inhabitant of *Pilot* $\vec{q} \vec{b}$. The partial algebraic type corresponding to ψ_j is called $R.D_j$.

Each constructor C_i of R has an instantiated pilot $\alpha_{i,\kappa}$. The latter is said to be *compatible* with the pattern ψ_j if one of the two following conditions holds:

1. $\alpha_{i,\kappa}$ is $D_j \vec{z}_j$, where $\vec{z}_j$ are terms with free variables in the arguments $\vec{x}_i$ of C_i . We define the substitution $\sigma_j = \{\vec{y}_i \mapsto \vec{z}_j\}$ such that $\alpha_{i,\kappa} = D_j \vec{z}_j = D_j \vec{y}_j \sigma_j = \psi_j \sigma_j$.

2. $\alpha_{i,\kappa}$ is $x_{i,\theta}$, where the variable $x_{i,\theta}$ is the argument of the constructor C_i at position θ . We define the substitution $\sigma_j = \{x_{i,\theta} \mapsto D_j \vec{y}_j\}$ such that $\alpha_{i,\kappa} \sigma_j = x_{i,\theta} \sigma_j = D_j \vec{y}_j = \psi_j$.

The partial algebraic type R_D_j is defined by gathering all constructors $C_i_D_j$ which mirror a constructor C_i whose instantiated pilot $\alpha_{i,\kappa}$ is compatible with ψ_j – intuitively, only “relevant” constructors of R are selected. The general form of R_D_j is:

Variant $R_D_j (\vec{p} : \vec{P}) : \forall \vec{a}_{<\kappa} \vec{y}_j, \text{ let } a_\kappa := \psi_j \text{ in } \forall \vec{a}_{>\kappa}, S :=$
 $| C_i_D_j : \forall \vec{x}_i, \text{ let } \vec{y}_j := \vec{z}_j \text{ in } R_D_j \vec{p} \vec{\alpha}_{i,<\kappa} \vec{y}_j \vec{\alpha}_{i,>\kappa} \quad \text{when } \alpha_{i,\kappa} \text{ is } D_j \vec{z}_j$
 $| C_i_D_j : \forall \vec{x}_{i,<\theta}, \forall \vec{y}_j, \text{ let } x_{i,\theta} := \psi_j \text{ in } \forall \vec{x}_{i,>\theta}, R_D_j \vec{p} \vec{\alpha}_{i,<\kappa} \vec{y}_j \vec{\alpha}_{i,>\kappa} \quad \text{when } \alpha_{i,\kappa} \text{ is } x_{i,\theta}.$

It is non-recursive, its parameters are the same as for R , its indices are the indices of R where the pilot index a_κ has been replaced by the (possibly empty) sequence of variables of the pattern ψ_j , and where the occurrences of a_κ in subsequent dependent indices have been replaced by ψ_j , using **let in** constructs. The types of the partial constructors are obtained in both cases by applying the substitution σ_j to the type telescope of each constructor C_i .

3.2.2 Proxy Type

The proxy type is defined by case analysis on the pilot a_κ , returning the suitable partial algebraic type corresponding to basic patterns of the pilot type. One has to be careful because the type of indices greater than κ , contained in $\vec{a}_{>\kappa}$, may depend on indices up to κ .

Definition $R_proxy_type : \forall (\vec{p} : \vec{P}) (\vec{a} : \vec{A}), S := \text{fun } \vec{p} \vec{a}_{<\kappa} a'_\kappa \Rightarrow$
 $\text{match } a'_\kappa \text{ as psi return let } a_\kappa := \text{psi in } \forall (\vec{a}_{>\kappa} : \vec{A}_{>\kappa}), S \text{ with}$
 $\dots$
 $| D_j \vec{y}_j \Rightarrow \text{let } a_\kappa := D_j \vec{q} \vec{y}_j \text{ in fun } (\vec{a}_{>\kappa} : \vec{A}_{>\kappa}) \Rightarrow [R_D_j] \vec{p} \vec{a}_{<\kappa} \vec{y}_j \vec{a}_{>\kappa}$
 end.

Here, $[R_D_j]$ is a placeholder for a variable of the corresponding $R_abproxy_type$, to be substituted by R_D_j or by an isomorphic term, as mentioned at the beginning of Section 3. If for a_κ we have a dependent type T that actually depends on previous indices in $\vec{a}_{<\kappa}$ (for instance, we have $a_\kappa : \text{vect } X (S \ n)$, where n is in $\vec{a}_{<\kappa}$), we define first a PBSI for T and then use it in the standard way in the above R_proxy_type :

$\text{match } T_proxy a'_\kappa \text{ as psi return let } a_\kappa := \text{psi in } \forall (\vec{a}_{>\kappa} : \vec{A}_{>\kappa}), S \text{ with } \dots \text{ end.}$

3.2.3 Proxy Substitution Functions

For each constructor C_i of R , we have a proxy substitution function that operates on placeholders $[C_i]$. According to the compatibility hypothesis for the constructor D_j given in 3.2.1, we have one of the two following cases:

1. When $\alpha_{i,\kappa}$ is $D_j \vec{z}_j$, the proxy substitution function operates as follows:

$$[C_i] \rightsquigarrow [C_i_D_j].$$

2. When $\alpha_{i,\kappa}$ is $x_{i,\theta}$, a further case analysis on $x_{i,\theta}$ is needed to select the appropriate partial algebraic type. The proxy substitution function operates then as follows:

$$[C_i] \rightsquigarrow \lambda(\vec{x}_i : \vec{X}_i) \Rightarrow \text{match } x_{i,\theta} \text{ return } R_proxy_type \vec{a} \{a_\kappa \mapsto x_{i,\theta}\} \text{ with}$$

$$\dots$$

$$| D_j \vec{z}_j \Rightarrow [C_i_D_j] \vec{\alpha}_{i,<\theta} \vec{z}_j \vec{\alpha}_{i,>\theta}$$

$$\text{end}$$

3.2.4 Example

To illustrate the second compatibility hypothesis above, consider the following inductive definition for “greater than or equal to”.

```
Inductive ge2 : N → N → Prop :=
| G0 : ∀ n, ge2 n 0
| GS : ∀ n m, ge2 n m → ge2 (S n) (S m).
```

We take the first index as the pilot index. In the first constructor `G0`, the corresponding pattern for the pilot index is a variable. It is therefore compatible with the two constructors of $\mathbb{N}$, and then the two partial algebraic types have a constructor corresponding to `G0`. They are respectively, before parameterization:

```
Variant ge2_0 : N → Prop :=
| G0_0 : ge2_0 0.
Variant ge2_S : N → N → Prop :=
| G0_S : ∀ n, ge2_S n 0
| GS_S : ∀ n m, ge2 n m → ge2_S n (S m).
```

There is nothing special about the proxy type.

```
Definition ge2_proxy_type (n : N) : N → Prop :=
  match n with 0 ⇒ ge2_0 | S n ⇒ ge2_S n end.
```

In the proxy, a further case analysis on `n` is needed to select the right mirror constructor corresponding to `G0`.

```
Definition ge2_proxy {n m} (g : ge2 n m) : ge2_proxy_type n m :=
  match g with
  | G0 n ⇒ match n with 0 ⇒ G0_0 | S n ⇒ G0_S n end
  | GS n m g ⇒ GS_S n m g
  end.
```

3.3 Final Transformation : Parameterization

The partial algebraic types generated by a series of specializations have indices that come from variables in a pattern of a pilot index. As indicated in Section 2, it is important to have parameters instead of those indices. To this effect, we consider a last transformation of independent interest, called *parameterization*, to be applied to the above partial algebraic types. Given an algebraic type R , it constructs a similar algebraic type R^p where indices are turned into parameters as much as possible. R^p has the same arity as R up to possible reordering, due to the fact that parameters always precede indices.

To be transformable into a parameter, an index must follow two conditions:

1. In the arity of R , its type can only depend on parameters.
2. In the conclusion of all constructors, the actual index must be a variable.

Indices in $\vec{a}$ are considered from left to right. If an index fulfills the above conditions, it is transformed into a parameter and then moved to the end of $\vec{p}$ (condition 1 on previous indices is not altered), otherwise it is left unchanged.

4 Implementation

The automated generation of our auxiliary definitions, in particular partial algebraic types, requires the manipulation of ROCQ representations and access to internal object definition APIs. We chose to use METAROCQ [25] because there is a reasonable hope that our code could be later translated in OCaml in order to improve its performances, since objects are represented in the same way in METAROCQ as in the OCaml source code of ROCQ. We have observed that our algorithm performs poorly in developments which depend on large libraries such as `mathcomp`. This has been traced back to the need in METAROCQ to translate a large part of the context in which the inverted inductive types are defined. Each translation involves a deep copy of the context from its native ROCQ internal representation to a reified representation as a ROCQ term. A native OCaml plugin could be a game changer in such cases.

Technical details about the architecture of our implementation and the required METAROCQ structures were presented in a previous article [11]. The technical aspects described there remain valid, even though this article describes a very preliminary version of our work dedicated to an explainable alternative of the `inversion` tactic. All transformations described in Section 3 are implemented, except the case where indices depend on indices that is described at the end of 3.2.2. This case requires more engineering work, and anyway it is not a theoretical limitation of our approach, as explained in 3.2.2. The idea is also illustrated in one of our handcrafted example dedicated to even bounded natural numbers. In the other examples we have examined, this feature was not necessary.

Our tool is available on github at <https://github.com/BasileGros/proxy-based-small-inversions>, as a ROCQ plugin. It includes a file `tutorial.v` that explains how to use it in common cases, as well as a gallery of more advanced examples. The main commands to be used are “`Derive InvProxy for R`” and “`Derive Dependent InvProxy for R`”, for basic and dependent PBSI, respectively. Options are available to tune pilot indices. The code generated is identical to that illustrated in Section 2 for `vect`. For convenience, the proxy type and the proxy function are also encapsulated within a `Rocq` class, allowing us to provide the `sinv` and `sdinv` tactics, which perform small inversion and dependent small inversion, respectively. Instructions are also provided to help users, if they wish, make their development independent of this plugin.

5 Related Work

The notion of inversion in ROCQ is first discussed in [13]. It provides the background of the extremely versatile and frequently used `inversion` tactic currently present in ROCQ. However, it is actually quite complex and its proof term generation produces sub-goals cluttered with an unpredictable number of equality hypotheses and the generated proof-terms have a large size, with reduced readability. As such it does not satisfy the trustworthy criteria discussed in 1.1. In addition, common cases in dependent programming lead to unsolvable issues, such as equality between dependent pairs that necessitates UIP (Uniqueness of Identity Proofs).

The initial focus of small inversions [22, 21] was to obtain shorter proof terms as well as proof scripts that were easier to maintain than those that could be obtained using the above `inversion` tactic, by properly using the return clause so that bulky additional equalities could be avoided. An important evolution of the elaboration of dependent pattern matching of ROCQ, partly inspired by [22], is under study and in the process of being implemented [1]. The version presented in [21] uses a generalized formula in the return clause, with a practical motivation in mind: to offer a fine-tuned elimination principle that is encoded separately and allows for more modular scripts by allowing to quantify on the conclusion of the current sub-goal. This

elimination principle can be thought about as the well-known functional impredicative encoding of basic PBSI. In fact, PBSI were originally designed as an improvement on [21] for use in an educational setting, specifically for teaching the fundamentals of semantics to students who had encountered inductive types and relations but were unfamiliar with continuations, in order to offer them a practical and easily understandable alternative to the *inversion* tactic. An English version of this course is available at [20]. The work reported in [11] aimed at supporting this approach to inverting inductive *relations*. The limitations described at the end of [11] have since been overcome, thanks to the decomposition into transformations described in Section 3, with one exception as explained in Section 4. The present paper also examines the more challenging setting of programming with precise dependent types, including in the case of co-inductive types. In particular, dependent PBSI were not considered in [11].

To some extent, PBSI can be considered as a special kind of *views*, a concept proposed by Wadler [30] then improved in [19] using dependent types. There, the idea is to translate a type A into a single isomorphic inductive type V (the view) in order to perform pattern matching on $a : A$ against patterns in V . The main construct is `match view_in a with ... end` to which our `match proxy a with ... end` is indeed similar. A distinctive point of PBSI is that the *type* of `proxy a` is *computed* (using a large elimination) towards a relevant partial algebraic type, so that we actually have a multiplicity of views that work together, a particularity which is beyond the motivations of [30] and [19] – it is actually beyond the limitations of the simply typed programming language considered in [30].

Programming with dependent types has been considered for a long time in the programming language community [2, 32, 5]. It has also been extensively studied in the framework of proof assistants coming from Type Theory, such as AGDA or ROCQ. Previous work on using dependent pattern matching in CIC has mainly focused on the design of suitable *return* clauses, see, e.g., the *convoy pattern* [6]. If we look at the ease of expression, the AGDA setting is undoubtedly more comfortable. For instance, `map2` can be defined by Equations (1). This is due to an advanced pattern matching mechanism based on first-order unification, as initially proposed by Coquand (see [9] and Figure 8 of [19]). However, UIP was hard-wired in the first implementations, raising issues related to the unification rule associated to provable instances of UIP (see [8] p. 26) and also its incompatibility with the univalence axiom coming from Homotopic Type Theory. This was repaired later using a more flexible approach due to Cockx and Devriese [7, 8]. Even then, Sozeau and Mangin argue in [26] that the latter is axiomatic, whereas their definitional approach for ROCQ – the EQUATIONS plugin, based on essentially the same algorithm – is by design more trustworthy, in light of the de Bruijn criterion. The reader is referred to [7, 8, 26] for more details.

An additional feature is common to all approaches based on first-order unification: in the tradition of first-order logic, as implemented in languages such as PROLOG, they include the well-known occur-check in order to prohibit cyclic structures. It is not innocuous, because proving $\forall x : \mathbb{N}, x = \mathbf{S} x \rightarrow \perp$ essentially requires an induction on x . If we consider a co-inductive version of natural numbers, we do have a term x such that $x = \mathbf{S} x$ and we want to be able to express pattern matching on such an x as well. Indeed, prohibiting cyclic structures at the level of pattern matching calls into question the possibility of using pattern matching on co-inductive structures, which can be viewed as conflicting with the separation of concerns mentioned in 1.2. By contrast, there is no occur-check in PBSI. As a result, for instance, we can define `map2` on co-vectors indexed by co-nats using PBSI using exactly the same piece of code as in Section 2.1. Co-inductive types are currently not in the scope of EQUATIONS.

Table 1 provides a summary of the conclusions we can draw from the file available at [12]. In this file, we compare the various approaches to dependent pattern matching discussed above

	inversion	SI'2013	EQUATIONS	PBSI
map2	5 8	6	3	6
map2 CIC	35	10	22	8
map2 proof	✗	✗✓	✓	✓
comap2 (cofixpoint)	✓	✓	✗	✓
remap2	✗	✗ 6	3	6
remap2 CIC	✗	✗10	55	12
remap2 proof	✗	✗✓	✓	✓
proj_even2 on bounded nat	✗	2	2	2*
proj_even2 CIC	✗	2	21	2
proj_sqeven4	✗	3	3	3*
proj_sqeven4 CIC	✗	3	54	4
source code readability	+	+	+++	++
trust criteria	XXX	✓✓✓	✓✗✗	✓✓✓

Table 1: comparison summary

for Rocq for three series of examples: first the function `map2`, presented in 2.1 – together with `comap2` which is a co-fixpoint version of `map2`; then `remap2`, presented in 2.2, and finally a simple program inspired by [21], that computes a projection `proj_even2` on even bounded natural numbers, together with its iterated version `proj_sqeven4`, so that we can illustrate pattern matching on terms with up to 3 levels of indices. Those programs are displayed in Figure 1. The ‘*’ used in the PBSI column for those programs recalls that the current automated version of PBSI cannot handle indexed indices – there we used handcrafted PBSI. For each program we give the number of lines of the corresponding code in [12], as a very rough complexity measure. The lines whose title includes “CIC” contain the size of terms displayed by ROCQ. As expected, the CIC code coming from programs using small inversions is quite close to the original code and is easy to recognize, whereas this is far from being the case for the other approaches.

The leftmost answers given in column “SI’2013” correspond to small inversions as presented in [21]: dependent inversions were not in their scope. However, dependent PBSI can be translated using continuations in the same manner as for basic PBSI; the corresponding answers are put on the right. For instance, “✗ 6” for `remap2` means that the translation using continuations of the PBSI program takes 6 lines.

The last lines “source code readability” and “trust criteria” provide a qualitative appreciation of each approach. For trust criteria, we give three answers corresponding to the criteria as stated in Section 1.1, namely the de Bruijn criterion, Pollack’s consistency and the source-level coherence criterion. The mark given to EQUATIONS corresponds to scenarios where you need to perform your own reduction steps in a proof script: you are then faced with sub-terms coming from the CIC translation of your source code that are difficult or impossible to understand. Indeed, such terms are clearly not intended to be directly exploited – as for the [inversion](#) tactic, CIC terms produced by EQUATIONS are much too long and complicated. A number of additional tools are provided in the EQUATIONS package, through which you can hope to carry out reasoning. All those tools, as well as the layer for translating the source language of EQUATIONS, do not belong to Rocq’s kernel, and then must be added to the trusted computing base.

```

(* Bounded natural numbers *)
Inductive bn :  $\mathbb{N} \rightarrow \text{Set}$  :=
| B0 :  $\forall \{n\}, \text{bn } (S \ n)$ 
| BS :  $\forall \{n\}, \text{bn } n \rightarrow \text{bn } (S \ n)$ .

(* Even bounded numbers, together with a projection function *)
Inductive even :  $\forall \{n\}, \text{bn } n \rightarrow \text{Type}$  :=
| even0 {n} : even (n := S n) B0
| even2 {n} {i: bn n} : even i  $\rightarrow$  even (BS (BS i)).

Definition proj_even2 n (i : bn n) (e : even (BS (BS i))) : even i :=
  let (e') := even_proxy e in e'.

(* Square even: same scenario on the even family rather than on bn *)
Inductive sqeven :  $\forall \{n\} \{i : \text{bn } n\}, (\text{even } i) \rightarrow \text{Type}$  :=
| sqeven0 {n} : sqeven (n := S n) (i := B0) even0
| sqeven4 {n} {i: bn n} {e : even i} : sqeven e  $\rightarrow$  sqeven (even2 (even2 e)).

Definition proj_sqeven4 n (i : bn n) (e : even (BS (BS i)))
  (s : sqeven (even2 (even2 e))) : sqeven e :=
  let (s') := sqeven_proxy s in s'.

```

Figure 1: even bounded natural numbers

6 Conclusion

Compared with what can be done with the `inversion` tactic, PBSI and EQUATIONS are much better approaches for programming with dependent types. EQUATIONS offer better ease of expression, an advantage shared with other tools like Agda that are based on first-order unification. However, like with `inversion`, the underlying CIC terms are quite large and hard to read. When they are exposed to the end-user, for instance after a computation step, there is no way to get the original source code. Using PBSI allows the user to shift the compromise line: the expression of programs is slightly less pleasant, but the criteria exposed in 1.1 are respected, including Pollack’s criterion and the coherence criterion. We use CIC “as is” without needing to introduce any specific syntactic constructs. Thanks to proxies, the generalization over indices of CIC pattern matching is no longer an issue and at the same time, information about actual indices is transported in parameters and then naturally preserved without additional equalities. Our approach is strengthened by a tool that relieves the user of the tedious part of PBSI. This is without compromising trust since we only have to check that the *results* provided by this tool are readable, axiom-free and well-typed; more precisely, as for the last two points, we only have to let ROCQ’s kernel check them for us, by de Bruijn criterion. A formal justification of the code dedicated to automation – for example, to ensure that the produced auxiliary definitions are always well-typed – would be nice from a theoretical standpoint, but it is by no means essential when it comes to trust. Of course, such a justification remains relevant to totality, predictable applicability, and faithfulness of the generated proxy.

Beyond that, regarding the interactive development of proofs, the automated PBSI offers an alternative to perform inversion on algebraic relations of independent interest.

References

- [1] W. Ait-Moussa, Pierre Boutiller, Hugo Herbelin, Meven Lennon-Bertrand, Thierry Martinez, and Gabriel Scherer. Compilation of dependent pattern-matching using small inversion. In *TYPES 2026 Abstracts*, Gothenburg, May 2026.
- [2] Lennart Augustsson. Cayenne - A language with dependent types. In S. Doaitse Swierstra, Pedro Rangel Henriques, and José Nuno Oliveira, editors, *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, volume 1608 of *Lecture Notes in Computer Science*, pages 240–267. Springer, 1998. doi:10.1007/10704973_6.
- [3] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.
- [4] The bbchallenge Collaboration, Justin Blanchard, Daniel Briggs, Konrad Deka, Nathan Fenner, Yannick Forster, Georgi Georgiev, Matthew L. House, Rachel Hunter, Iijil, Maja Kadziolka, Pavel Kropitz, Shawn Ligocki, mx dys, Mateusz Naćiszewski, savask, Tristan Stérin, Chris Xu, Jason Yuen, and Théo Zimmermann. Determination of the fifth busy beaver value, 2026. URL: <https://arxiv.org/abs/2509.12337>, arXiv:2509.12337.
- [5] Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.
- [6] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: <http://mitpress.mit.edu/books/certified-programming-dependent-types>.
- [7] Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017. URL: <https://lirias.kuleuven.be/handle/123456789/583556>.
- [8] Jesper Cockx and Dominique Devriese. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *J. Funct. Program.*, 28:e12, 2018.
- [9] Thierry Coquand. Pattern matching with dependent types. In B. Nordstrom, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Program*, pages 66–79. Chalmers University of Technology, 1992.
- [10] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. doi:10.1007/3-540-52335-9_47.
- [11] Pierre Corbineau, Basile Gros, and Jean-François Monin. Proxy-based small inversions: a case study in metacoq programming. In Adrien Guatto and Marie Kerjean, editors, *JFLA 2025 - 36es Journées Francophones des Langages Applicatifs*, Jan 2025.
- [12] Pierre Corbineau, Basile Gros, and Jean-François Monin, 2026. URL: <https://www-verimag.imag.fr/DIST-TOOLS/PBSI/>.
- [13] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in Coq. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 85–104, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- [14] Peter Dybjer. Inductive families. *Formal Aspects Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.
- [15] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [16] Eduardo Giménez. Structural recursive definitions in type theory. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 397–408, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

- [17] Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics*, pages 333–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [18] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.
- [19] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.
- [20] Jean-François Monin, 2024. URL: <https://www-verimag.imag.fr/~monin/Teaching/ECNU-2024/>.
- [21] Jean-François Monin and Xiaomu Shi. Handcrafted Inversions Made Operational on Operational Semantics. In S. Blazy, C. Paulin, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 338–353, Rennes, France, July 2013. Springer.
- [22] Jean-François Monin. Proof Trick: Small Inversions. In Yves Bertot, editor, *Second Coq Workshop*, Edinburgh, United Kingdom, July 2010. URL: <https://hal.inria.fr/inria-00489412/en/>.
- [23] Robert Pollack. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 10 1998.
- [24] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018. doi:10.1145/3158104.
- [25] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM*, 72(1), January 2025. doi:10.1145/3706056.
- [26] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341690.
- [27] The CompCert Development Team. Main website of the CompCert project. <https://compcert.org/>, 2024.
- [28] The Coq Development Team. Main website of the Rocq project. <https://rocq-prover.org/>, 2025.
- [29] The Coq Development Team. The Coq reference manual, release 9.2.0, 2026.
- [30] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 307–313. ACM Press, 1987. doi:10.1145/41625.41653.
- [31] Benjamin Werner. *Une Théorie des Constructions Inductives*. Theses, Université Paris-Diderot - Paris VII, May 1994. URL: <https://theses.hal.science/tel-00196524>.
- [32] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 214–227. ACM, 1999. doi:10.1145/292540.292560.