

# Proof Trick: Small Inversions\*

Jean-François Monin

Université de Grenoble and CNRS-LIAMA

Beijing

jean-francois.monin@imag.fr

We show how an inductive hypothesis can be inverted with small proof terms, using just dependent elimination with a diagonal predicate. The technique works without any auxiliary type such as `True`, `False`, `eq`. It can also be used to discriminate, in some sense, the constructors of an inductive type of sort `Prop` in Coq.

## 1 Introduction

In the process of proving some goal  $\dots H_i : T_i \dots \vdash C$ , we often need to invert an hypothesis  $H_i$ . More precisely, if its type  $T_i$  is an instance  $I(\mathbf{a})$  of an inductive type family  $I(\mathbf{x})$ , we want to select the only possible constructors for  $I(\mathbf{a})$  and to get the corresponding components. Common instances of this situation occur when reasoning on language operational semantics given in small-step or big-step style [7]. For the sake of simplicity, let us consider a predicate for multiples of 3 defined (in Coq syntax [9, 1]) by

```
Inductive mul3 : nat -> Prop :=
  | T0 : mul3 0
  | T3 : forall n, mul3 n -> mul3 (3 + n).
```

If we have an hypothesis of type, say, `mul3 (S 0)`, we would like to consider it as absurd. Or, from `mul3 13` we would like to infer `mul3 10`, `mul3 7`, etc., yielding the absurd.

In informal reasonings, we generally consider such consequences as obvious, because “no rule yields `mul3 1`” and “`mul3 13` can only come from the second constructor `T3`, applied to `10`”. But formally the justification has to be given using regular case analysis. Finding the right pattern is often tricky. In Coq, this can be done automatically thanks to a very useful tactic called `inversion` [3]. However both the proof term and the underlying reasoning are quite large in the current implementation, which makes the corresponding *explanation* somewhat hard to follow. From a practical perspective, inversion is useful in writing programs with dependent types [6, 2]; such programs cannot be written directly – we have to switch to interactive mode<sup>1</sup> – and running them entails heavy computations.

Indeed, experimented Coq users prefer to specify their problem using functions rather than relations whenever possible, because in some sense, conversion provides inversion for free. This may require some work. For example in the previous example, `mul3` can be seen as a *partial* function. Considering the following total function `mod3`:

```
Fixpoint mod3 n : nat := match n with +3 n => mod3 n | n => n end
```

we get a better equivalent formulation for `mul3 n` which is `mod3 n = 0`.

However it is not always so easy to guess a suitable generalization, for instance if we work with types different from `nat`, or with non-deterministic relations such as the rules for structural operational semantics of a programming language or of a process calculus.

\*This work was supported by ANR project SIVES, ANR-08-BLAN-0326-01.

<sup>1</sup> In many situations, it is possible to mix direct and interactive mode using the tactics `refine` or `program`.

The purpose of the present paper is to show that inversion can be implemented just by dependent elimination with an auxiliary diagonal function, and that tailoring this function to the intended use of the inversion results in manageable terms, shorter explanations and in more efficient programs.

**Related work.** The inversion tactic of Coq was first implemented by C. Murthy, then by C. Cornes [3]. C. McBride did a similar work for LEGO [5].

## 2 Two Facets of Inversion

Recall that inversion has two facets, which are illustrated on the previous example of multiples of 3:

- Extracting arguments (more generally: components) in relevant cases.
- Removing irrelevant cases.

The first facet is the easiest. The point is to ensure that the terms occurring in the hypothesis to be inverted occur in the conclusion as well. For example, in order to prove  $\text{mul3 } (3 + n) \vdash \text{mul3 } n$  we can first convert the conclusion to  $\text{mul3 } (\text{pred } (\text{pred } (\text{pred } (3 + n))))$ , that is, a well-suited predicate – let us call it  $P$  – applied to  $(3 + n)$ . We then proceed by cases on  $\text{mul3 } (3 + n)$ , which boils down to proving  $P x$  by cases on  $\text{mul3 } x$ . We then get two proof obligations which can be solved trivially:

- $P 0$ , which converts to  $\text{mul3 } 0$
- $\forall n, \text{mul3 } n \rightarrow P (3 + n)$ , which converts to  $\forall n, \text{mul3 } n \rightarrow \text{mul3 } n$ .

We will see later a systematical way for obtaining that. Note that, though the first ( $\text{mul3 } 0$ ) is trivial, it may seem a bit strange that we have to consider it, since  $0$  does not have the shape  $\text{mul3 } (3 + n)$ . Indeed, this case is rightly ignored by the standard inversion of Coq. This is clearly related to discrimination and absurdity, that is, the second facet. As this problem has to be coped with in the case of fully absurd hypotheses, such as  $\text{mul3 } 1$ , we mainly focus on such situations.

## 3 Inversion by Diagonalization of Absurd Hypotheses

For simplicity we consider inductive predicates on one argument. We then assume a data type  $D$  and an inductive predicate  $P : D \rightarrow \text{Prop}$  having  $n$  constructors  $C_i$ .

**Definition 3.1 (Absurd)** Let  $P t_i, 1 \leq i \leq n$ , the (type of the) conclusion of  $C_i$ . We say that an hypothesis  $H : P t$  is absurd when the term  $t$  matches no term among  $\{t_i \mid 1 \leq i \leq n\}$ .

In many practical cases, the  $t_i$  reduce to terms made only of constructors in  $D$ .

A short method for inverting  $H : P t$  is as follows. Consider the function *diag* defined by

```
diag x := match x with t => C | _ => P t end.
```

where  $C$  is the conclusion of the current subgoal. The latter is then convertible to  $\text{diag } t$ . A simple case analysis on  $H$  yields  $n$  subgoals which are all trivially solved since they are convertible to  $P t$ . The corresponding proof term can be written directly as follows, where all branches have the same shape, as indicated for the constructor  $C_i$ :

```

let diag x := match x with t => C | _ => P t in
match H in (P x) return (diag x) with
...
| C.i _ ... => H
...
end.

```

For example, a proof that anything can be deduced from `mul3 2` is:

```

Definition absurd2_prog : forall C: Prop, mul3 2 -> C :=
  fun C H =>
    let diag x := match x with 2 => C | _ => mul3 2 end in
      match H in (mul3 n) return (diag n) with
        | T0 => H
        | T3 _ _ => H
      end.

```

We see that there is no need for `False` nor for `True`. The type of  $H$  is not changed inside the branches of the pattern matching construct, so  $H$  has type `mul3 2`, which is convertible both to `diag 0` and to `diag (S(S(Sx)))`, the latter being convertible to `diag (3+x)`.

For a simple example of a non-deterministic relation, consider the following predicate `nd`, which expresses that a number is obtained by successive applications of functions `f` and `g` to `c`.

```

Inductive nd (f g: nat-> nat) (c: nat) : nat -> Prop :=
  | Nc: nd f g c c
  | Nf: forall n, nd f g c n -> nd f g c (f n)
  | Ng: forall n, nd f g c n -> nd f g c (g n).

```

The following script yields a 10 lines function, which, though somewhat tricky, is certainly more pleasant and efficient than the 102 lines obtained using standard inversion (the issue about size of proof terms is discussed in section 6).

```
Definition pl3 := plus 3.
```

```
Definition pl5 := plus 5.
```

```
Lemma absurd_nd4_interac : forall C: Prop, nd pl3 pl5 7 4 -> C.
```

```
Proof.
```

```
intros C H.
```

```
pose (diag x := match x with 4 => C | _ => nd pl3 pl5 7 4 end).
```

```
change (diag 4).
```

```
case H; (intros; exact H) || (intros n N; case N; intros; exact H).
```

```
Qed.
```

```
Definition absurd_nd4_prog : forall C: Prop, nd pl3 pl5 7 4 -> C :=
```

```
  fun C H =>
```

```
    let diag x := match x with 4 => C | _ => nd pl3 pl5 7 4 end in
```

```
    match H in (nd _ _ _ n) return diag n with
```

```
      | Nf _ N =>
```

```
        match N in (nd _ _ _ n) return diag (pl3 n) with
```

```
          | Nc => H
```

```

      | _ => H
    end
  | _ => H
end.

```

## 4 Eliminating Strong Elimination

The previous method, as well as the standard inversion mechanism of Coq, makes an essential use of strong elimination. That is, the `match` construct is applied to regular data in  $D$  (e.g. natural numbers, Booleans, lists, trees, etc.), to produce a result in `Prop`, which inhabits a higher universe. It is well known that this feature is needed for proving that constructors are different (unless it is axiomatically assumed, for instance in Peano arithmetics). If we redefine similar data structures in `Prop` instead of `Set`, for instance “hidden” natural numbers:

```
Inductive nat' : Prop := 0' : nat' | S' : nat' -> nat'.
```

We cannot prove that  $0' \neq S' 0'$ . This is precisely why proof irrelevance can be safely added as an axiom. At the same time this makes users reluctant to use such data structures.

Consider for example the version of `mul3` called `mul3'` using `nat'` instead of `nat`. Can we make any useful use of `mul3'`?

The point of the absurd is that anything can be deduced from it. There is little interest to try to infer an already known fact, hence in many cases, the most interesting consequences of an absurdity are just other absurdities. Clearly, `False` cannot be inferred, but this should not prevent us to try to infer other absurd facts. For instance, from `mul3 1`, we can infer `mul3 (1 + 3n)` by repeated applications of `T3`. We could also infer `mul3 (2 + 3n)` as soon as we get `mul3 x → mul3 y → mul3 (x + y)`, but it seems to be a peculiarity of arithmetics and there is no evidence how to generalize this method to other inductive types.

For a more useful example, aiming at defining a total function which computes the third of any multiple of 3, consider the binary relation `div3spec`, which ranges here over  $\text{nat}' \times \text{nat}'$  in order to make sure that strong elimination is not used (of course, the technique works on `nat` as well).

```
Inductive div3spec : nat' -> nat' -> Prop :=
  | D0 : div3spec 0 0
  | D3 : forall d n, div3spec d n -> div3spec (S d) (3 + n).
```

When proving  $\forall n, \text{mul3 } n \rightarrow \exists d, \text{div3spec } d n$ , by induction on  $n$ , we provide fake answers for numbers which are not multiples of 3, yielding goals such as `mul3 1 ⊢ div3spec fake 1`.

Fortunately, we can adapt the diagonal argument given above for proving directly `mul3 x` for any  $x$  and, more generally, any proposition  $P y$  provided that there is at least one element  $b$  such that  $P b$ . It should cover most practical situations – at least, the ones where a positive information is eventually expected.

This time the function `diag` has a small type as its codomain. We start with the case where the domain and the codomain are the same type  $D$ . Reusing the notation of section 3, we consider a goal  $\dots H : Pt \dots \vdash C$  where  $C$  is  $P s$  for some  $s$  in  $D$  and  $P t$  is absurd. The `diag` function that we consider is now

```
diag x := match x with t => s | _ => t end.
```

The current conclusion then converts to  $P(\text{diag } s)$  and the rest follows the same line as at the end of section 3. However note that, the use of  $H$  for discharging the subgoals provided by the constructors  $C_i$

is no longer artificial. We are also satisfied that inferring  $Ps$  from  $Pt$  takes essentially one step, as for the usual absurd-elimination rule.

For a more general case where  $C$  is  $Qy$ , given that we can prove  $Qb$ , we take

```
diag x := match x with t => y | _ => b end.
```

For example, we can infer  $0 = 1$  by converting it to  $0 = \text{diag } t$  with

```
diag =  $\lambda x.$  match  $x$  with  $t \Rightarrow 1$  |  $_ \rightarrow 0$  end.
```

## 5 Recursively Absurd Hypotheses

There are some differences between standard inversion and the previous method, when we want to reduce recursively absurd hypotheses, such as `mul3 10`. Repeating standard inversion yields successively `mul3 7`, `mul3 4`, `mul3 1`, no subgoals. With the method given here, it is more naturel, and indeed better, to define the diagonal function once for all from the beginning. For instance, if we want to prove `mul3 10 ⊢ 0 = 1`, we take:

```
pose (diag x := match x with 10 => 1 | _ => 0 end).
```

This amounts to say that we guess from the beginning that `mul3 10` is absurd, anticipating several applications of the constructors `T3` and `T0`. A still better – more efficient, in particular for values larger than `10` – option is to use a program instead of a `match`<sup>2</sup>:

```
pose (diag x := if nat_beq x 10 then 1 else 0).
```

A further advantage of the latter form is that it can be handled by the tactics language of Coq [4]. After converting the conclusion to  $0 = \text{diag } 10$  we get successively the goals  $\forall n, \text{mul3 } n \rightarrow 0 = \text{diag}(3 + n)$ ,  $\forall n, \text{mul3 } n \rightarrow 0 = \text{diag}(3 + (3 + n))$ , etc. – at each step we destruct the premise `mul3 n`. The intermediate goals are less readable than with standard inversion. However, it is easy to automatize the proof search with `Ltac` and we don't really need to read the intermediate steps.

```
Lemma invmul10 : mul3 10 -> 0 = 1.
```

```
Proof.
```

```
intro m.
```

```
pose (diag x := if nat_beq x 10 then 1 else 0).
```

```
change (0 = diag 10).
```

```
case m; clear m; reflexivity || intros n m.
```

```
repeat (case m; clear m n; reflexivity || intros n m).
```

```
Qed.
```

We see again that the resulting proof term is quite small (20 lines, see appendix). In contrast, the proof term we get by standard inversion takes at least 87 lines (some subterms are hidden by the pretty printer).

## 6 Size of Proof Terms

The printed version of proof terms is only a hint, but it may be misleading if we really want to compare the size of proof terms, because the implementation of Coq involves structure sharing (thanks “hash consing”, in particular). In order to compare the sizes more extensively, we did some experiments with

<sup>2</sup>Here we use `nat_beq` provided by `Scheme Equality` for `nat`.

the predicate `nd` given in the introduction: in this example, the combinatorics of inversions is much more important, since each inversion produces 2 subgoals if the argument of `tr15` is large enough. Here are the scripts.

```
Variable p : nat.
```

```
Definition tr15 := nd (plus 6) (plus 9) 15.
```

```
Lemma tr15_beq : tr15 40 -> tr15 p.
```

```
Proof.
```

```
intros t40.
```

```
pose (diag n := if nat_beq n 40 then p else 40).
```

```
change (tr15 (diag 40)).
```

```
assert (gen: forall n, tr15 n -> tr15 (diag n)).
```

```
  repeat (exact t40 ||
```

```
    intros n t; exact t40 || (case t; clear t n)).
```

```
  apply gen; exact t40.
```

```
Defined.
```

```
Lemma tr15_inv : tr15 n40 -> tr15 p.
```

```
Proof.
```

```
unfold tr5; intros t40.
```

```
repeat (match goal with [ H : nd _ _ _ _ |- _ ] =>
        inversion_clear H end).
```

```
Qed.
```

In this version, the produced `.vo` files are 1.7 times smaller with our method : 40 Ko against 68 K. But the difference turns out to be much higher if `f`, `g` and `c` are inlined: the produced `.vo` files are then roughly 10 times smaller with our method : 36 Ko against 364 K.

## 7 Other Experiments

In practice we have to cope with  $n$ -ary predicates, extraction of subterms in some branches while cutting off the others, non-injective functions, etc. The only puzzle is to find the right `diag` functions. We experienced this technique on several toy examples in order to assess its usability. We briefly present two of them, as well as an application to programming language semantics in the next section.

Our first example is the program for division by 3, using the definitions of `mul3` and `div3spec` given above. The statement of the problem has no equality in it, and the reader can check that so does the solution. We avoid strong elimination by using two `diag` functions. This development can then be done with `nat:Prop`. The corresponding proof term takes 61 lines.

With standard inversion (which requires `nat:Set`), the proof script is half the size, and the corresponding proof term takes 134 lines. The script is given in the appendix.

Our second example involves a non-injective function (`pred`), and an additional free predicate (`A`) – hence strong elimination. The `diag` function is guessed by a typical `Ltac` expression. The corresponding proof term is displayed on 17 lines. Our proof with standard inversion is slightly longer and the corresponding proof term is displayed on 55 lines.

```

Inductive l10 : nat -> Prop :=
  | l10ini : l10 10
  | l10rec : forall n, A n -> l10 (pred n).

Lemma inv_lA0 : l10 0 -> A 0 ∨ A 1.
Proof.
intro l.
match goal with [ l: ?P|- ?Q ] =>
  pose (diag n := match n with 0 => Q | _ => P end) end.
change (diag 0). case l; simpl; auto.
repeat (destruct n; simpl; auto).
Qed.

```

## 8 Application to Operational Semantics

Operational semantics of programming languages are an interesting application field for our technique for several reasons:

- Problems are naturally stated in terms of inductive relations.
- Examples are realistic, both in terms of size and of complexity.
- We often not only want to remove irrelevant cases, but also to get the components in the relevant case(s) (the two facets of inversion presented in section 2 are needed).

We relate here a preliminary experiment in this area. We decided to start from a relatively small but independent development, namely the material provided by B. Pierce for his course *Software Foundations* [8]. A simple imperative programming language is described in the file `Imp.v`. Here is the abstract syntax of commands.

```

Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

```

The types `aexp` and `bexp` represent arithmetic and Boolean expressions, respectively. Their semantics are given by two functions, `aeval` and `beval`. In contrast, the semantics of `com` is inductively defined by a ternary relation between two states and a command. The file `Imp.v` provides a big-step semantics as follows.

```

Inductive ceval : state -> com -> state -> Prop :=
  | CESkip : forall st, ceval st CSkip st
  | CEAss : forall st a1 n l, aeval st a1 = n ->
    (CAss l a1) / st ~> (override st l n)
  | CESeq : forall c1 c2 st st' st'',
    c1 / st ~> st' -> c2 / st' ~> st'' ->
    (CSeq c1 c2) / st ~> st''

```

*etc.*

```

where "c1 '/' st '~>' st'" := (ceval st c1 st').

```

The file `Imp.v` includes convenient notations for expressions and commands. For instance, the command for incrementing the variable  $X$  by 2 is given by

```
Definition plus2 : com :=
  X ::= !X +++ A2.
```

The first theorem of interest for us states that the value of  $X$  after running this program is its previous value plus 2:

Theorem `plus2_spec`:  $\forall st\ n\ st', st\ X = n \rightarrow plus2 / st \rightsquigarrow st' \rightarrow st'\ X = n + 2$ .

This requires an inversion of the hypothesis `plus2 / st  $\rightsquigarrow$  st'`. With the diagonalization technique, the full proof term takes 33 lines (240 lines with standard inversion). The part related to inversion of this term is even shorter: 11 lines, the additional lines are just rewriting steps in order to solve the remaining subgoal.

```
refine
  ( (let e := !X +++ A2 in
    let Heqe := refl_equal e in
      (let diag x y z :=
        match y with
        | v' ::= e' => e' = !v' +++ A2 -> x v' = n -> z v' = n + 2
        | _ => (X ::= e) / st  $\rightsquigarrow$  st'
        end in
      match Heval in (c / s  $\rightsquigarrow$  s') return (diag s c s') with
      | CEAss st0 a1 n0 l Heqn0 => _ (* remaining subgoal *)
      | _ => Heval
      end) Heqe) HX).
```

Some hints are necessary for explaining how to get this proof term. In order to get maximal information from a case analysis, the conclusion of the goal should contain occurrences of all relevant arguments. In our case, this goal is an application of `diag` to arguments, hence `diag` has 3 parameters. Here, as the initial conclusion contains nothing about the contents of the command to be analyzed (`plus2`), the right-hand side `!X +++ A2` of this assignment is first remembered under the form of an equality `e = !X +++ A2`. Then the conclusion can be seen as a function of the left-hand side  $X$ , the expression `!X +++ A2` and the states `st` and `st'`, and we can design a suitable definition of `diag`. The whole process is described in the Ltac language as follows.

Theorem `plus2_spec`:  $\forall st\ n\ st', st\ X = n \rightarrow plus2 / st \rightsquigarrow st' \rightarrow st'\ X = n + 2$ .

Proof.

```
intros st n st' HX Heval. revert HX.
match type of Heval with ?c / _  $\rightsquigarrow$  _ => unfold c in Heval end.
let tyHeval := type of Heval in
match tyHeval with (?v ::= ?exp) / ?s  $\rightsquigarrow$  ?s' =>
  pose (e := exp); generalize (refl_equal e: e = exp);
  pattern v, e, st, st';
  match goal with [ |- ?concl v e st st' ] =>
    pose (diag x y z := match y with
      | v' ::= e' => concl v' e' x z
      | _ => tyHeval end)
```

```

    end;
    change (diag s (v ::= exp) s')
  end;
  case Heval; try (intros; exact Heval); unfold diag; clear diag.
  intros; subst. apply override_eq.
Qed.

```

## 9 Conclusion and Further Work

We have shown on different examples that inversion can be efficiently implemented using dependent elimination with a diagonal predicate, providing better control of the user on the details and on the size of proof terms. We do not pretend to address the fully general problem of inversion. The standard mechanism offered in Coq is perfect for performing proofs quickly. And we can reasonably expect that the price to be paid for this generality is heavier terms. Nevertheless the scheme we use is quite systematic, and is already automated to some extent using Ltac. Our preliminary experiment on programming language operational semantics seems to show that the scope of inversion by diagonalization is broad enough for many practical usages. Further experiments are needed in order to confirm this assessment.

## Acknowledgement

The author wish to thanks the reviewers of Coq-2 for their many detailed comments and constructive suggestions.

## References

- [1] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag. Available at <http://www.labri.fr/publications/13a/2004/BC04>.
- [2] Frédéric Blanqui, Jean-Pierre Jouannaud & Pierre-Yves Strub (2008): *From Formal Proofs to Mathematical Proofs: A Safe, Incremental Way for Building in First-order Decision Procedures*. In: Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri & C.-H. Luke Ong, editors: *IFIP TCS, IFIP 273*, Springer, pp. 349–365. Available at [http://dx.doi.org/10.1007/978-0-387-09680-3\\_24](http://dx.doi.org/10.1007/978-0-387-09680-3_24).
- [3] Cristina Cornes & Delphine Terrasse (1995): *Automating Inversion of Inductive Predicates in Coq*. In: *TYPES*, pp. 85–104.
- [4] David Delahaye (2000): *A Tactic Language for the System Coq*. In: Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning (LPAR), Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI) 1955*, Springer, Reunion Island (France), pp. 85–95.
- [5] Conor McBride (1996): *Inverting Inductively Defined Relations in LEGO*. In: Eduardo Giménez & Christine Paulin-Mohring, editors: *TYPES, Lecture Notes in Computer Science 1512*, Springer, pp. 236–253.
- [6] James McKinna (2006): *Why dependent types matter*. *SIGPLAN Not.* 41(1), pp. 1–1.
- [7] Hanne Riis Nielson & Flemming Nielson (1992): *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA.
- [8] Benjamin C. Pierce, Chris Casinghino & Michael Greenberg (2009). *Software Foundations*. Available at <http://www.cis.upenn.edu/~bcpierce/sf>.

[9] The Coq Development Team (2008): *The Coq Proof Assistant Reference Manual – Version V8.2*. Available at <http://coq.inria.fr>. <http://coq.inria.fr>.

## Appendix

### Proof term for inverting mul3 10

```

fun m : mul3 10 =>
let diag x := if nat_eq x 10 then 1 else 0 in
match m in (mul3 n) return (0 = diag n) with
| T0 => refl_equal (diag 0)
| T3 _ m0 =>
  match m0 in (mul3 n0) return (0 = diag (3 + n0)) with
  | T0 => refl_equal (diag (3 + 0))
  | T3 _ m1 =>
    match m1 in (mul3 n1) return (0 = diag (3 + (3 + n1))) with
    | T0 => refl_equal (diag (3 + (3 + 0)))
    | T3 _ m2 =>
      match
        m2 in (mul3 n2) return (0 = diag (3 + (3 + (3 + n2))))
      with
      | T0 => refl_equal (diag (3 + (3 + (3 + 0))))
      | T3 n2 _ => refl_equal (diag (3 + (3 + (3 + (3 + n2)))))
    end
  end
end
end.

```

### Script for div3

```

Definition div3Set_small : forall n, mul3 n -> {t | div3spec t n}.
fix 1.
intros n; case n; clear n.
  intro m; exists 0. (* m: mul3 0 |- div3spec 0 0 *)
  pose (diag (x: nat) := 0). change (div3spec (diag 0) (diag 0)).
  case mv; intros; apply D0.

  intros n ; case n; clear n.
  intro m; exists 7 (* fake *). (* m: mul3 1 |- div3spec 7 1 *)
  pose (diag1 x := if nat_eq x 1 then 7 else 0);
  pose (diag2 x := if nat_eq x 1 then 1 else 0).
  change (div3spec (diag1 1) (diag2 1)). case m; intros; apply D0.

  intros n ; case n; clear n.
  intro m; exists 5 (* fake *). (* m: mul3 2 |- div3spec 5 1 *)
  pose (diag1 x := if nat_eq x 2 then 5 else 0);

```

```
pose (diag2 x := if nat_beq x 2 then 2 else 0).
change (div3spec (diag1 2) (diag2 2)). case m; intros; apply D0.

intros n m; case (div3Set_small n). (* m: mul3 (+3 n) /- mul3 n *)
pose (diag x := match x with S (S (S y)) => y | _ => 3 + n end).
change (mul3 (diag (S (S (S n)))))). case m; simpl.
  intros; exact m.
  intros; assumption.
intros t d3. exists (S t). apply (D3 t n d3).
Defined.
```