A step towards the mechanization of partial functions : domains as inductive predicates

Catherine Dubois[1] and Véronique Viguié Donzeau-Gouge[2]

[1] LaMI, CNRS EP738, Univ. d'Évry, Bd des Coquibus, 91025 Évry Cedex, France.
email: Catherine.Dubois@lami.univ-evry.fr
[2] Cedric, CNAM, Rue Saint-Martin, 75141 Paris Cedex, France.
email: donzeau@cnam.fr

Abstract. This work is centred on the specification of partial operations in a system based on a classical logic with total functions. We present a style with pre-conditions: our method enables calculation of the domain of a partial function `f` independently of calculation of `f`. We also study the influence of this style upon the proof facility and the later use of the specification.

1 Introduction

In this paper we are in the context of a logic which *does not* incorporate the notion of partiality and where any function is total. This choice is justified by the power of the underlying logic and by the expressive power of the associated languages.

In this context, various tricks are used to encode the partiality. In a typed world, a total function of type $\tau \rightarrow \tau'$ is defined for every value of type τ. Thus we have to encode a partial function whose arguments and result are respectively of type τ_1 and τ_2 into a total function of type $\tau \rightarrow \tau'$. Usually τ_1 and τ are identical but τ_2 and τ' are not: τ' is intended to encapsulate the fact that for some elements of type τ_1, the function has no effective result. The nature of τ' depends on the style chosen to encode the partiality.

A function can be simply described by the equations that define the function itself: it is the case when the defining equations may be directly translated into a predicate, in a Prolog style. But the functional aspect is lost. A functional encoding, close to an ML implementation can also be used and simulates the exception mechanism. This style may be easily translated into a program, however it is often clumsy, essentially because we have to explicitly propagate the exceptions. Thus it influences the surrounding specifications and proofs.

In [8] Müller and Slind discuss and exemplify some techniques usable in the proof system Isabelle/HOL.

1.1 Our Objectives

We start from the defining equations of the partial function, we call this first specification the *informal* specification of the function. The word informal emphasizes here that it is not yet a specification that the proof system can process.

The style we put forward maintains the functional flavour and incorporates the notion of pre-condition. More precisely, to any partial function, a supplementary parameter is added : the proof of the membership of the parameters (the original ones) to the domain of the function. Then, whenever the function is applied, a proof that the arguments are well formed has to be provided. The contribution of our work enables to generate automatically the type of the expected proof from the defining equations of the function. The computation of the type of the pre-condition is based on the work done by Finn, Fourman and Longley [6] that we have adapted and extended to take into account any kind of recursion. Our method enables calculation of the domain of the function `f` independently of calculation of `f`. To achieve this, we assume given a post-condition in addition to the defining equations for `f`. Clearly this approach is more *liberal* than approaches that require precise knowledge of the values of the function in question.

1.2 Specification language and proof system

Our approach demands the manipulation of proofs as objects and requires dependent types. In short, the specification language must be a typed language with inductive definitions, dependent types and functions. Lastly a tactic-driven system is recommended for an easier writing of the proofs and introducing some automation. To illustrate this work, we have chosen to use the proof assistant Coq [1].

1.3 Overview of the paper

After a brief presentation of the Coq system that emphasizes the tools used in this paper e.g. inductive definitions, dependent types, we describe formally the informal specifications we start from. We develop an algorithm to compute the type of the pre-condition and establish a soundness property.

2 The Coq proof assistant

We give here a brief presentation of the Coq interactive proof assistant. A more detailed description can be found in [1]. However, here we provide in detail some fundamental notions used in this article.

The Coq system allows the development of verified formal proofs. The axiomatizations and specifications are written in the logical language Gallina, the foundation of which is the Calculus of Inductive Constructions [10], a version of higher order typed λ-calculus whose types are themselves typed terms of the language. The system provides the user with inductive types, very close to ML datatypes and inductive relations which can be compared to Prolog predicates. For example, the type of lists whose elements are of type `A` is described in Coq by:

```
Inductive list: Set:= nil: list | cons: A -> list -> list.
```

The relation that expresses that an element belongs to a list may be introduced through the following inductive definition:

```
Inductive In: A -> list -> Prop:=
  In_head: (x: A)(l: list) (In x (cons x l))
| In_tail: (x, y: A)(l: list) (In x l) -> (In x (cons y l)).
```

This declaration must be read as the definition of the smallest relation verifying the two inference rules named `In_head` and `In_tail`. From the definition of an inductive construction, the Coq system automatically generates the associated induction principle and provides proof tools to manipulate them (e.g. the `Induction` and `Inversion` tactics). The user can define functions and also recursive functions: facilities are provided to write functions defined with structural recursion, for instance the `Fixpoint` construct.

Intuitively, we can summarize the notion of dependent type as a type indexed by a value. The typical example in this field concerns the type of the arrays of elements of a set A. In fact we define a family $\texttt{array}_A$ indexed on naturals and denoted in Coq by $\texttt{(n : nat)(array}_A\texttt{ n)}$.

3 Defining Equations

A function is defined by a bunch of equations as in [6]:

$$\begin{array}{llll}\texttt{fun} \; f\; pat_{11} \ldots pat_{1n} & \texttt{when} \; cond_1 & \texttt{=} & exp_1 \\ \quad\vdots & & & \\ \quad f\; pat_{m1} \ldots pat_{mn} & \texttt{when} \; cond_m & \texttt{=} & exp_m\end{array}$$

The pat_{ij} are linear ML-style patterns built from variables and constructors Ctr. The optional guards $cond_i$ are boolean expressions assumed well-formed, and they contain variables appearing in the related patterns $pat_{i1} \ldots pat_{in}$. There is no notion of sequential execution so the order of the equations does not matter. Consequently two distinct clauses cannot contain overlapping patterns. Of course, the patterns covered by such a definition are not necessarily exhaustive.

The expressions exp_i are ML-expressions and follow the usual syntax:

exp ::= $x \mid Ctr\ exp \mid \texttt{let}\ x\texttt{=}\ e_1\ \texttt{in}\ e_2 \mid f(e_1\ \ldots\ e_n) \mid g(e_1\ \ldots e_m)$

The formal semantics of this language is not given here and is quite usual.

Any function is *a priori* partial. However in the following, for a greater readability, g denotes a symbol of total function whereas f and h will be always associated to partial functions. The defining equations are implicitly recursive. Furthermore there is no restriction on the ways in which f can appear in the right hand-side expressions. We allow all partial general recursive definitions. The manipulated values are defined inductively and therefore, correspond to the closed terms of a free algebra.

Thus a function may be partial either because the patterns involved in the definition are not exhaustive or because some exp_i is not well-formed and consequently contains forbidden applications of other partial functions.

We give below the defining equations of some illustrating examples:
• the function `head` returns the first element of a list:

```
fun  head  e::l = e
```

• the function `nth` computes the `n`th element of a list. This function only makes sense when `n` is a natural number strictly less than the length of a non empty list. The function `nth` implements a structural recursion.

```
fun  nth  0    x::l = x
   | nth (S n) x::l = nth n l
```

• lastly, the function `mgu` computes the principal unifier of two terms built from the constant `Cst`, the function symbol `Op` and variables. It is only defined for terms that can be unified. The possible result is a substitution pretty-printed as a list of pairs. In the following definition, `(s t)` denotes the application of the substitution `s` on the term `t` and `o` the composition of substitutions. The function `mgu` illustrates the case of a nested recursion (see the last clause).

```
fun mgu Cst Cst = empty_subst
   |mgu Cst (Var x) = [(x,Cst)]
   |mgu (Var x) (Var y) when x=y = empty_subst
   |mgu (Var x) t when (notin x t) = [(x,t)]
   |mgu (Op t1 t2) (Var x) when (notin x (Op t1 t2)) = [(x, (Op t1 t2))]
   |mgu (Op t1 t2) (Op x1 x2) = let u1 = mgu t1 x1 in
                                   (mgu (u1 t2) (u1 x2)) o u1
```

4 Introduction of Pre-conditions

In this section, the idea is to provide the specification with the proof that the arguments belong to the domain of the function. For this purpose, we introduce a predicate which characterizes the membership to the domain. We can consider that predicate as a pre-condition of the function.

The definition of the predicate influences the expression of the function and consequently, its uses to come. Then we propose to compute this predicate automatically from the defining equations of the function and a possible post-condition. Let us illustrate this process with the simple `head` function. Intuitively, the membership predicate to the domain of this function specifies the non empty lists. Its natural notation is `( not l= nil)` which leads to a dependent function type `(l:list)( not l=nil) -> A`. Its definition realises a pattern matching of `l`: the case `l=nil` is fixed via a proof by contradiction. Alternatively, we can define the membership predicate as an inductive type with a unique constructor `dom_head1`. It is written in Coq as:

```
Inductive dom_head : list -> Set:=
dom_head1: (l: list)(x: A)(dom_head (cons x l)) .
```

Then the type of the function becomes: `(l:list)(dom_head l) -> A`; it is written as follows:

```
Definition head_proof:= [l: list] [p: (dom_head l)]
   Cases p of (dom_head1 l0 x) => x  end.
```

It is obtained by a simple pattern matching of the predicate. Now, we are going to present a way to compute the ad hoc appropriate predicate in order to obtain the function definition.

4.1 Ad hoc inductive types for the pre-conditions

Our goal is to extract from the defining equations of an operation f, the predicate *dom_f* which characterizes the membership to the domain definition. The construction of the function in this last style is based on the structure of the deduced inductive type. The computation of *dom_f* is drawn from [6].

However, in a general case, to define a partial function, we need to provide not only its defining equations but also a post-condition which characterizes its results. For instance, a post-condition for `mgu` may be that the `mgu` of two terms unifies the two terms. In this style, the function definition encapsulates the correctness proof of the function with respect to its post-condition.

Post-conditions are necessary as soon as the computation generates nested recursive calls, in particular, when non-structural recursion is used. Then the result of the nested recursive call is characterized by the post-condition of the called function. If the inductive predicate can be synthesized systematically, the building of the function is semi-automatic. Some verifications with respect to the post-condition are left to the programmer.

We can always compute the definition domain of a function, even if it does not terminate. But we do not know what is the meaning of a post-condition in this case.

4.2 Definition of *dom_f*

To each equation of the informal specification corresponds a constructor of the inductive predicate.
More formally, the equation $f\ pat_{i1} \ldots pat_{in}$ `when` $cond_i$ `=` exp_i
leads to the rule $dom\_f_i : \forall x_1, x_2 \ldots x_k . \Delta(exp_i) \wedge cond_i \rightarrow dom\_f(pat_{i1}, \ldots, pat_{in})$ where $x_1, x_2 \ldots x_k$ are the pattern variables and $\Delta(exp_i)$ denotes the predicate of well-formedness of exp_i.

For instance, if exp_i is the application $(h\ e)$, $\Delta(h\ e)$ states that e is well formed and belongs to the domain of h.
Consequently, if e is the application $(f\ e')$, $\Delta(h\ e)$ expresses that $(f\ e')$ is well-formed and belongs to *dom_h*. As it is, we obtain $dom\_h(f\ e')$ and, hence, in the definition of *dom_f* the symbol f appears. This situation would oblige us to define simultaneously f and *dom_f* but we want to avoid that. From here, our strategy diverges from the approach of Finn, Fourman and Longley exposed in [6].

In order to cope with this situation and also to associate some semantics to f, we replace a call $(f\ e')$ by a fresh variable, let us say y, which satisfies the post-condition associated to f, namely P_f, thus $dom\_h(f\ e')$ becomes $\forall y . P_f(e', y) \Rightarrow dom\_h(y)$.

Let us illustrate this with two simple examples :
• $f\ (S\ n)$ = $f(n)$ generates the following predicate: $dom\_f(n)$. (We do not need here any post-condition)
• for $f\ (S\ n)$ = $f(f(n))$, the predicate will be
$dom\_f(n) \Rightarrow (\forall m.P_f(n,m) \Rightarrow dom\_f(m))$.

4.3 Definition of Δ

In order to define Δ syntactically without increasing too much the complexity of its definition and the proof of soundness, we work on a canonical form of the expressions exp_i which emphasizes syntactically the call by value semantics and the structure of the computation. In particular, all the nested calls are named with fresh identifiers and introduced by `let` constructs. It is easy to convince ourself that this canonical form is semantically equivalent to the initial one (assuming a call by value semantics). For instance, the canonical form of f (`let` z = $(g\ (f\ x)\ (f\ y))$ `in` $z * z$) is[3]:

```
let y1 = f x in
 let y2 = f y in
  let z = (g y1 y2) in
   let y3 = z * z  in
    let y4 = f y3 in y4
```

Thus the canonical form of an expression contains a succession of nested `let` statements where each defining expression is a call to a function with arguments which are constants or identifiers i.e. always defined expressions. From this canonical form, we produce another syntactical expression denoted by $\widehat{e}$: it will be used to compute the Δ predicate, and it is obtained by replacing each defining expression `y=f x` by the condition $y|P_f(x,y)$ where P_f is the post condition associated to f. Everywhere $\widehat{e}$ and e are identical we note e instead of $\widehat{e}$. From the previous example we obtain:

```
let y1 | P_f(x, y1) in
 let y2 | P_f(y, y2) in
  let z = (g y1 y2) in
   let y3 | P_f(z * z, y3) in
    let y4 = f y3 in y4
```

Definition of the well-formedness of an expression .
$\Delta(e)$ returns a formula which expresses the well-formedness of e computed from $\widehat{e}$ with the following rules:
•$\Delta(x) = true$
•$\Delta(Ctr\ e) = \Delta(e)$
•$\Delta($`let` $y|P_h(e_1,y)$ `in` $\widehat{e_2}) = dom\_h(e_1) \wedge (\forall y.P_h(e_1,y) \Rightarrow \Delta(\widehat{e_2}))$

[3] If a lazy semantics had been chosen we would work with the canonical expression $(f\ (g\ (f\ x)\ (f\ y))\ (g\ (f\ x)\ (f\ y)))$

•Δ(`let` x= e_1 `in` $\widehat{e_2}$) = $(\forall x.x = e_1 \Rightarrow \Delta(\widehat{e_2}))$
•$\Delta(g\ e_1) = \Delta(e_1)$

Let us come back to the examples .
We compute the pre-conditions for `nth` and `mgu` from the informal definition.
For `nth`, the canonical form is

```
fun  nth  0    x::l = x
   | nth (S n) x::l = let y = nth n l  in y
```

which becomes the expression $\widehat{\texttt{nth}}$

```
fun  nth  0    x::l = x
   | nth (S n) x::l = let y | P_nth(n, l, y)  in y
```

Δ(`let` $y|\ P_{nth}(n, l, y)$ `in` y) $= dom\_nth(n, l) \wedge (\forall y.P_{nth}(n, l, y) \Rightarrow true)$ which can be simplified as $dom\_nth(n, l)$.
This computation suggests the Coq declaration given below. The conjunctions generated by Δ are replaced in the inductive Coq definition by implications. This choice will provide us with a best comfort in the proofs to come involving the dom_f predicate.

```
Inductive dom_nth: nat -> list -> Set :=
  dom_nth1: (a: A)(l: list) (dom_nth O (cons a l))
| dom_nth2: (p: nat)(a: A)(l: list)
    (dom_nth p l) -> (dom_nth (S p) (cons a l)).
```

For `mgu`, the canonical form of the last equation is:

```
   |mgu (Op t1 t2) (Op x1 x2) = let u1 = mgu t1 x1  in
                                 let y1 = u1 t2 in
                                  let y2 = u1 x2 in
                                   let y3 = mgu y1 y2 in
                                    let y4 = y3 o u1 in y4
```

and the corresponding $\widehat{mgu}$ is:

```
   |mgu (Op t1 t2) (Op x1 x2) -> let u1 |P_mgu(t1,x1,u1)  in
                                  let y1 = u1 t2 in
                                   let y2 = u1 x2 in
                                    let y3 |P_mgu(y1, y2,y3) in
                                     let y4 = y3 o u1 in y4
```

By applying the previous computation rules, Δ applied to the last equation returns after simplification:
$dom\_mgu(t_1, x_1)\wedge$
$(\forall u_1.P_{mgu}(t_1, x_1, u_1) \Rightarrow (\forall y_1.y_1 = (u_1\ t_2) \Rightarrow (\forall y_2.y_2 = (u_1\ x_2) \Rightarrow dom\_mgu(y_1, y_2))))$
We deduce the following Coq declaration:

```
Inductive dom_mgu : term -> term -> Set :=
  dom_mgu1 : (x: nat)(dom_mgu (Var x) (Var x))
| dom_mgu2 : (x: nat)(t: term)(notin x t) -> (dom_mgu (Var x) t)
| dom_mgu3 : (x: nat)(dom_mgu Cst (Var x))
| dom_mgu4 : (dom_mgu Cst Cst)
| dom_mgu5 : (t1, t2: term)(x : nat)
             (notin x (Op t1 t2)) -> (dom_mgu (Op t1 t2) (Var x))
| dom_mgu6 : (t1, t2, x1, x2: term)
             (dom_mgu t1 x1) ->
             ((s : substitution) (s t1) = (s x1)) ->
             (dom_mgu (s t2) (s x2)) ->
               (dom_mgu (Op t1 t2) (Op x1 x2)).
```

4.4 Soundness of Δ

Soundness property establishes that under the hypothesis that f terminates, if *dom_f*$(e_1, \ldots, e_n)$ holds, then $(f e_1 \ldots e_n)$ is defined. To prove this property, we consider an environment Γ which contains the definition of the free variables appearing in f and $e_1 ... e_n$ as well as the post-conditions for all the used partial functions, and satisfying *dom_f*$(e_1 \ldots e_n)$. This property can be formally defined and proved.

4.5 Encoding the functions

The type of a partial function f of arity n becomes in Coq the dependent type : $(x_1 : t_1)(x_2 : t_2) \ldots (x_n : t_n)(dom\_f\ x_1 x_2 \ldots x_n) \rightarrow T$ where T is $(r : t) P_f(x_1, x_2 \ldots x_n, r)$. For instance the mgu function receives the type (u, v: term) (dom_mgu u v) -> (unifier u v) when unifier is defined as follows:

```
Inductive unifier [u, v : term] : Set :=
  C: (s: substitution)(s u) = (s v) -> (unifier u v).
```

When initialy, a function is recursive, it remains recursive in this style, but its recursive argument is the proof pi of the pre-condition. In most of the cases, it can be defined in Coq with a Fixpoint: indeed, the recursive calls are done on strict sub-terms of the proof pi. For instance, in the case of nth, when the proof is built up using the constructor dom_nth2, its pattern matching gives the proof that the arguments of the recursive call belong to the domain of definition. The function nth is written :

```
Fixpoint nth_proof [n: nat; l: list; pi: (dom_nth n l)]: A:=
Cases pi of
  (dom_nth1 a l0) => a
| (dom_nth2 m a l0 h) => (nth_proof m l0 h)
end.
```

4.6 Proof schemes

The tools used in the proof development relate here in the inductive calculus and the associated tactics but in a context of dependent types.

In the expression of the surrounding lemmas, we can take advantage of the real functional aspect of the specification. We illustrate below some lemmas related to our examples :

1. the first element of a list belongs to that list
2. the nth element of a list belongs to that list
3. the free variables of the principal unifier of two terms belong to the union of the free variables of the terms.

```
Lemma head_proof: (l: list)(p: (dom_head l))
  (In (head_proof l p) l).

Lemma nth_proof: (l: list)(n: nat)(p: (dom_nth n l))
  (In (nth_proof n l p) l).

Lemma mgu_proof: (t1,t2: term)(p: (dom_mgu t1 t2))
 (is_included (FV (subst_of (mgu t1 t2))) ((FV t1) U (FV t2))).
```

In these three lemmas, the last quantified variable `p` denotes a proof, its type is the computed inductive predicate `dom`.
Generally speaking, the proofs of such lemmas are developed by induction (inversion may suffice when non recursive function) on the pre-condition `p`.

5 Conclusion

In this paper, we extend the approach of Finn, Fourney and Longley to generate definedness conditions and we present a way to deal with nested recursion by using post-conditions. More precisely, we underline a systematic way to compute *dom_f* as an inductive relation, that allows us to use the power of the inductive calculus. The computation of *dom_f* can be automated. A perspective to our work is to study the development of a tactic related to this automation.

The required post-condition must be given by the user, it is not synthezised automatically. It corresponds to older approaches for termination proofs, where termination of a nested recursive function could only be verified if its correctness was proved simultaneously. However, our post-condition is not necessarily a correctness property, it could be a weaker property (for instance for `mgu`, we could assert that the domain of the most general unifier of `t1` and `t2` is a subset of the free variables of `t1` and `t2`). Furthermore, we feel that simple typing conditions used as post-conditions may be sufficient.

Computing the domain of a function and proving its termination may be related. In particular, if the post-condition is an induction lemma as defined by Giesl [7], we obtain simustaneously a definition of the function and a proof that this induction lemma is partially correct in the sense of [7]. Furthermore, in this case, the generated proof obligations are arithmetic properties (about measures) which may be automated. To prove the termination of the function, it remains at least to establish that the recursive calls decrease according to the chosen measure.

We have to study more precisely the impact of the chosen post-condition upon the defined function.

This style may be drawn nearer to the notion of subtypes and the associated proof obligations available in the system PVS. But the PVS subtypes have to be given by the user.

All the proofs mentioned in this paper have been verified with Coq and their script should be asked from the authors.

Finally, we want to emphasize the fact that the examples are specified and proved in Coq but the results can be easily transposed in any proof system that provides total functions, inductive definitions and dependent types.

Acknowledgements We would like to thank the anonymous referees for their helpful comments.

References

1. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Pascal Manoury, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi and Benjamin Werner. The Coq Proof Assistant, Reference Manual, Version 6.1. INRIA, Rocquencourt, December 1996.
2. J. Brauburger and J. Giesl. Termination Analysis for Partial Functions. In Proceedings of the Third International Static Analysis Symposium (SAS'96). Aachen, Germany, Lecture Notes in Computer Science 1145, Springer-Verlag, 1996.
3. P. Behm, L. Burdy and J.-M. Meynadier. Well Defined B. *Proceedings of Second International B conference*, Montpellier, 21-24 April 1998.
4. C. Cornes and D. Terrasse. Automating Inversion of Inductive Predicates in Coq. *In BRA Workshop on Types for Proofs and Programs*, Turin, June 1995.
5. W. M. Farmer. Partial functions version of Church's simple theory of types. *Journal of Symbolic Logic.* 55(3), 1269-1291,1990.
6. S. Finn, M. Fourman, J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning.* 18(1), 85-104, 1997.
7. J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *Journal of Automated Reasoning.* 19, 1-29, 1997.
8. O. Müller, K. Slind. Treating Partiality in a Logic of Total Functions. *The Computer Journal*, Vol 36, No 5, 1997.
9. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML Programs in the System Coq. *Journal of Symbolic Computation–special issue on automated programming*, 15(5&6):607–640, May&June 1993.
10. Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, LNCS 442. Springer-Verlag, 1990. Also available as technical report CMU-CS-89-209.
11. F.Regensburger. HOLCF:Higher Order Logic of Computable Functions. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293-307. Springer-Verlag, 1995.
12. K. Slind. Function Definition in Higher-Order Logic In J. von Wrigh, J. Grundy, and J. Harrison, editors, *Proc. of the 9th Int. Conf. Theorem Proving in Higher Order Logics,*, volume 1125 of *Lecture Notes in Computer Science*, pages 381-398. Springer-Verlag, 1996.