PUBLICATION INTERNE

N^o 1766

DEFINING AND REASONING ABOUT GENERAL RECURSIVE FUNCTIONS IN TYPE THEORY: A PRACTICAL METHOD

DAVID PICHARDIE AND VLAD RUSU

ISSN 1166-8687

Defining and Reasoning About General Recursive Functions in Type Theory: a Practical Method

David Pichardie[*] and Vlad Rusu

Systèmes symboliques — Systèmes communicants
Projets Lande and Vertecs

Publication interne n1766 — November 2005 — 15 pages

Abstract: We propose a practical method for defining and proving properties of general (i.e., not necessarily structural) recursive functions in proof assistants based on type theory. The idea is to define the graph of the intended function as an inductive relation, and to prove that the relation actually represents a function, which is by construction the function that we are trying to define. Then, we generate induction principles for proving other arbitrary properties of the function.

The approach has been experimented in the Coq proof assistant, but should work in like-minded proof asistants as well. It allows for functions with mutual recursive calls, nested recursive calls, and works also for the standard encoding of partial functions using total functions over a dependent type that restricts the original function's domain.

We present simple examples and report on a larger case study (sets of integers represented as ordered lists of intervals) that we have conducted in the context of certified static analyses.

Key-words: non-structural recursive functions, induction principles, Coq proof assistant

(Résumé : tsvp)

[*] Currently member of the Everest project at Inria Sophia Antipolis

Une approche pratique pour la définition et la preuve de propriétés de fonctions recursives générales en théorie des types

Résumé : Nous proposons une approche pratique pour définir et prouver des propriétés de fonctions récursives générales (i.e., non nécessairement structurelles) dans des assistants de preuve basés sur la théorie des types. L'idée principale est de définir le graphe de la fonction en cours de définition comme une relation inductive, et de prouver que cette relation représente une fonction qui est, par construction, celle qu'on essaie de définir. Ensuite, nous définissons des principes d'induction qui permettent de prouver des propriétés quelconques sur la fonction.

Nous avons expérimenté l'approche dans l'assistant de preuve Coq. Cependant, elle devrait fonctionner dans d'autres systèmes basés sur la théorie des types. L'approche permet de traiter des appels mutuellement récursifs et/ou imbriqués, ainsi que des fonctions partielles via le codage standard par des fonctions totales au type dépendant qui restreint le domaine de définition.

Nous présentons des exemples simples ainsi qu'une étude de cas plus complexes (le codage d'ensembles d'entiers par des listes d'intervalles) que nous avons menée dans le contexte du développement d'analyses statiques certifiées.

Mots clés : fonctions récursives non-structurelles, principes d'induction, assistant aux preuves Coq

1 Introduction

Defining and reasoning about some relatively simple recursive functions (such as Quicksort) in proof assistants based on type theory (such as the Coq proof assistant [5]) is surprisingly hard. For a user interested in *using* such tools for the purpose of verification (and not in the subtleties of type theory), this difficulty may rapidly become overwhelming, as too much effort is spent on dealing with difficult concepts not related to her verification problem. Moreover, in other proof assistants based on different logicals frameworks, these problems are considerably easier [11].

The main problem arises from the fact that type theory imposes that all functions terminate; moreover, the type-checking algorithm must be able to check this fact automatically. This restricts the class of functions that are definable in a simple manner to recursive functions with structural recursion over a parameter (for example, in a function with formal parameter $p \in \mathbb{N}$, a recursive call may be performed with the actual parameter $p-1$ but not, e.g., $p \ div \ 2$).

In this paper we propose a practical method for defining non-structural recursive functions in the Coq proof assistant and other like-minded systems. The idea is to define the graph of the intended function as an inductive relation, and to prove that the relation actually represents a function, which is by construction the function that we are trying to define. We also generate induction principles for the function, which help prove other properties that the function satisfies. In particular, the induction principle is stronger that the function's own fixpoint equation, i.e., that which allows to unfold the function's definition in a proof. Synthetising and using induction principles for *structural* recursive functions in Coq has already been proposed in [3].

The method allows for functions with mutual recursive calls, nested recursive calls, and works also for "partial" functions via the standard encoding as total functions over a dependent type restricting membership in the original function's domain.

Last but not least, our approach is relatively simple to understand and use - it uses only the simplest concepts and constructions on well-founded orders - and can be automated, at least for total functions without nested/mutual recursive calls. In the envisaged tool, the user writes her function in a pseudocode notation similar to that of Coq, and provides a well-founded order on the function's argument(s). The tool would then automatically generate the induction principle associated to the function (and, optionally, the function's fixpoint equation). In this way, writing recursive functions in Coq would become almost as easy as in, e.g., PVS [11].

Related Work

The "converging iterations" approach In [2], a recursive function f is defined as the solution of a fixpoint equation involving a functional F. The definition involves proving that the fixpoint equation terminates in a finite number of iterations. The fixpoint equation of f, which allows to replace f by its definition in a proof, is then obtained almost for free. Induction principles are not provided. The approach has been implemented in a prototype tool that automatically generates some of the definitions, proof obligations, and corresponding proofs [1].

The "ad-hoc predicate" approach In [6], the domain of the function f under definition is encoded using an inductive type. Then, f is defined by *structural induction* over the *proof of membership of values in the domain of f (on which f terminates).* This appoach is very elegant, and uses some subtle features of type theory, but has a major drawback: to be applicable in general it requires a specific type theory including the so-called "Dybjer's scheme" [10] for simultaneous recursive/inductive definitions; this scheme is available in few proof assistants. A restricted version for the Calculs of Constructions (disallowing nested recursive calls) has been proposed ([5], Section

15.4) but this version requires to prove so-called inversion theorems in a very specific manner, such that structural recursion can be performed on them for defining f. This pushes the Calculus of Construction to its limits and requires a lot of expertise. On the other hand, an induction principle for f is obtained for free from that of the function's domain.

Combining [6] and [2] In [4], the "converging iterations" and "ad-hoc predicate" approaches are merged into a powerful technique for recursive function definition, allowing for partial functions with nested recursive calls, yet without requiring Dybjer's scheme. The method is therefore implementable in Coq. However, it also adds up all the difficulties of the two combined approaches, which makes it quite hard to grasp and to use for practical verification purposes.

Synthesising Functions from Relations In ongoing independent work, yet unpublished, Barthe and Forest from the Everest project at Inria Sophia Antipolis are developing an approach for sythesising a general recursive function and its induction principle from an arbitrary inductive relation.

Comparison with our method Unlike the *converging iterations method*, our functions are "solutions" of an inductive relation (the function's own graph), not of a fixpoint equation involving iteration of a functional. This saves us complex proofs about terminating iterations. Unlike the *ad-hoc predicate* method and its extension [4], ours does not require advanced knowledge of the subtleties of type theory. The *synthesising functions from relations* approach appears to be quite close to ours. However, they use a version of the *ad-hoc predicate* method, whereas we use well-founded induction. We are currently working together on a general framework that would encompass both our approaches, as well as on implementing these techniques in a prototype tool within Coq.

Summary The rest of the paper is organised as follows. In Section 2 we briefly present the infrastructure that Coq provides for defining non-structural recursive functions, and discuss its merits and its limits. In Section 3 we present our method in detail on a simple example, and in Section 4 we indicate how the method can be used for dealing with nested and mutual recursive calls and with "partial" functions (encoded as total functions over a dependent type restricting their domain). In Section 5 we present a case study: encoding sets of integers using ordered lists of intervals, which we planned to use for certified static analyses [7, 8]. Finally, in Section 6 we conclude and present future work.

2 Defining General Recursive Functions using Coq's library

The Coq system's standard library (in particular, that dealing with well-founded induction and recursion, available at `http://coq.inria.fr/library/Coq.Init.Wf.html`) offers reasonably good support for defining general recursive functions. However, difficulties arise when trying to reason about those functions. Some solutions are proposed in, e.g., Chapter 5 of [1], but they, in the author's own words, exceedingly complex. This is the motivation for the author's subsequent works on the *converging iterations* method ([4], Part 4 of [1], and Section 15.3 of [5]).

A (total, thus terminating) recursive function can be built using a well-founded relation on its domain, which ensures that parameters of all recursive calls within it are *smaller*, according to the order, than the function's own formal parameters. The `well_founded_induction` theorem[1] from

[1]Simplified for the need of the presentation; see [5], Section 15.2 for details. We assume a basic knowledge of type theory. Details specific to the Calculus of Constructions and the Coq syntax are introduced on a by need basis.

the Coq library says precisely this, and can be used to define general recursive functions. Given a domain `A` and a well-founded relation `R` $\subseteq$ `A`$\times$`A`, it allows to define functions `f`: A $\rightarrow$ B:

(∀ `y:A, (`∀ `z: A, z R y` → `B)` → `B)` → `(`∀ `x : A, B).`

That is, to obtain an element in `B` for every element `x` in the domain `A`, it is enough that `f` be defined in each `y` in `A` *as soon as* `f` is defined in every `z` less than `y`. Then, to define `f`, the user provides:

- the domain `A`, a binary relation `R` $\subseteq$ `A`$\times$`A`, and co-domain `B`
- a proof `P` that `R` is well-founded
- the *induction step* `I`, i.e., how to build `f y` from recursive calls `f z` applied to *smaller* `z`.

Then, `(well_founded_induction A R P I)` constitutes the definition of `f: A` → `B`.

This definition can be used, e.g., to evaluate `(f a)` for concrete values `a` in `A`, but it is not directly useable in *proofs*. Indeed, in a proof by induction one usully needs to unfold the recursive definition of `(f x)`; typically, a given proof context implies that the actual recursive call being made is, e.g., `(f y)`, and, by induction, whatever needs to be proved holds for `y` and `(f y)`. And since `(f x)` is not directly defined via its recursive calls, but only indirectly as `f` $\triangleq$ `(well_founded_induction A R P I)`, this most elementary reasoning cannot be used!

There are a number of ways to circumvent this problem. First, using the expressive type system of Coq, one can encode whatever properties one wishes about `f` using dependent types. For example, assume that we want to prove that some post-condition `(Post (x (f x)))` holds, whenever the argument `x` satisfies a pre-condition `(Pre x)`. Then, it is enough we define `f` right from the start to have the dependent type ∀ `x: A, (Pre x)` → `{y : B | Post x y))}`.

In this way, the pre/post condition information about `f` is immediately available. However, this approach assumes that one knows already when defining `f` all the properties that one will ever need about it! This is not realistic, because any nontrivial development typically requires the user to state and prove many intermediate lemmas, *not known in advance*, between the definition of a function and the final theorem about the function. The above approach would then require to *re-define* `f` with each new lemma, and this definition requires to prove the new lemma while the function is defined, in addition to re-proving all lemmas previously proved.

Moreover, if one needs to defined and prove properties about two functions, say, `f` and `g`, we are faced with a vicious circle (the dependent-type definition of `f` may require that of `g` and reciprocally). Hence, this approach is clearly not useable in any nontrivial development.

3 Defining a General Recursive Function via its Graph

However, there is something to be retained from the approach presented in the previous section. Ideally, if one were able to define a function `f` with a dependent type representing its "most general" property, then it would be enough to define the function once and for all with the property, and then all (true) lemmas in a development would be provable from that property.

A natural candidate for "the most general property of a function `f: A` → `B`" is given by the graph of `f`, that is, a relation `fR` satisfying ∀ `(x:A) (y:B), y = (f x)` ↔ `(fR x y)`. And since `fR` is a relation, not a function, it becomes possible to use the powerful mechanisms of `Coq` for defining and reasoning about inductive relations. The method then proceeds as follows.

1. Define an inductive relation `fR`, which is the graph of the intended function `f`.
2. Define an auxiliary function with a dependent type encoding the fact that `f` satisfies its graph, i.e., define `f_rich: A` → `{y : B | fR x y}` as explained in Section 2.

3. Define `f` by removing the dependent type of `f_rich`, thus reducing its type to A → B.
4. Prove that `fR` defines a function in the usual mathematical sense (univoque relation).
5. Finally, from the definition `fR` we obtain the fixpoint equation of `f`, and from the induction principles of `fR` we derive induction principles for `f`, which can be used in other proofs.

The fixpoint equation allows to unfold the definition of `f` in proofs by induction. Better, the induction principle automatically does the induction (splitting the proof into adequate subproofs).

3.1 Example: Powers of Two

The powers of 2 can be defined as $2^n = 2 \cdot 2^{n-1}$ if n is odd and $(2^{n/2})^2$ if n is even. In "Coq",

```
Fixpoint pow2 (n: nat) :  nat  :=
  match  n  with
   |0 =>  1
   |S  q =>  match (even_odd_dec (S q))  with
               |left _ => (pow2 (div2 (S  q)))*(pow2 (div2 (S q)))
               |right _ => n * (pow2 q)
             end
  end
```

Of course, this definition is not accepted by Coq because the first recursive calls are not performed on structurally smaller arguments. Here, `even_odd_dec` is a Coq *decision*, returning either `left` and a proof of the fact that its argument is even, or `right` and a proof of the fact that its argument is odd. Proofs are irrelevant here and are replaced by the `_` placeholder.

Step 1: Defining the Draph of the pow2 Function The first step in the method defined at the beginning of the section consists in defining the graph of `pow2` as an inductive relation, which completely "mimics" the intended definition of `pow2` with its three cases:

```
Inductive pow2_rel : nat -> nat -> Prop :=
|pow2_0 : pow2_rel 0 1
|pow2_S_even : forall n q  pow2_q, n = S q -> (even_odd_dec (S q)) = left _ ->
               pow2_rel (div2 (S q))  pow2_q -> (pow2_rel (S q) (pow2_q*pow2_q))
|pow2_S_odd : forall n q pow2_q, n = S q ->  (even_odd_dec (S q)) = right _ ->
                          pow2_rel q pow2_q -> pow2_rel (S q)  (2*pow2_q).
```

The `pow2_0` case (or *constructor*) deals with the base case, whereas the `pow2_S_even` and `pow2_S_odd` constructors deal with the cases where the argument is even, respectively odd.

Step 2: Auxiliary Function with Rich Dependent Type The type for our intended function is a dependent type specifying that the function satisfies to its graph relation.

```
Definition pow2_type (n:nat) :=  {pow2_n: nat | pow2_rel n pow2_n}.
```

We also define an auxiliary function `pow2_wf`, which will serve as *induction step* for the Coq library's `well_founded_induction` theorem (already discussed in Section 2). `pow2_wf` says that if one knows how to define `pow2` for smaller values, then one knows how to define it for the current value A well founded order is assumed, here, it is the usual < on natural numbers.

The definition of pow2_wf is, of course, a constructive one and requires a proof of existence[2].

```
Definition pow2_wf  : forall n,  (forall m, m<n -> pow2_type m) -> pow2_type n.
...
Defined.
```

The proof consists in considering three cases, which correspond to those in the pseudo-fixpoint definition of pow2 and to the same three constructors in the definition of the pow2_rel graph.

Then, by standard use of the well_founded_induction theorem, as explained in in Section 2,

```
Definition pow2_rich:=(well_founded_induction lt_wf (fun x=>pow2_type x) pow2_wf).
```

Here, lt_wf is a proof from Coq's libraries that the < order on natural numbers is well founded.

Step 3: Eliminating the Dependent Type To define pow2 we eliminate the dependent type information of pow2_rich in a simple command, thus making it a function from nat to nat. The Caml code generated by Coq's code generation tool, indicates that we are doing well:

```
Definition pow2 (n:nat) :  nat := let (f,_) := pow2_rich n in f.

 (** val pow2  : nat -> nat **)
let rec pow2 = function
  | O -> S O
  |  S n -> (match even_odd_dec (S n)  with
             | Left -> let x0 = pow2  (div2 (S n)) in mult x0 x0
             | Right -> mult (S (S O)) (pow2 n))
```

Step 4: Proving that pow2_rel is Functional Next, we show that pow2_rel actually defines some (for now, partial) function in the usual mathematical sense, that is, to each value in the domain it associates at most one value in the co-domain. For this, we shall require in general that the cases (constructors) defining the graph relation be mutually exclusive. This is reasonable, since the constructors match the cases in the function's pseudocode definition, which have to be mutually exclusive if we are going to define a function at all.

```
Lemma pow2_functional: forall x z1 z2, pow2_rel x z1 -> pow2_rel x z2 -> z1 = z2.
...
Qed.
```

Step 5: Fixpoint Equation and Induction Principle The crucial fact that the pow2_rel relation is functional allows us now to obtain the fixpoint equation of pow2. Intuitively, pow2_rel and pow2 represent the same function, and pow2_rel is very close from the fixpoint equation of pow2. As a matter of fact, for a methodological point of view it is better to separate the contexts of the recursive calls in individual lemmas, also useful for rewriting.

[2]The half-page proof is omitted for simplicity. All complete examples in the paper can be downloaded from http://www.irisa.fr/vertecs/Equipe/Rusu/rap1766/. We would like to point out that proofs are completely generic - they have been reproduced many times for many examples and always follow the same pattern.

```
(*case: argument = 0*)
Lemma pow2_0 : pow2 0 = 1.
...
Qed.
Hint  Resolve pow2_0: base. (*for automatic rewriting*)

(*case: even, non-zero argument*)
Lemma pow2_S_even : forall q,  even_odd_dec (S q) = left _->
 pow2 (S q) = (pow2 (div2 (S q) ))* (pow2 (div2 (S q))).
...
Qed.
Hint Resolve pow2_S_even: base.

(*case:  odd argument*)
Lemma pow2_S_odd : forall q, even_odd_dec (S q) = right _ ->
 pow2 (S q) =  2* (pow2 ( q)).
...
Qed.
Hint Resolve pow2_S_odd: base.

(*Fixpoint equation*)
Lemma pow2_fixpoint : forall n, pow2 n  =
match n with
|O => 1
|S q =>
   match (even_odd_dec (S q)) with
        |left _ =>  (pow2 (div2 (S q))) * (pow2 (div2 (S q)))
        |right _ => 2 * (pow2 q)
   end
end.
intros.
destruct n ; auto with base.
case (even_odd_dec (S n)) ; auto with base.
Qed.
```

We have obtained the fixpoint equation, which allows to unfold the definition of a function in a proof, a crucial feature in proofs by induction. But we can do even more: we can obtain an *induction principle* that does the induction for us, in any circumstance where a proof on `pow2` is required, by splitting the proof into adequate subproofs. For this, we slightly modify the induction principle automatically generated by Coq for the inductive relation `pow2_rel`, which is given by:

```
pow2_rel_ind : forall P : nat -> nat -> Prop,
       P 0 1 ->
       (forall n q pow2_q : nat, n = S q -> even_odd_dec (S q) = left _ ->
        pow2_rel (div2 (S q)) pow2_q ->
        P (div2 (S q)) pow2_q -> P (S q) (pow2_q * pow2_q))
        ->
```

```
         (forall n q pow2_q : nat, n = S q -> even_odd_dec (S q) = right _ ->
         pow2_rel q pow2_q -> P q pow2_q -> P (S q) (2 * pow2_q))
       -> forall n n0 : nat, pow2_rel n n0 -> P n n0
```

Intuitively, the `pow2_rel_ind` induction principle says that an arbitrary predicate `P` holds for numbers `n` and `n0` satisfying `pow2_rel n n0` whenever `P` satisfies some constraints that correspond to the recursive definition of `pow2_rel`. The induction principle for the `pow2` *function* is then

```
Lemma pow2_ind : forall P : nat ->nat-> Prop,
       P 0 (pow2 0) ->
       (forall n q,  n = S q  -> even_odd_dec (S q) = left _ ->
        P (div2 (S q)) (pow2 (div2 (S q))) ->
        P (S q) ((pow2 (div2 (S q)))* (pow2 (div2 (S q)))))
       ->
       (forall n q, n = S q -> even_odd_dec (S q) = right _ ->
       P q (pow2 q) -> P (S q) (2*pow2 q))
       -> forall n, P n (pow2 n).
...
Qed.
```

It is very similar to that of the `pow2_rel` relation (and uses `pow2_rel_ind` in its proof!). The main difference is that `pow2_ind` now directly refers to the values of the recursive calls of `pow2`.

Automation The approach can be automated, at least for total functions without nested or mutual recursive calls. In the envisaged tool, the user writes her function in pseudocode, and provides a well-founded order on (one of) the function's arguments. The tool would then automatically generate the induction principle associated to the function (and, optionally, the function's fixpoint equation), which can then be used to prove other propperties of the function.

4 Nested, Mutual Recursive Calls, and Partiality

The method also works for more complex recursion patterns such as nested and/or mutual recursive calls and can deal with partial functions, albeit less automatically.

Nested Recursive Calls Consider the following pseudocode wih nested calls

```
Fixpoint f (n:nat) : nat :=
match (le_gt_dec n 0) with
|left _ => 0
|right _ => 3* (f (2*(f (n-1)))
end
```

Here, `le_gt_dec n 0` is a decision which either returns `left` and a (here, irrelevant) proof of `n <= 0`, or `right` and a (likewise, irrelevant) proof of `n > 0`. The corresponding graph relation is:

```
Inductive fR : nat-> nat-> Prop :=
|fR_left : forall n H, le_gt_dec n 0 =  left _ H -> fR n 0
|fR_right : forall n m p H, le_gt_dec n 0 =
right _ H -> fR (n-1) m -> fR (2*m) p  -> fR n (3*p).
```

Note how the nested recursive called has disappeared from the relation. Now, if we just apply our method and try to define a depedent-typed function

```
Definition f_rich_type (n:nat) := {p : nat | fR n p}.
Definition f_rich_ : forall n,   f_rich_type n.
```

we are immediately faced with a difficulty: we have to prove that the nested call returns something smaller, so that we can legally apply `f` to it, i.e., we need that `2*(f(n-1)) < n` holds.

Of course, we cannot prove this since `f` is not yet defined at that point. However, we can prove the equivalent of that constraint in the function's graph, which we have just defined:

```
Lemma nested_proof_obligation : forall n m, fR n m -> 2*m < n+1.
```

and use this lemma in the subsequent application of our method, which proceeds smoothly from there on. We eventually obtain the expected induction principle and expected Caml code

```
(** val f : nat -> nat **)
let rec f x =
  match le_gt_dec x O with
    | Left -> O
    | Right -> mult (S (S (S O))) (f (mult (S (S O)) (f (minus x (S O)))))
```

Hence, for nested function calls the expected level of automation is more limited - it merely consists in generating the function's graph and the `nested_proof_obligation` - but still useful.

Mutual Recursive Calls For functions with mutual recursive calls, we can use the mutually inductive relations feature of Coq to define their graph. Consider the pseudocode wih mutual recursion:

```
Fixpoint f(n: Z) : Z :=
match (le_gt_dec n 0) with
|left _ => 0
|right _ => g (n-1)
end

Fixpoint g(n: nat) : nat :=
match (le_gt_dec n 0) with
|left _ => 0
|right _ => f (n-1)
end.
```

The corresponding graph is then given by the corresponding mutually inductive relations

```
Inductive fR : nat -> nat -> Prop :=
|f_0 : forall n H, le_gt_dec n 0= left _ H -> fR n 0
|f_gt : forall n H x, le_gt_dec n 0= right  _ H  -> gR (n-1) x -> fR n x
with
gR : nat -> nat -> Prop :=
|g_0 : forall n H, le_gt_dec n 0= left _ H -> gR n 0
|g_gt : forall n H x, le_gt_dec n 0= right _ H -> fR (n-1) x -> gR n x.
```

It is then natural to define both functions `f` and `g` simultaneously (in order to avoid a vicious circle in which the dependent-typed definition of `f` requires that of `g`, and reciprocally):

```
Definition fgR : nat -> nat*nat -> Prop :=  fun n p=>fR n (fst p)/\gR n (snd p).
Definition fgR_type (n : nat) := {m:nat*nat| fgR n m}.
Definition fg_rich : forall x,  fgR_type x.
```

The method proceeds from here on. The generated Caml syntax does not exactly match that of Coq, as we cannot expect the generator to guess that we are after mutually recursive functions:

```
(** val f : nat -> nat **)
let f n =
  fst
    (let rec acc_iter x =
       match le_gt_dec x O with
         | Left -> Pair (O, O)
         | Right ->
             let x0 = acc_iter (minus x (S O)) in Pair ((snd x0), (fst x0))
     in acc_iter n)
```

However, the semantics of the Caml code is that expected. As for induction principles, we can generated those corresponding to the (mutually inductive) graphs as "prescribed" by our method.

In some cases, however, these induction principles are very good at proving anything - we might need to draw inspiration from stronger induction schemes of the mutually inductive relations (as generated by the `Scheme` command of Coq). In even more rare cases, such as those described in [5], Section 14.3.3, even the `Scheme`-generated induction principles are too weak, and we may need to define specialised induction principles and to prove them manually.

Partial Functions Finally, we briefly discuss the application of our method to partial functions. A "partial" function `f :  A` $\rightarrow$ `B`, whose actual domain is that satisfying a predicate `P : A` $\rightarrow$ `Prop`, is simulated by a dependently-typed total function with two arguments `f': forall (x:A), (P x)` $\rightarrow$ `B`, where the second argument is a proof of `(P x)` (that the function is "defined" for the first argument). If `f` contains recursive calls the situation is more complicated than before, because we have to provide proofs that the arguments to recursive calls is also in the subdomain. However, this difficulty is somewhat orthogonal to our method - the same difficulty occurs for structurally recursive functions as well. Apart from this problem, the method works reasonably well in its spirit, although automation might become more difficult.

We illustrate partial functions with a version of the integer logarithm function, which computes the logarithm in base 2 of the power of 2 closest (from below) to its input. We choose to make an extensive use of dependent types in order to test the limits of our method. We first define a Coq decision, which for all strictly positive `n` returns either `n-2` or a proof that `n=1`:

```
Definition strictpos_dec : forall n, n > 0-> {p : nat | n = p+2}  +{n = 1}.
...
Defined.
```

The pseudocode or our function follows. The cases are identified by comments in the code.

```
Fixpoint log (n:nat) (h : n > 0) : nat :=
match strictpos_dec n h with
|inright _  => 0 (*case n=1*)
|inleft H => match H with (exist x _) =>
                     (*case n > 1; here; x = n-2*)
                      1+(log (1+div2 x)
                              (le_plus_l 1 (div2 x))) (*<-proof*)
                   end
end
```

The `log` function is "partial" in that it is only defined for `n > 0`. The second argument `h : n > 0` is a proof encoding this partiality. Then, a proof needs also to be passed to the recursive call. It is a proof that `1+ div2 x > 0` where `div2` is the integer division by 2.

We need a few more steps before we can apply our approach. First, the approach is based on the `well_founded_induction` theorem from Coq's library, which only allows to build functions with one argument. Hence, we group the two arguments of the `log` function into one of type:

```
Inductive strictpos : Set := strictpos_ : forall (n:nat) (h : n > 0), strictpos.
```

Then, we define the graph of a function denoted `log'`, which is a version of `log` with one argument of type `strictpos`:

```
Inductive log'_rel : strictpos -> nat -> Prop :=
log'_rel_1 : forall n h  H,
strictpos_dec n h= inright _ H -> log'_rel (strictpos_ n h)  0
| log'_rel_2 :
forall n p q h h' H, strictpos_dec n h = inleft _ (exist _ p H) ->
 log'_rel (strictpos_ (1+ div2 p) (le_plus_l 1 (div2 p))) q ->
log'_rel  (strictpos_ n h') (q+1).
```

From here on, we apply our method for defining the function `log'` with the following type

```
Definition log'_rich_type (n: strictpos) :=  {m : nat | log'_rel n m}.
```

Finally, once the `log'` function is defined, we define `log` by splitting the composed argument of type `strictpos` into its components:

```
Definition log : forall n, n >0 -> nat := fun n h => log' (strictpos_ n h).
```

The resulting induction principle is given below. Together with that of the `pow2` function, given in Section 3, we have used to prove a simple property:

```
Theorem log_pow2 : forall x h, log (pow2 x) h  = x.

(*induction principle for the log function*)
Lemma log_ind : forall P : forall n, n>0 -> nat-> Prop,
       (forall n h H, strictpos_dec n h = inright _ H -> P  n h 0) ->
       (forall n p q h h' H, strictpos_dec n h = inleft _ (exist _ p H) ->
        P  (1 + div2 p) (le_plus_l 1 (div2 p)) q ->
        P n h' (q + 1)) ->
       forall n h,  P n h (log n h).
```

The complete development can be found at `http://www.irisa.fr/vertecs/Equipe/Rusu/rap1766/`.

5 Case Study: Interval Sets

We have built a library for finite sets of natural numbers encoded using ordered lists of intervals. The library was planned to be part of the development of a framework for certified static analyses [7, 8]. It includes functions for union, intersection, inclusion, and membership tests.

Among all those operations, the union operation is the most involved. We describe in some detail its definiton and some of the proofs that we have developed around it using our method.

First, interval sets are defined using the following two inductive relations (`::` is the Coq notation for cons on lists, and `fst`, `snd` return the first, resp. the second element of a pair):

```
Inductive  is_interval (i:Z*Z) : Prop :=
  |is_interval_cons :  fst i <= snd i -> is_interval i.

Inductive is_interval_set : list (Z*Z) -> Prop :=
    |Nil_set : is_interval_set nil
    |Single_set : forall i, is_interval i -> is_interval_set (i::nil)
    |Cons_set : forall i j l,
      is_interval i -> is_interval j ->  (1+ snd i) < fst j ->
      is_interval_set (j::l) -> is_interval_set (i::j::l).
```

In particular, note that successive intervals in an interval set may not intersect each other, touch each other, or even be adjacent: $1 + snd\ i < fst\ j$, otherwise, they would form a single interval. We use the following inductive definition, which compares the relative positions of two intervals:

```
Inductive compare_intervals  (a1 a2 : (Z*Z)) : Set :=
|snd_far_after : snd a1 +1 < fst a2 -> compare_intervals a1 a2
|snd_close_after :  fst a1 < fst a2 -> snd a1 < snd a2 -> snd a1 +1 >=
fst a2 -> compare_intervals a1 a2
|snd_includes :  fst a1 >= fst a2 -> snd a1 <= snd a2 ->
(~fst a1 = fst a2 \/ ~(snd a1 = snd a2) ) -> compare_intervals a1 a2
|snd_equal_fst : fst a1 = fst a2 -> snd a1 = snd a2 ->
compare_intervals a1 a2
|fst_includes : fst a1 <=fst a2  -> snd a1 >= snd a2 ->
(~fst a1 = fst a2 \/ ~(snd a1 = snd a2) )  -> compare_intervals a1 a2
|fst_close_after :   fst a2 < fst a1 -> snd a2 < snd a1 -> snd a2 +1 >=
fst a1 -> compare_intervals a1 a2
|fst_far_after: snd a2 +1 < fst a1 -> compare_intervals a1 a2.
```

We are now ready to give the pseudocode for the union operation. An implementation linear in the size of its arguments cannot be structural, as it cannot predict in advance how the lists begin (more specifically, how many initial intervals of one list are included in the second one):

```
Fixpoint union (li1 li2: list (Z*Z)) : list (Z*Z) :=
match li1, li2 with
|nil, l' => l'
|l'', nil => l''
|a1::l1, a2::l2 =>
  match compare_intervals_dec a1 a2 with
   |snd_far_after  _  => a1::(union l1 (a2::l2))
```

```
    |snd_close_after _ _ _=> union l1 ((fst a1,snd a2)::l2) (*not structural*)
    |snd_includes _ _ _  => union l1 (a2::l2)
    |snd_equal_fst  _ _ => a1::(union l1 l2)
    |fst_includes _ _ _  =>  union (a1::l1) l2
    |fst_close_after _ _ _=> union ((fst a2,snd a1)::l1) l2 (*not structural*)
    |fst_far_after _ => a2::(union (a1::l1) l2)
       end
end.
```

One property that we need to prove about the `union` function is that, although its arguments are lists of pairs of integers, by applying it to two `interval_set`s one obtains an `interval_set`.

This turned out to be a nontrivial exercise. We have solved it as follows: we have defined a weak interval set structure, which is a list of intervals in which consecutive intervals may overlap, and a *weak union* operation, which is closed on weak interval sets (but not on proper interval sets). Then, a grouping operation transaforms a weak interval set into a proper interval set. Weak union and grouping are also non-strucuturally recursive, but somewhat simpler that the proper union. We then show that the proper union is equal to the composition of the weak union and the grouping operations, and that the result of the composition is an interval set.

For this, we make a heavy use of the induction principles for proper union, union, and grouping functions. Several dozens of lemmas were proved using those principles; without them, we could not have completed the development. In fact, we have come across the presented method after trying (and failing) to use some of the existing approaches in a practical setting.

6 Conclusion and Future Work

We present a practical method for defining and proving properties of general recursive functions in proof assistants such as Coq and other proof assistant based on type theory. The graph of the intended function is defined as an inductive relation, and the function is defined using a dependent type, specifying that the function "satisfies" its graph. This property allows us to obtain the function's fixpoint equation, and from the graph's automatically generated induction principle, we generate induction principles specific to the function, which allows to prove other properties of the function. We demonstrate the approach for functions with mutual recursive calls, nested recursive calls, and for partial functions seen as total functions over a dependent type restricting its domain.

As for future work, we are building a prototype tool implementing our method for the relatively simple case of total functions without nested/mutual recursive calls. In the tool, the user writes her function in a pseudocode notation similar to that of Coq, and provides a well-founded order on the function's argument(s). The tool then automatically generates the induction principle associated to the function (and, optionally, the function's fixpoint equation). This considerably simplifies working with general recursive functions in Coq, a feature most desirable as demonstrated by our case study reported in Section 5.

References

[1] Antonia Balaa. *Fonctions récursives générales dans le calcul des constructions.* PhD thesis, University of Nice, 2002.

[2] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2000.

[3] G. Barthe and P. Courtieu. Efficient reasoning about executable specifications in coq. In Carreño et al. [9], pages 31–46.

[4] Yves Bertot, Venanzio Capretta, and Kuntal Das Barman. Type-theoretic functional semantics. In Carreño et al. [9], pages 83–98.

[5] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions.* springer, 2004.

[6] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708, February 2005. Cambridge University Press.

[7] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, September 2005.

[8] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2005.

[9] Victor Carreño, César Muñoz, and Sofiène Tashar, editors. *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*. Springer, 2002.

[10] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

[11] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer.