MASTER

Formally proving the correctness of functional programs
a comparison of different methods in the proof assistant CoQ

Saidi, S.

Award date:
2010

Formally Proving the Correctness of Functional Programs

A Comparison of Different Methods in the Proof Assistant Coq

Sepideh Saidi

Eindhoven, August 2010

Formally Proving the Correctness of Functional Programs

A Comparison of Different Methods in the Proof Assistant Coq

Author:
Sepideh Saidi (0559938)

Supervisor:
prof.dr.J.H.(Herman) Geuvers

Other members of the examination committee:
dr.ir. C. (Kees) Hemerik
dr.ir. M.G.J. (Michael) Franssen

August 27, 2010

Abstract

The objective of this thesis is to investigate methods for proving functional programs correct in the proof assistant Coq. Coq and its underlying type theory pCic, the predicative calculus of inductive constructions, combines a small core dependently typed functional programming language with a proof assistant (also using dependent types) for specifying properties of these programs and proving them.

During the course of this project various methods for defining functional programs in Coq and the mechanisms for proving properties of these programs are investigated and compared. The underlying theory of Coq does not allow general recursion (but only structural recursion), all programs should be terminating and total, so there is no direct translation of every functional program into Coq. A bonus of Coq is that the functional programs are provably correct.

There are a number of methods which are capable of dealing with terminating functions which are not structurally recursive. These methods are compared to each other by means of four major criteria. Among others, the closeness of the Coq programs to the "original" functional programs is studied. We also study how the correctness proofs are given and how the programming and proving are integrated in the various methods.

Preface

Type theory is a fascinating and broad area of fundamental theory. During the last six months, I was pleased to have the opportunity to discover more about the powerful and abstract concepts of this theory. In particular, I acquired more insight into what (a subset of) this theory has to offer with regard to the proof assistant Coq. The focus point was initially to investigate the relatively new framework called Russell in this respect. Lots of initial investigations are therefore devoted to this framework as the reader might notice from the amount of presented details with regard to this framework. However, since most of the available documentation about this framework was written in French, which is not an accessible language for me, the focus has changed. The investigation has been extended with other methods, based on which a comparison has been carried out with regard to these methods. I am pleased to announce that this focus change, contemplating the whole process, was satisfactory. This satisfaction has two reasons. First of all, the involvement of the other methods has helped me to gain more perspective of the possibilities. Secondly, only after studying those other possibilities, I started to thoroughly appreciate the ease and comfort which the Russell framework offers.

Herman Geuvers was my indispensable supervisor. His knowledge, incredible insight and valuable comments were essential to accomplish this task. Therefore, I want to express my gratitude towards him for guiding me through this quest and helping me to put everything in perspective consecutively. Furthermore, I would like to thank Kees Hemerik for his valuable comments on the last draft of this thesis.

Eindhoven, August 2010

Contents

Chapter 1

Introduction

The number of critical software applications is increasing rapidly in various application domains. The correctness of these software applications is not evident specially when the systems tend to get more complex. Considering the critical role that software applications fulfill in various apparatus nowadays, it is of crucial importance that these applications are not due to any failure what so ever; especially in order to overcome the catastrophic disasters which these applications may induce[1].

Due to the system's complexity, it is mandatory to automatize a large part of (preferably the whole of) the verification process using computers. In order to achieve a reliable level of trust and guarantee regarding the behavior of these systems, formal verification techniques are required. Testing methods are by far the most accustomed methods when it comes to the validation of software applications. Although these methods are capable of detecting common bugs of an application, they are not suitable for ensuring the total absence of bugs. Hence to assure the absence of bugs, the testing frameworks aren't competent enough and thus substantiated formal verification techniques are required.

Among the formal automated verification techniques, Model Checking technique can be mentioned; a technique which is used to ascertain that a given model of a system meets certain safety requirements. For instance the absence of deadlock, which causes the system to crash, can be ensured using model checking techniques. More specifically, the model of a system is formulated in a particular formal specification language and the requirements are specified as properties in some temporal logic for instance in CTL [EMCS86] or in LTL [Pnu77]. Model Checking is then performed by exhaustive exploration of the entire state space, generated by means of the system's model, to validate the desired property. Unfortunately, model checking techniques suffer from a combinatorial blow up of the state-space, known as the space explosion problem. Among others, abstraction , symbolic model checking and partial order reduction are approaches to combat this problem.

Another way of verifying a system formally, is by means of so called Interactive Theorem Provers (ITP). These provers seek verification of the correctness of programs using mathematically based reasoning and proof techniques. Some modern ITP's are COQ [Coq, The09], ISABELLE/HOL [TNW02], PVS [SOS92], ACL2 [MKM00] and TWELF [Sch09]. In the course of ITP's the following steps are undertaken in order to proof the correctness of the programs:

1. The user gives the definition of a program/system
2. The user describes the properties to which the program/system should adhere to

[1] A list of catastrophic disasters due to the software failure
http://www.sereferences.com/software-failure-list.php

3. Interactively with the ITP, the user provides correctness proof of the program/system

In the following section, we're going to discuss the problems which one encounters when using the ITP's.

1.1 Problem Description

For the fully detailed problem description more technical context is required, which is explained in chapter [2]. Therefore, a less formal problem description is provided in this section. This problem description is reformulated in the next chapter where the required context is available.

From a historic point of view, constructive type theories have been used vastly for theorem proving. Proof assistants based on constructive type theory such as COQ and LEGO [Pol94], provide a combination of a programming language and a powerful proof language on the basis of which one can prove, specify and execute the programs. One aspect of constructive type theory from which a proof language owes its power, is the fact that it combines deduction with computation. This aspect makes constructive type theory also a convenient candidate for programming i.e. the strength of the type system allows properties of programs to be defined, and the computational properties provide an operational semantics for the programs.
Despite of its power, constructive type theory brings with it a limitation namely: only structurally recursive programs can be defined due to the fact that termination must be guaranteed for those recursive functions.

Generally speaking, a proof assistant based on type theory works as follows: in order to prove the correctness of a given program, the user is provided with a bunch of so called *tactics*. These *tactics* assist the user in constructing a proof term. Internally, these *tactics* are sent to a proof engine which constructs this proof term. The proof term is sent to the type checker and the new goals are generated for the user. The type checker decides upon the correctness the proof term. (see figure [1.1]).

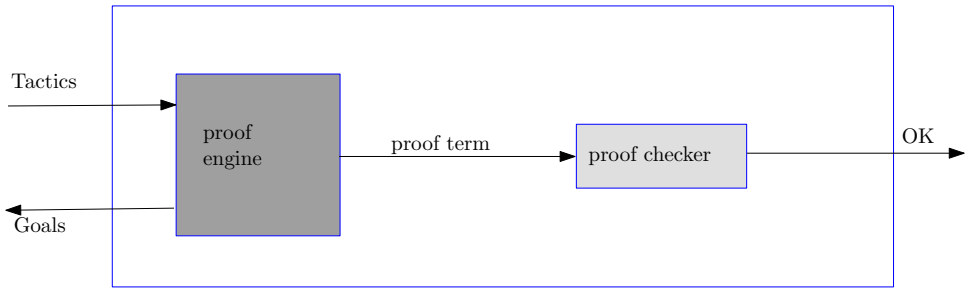

Figure 1.1: Proof development in type theory based proof assistants

In the course of this document, our focus is on proving the correctness of functional programs. As ITP we choose the proof assistant COQ[The09].

Motivation: why Coq? The first reason for this choice is the underlying foundation of the COQ proof assistant namely higher-order logic with inductive data types which provides a powerful framework for functional programming. The Proof assistant ACL2 is based on first order logic and hence less interesting. Another remarkable aspect about COQ, is the presence of the invaluable notion of dependent types. Dependent types are types that depend on the values of other types

by means of which one can express correctness properties in terms of the types. ACL2 and HOL lack the notion of dependent types and PVS and TWELF each support a different strict subset of COQ's dependently typed language. Secondly, it is desirable that a proof assistant has a reliable (relatively small) proof-checking kernel, the correctness of which can be checked by a user. This desirable property, known as "Bruijn criterion" [BG01], is absent in proof assistants like PVS and Acl2 due to the existence of fancy decision procedures that produce no evidence trace justifying their results.

In functional programming languages, pattern matching on a term's structure provides a concise notation for defining functions. Each branch of a pattern-matching construct, has to address one of the constructors of the Inductive type. Consider the simple pattern matching construct in COQ on the term t of an inductive type with constructors c_1, c_2, $c_3, ..., c_n$ and $\vec{x_i}$ as the linear factor of (distinct) variables:

$$
\begin{array}{l}
\mathit{match}\ t\ \mathit{with} \\
\quad |\ (c_1\ \vec{x}_1) \quad \Rightarrow \quad e_1 \\
\quad |\ (c_2\ \vec{x}_2) \quad \Rightarrow \quad e_2 \\
\quad \vdots \\
\quad |\ (c_n\ \vec{x}_n) \quad \Rightarrow \quad e_n \\
\mathit{end}.
\end{array}
$$

For recursive function definitions the above construction can be refined as:

$$
\begin{array}{l}
\mathit{Fixpoint}\ f\ (t : \mathit{Ind\_type}) : \mathit{Type} := \\
\mathit{match}\ t\ \mathit{with} \\
\quad |\ (c_1\ \vec{x}_1) \quad \Rightarrow \quad e_1 \\
\quad |\ (c_2\ \vec{x}_2) \quad \Rightarrow \quad e_2 \\
\quad \vdots \\
\quad |\ (c_n\ \vec{x}_n) \quad \Rightarrow \quad e_n \\
\mathit{end}.
\end{array}
$$

Where the expressions (e_i) can have the form $(f(\vec{x_i}))$. Generally speaking, in order to ensure the termination of a recursive function definition, one needs to assure that the recursive call is invoked on a smaller argument. Strictly speaking, this is achieved by peeling off the constructor once as is the case in above *Fixpoint* definition: $((c_i\,\vec{x_i}) \Rightarrow f(\vec{x_i}))$.
The restriction on the number of times that the constructor is peeled off is however unnecessary. In other words, the constructor can be peeled off several times as long as the argument to the recursive call is structurally smaller.

In the context of proof assistants based on type theory one can distinguish two types of recursive definitions:

- Structural recursion which, as the name implies, means that all recursive calls need to be made on structurally smaller arguments. Function $div2$ is an instance of a structurally recursive function. Note that the argument to the recursive call p is structurally smaller than the initial argument $S(S\ p)$. Hereby the constructor S on natural numbers is peeled off twice and hence the decreasing character of the argument to the recursion call is ensured.

```
Fixpoint div2 (n : nat) : nat :=
match n with
  | S (S p)  ⇒  S (div2 p)
  | _  ⇒  0
end.
```

- General recursion where a function is defined by using the same function in the definition body (function invokes itself recursively). In cases of general recursion, the decreasing character of the argument to the recursive call can't be inferred by structurally. A function defined by general recursion need not to be terminating for all input values. Function $log2$ is an instance of a general recursive function. The argument to the recursive call $S(div2\ p)$ is not structurally smaller than the initial argument $S(S\ p)$. Hence in order to prove the termination of this function one needs to prove that: $\forall\ p\ :\ nat.\ S(div2\ p)\ <\ S(S\ p)$

```
Fixpoint log2 (n : nat) : nat :=
match n with
  | S 0  ⇒  0
  | S (S p)  ⇒ S (log2 (S (div2 p)))
end.
```

The main issues: In order to ensure the decidability of type checking (and hence of proof checking), proof assistants based on type theory require that the functions are provably terminating, total and deterministic. Fulfilling these requirements is not always evident and it can give rise to some difficulties for certain functions. In the Coq proof assistant for instance, totality and determinacy are enforced by requiring definitions by pattern matching to be exhaustive and non-ambiguous. When it comes to recursion in Coq, termination is syntactically ensured by imposing that all recursive calls need to be made on structurally smaller arguments (structural recursion). However, many interesting terminating functions are not structurally recursive and/or are partial functions (e.g. div2, log2, pow2, head, etc.). For general recursive functions, termination is an issue because the guard predicate isn't able to infer the decreasing character of the recursion any more (e.g. log2, pow2). For partial functions, type checking becomes an issue because one needs to provide exhaustive pattern matching cases which isn't possible for partial functions (e.g. head, log2). Nevertheless, Coq also allows the use of non-structural functions if they are provided with a prove of well-foundedness [RMDA07] or a measure function which guarantees the decreasing of the arguments at each recursive call. The latter provides the user with the possibility to define more complex functions without following the tedious encoding using structural recursion. Algorithms for computing Gröbner bases, as described by Théry [Thé98] and Coquand et al. [CP98] are instances hereof.

Hence the main issues one has to deal with when formalizing a functional program as a Coq function are:

1. partiality
2. non-structural (general) recursion
3. termination

Definition 1.1.1 (partiality)**.** *In mathematics, a partial function from X to Y is a function $f : X' \to Y$, where X' is a subset of X. It generalizes the concept of a function by not requiring f to map every element of X to an element of Y (only some subset $X' \subseteq X$). If $X' = X$, then*

f is called a total function and is equivalent to a function. Partial functions are often used when the exact domain, X' , is not known (e.g. many functions in computability theory).

Definition 1.1.2 (non-structural (general) recursion)**.** *A general recursive function f whose left and right sides are composed from the same function symbols (for example, f, g, h, etc.). In other words general recursive functions are functions which are defined by using the same function in the definition body (function invokes itself recursively). In cases of general recursion, the decreasing character of the argument to the recursive call can't be inferred by the guard predicate. Hence, a function defined by general recursion need not to be terminating for all input values.*

Definition 1.1.3 (termination)**.** *Given the starting state s, the program is called terminating if it starts its execution initially in s and reaches the final state s', where the execution stops.*

In cases where the function definition is not structurally recursive, its termination isn't guaranteed and needs to be proven explicitly by means of a measure function or a well-founded ordering on recursion arguments.

Objectives: There are a number of methods to tackle the aforementioned problems. Hereby two main categories can be distinguished:

1. category 1: fully specified function : a function definition is given along with its complete specification.
2. category 2: total separation of function specification and properties (separation of concerns)

Throughout this document, a comparison is made between methods belonging to one of the two categories. Based on this comparison pros and cons of the various methods are discussed thoroughly. For clarification, a number of examples is provided [2.5].

Methods under consideration: The methods under consideration are:

1. The Ad-hoc Predicate method due to Ana Bove and Venanzio Capretta [BC05]
2. The Converging Iterations Method due to Antonia Balaa and Yves Bertot [BB00]
3. The FUNCTION Framework due to Gilles Barthe et al. [GBR06]
4. The RUSSELL Framework due to Matthieu Sozeau [Soz06]

The criteria: There is a number of criteria on the base of which these methods are compared to each other:

1. function definition
2. function properties
3. program extraction
4. function executability

Now we are going to give a definition for each criterion, and accordingly we formulate the raised questions with respect to that criterion.

Definition 1.1.4 (function definition)**.** *By function definition we mean the actual implementation of the given function(code).*

RQ1: How difficult is it to write the definition of the function?
RQ2: How close is the function definition to the original functional program expressed in a practical functional programming language , where there is no restriction on recursive calls?

Definition 1.1.5 (function properties)**.** *Each function should satisfy certain properties: the result of its execution can be formulated by means of formal properties to which it should adhere. For imperative programs, these properties are given via pre- and post-conditions. For functional programs, these properties are given by logical formula's (which are expressed as types in the proof assistant* Coq*).*

RQ3: How difficult is it to formulate and prove the properties of the functions?

Definition 1.1.6 (program extraction)**.** *The automatic generation of functional code from* Coq *proofs is called program extraction. The main motivation behind this mechanism is to produce certified programs i.e. programs for which the correctness is already proved by means of* Coq*'s powerful proof apparatus, hence the resulted extracted code is a certified functional program; as one may have written down in a ML-like language [PM89, MW93]. The current supported languages for the extraction mechanism are Objective Caml(*Ocaml*) [Lea04],* Haskell*[ea92] and* Scheme *[Aea98]. In chapter [2], program extraction and the theory behind it are explained in more detail.*

RQ4: Is the extracted code in accordance with our expectation?

Definition 1.1.7 (function executability)**.** *Running the code for actual input is called function execution. Having the certified code, one has to put it in use in order to get the results.*

RQ5: Can one execute the code in Coq *sufficiently well in terms efficiency?*

1.1.1 Outline

The rest of this thesis is organized as follows. Chapter [2] is devoted to background information about the Coq proof assistant and its underlying type theory. Chapter [3] provides an overview of the current methods from the literature. In Chapter [4] a comparison between these methods is provided through extensive experimental studies and their pros and cons with respect to each other are discussed. It also outlines the difference between the two different approaches (4.1,4.1) and elaborates on advantages and disadvantages of these two by means of the existing methods. Finally, in chapter [5] the conclusions that follow from earlier chapters are summarized and some suggestions for future improvements are provided.

Chapter 2

Context of the Investigation

2.1 Introduction

Type theory is a broad and active area of theoretical research. In this chapter, we restrict ourselves to a snapshot of it which is required as background information for the upcoming chapters. For the other notions which are not directly related to the content of this thesis, the reader is provided with appropriate resources.

Constructive type theory is a typed λ-calculus for higher-order logic [Geu95]. The Coq proof assistant has constructive type theory as a theoretical basis, a formalism which gives a clear semantics for representing the programs on a computer and contains both computation and deduction aspects trough the Curry-Howard and De Bruijn correspondence [SU06]. This correspondence (also referred to as proofs-as-programs or formula-as-types correspondence in the literature) relates the systems of formal logic, as present in the proof theory, and computational calculi like the typed λ-calculus as found in the type theory to each other. In terms of this correspondence, formulas correspond to types, proofs correspond to terms (algorithms/ programs), provability corresponds to inhabitation (of a type) and proof normalization corresponds to term reduction.

In essence, the Curry-Howard and De Bruijn correspondence implies that a constructive proof can be transferred to a program and the formula it proves is a type for that program. In other words, if the user can supply a constructive proof that a value of that type exists, then the proof can be checked and converted into executable computer code that computes the value by carrying out the construction. The proof checking feature makes dependently typed languages closely related to proof assistants. The code-generation aspect provides a powerful approach to formal program verification and proof-carrying code, since the code is derived directly from a mechanically verified mathematical proof.

Hence a judgment in type theory of the form:

$$M : A$$

can be read in different ways namely:

1. M is an element of the set A
2. M is a proof (construction) of the formula A
3. M is an algorithm which implements the abstract data type specification A

4. M is a functional program of type A

In case M denotes a proof, one can actually constructs a proof term M representing the deduction of A. For instance, a proof of an implication $A \to B$ is a term $f : A \to B$.

The main consequences of this correspondence in theorem proving are :

- proof checking is type checking which is performed by means of a type checking algorithm. This algorithm checks whether term M is well typed. If the type checking algorithm returns A as the result, than one can conclude that $M : A$. The failure of the type checking indicates that the term M is not typable.
- Interactive Theorem Proving is the interactive construction of a term of a given type. Hence, the user is provided with powerful *tactics* to construct a proof term.

Consequently the class of formal systems that are particularly convenient for Interactive Theorem Proving are the systems based on dependent type theory.

The rest of this chapter is organized as follows: In section [2.2] Coq's type system is introduced. Section [2.3] elaborates on the notion of dependent types and their suitability for the goal of theorem proving. Section [2.4] illustrates the conceptual idea behind the methods as introduced by Bengt Nordström. In section [2.5] the problem statement is reformulated using the formal context provided in this chapter. Finally, section [2.6] is devoted to introducing some notions which are required in the rest of this thesis.

2.2 Coq's type system

The underlying type system of Coq is a Calculus of Constructions with Inductive definition which provides an expressive basis for Coq's proof language [The09]. In terms of this calculus, types are seen as terms of the language and they should belong to another type. The type of a type is always a constant of the language called a sort. There are two basic sorts in this language namely the sort `Set` and the sort `Prop`. The sort `Set` is predicative (informative) which is intended to be used for computational purposes while the sort `Prop` is impredicative (non-informative) and thus it is suitable for logical reasoning about the program. More concretely, the sort `Set` includes the programs and usual sets (such as booleans, naturals, lists etc.). While the sort `Prop` includes the class of terms representing the proofs. In other words, a specification is identified with the set of all programs satisfying the specification and a proposition is identified with the set of its proof objects.
For Coq version V7, the underlying Calculus was known as the Calculus of (Co)Inductive Constructions (Cic) in which the sort `Set` was also impredicative. Recent versions of Coq (version V8.0 and later) are based on a weaker calculus where the sort `Set` satisfies predicative rules. More precisely, the new calculus, Predicative Calculus of (Co)Inductive Constructions (pCic), distinguishes the computational context represented by sort `Set` from the logical context, which is presented by `Prop`. This distinction was initially motivated by the extraction mechanism due to Chiristine Paulin [PM89]. This ensures that the algorithmic part of the function which is responsible for the computation, is separated from the proof components. Once the correctness of the program has been proven, these proof components are not interesting any more. The extraction mechanism of Coq is explained in section [2.3.2] in more detail.

Some important properties of this new type system are:

- every well-formed term has (at least) a type
- every well-formed type is also a term of the calculus

These properties lead to the conclusion that every well-formed type has also a type, which in its turn has a type and so on. In other words, all objects are required to have a type. Consequently sorts also should be given a type because they can be manipulated as ordinary terms. Because assuming simply that `Set` has type `Set` leads to an inconsistent theory (Russell paradox [Men09]), there are infinitely many sorts in the language of (pCic). These are, in addition to `Set` and `Prop` a hierarchy of universes $Type(i)$ for any integer i. The set of sorts S which is defined as $\{Set, Prop, Type(i) \mid i \in \mathbb{N}\}$ has the following properties:

$$Prop : Type_0,\ Set : Type_0,\ Type_i : Type_{i+1}$$

The universe levels, denoted by indexes, are used internally for avoiding the inconsistencies and are not visible to the user.

In the sequel, the syntax and deduction rules of this calculus are provided.

Syntax of pCIC:

Terms are built from variables, global names, constructors, abstraction, application, local declarations bindings, product, sigma type and projections.

$$\begin{aligned}\tau ::= {} & x \mid \tau\ \tau \mid \lambda x : \tau.\, \tau \mid \Pi\ x : \tau.\, \tau \mid \Sigma\ x : \tau.\, \tau \mid (\alpha, \alpha)_{\ \Sigma x:\tau.\tau} \mid \pi_1\ \alpha \mid \pi_2\ \alpha \mid \textbf{let}\ x := \tau\ \textbf{in}\ \tau \\ & \mid Set \mid Prop \mid Type\end{aligned}$$

Note that from a syntactic point of view, types cannot be distinguished from terms, except that they cannot start by an abstraction, and that if a term is a sort or a product, it should be a type.

pCIC Deduction Rules:

There are two judgments defined simultaneously. The judgment $\Gamma \vdash t : T$ means that t is a well-typed term of type T in context Γ. The second judgment $\vdash \Gamma\,(\mathbf{wf})$ means that the context Γ is a valid and well-founded context. Figure [2.1] depicts the pCic's typing rules.

$$\textsc{Wf\_Empty}\ \frac{}{\vdash [\,]\ (\mathbf{wf})} \qquad \textsc{Wf\_Var}\ \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A\ (\mathbf{wf})}\ s \in S \wedge x \notin \Gamma$$

$$\textsc{Var}\ \frac{\vdash \Gamma(\mathbf{wf}) \quad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Axiom}\ \frac{\vdash \Gamma(\mathbf{wf})}{\Gamma \vdash s_1 : s_2}\ (s_1, s_2) \in \mathcal{A}$$

$$\textsc{Prod}\ \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash \Pi x : T.U : s_2}\ (s_1, s_2) \in \mathcal{R}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash \Pi x : T.U : s \quad \Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U} \qquad \textsc{App}\ \frac{\Gamma \vdash f : \Pi x : V.W \quad \Gamma \vdash u : V}{\Gamma \vdash (f\,u) : W[u/x]}$$

$$\textsc{Sum}\ \frac{\Gamma \vdash T : s \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash \Sigma x : T.U : s}\ s \in \{Prop, Set\}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash \Sigma x : T.U : s \quad \Gamma \vdash t : T \qquad \Gamma \vdash u : U[t/x]}{\Gamma \vdash (t, u)_{\Sigma x : T.U} : T.U}$$

$$\textsc{Pi\_1}\ \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_1 t : T} \qquad \textsc{Pi\_2}\ \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_2 t : U[\pi_1 t/x]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash t : U \qquad \Gamma \vdash U \equiv_{\beta\,\pi} T : s}{\Gamma \vdash t : T}$$

$$\textsc{Let}\ \frac{\Gamma \vdash t : T \qquad \Gamma, x := t : T \vdash u : U}{\Gamma \vdash \mathbf{let}\ x := t\ \mathbf{in}\ u : U[t/x]}$$

$$\textsc{Ind}\ \frac{\Gamma, \Gamma_P(\Gamma_I := \Gamma_C)\,(\mathbf{Ind}) \quad j = 1...k}{I_j : A_j} \qquad \frac{\Gamma, \Gamma_P(\Gamma_I := \Gamma_C)\ (\mathbf{Ind}) \qquad i = 1...n}{c_i : C_i}$$

$$\textsc{Match}\ \frac{\Gamma \vdash c : (I\,q_1...q_r\,t_1...t_s) \quad \Gamma \vdash P : B[(I\,q_1 \ldots q_r) \mid B] \quad (\Gamma \vdash f_i : \{(c_{pi}\,q_1 \ldots q_r)^P\})_{i=1..l}}{\Gamma \vdash \mathbf{case}(c,\, P,\, f_1 \mid \ldots \mid f_l) : (P\,t_1...t_s\,c)}$$

$$\textsc{Fix}\ \frac{(\Gamma\ ,\vdash A_i : s_i)_{i=1..n} \qquad (\Gamma\ , f_1 : A_1, \ldots, f_n : A_n \vdash t_i : A_i)_{i=1..n}}{(\Gamma\ ,\vdash Fix\, f_i\{f_1 : A_1 := t_1 \ldots f_n : A_n := t_n\} : A_i}$$

Figure 2.1: pCIC deduction rules

Actually, the rules till the Let rule belong to the Calculus of Constructions. Stating the rules for inductive definitions, match and fix in general form needs quite tedious definitions. We shall

try to give a concrete understanding of the rules by precising them on some examples. In the context Γ the variables that occur in M and A are given a type. In de case that A is a proposition ($\Gamma \vdash A : Prop$), M is called the proof term of the proposition A. In the other case, where A is a set ($\Gamma \vdash A : Set$), M is called to be an element of the set A.
he sorts $s, s_i \in S$ are elements of the set of sorts S further other restrictions are mentioned.
Furthermore, the set of axioms is $\mathcal{A} = \{(Set, Type), (Prop, Type), (Type_i, Type_j \mid i < j)\}$. Finally the set of rules $\mathcal{R}$ is defined on the set of sorts with the following possible combinations:
$\mathcal{R} = \{(Set, Set), (Prop, Set), (S, Prop), (Type_i, Type_j \mid i < j)\}$.

To give an inductive definition, one needs to define the names and the type of the inductive sets to be defined and the names and types of the constructors of the inductive predicates. An inductive declaration can consequently be represented with two contexts (one for inductive definitions ($\Gamma_I = [I_1 : A_1; \ldots; I_k : A_k]$) and one for constructors ($\Gamma_C = [c_1 : C_1; \ldots; c_n : C_n]$)). Furthermore, if an inductive definition admits r inductive parameters, then there exists a context of parameters (Γ_P) of size r, such that $\Gamma_P = [p_1 : P_1; \ldots; p_r : P_r]$.

pCIC VS. Coq's concrete syntax:

The concrete syntax of Coq is in some places different from its abstract syntax simply because some symbols need to be represented in words. Figure [2.2] illustrates the pCIC counterparts in Coq's concrete syntax or elements out of the syntax which are used in this thesis:

pCIC	Coq
$\Pi x : T . U$	$forall\, x : T , U$
$\Sigma\, x : T . U$	$exists\, x : T , U$
$\lambda x : M : T$	$fun\, x : T \Rightarrow M$
$\pi_1\, t$	$proj1\_sig\ t$
$\pi_2\, t$	$proj2\_sig\ t$

Figure 2.2: pCIC VS. Coq

For the inductive definitions, match construct and the fixpoint construct these correspondences are more complicated. The concrete syntax of these constructs and some examples are provided below.
The inductive definitions are represented as follows in Coq:

Inductive $I\ p_1 \ldots p_r : A_i := c_1\, \vec{p_1} : A_1 \mid \ldots \mid c_n\, \vec{p_1} : A_n.$

For instance the natural numbers are defined inductively as:

Inductive $nat : Set := 0 : nat \mid S : nat \rightarrow nat.$

The match construct is presented as follows:

PCIC $\quad$ **case**$(m,\ (\lambda x\ ,\ P),\ \lambda x_{11}\ldots x_{1p_1},\ f_1\ \mid\ \ldots\ \mid\ \lambda x_{n_1}\ldots x_{np_n},\ f_n)$

COQ $\quad$ **match** m **as** x **return** $(P\,x)$ **with** $(c_1\,x_{11}\ldots x_{1p_1})\ \Rightarrow\ f_1\ \mid\ \ldots\ \mid\ (c_n\,x_{n1}\ldots x_{npn})\ \Rightarrow\ f_n$ **end**

In this match construct expression, if m is a term built from a constructor $(c_i\,u_1\ldots u_{pi})$ then the expression will behave as it is specified by the i^{th} branch and will reduce to f_i, where the $x_{i1}\ldots x_{ipi}$ are replaced by $u_1\ldots u_p$. The keywords **as** and **return** can be omitted in cases where the result type of predicate P can be inferred from the type of all the right hand sides. If the type of all right hand sides is not the same everywhere, then the result type should be enforced by means of the predicate P in **return** part. The **as** part can be omitted if either the result type does not depend on m (non-dependent elimination) or m is a variable.

The concrete syntax of fixpoint definitions can be given as:

$$(\Gamma\ ,\vdash\ Fix\,f_i\{f_1\ :\ A_1\ :=\ t_1\ldots f_n\ :\ A_n\ :=\ t_n\}\ :\ A_i$$

$$\textbf{Fixpoint}\ f_i\ (A_1\ldots A_n\ :Ind_A)\ \{struct\ A_i\}\ :\ Type\ :=\ f_1\ :\ A_1\ :=t_1\ldots f_n\ :\ A_n\ :=\ t_n$$

Here the *struct* points out the identifier for which the structural recursion is carried out. A fixpoint definition on natural numbers looks for instance like:

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (plus p m)
end.
```

Here we have briefly explained the COQ's constructs which will be used further on in this thesis. For more information regarding these language constructs, we encourage the reader to read chapter 4 of COQ's reference manual [The09].

2.3 Dependent types

Constructive type theories (like primarily Martin-Löf's type theory [Set04] or Coquand [CH88] and Huet's type theory [BNS00]) are enriched with dependent types. Dependent types are types that depend on the values of other types. By means of conventional type systems such as Hindley_Milner type system ([Mea09]), the user has to provide the type of the program, write the program, write the specifications, and then prove that the program satisfies its specifications. The type system validates then their type. In other words, the program is type checked with respect to a fixed set of criteria and the well typed programs are distinguished from ill formed ones. The dependent type systems go further than that: they enable the user to define a specification about the function along with the function definition and this specification can be as rich as one might desire. Hence, the role of dependent types becomes important when a wider continuum of precision is desired in defining the specifications. By means of dependent types one can realize a continuum of precision from the basic assertions we are used to expect from types up to a complete specification of the

programs' behavior. The choice of how expressive the specification is desired to be, is left to the programmer and depends on to what extend one wants to exploit the expressiveness of such a powerful type discipline.

To give a flavour of this powerful discipline, assume one wants to sort a list of integers. The type specification can be given as follows:

$$sort \ : \ List_{\mathbb{N}} \ \rightarrow \ List_{\mathbb{N}}$$

This is the most general possible typing of the function sort, as we are familiar with from conventional type systems. The specification to which this sort program should then adhere to can be expressed in classical logic as:

$$\forall_{l \ : \ List_{\mathbb{N}}}. \ (Sorted \ (sort \ l))$$

Where $(Sorted \ y)$ is a predicate on the output type of sort, which states that there is a sorted output list which will be returned as the result of this function. By restricting the output type, one can assure that the function produces the desired result. While the outcome of the first variant can be any list (in case that the program doesn't produce what it supposed to produce), the result of second variant is for sure a sorted list. The specification can be refined even more, i.e. the resulted list should not only be a randomly sorted list. It should be the sorted version of the input list. Thus the specification can be refined further as:

$$\forall_{l \ : \ List_{\mathbb{N}}}. \ (Sorted(sort \ l) \ \wedge \ Permutation(l, (sort \ l)))$$

The latter specification, is the most complete specification one can think of for a sorting program. Stated more generally:

$$\forall_{l \ : \ List_N}.\exists_{l' \ : \ List_N}. \ \underbrace{(Sorted(l') \ \wedge \ Permutation(l, l'))}_{P}$$

If we solve the above problem by a program called *sort*, then *sort* is function that, when applied to a list l of natural numbers, produces a pair $< l', p >$ where the first component l' is an ordered permutation of l and the second component p is a proof statement of proposition P.
In most of the cases, however, when reading the existential statement as the specification of a program, one is only interested in the first component being the computational part of it. The second component is only a kind of (formal) witness stating that the first component has a property (in this case property P).
Furthermore, the programs satisfying this specification are exactly the set of all sorting programs, i.e. programs which when applied to a list of natural numbers, produce an ordered permutation of that list as an output.

By means of constructive type theory, the logical part of the specification (proposition P) can be integrated into the type of the program and the type of the program is called to be dependent type because it depends on proposition P. The above specification can be expressed in type theory.

$$sort : \Pi x : List_N.\Sigma y : List_N \underbrace{(Sorted(y) \wedge Permutation(x, y))}_{P}$$

This function sort is defined as a projection of the input list l to a pair $< l', p >$ where the first component l' begin the ordered permutation of l and the second component p being the proof statement of proposition P ($l \mapsto < l', p >$).

As argued earlier, the programmer is interested in a projection like $l \mapsto l'$ in which the formal proof comment is omitted.

This is achieved (in systems like the RUSSELL 3.5) by forming the so called subset types instead of Σ-types by means of which the logical part can be removed. The result is a partial COQ term, where the missing proof statements are represented by meta-variables. This partial COQ term can be made complete by instantiating these holes with the actual proof statements in a later stage[1]. The subset type representation of the above Σ-type is given as:

$$\{y : List_{\mathbb{N}} \mid P\}$$

The type of the *sort* can be given accordingly as:

$$sort : \Pi x : List_N.\{y : List_N \mid (Sorted(y) \wedge Permutation(x, y))\}$$

2.3.1 Subset types VS. Σ-types

When it comes to the computationally relevant content, the powerful Σ- types which merge the data type with the logical propositions becomes less wieldy. Hence the subset types seem to be more suitable candidate in this perspective.There are two reasons, the subset types are the best candidates for expressing the dependently typed programs in COQ.

1. As mentioned before, the second component (proof statement) of the produced pair is often not desired, in subset types the second component can be removed from each element. So they are more suitable for the distinction which needs to be made between programs and proofs.

[1] This shows some resemblance with imperative program construction by means of Hoare Logic [Hoa69]. A Hoare triple $\{P\}\, S\, \{Q\}$ is in fact a partial annotated program with the specifications, initially with only the pre-condition P and the post-condition Q. Using Hoare's deduction rules imperative programs can be constructed gradually. The nodes in the syntax tree of the program are annotated by specifications and at each refining step (using the Hoare's deduction rules), there maybe some side conditions which are added to constructed node. These side conditions can be seen as verification conditions or proof obligations. Initially these verification conditions are marked with holes (meta-variables) which gradually will be filled in with the appropriate proofs i.e. in to drive a correct program, these verification conditions needs to be discharged and initiated by the actual proof statements. There are interactive tools for imperative programs, like Cocktail [Fra00], by means of which this partial annotated program can be filled in with the proofs and eventually result in a complete syntax tree of the program (which can be compared with a complete lambda term in COQ). The proofs in the annotation provide enough information to check a posteriori if a constructed program does indeed meet its specification. Cocktail satisfies further the "Bruijn criterion" because like when a simple type checking algorithm can ensure that a constructed proof (lambda-term) is correct, a simple program checking algorithm ensures an annotated program is correct.

2. A more subtle difference is the desire to make distinction between `Set` and `Prop`. In the set $\Sigma x : T.U$ both T and U are defined to belong to both sorts `Set` and `Prop`. That is while in the subset $\{x : T \mid U\}$ ($T : Set$, $U : Prop$). The reason to this restriction is simply because it corresponds to a pair where the second component contains computationally irrelevant information.

Subset types Deduction Rules:

The deduction rules of subset types in pCic are given in figure [2.3]:

$$\textsc{subset} \quad \frac{\Gamma \vdash A : Set \qquad \Gamma, x : A \vdash P : Prop}{\Gamma \vdash \{x : A \mid P\} : Set}$$

$$\textsc{Element} \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash \{x : A \mid P\} : Set \Gamma \vdash p : P[a/x]}{\Gamma \vdash elt\, A\, (\lambda x : A.P)\, a\, p : \{x : A \mid P\}}$$

$$\textsc{Subset\_}\sigma_1 \quad \frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash \sigma_1\, t : A} \qquad \textsc{Subset\_}\sigma_2 \quad \frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash \sigma_2\, t : P[\sigma_1\, t/x]}$$

Figure 2.3: Subset type in pCic

Provided the motivation to distinguish the computational content from the logical content, brings us to the so called *program extraction* which means the automatic generation of functional code from Coq proofs.

2.3.2 Extraction mechanism in Coq

The Curry-Howard correspondence gives us the possibility to extract programs from their corresponding proofs [Let03]. The distinction which is made between the computational and the logical content of a specification in the underlying type system of Coq facilitates this extraction mechanism. For the *sort* specification for instance the extraction mechanism is illustrated in figure [2.4].

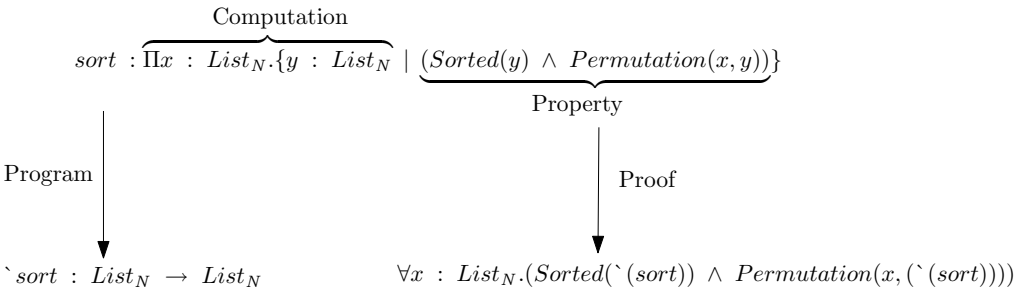

Figure 2.4: Program Extraction

Note that the *grave* in the notation above: (` (*sort*)) indicates the first projection from the decorated subset type.

The current extraction mechanism of Coq supports the languages Objective Caml(Ocaml) [Lea04], Haskell[ea92] and Scheme [Aea98].
The question which may arise here is the following:
Why does the program need to be extracted to an external target language? There are three reasons to motivate this extraction.

1. the first reason behind extracting to an external target language is efficiency. In Coq there is a possibility to execute the program using a so called (`Eval Compute in`) tactic which consecutively applies a series of reductions (see section 4.3 of [The09]). This way of execution is comparable with interpreting, which is by far the most inefficient way of executing. External compilers like OCaml or Haskell are considerably more efficient.
2. the second reason is that the certified code obtained in this way, can be used in larger developments like certified libraries which can be reused several times in different applications.

2.4 Nordström Model of General Recursion

Although not always mentioned explicitly by the followers, the work of Bengt Nordström [Nor88] has played a crucial role in the establishment of a for involvement of general recursion in type theory. This is achieved by generalization of the strong induction principle (noetherian induction principle figure [2.5]) to a so called well-founded induction principle and the introduction of the set of accessible elements.

$$\frac{\forall\, x : N\,.\,(\forall\, y : N\,.\,y < x \;\rightarrow\; P\,y) \rightarrow P\,x}{\forall\, x \;:\; N\,.\,P\,x}$$

Figure 2.5: Strong Induction Principle

The generalization is based on the observation that in the rule for strong induction, there is nothing particular which makes the rule only suitable for natural numbers, beside the $(<)$ ordering which is a well-founded ordering on natural numbers. Hence, by generalizing the $(<)$ to any well-founded relation $(\prec_A)$ on an arbitrary set A, the rule is still applicable (figure [2.6]).

$$\frac{\forall\, x : A\,.\,(\forall\, y \;:\; A\,.y \prec_A x \;\rightarrow\; P\,y) \rightarrow P\,x}{\forall\, x \;:\; A\,.\,P\,x}$$

Figure 2.6: Well-founded Induction Principle

Before giving the definition of the set of accessible elements, some notions need to be formalized.

Definition 2.4.1 (well-founded set)**.** *Let $(\prec_A)$ be a binary relation on the set A and let $(A, \prec_A)$ be a partially ordered set. The relation $(a \prec_A b)$ can be read as "a precedes b". We call $(A, \prec_A)$ a well-founded set if and only if every non-empty subset of A contains minimal element m with respect to the order relation $(\prec_A)$.*

Definition 2.4.2 (R-decreasing chain)**.** *Let $(\prec_A)$ be a binary relation on the set A. R is called to be a R-decreasing chain in a partially ordered set $(A, \prec_A)$, if a totally ordered subset of A has no minimal element. In other words, the sequence $a_i (i \in \mathbb{N})$ in R forms a R-decreasing chain if $(a_{i+1} \prec_A a_i)$ for every index i.*

Proposition 2.4.3. *Let $(A, \prec_A)$ be a partially ordered set. Then the following two statements are equivalent:*

(i) $(A, \prec_A)$ *is a well-founded set.*

(ii) there does not exist any infinite R-descending chain in A.

Proof. (by contradiction).

$(i) \Rightarrow (ii)$. Assume that (i) holds but that there exists an infinite descending chain c in A. However, c is a non-empty subset of A without minimal element, contradicting the fact that A is a well-founded set.

$(ii) \Rightarrow (i)$. Assume that (ii) holds, but that $(A, \prec_A)$ is not a well-founded set. Therefore, there exists a non-empty subset s of A that does not contain a minimal element. However, a non-empty set that does not contain a minimal element must contain an infinite R-decreasing chain, contradicting our assumption (ii). □

The set of accessible elements can now be defined using the above definitions:

Definition 2.4.4 (accessible elements)**.** *The set* $\boldsymbol{Acc}(A, \prec_A)$ *of accessible elements of* $(\prec_A)$ *in A is the set of elements* $(a_i \in A)$ *such that there is no infinite R-decreasing chain of these elements in A. The set* $\boldsymbol{Acc}(A, \prec_A)$ *is called the well-founded part of* $\prec_A$ *by Aczel [Acz77]. The set A is well ordered by* $(\prec_A)$ *if* $A = \boldsymbol{Acc}(A, \prec_A)$

Using this definition, the well-founded induction rule can be refined as follows:

$$\frac{\forall\, x : A\,.\,(\forall\, y :\, A,\, y \prec_A\, x \,\rightarrow\, \mathbf{Acc}\; y)}{\forall\, x \,:\, A\,.\,\mathbf{Acc}\; x}$$

Figure 2.7: Accessibility-intro

Using the set of accessible elements, one can define the function upon this set and hence guarantee the termination of a general recursive function. The work of Nordström forms hence the theoretical basis for the methods we are going to investigate in the upcoming chapter [3].

2.5 Reformulation of problem statement

Having provided enough background information, the problem statement can be made formal. Because the propositions are represented as types and proofs as programs, the non-terminating proofs are not allowed in Coq. In other words due to this intimate relationship between proofs and programs, no Coq object is allowed to trigger an infinite R-decreasing chain. For structural recursive functions, the termination is ensured syntactically i.e. the argument to the recursive call is made structurally smaller by peeling off the constructor once or several times. For general recursive functions on the other hand, the decreasing character can not be inferred syntactically. Hence, to ensure the termination of these kind of recursive calls, one needs to provide a termination proof. Besides the mentioned problems with termination of general recursion, there is a difficulty to define partial functions in type theory. The functions are defined with pattern matching on the constructors of an inductive type. For each constructor, there should be a branch in the match

construct which provides the result for that branch. Because partial functions are not defined for some constructors of the inductive type, pattern matching becomes an issue. The problem is namely what to return as the result for the branches for which there is no function result.

There are two main category of approaches to circumvent these problems:

category 1: give the fully specified function i.e. define the function f such that it satisfies the following specification:

$$f : \Pi\, x :\ A.Px \ \rightarrow\ \{y\ :\ B, R(x, y)\}$$

category 2: separate function specification and properties i.e. define the function f of type $f : A \rightarrow B$ and then prove that it satisfies the following property:

$$\forall\, x :\ A\,.\,P\,x \Rightarrow\ R(x, (f\ x))$$

The first category can use the powerful discipline of dependent types as introduced earlier in this chapter. However, the dependently typed programs are by their nature, proof carrying code. This exhibits their only drawback mainly because of two reasons:

1. the construction of such a code becomes a tedious task. From the programming methodological point of view, the programmer needs to provide (at least) the termination proof simultaneously with the program derivation.
2. because the code is polluted with the proof (which is of no concern to the execution), such code is not easily readable.

In chapter [3] we are going to investigate four methods which deal with this problems in different ways. One of these methods known as the RUSSELL framework, belongs to the first category of approaches (4.1) and the other three belong to the second category(4.1).

The different methods are applied to two running examples which we introduce here. These two running examples, although simple, are representative for our problem domain, i.e. they cover the termination with general recursion functions and partiality. As the reader will notice, understanding of these two examples is overwhelming enough for unexperienced readers in the area of type theory. Hence, these two reasons motivate our choice for these particular examples.

The Running Examples: In order to make the comparisons more clear, two terminating functions are used which belong to the category of interesting cases in which partiality and/or general recursion are addressed. Hence the proof of termination and the function representation in the context of type theory are the main issues which the user will have to deal with. Both examples use the definition of $div2$ from the COQ's standard library.

$div2 = fix\ div2\ (n : nat) : nat :=$
  $match\ n\ with$
  $| 0 \Rightarrow 0$
  $| 1 \Rightarrow 0$
  $| S\ (S\ n') \Rightarrow S\ (div2\ n')$
  $end : nat \rightarrow nat$
$Argument\ scope\ is\ [nat\_scope]$

Running Example 1: log2
This function is a terminating partial function which can be expressed by means of non-structural recursion. Partiality is due to the fact that the function is not defined for $n = 0$ and it is general recursive because the guard predicate can't detect that the argument to the recursive call actually decreases i.e. for $n = S(S\ p)$, $S(div2\ p)$ is not *structurally* smaller than $S(S\ p)$; although it is obviously smaller.

The encoding in OCAML will look like:

```
(**val log2 : nat → nat**)
let rec log2 x = match x with
  | S 0 → 0
  | S (S p) → S (log2 (S (div2 p)))
```

Running Example 2: twoPowN
The second function is also a terminating function which can be expressed by means of general recursion.

The encoding in OCAML will look like:

```
(**val twoPowN : nat → nat**)
let rec twoPowN = function
  | 0 → S 0
  | S p →
    (match even_odd_dec p with
      | Left → mult (S (S 0)) (twoPowN p)
      | Right → square (twoPowN (div2 (S p))))
```

Where *even_odd_dec* makes a decision on whether p is even or odd and is defined in the standard library as:

Lemma 2.5.1. *Lemma even_odd_dec : ∀ n, {even n} + {odd n}.*

It is more intuitive for a programmer to choose to encode the second branch (decision on whether the input is even or odd) as follows:

```
if (even_odd_dec p)
then 2 * twoPowN
else square (twoPowN (div2 (S p))))
```

Or even the version from the standard library:

```
Fixpoint two_power (m : nat) : nat :=
  match m with
  | 0 ⇒ 1
  | S n ⇒ two * two_power n
  end.
```

The encoding we have chosen here is more efficient and is implemented using general recursion which we want to investigate. Furthermore, the choice of branches with the match construct is due to the observation that it bears more resemblance with the extracted OCAML code and hence eases the comparison process.

After providing the application of these examples to four methods, in [4], we are going to compare these four methods by means of the criteria and answer the research questions as introduced in chapter [1].

2.6 Some definitions

In this section, we are going to introduce a number of definitions which will come across in the rest of this thesis.

Definition 2.6.1 (Proof Irrelevance Principle)**.** *Given the function f of the following type:*

$f : \Pi\ x : A.\ P\ x \rightarrow B$ *with* $A, B : Set, P\ x : Prop$

The proof irrelevance principle holds if f a q = f a q' for all a,a':A and q,q' : P a.

Definition 2.6.2 (False_rec)**.** *This* Coq *term is used to deal with any absurd sub-cases. For example, when defining a function* $f : \{x : nat \mid x <> 0\} \rightarrow nat$ *False_rec can be used in any sub-case where* $x = 0$*. During extraction this False_rec is translated into an exception, meaning that execution should never come to this point. Now the Coq partial application* $(f\ 0)$ *is legal, despite the fact that it will be impossible to provide a second argument of type* $0 = 0$*. The definition of* $False\_rec$ *is given as:*

```
False_rec = fun P : Set ⇒ False_rect P
  : forall P : Set, False → P
Argument scopes are [type_scope _]
False_rect =
fun (P : Type) (f : False) ⇒ match f return P with
  end
  : forall P : Type, False → P
Argument scopes are [type_scope _]
```

Definition 2.6.3 (Sigma types)**.** *The* Σ*-types are also defined in* Coq*'s standard library as:*

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
  exist : forall x : A, P x → sig P
```

Where exist can be used to make an element of Σ*-type.*

Definition 2.6.4 (equal_refl)**.** *Sometimes, the resulted type of a function contains an equality. Accordingly, the* $equal\_refl$ *can be used to remove those equalities from the result type.*

```
Notation refl_equal := eq_refl
eq_refl
  : forall (A : Type) (x : A), x = x
```

Definition 2.6.5 (Extraction Inline)**.** *Extraction mechanism performs constants inlining and reductions. In addition to the automatic inline feature, sometimes one needs to explicitly indicate which constants need to be inlined by the* `Extraction Inline` *command.*

Chapter 3

The Methods under Consideration

3.1 Introduction

This chapter provides an overview of the current methods from the literature and introduces(where possible) extensions to the existing methods in order to cover the three main issues as listed in chapter [1] namely:

1. partiality
2. non-structural (general) recursion
3. termination

Furthermore, the application of all the methods in Coq is illustrated by means of the running examples (2.5).

The rest of this chapter is organized as follows:
Section [3.2] describes the converging iteration methods and introduces an extension for the partial functions based on this method. Section [3.3] explains the ad-hoc predicate method and its extension to Cic as suggested by Bertot and Castéran. Section [3.4] introduces the function framework. Finally, section [3.5] investigates the Russell framework. This investigation is more detailed than for the rest of the methods. In addition to the running examples, some extra examples on lists are studied to gain more sight in this framework.

3.2 The Converging Iterations Method

3.2.1 Description of the Method

This method belongs to the second category of approaches (4.1). Balaa and Bertot [BB00] have based their work on the so called *accessibility predicate* (see definition [3.2.1]). In order to deal with general recursion, the function is defined by induction on some well-founded relation for which the recursive calls are decreasing. More precisely, if we provide a function such that recursive calls are only performed on terms that are smaller than the initial argument for some well-founded relation, then we can ensure its termination. In practice, defining a function using well-founded recursion could also be cumbersome. In fact, these functions are also defined by structural recursion, but

rather than following the structure of the input argument, the recursion follows the structure of the proof that there is no infinite descending chain starting from the initial function argument.

The difficulty one has to deal with in this approach is the fact that the full description of the proof is required in order to follow its structure. The *converging iteration method* provides therefore a so called *fix-point equality theorem* automatically, which provides a reduction rule and can be used without any knowledge of the proofs' structure. By means of this method the main issues are covered as follows:

1. partiality: although not addressed by Balaa and Bertot, there is a possibility to refine the accessible domain of input elements which allows us to deal with partiality.
2. general recursion: by induction on some well-founded relation for which the recursive calls are decreasing.
3. termination: the function is defined such that recursive calls are only performed on terms that are smaller for some well-founded relation and therefore the termination is ensured.

Definition 3.2.1 (accessibility)**.** *Given the binary relation R over a set A, an element $a \in A$ is R-accessible if every element smaller (under R) than a is R-accessible. In particular, all minimal elements are R-accessible. The* Coq *definition of accessibility is as follows:*

Inductive Acc [A : Set; R : A → A → Prop] : A → Prop :=
Acc_intro : ∀ x : A.(∀ y : A.(R y x) ⇒ (Acc A R y)) ⇒ (Acc A R x)

In particular, accessibility describes the elements from which one can't start an infinite R-decreasing chain. The *Acc_intro* gives the induction principle for accessible elements: if all elements smaller than x are accessible, then x is accessible.

In order to ease the notation, the relation R is chosen to be $(<)$ in Balaa and Bertot's work. The recursion can now be carried out on the domain of accessible elements $(Acc\ x)$ with strong induction on the well-founded order $(<)$.
The reduction step is:

$$(Acc\_rec\ B\ \phi\ x\ (Acc\_intro\ x\ h)) \rightsquigarrow (\phi\ x\ h\ \ \lambda y : A.\, \lambda p : y < x.\, (Acc\_rec\ B\ \phi\ y\ (h\ y\ p)))$$

Where $\phi : \Phi$ with

$$\begin{aligned}\Phi \equiv\ & [\Pi\ x : A.\, (\Pi\ y : A.\, y < x \Rightarrow (Acc\ y)) \rightarrow \\ & (\Pi\ y : A.\, y < x \rightarrow (B\ y)) \rightarrow (B\ x)]\end{aligned}$$

Acc_rec is the fixpoint definition which uses the *Acc_intro* as the induction principle. The type of *Acc_rec* can be given accordingly as follows:

$$Acc\_rec : \Pi\ B : A \rightarrow Set.\ \ \Phi \rightarrow \Pi\ x : A\ (Acc\ x) \rightarrow (B\ x)$$

The type Φ states that in order to unroll the recursion on x, one can use that all smaller elements are accessible and that for all smaller elements we have already a value. Note that $(h\,y\,p)$ is the proof of y's accessibility. This definition bears analogy with the a Lattice structure (see figure [3.1]).

The function ϕ is defined by means of three arguments in the above definition: an object x of type A, a proof h that all elements smaller than x are accessible (strong induction), and a function that states that for all y smaller than x, $(B\,y)$ is already proved. Having these three ingredients, a value of type x can be produced.

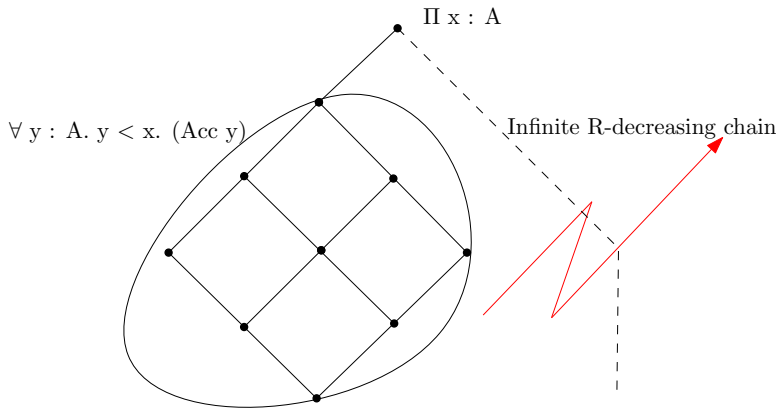

Figure 3.1: Illustration of accessibility predicate

Definition 3.2.2 (well-founded relation)**.** *A relation R is called to be well-founded if the accessibility predicate is verified for all elements of the type.*

Theorem 3.2.3. $wf_<$

$wf_< : (\Pi\ x\ :\ A\,.\,(Acc\ x))$.

If $Acc\,x$ can be defined on the elements of a type, then the function ϕ can be simplified to a term ϕ' of type:

$$\phi^{'} : (\Pi\ x : A.\,(\Pi\ y : A.\,y < x \rightarrow (B\ y)) \rightarrow (B\ x))$$

Having this new function, a simplified version of *Acc_rec*, $Rec_{wf<}$, can be provided with the following type:

$$Rec_{wf} <:(\Pi\ x : A.\,(\Pi\ y : A.\,y < x \rightarrow (B\ y)) \rightarrow (B\ x)) \rightarrow (\Pi\ x : A.\,(B\ x))$$

Rec_{wf} satisfies the following equality:

$$(Rec_{wf} < \phi^{'}\ x) = (\phi^{'}\ x\ \ \lambda y : A.\,\lambda p : y < x.\,(Rec_{wf} < \phi^{'}\ y))$$

The later is present in Coq's standard library and is called *well_founded_induction*. For the completeness matters we provide its definition below:

```
well_founded_induction =
fun (A : Type) (R : A → A → Prop) (Rwf : well_founded R) (P : A → Set) ⇒
well_founded_induction_type Rwf P
   : forall (A : Type) (R : A → A → Prop),
     well_founded R →
     forall P : A → Set,
     (forall x : A, (forall y : A, R y x → P y) → P x) →
     forall a : A, P a
Arguments A, R are implicit
Argument scopes are [type_scope _ _ _ _ _]
```

where *well_founded_induction_type* is defined as:

```
well_founded_induction_type =
fun (A : Type) (R : A → A → Prop) (Rwf : well_founded R)
  (P : A → Type) (X : forall x : A, (forall y : A, R y x → P y) → P x)
  (a : A) ⇒
Acc_rect P
  (fun (x : A) (_ : forall y : A, R y x → Acc R y)
    (X0 : forall y : A, R y x → P y) ⇒ X x X0) (Rwf a)
   : forall (A : Type) (R : A → A → Prop),
     well_founded R →
     forall P : A → Type,
     (forall x : A, (forall y : A, R y x → P y) → P x) →
     forall a : A, P a
Arguments A, R are implicit
Argument scopes are [type_scope _ _ _ _ _]
```

and *well_founded* is defined as:

```
well_founded =
fun (A : Type) (R : A → A → Prop) ⇒ forall a : A, Acc R a
   : forall A : Type, (A → A → Prop) → Prop
Argument A is implicit
Argument scopes are [type_scope _]
```

and finally *Acc_rect* definition is given as:

```
Acc_rect =
fun (A : Type) (R : A → A → Prop) (P : A → Type)
  (f : forall x : A,
       (forall y : A, R y x → Acc R y) →
       (forall y : A, R y x → P y) → P x) ⇒
fix F (x : A) (a : Acc R x) {struct a} : P x :=
  match a with
  | Acc_intro a0 ⇒ f x a0 (fun (y : A) (r : R y x) ⇒ F y (a0 y r))
  end
     : forall (A : Type) (R : A → A → Prop) (P : A → Type),
       (forall x : A,
          (forall y : A, R y x → Acc R y) →
          (forall y : A, R y x → P y) → P x) → forall x : A, Acc R x → P x
Arguments A, R, x are implicit
Argument scopes are [type_scope _ _ _ _ _]
```

Based on Balaa and Bertot method, the *well_founded_induction* receives as parameters 1) the input type (A), 2) the relation on this type (<), 3) the theorem which states that this relation is indeed well-founded(lt_wf), 4) a function (P) which gives the result type for each input element and finally 5) a function ϕ' which describes the computation part. From these parameters, the first two are implicit and are inferred by Coq. The user needs to specify the last three parameters only.

In practice the summery of steps one needs to undertake are:

1. provide and prove a theorem which states that the argument to the recursion call decreases at each step.
2. define a function ϕ' which describes how to compute $(\phi'\, x)$, knowing $(\phi'\, y)$ for all $(y < x)$.

3. use *well_founded_induction* with the input parameters as described above.

In the sequel, the first running example is used to illustrate this method. Initially, partiality is ignored and the "*log*2 0" is defined to be "0". Afterwards, a variant of this method is encoded which is capable of dealing with the partiality.

3.2.2 Application of the Method

Running Example 1: log2
First of all a theorem is required which states that the argument to the recursive call decrease at each step.

Theorem 3.2.4.

$$\mathit{Theorem}\ \mathit{div2\_lt} : \mathit{forall}\ (x : \mathit{nat}), (\mathit{lt}\ (\mathit{div2}\ (S\ x))\ (S\ x)).$$

The next step is to provide the ϕ' function which does the computations (using the theorem [3.2.4]). Considering the type of ϕ', this function involves the reasoning step as well i.e. the fact that for computing x all smaller elements should be accessible. The resulted type needs therefore to be refined by means of the *return* construct to ensure this requirement.

$$\begin{array}{l}
\mathit{Definition}\ \mathit{log2_comp}\ (x : \mathit{nat}) := \\
\quad \mathit{match}\ x\ \mathit{return}\ (\mathit{forall}\ f : (\mathit{forall}\ y : \mathit{nat}, \mathit{lt}\ y\ x \to \mathit{nat}), \mathit{nat})\ \mathit{with} \\
\quad\quad \mid 0 \Rightarrow \\
\quad\quad\quad \mathit{fun}\ f : (\mathit{forall}\ y : \mathit{nat}, \mathit{lt}\ y\ 0 \to \mathit{nat}) \Rightarrow 0 \\
\quad\quad \mid S\ 0 \Rightarrow \\
\quad\quad\quad \mathit{fun}\ f : (\mathit{forall}\ y : \mathit{nat}, \mathit{lt}\ y\ (S\ 0) \to \mathit{nat}) \Rightarrow 0 \\
\quad\quad \mid S\ (S\ m) \Rightarrow \\
\quad\quad\quad \mathit{fun}\ f : (\mathit{forall}\ y : \mathit{nat}, \mathit{lt}\ y\ (S\ (S\ m)) \to \mathit{nat}) \Rightarrow \\
\quad\quad\quad\quad (S\ (f\ (\mathit{div2}\ (S\ (S\ m)))\ (\mathit{div2_lt}\ (S\ m)))) \\
\mathit{end} \\
\quad : \mathit{forall}\ x : \mathit{nat}, (\mathit{forall}\ y : \mathit{nat}, y < x \to \mathit{nat}) \to \mathit{nat}
\end{array}$$

The good news is that this definition can be simplified. Due to the fact that the only case which requires a proof is the last branch where the decreasing character of the argument can't be inferred automatically. Hence the simplified version is given as:

$$\begin{array}{l}
\mathit{Definition}\ \mathit{log2_comp}\ (x : \mathit{nat}) := \\
\quad \mathit{match}\ x\ \mathit{return}\ (\mathit{forall}\ f : (\mathit{forall}\ y : \mathit{nat}, \mathit{lt}\ y\ x \to \mathit{nat}), \mathit{nat})\ \mathit{with} \\
\quad\quad \mid 0 \Rightarrow \mathit{fun}\ f : _ \Rightarrow 0 \\
\quad\quad \mid S\ 0 \Rightarrow \mathit{fun}\ f : _ \Rightarrow 0 \\
\quad\quad \mid S\ (S\ m) \Rightarrow \\
\quad\quad\quad \mathit{fun}\ f : (\mathit{forall}\ y : \mathit{nat}, \mathit{lt}\ y\ (S\ (S\ m)) \to \mathit{nat}) \Rightarrow (S\ (f\ (\mathit{div2}\ (S\ (S\ m)))\ (\mathit{div2_lt}\ (S\ m)))) \\
\mathit{end} \\
\quad : \mathit{forall}\ x : \mathit{nat}, (\mathit{forall}\ y : \mathit{nat}, y < x \to \mathit{nat}) \to \mathit{nat}
\end{array}$$

Finally, the definition of *log2* can be given using the *well_founded_induction* which receives the well-foundedness of the relation (<) plus a function which maps the elements of input to the output and of course the computation function *log2_comp*. The *well_founded_induction* is the function that takes care of the recursive calls.

```
log2 : forall a : nat, (fun _ : nat ⇒ nat) a
Definition log2 := (well_founded_induction lt_wf (fun _ : nat ⇒ nat) log2_comp).
```

Running Example 2: twoPowN
First of all a theorem is required which states that the argument to the recursive call decrease at each step. Note that in this case, there are two recursive calls both of which need to be filled in with required proof statement. The first recursive call is however a structural recursion and the desired theorem is already present in `Coq.Arith.Lt` library of Coq as follows:

Theorem 3.2.5.

```
Theorem lt_n_Sn : forall n, n < S n.
```

The second recursive call is not structural and hence the decrement of the argument to the recursive call needs to be proved first:

Theorem 3.2.6.

```
Theorem div2_lt : forall (x : nat), (lt (div2 (S x)) (S x)).
```

The next step is to provide the ϕ' function which does the computations (using the theorem [3.2.6]). Considering the type of ϕ', this function involves the reasoning step as well i.e. the fact that for computing x all smaller elements should be accessible. The resulted type needs therefore to be refined by means of the *return* construct to ensure this requirement. We provide here the simplified version:

```
Definition twoPowN_comp (x : nat) :=
match x return
(forall f : (forall y : nat, lt y x → nat), nat) with
| 0 ⇒ fun f : _ ⇒ 1
| S p ⇒ fun f : (forall y : nat, lt y (S p) → nat) ⇒
  if (even_odd_dec p) then 2 * (f p (lt_n_Sn p)) else (square (f (div2 (S p)) (div2_lt p)))
end.
```

Finally, the definition of *twoPowN* can be given using the *well_founded_induction* which receives the well-foundedness of the relation (<) plus a function which maps the elements of input to the output and of course the computation function *twoPowN_comp*. The *well_founded_induction* is the function that takes care of the recursive calls.

```
Definition twoPowN := (well_founded_induction lt_wf (fun _ : nat ⇒ nat) twoPowN_comp).
```

3.2.3 The Converging iteration method on partial functions

3.2.3.1 Description of the Method

In order to make the converging iteration method a suitable candidate for partial function definitions, the accessibility predicate can be used to impose a restriction on the input elements. The most natural way to achieve this is to provide the function with the domain of acceptable elements:

Theorem 3.2.7. $wf_{<part}$

$wf_{<part}$: $(\forall\, x\ :\ A\,.\,(Dom\ x) \rightarrow (Acc x))$.

After which the type of *Acc_rec* can be refined:

$$Acc\_rec : \Pi\ B : A \rightarrow Set.\ \ \Phi \rightarrow \Pi\ x : A\ (Dom\ x) \rightarrow (B\ x)$$

Let's take a look at our running example to see whether this is applicable.

3.2.3.2 Application of the Method

Running Example 1: log2
Putting the restriction on the acceptable input elements via a hypothesis which states that the input value is not supposed to be a 0, will result in the following encoding of the function ϕ'.

```
Definition log2_comp_part (x : nat) (domain : x <> 0) :=
  match x as z return x = z → (forall y : nat, lt y z → nat) → nat with
   | 0 ⇒ fun g ⇒ fun f ⇒ False_rec nat (domain g)
   | S 0 ⇒ fun g ⇒ fun f ⇒ 0
   | S (S m) ⇒ fun g ⇒
     fun f : (forall y : nat, lt y (S (S m)) → nat) ⇒ (S (f (div2 (S (S m))) (div2_lt (S m))))
end (refl_equal x)
  : forall x : nat, x <> 0 → (forall y : nat, y < x → nat) → nat
```

Which before applying the $(refl\_equal\ x)$ is of the type:

$$: x = x \rightarrow forall\ x : nat, x <> 0 \rightarrow (forall\ y : nat, y < x \rightarrow nat) \rightarrow nat$$

In order to apply the hypothesis to the first branch, we need an extra function g which takes care of the equality in result type, the type of function f remains the same:

$$\lambda z\ :\ nat.\ \underbrace{x = z}_{\text{g}} \rightarrow \underbrace{(\Pi y\ :\ nat.\, y\ <\ z\ \rightarrow\ nat)}_{\text{f}} \rightarrow\ nat$$

Furthermore in the above definition, the function *False_rec* which is borrowed from the Coq's standard library enables us to drive a contradiction in case of 0 and yet returning the desired type which is the type *nat* in this case.

The right hand side of the first branch which uses *False_rec* simplifies as follows:

$(domain\ g) \rightsquigarrow False$
and hence
$False\_rec\ nat\ (domain\ g) \rightsquigarrow nat$

Which is exactly of the required type.

So far, it seems that the introduction of domain is a useful way to adapt the Balaa and Bertot method in such a way that it can deal with partial function definitions. The next step is to provide the actual function definition by means of $well\_founded\_induction$. Following the guidelines will lead us to the following definition:

$Definition\ log2\_part := (well\_founded\_induction\ lt\_wf\ (fun\ \_ : nat \Rightarrow nat)\ log2\_comp\_part).$

However, the type of this definition doesn't comply with the type that $well\_founded\_induction$ expects and therefore an error occurs.

By adapting the return type, one has to ensure that it becomes of the type: $x <> 0 \rightarrow nat$. Furthermore, the input of function f needs to be non-zero. For our example this amounts to an axiom which is used in the last branch of the pattern matching.

Axiom 3.2.8.

$Axiom\ non\_zero : forall\ x : nat, div2\ (S\ (S\ x)) <> 0.$

As mentioned before, one needs to change the return type to ensure that all elements are non-zero. Hence, the number of input elements of the function f increases to three, 1) a natural number, 2) a lemma which states that y is smaller than z and hence accessible, 3) the new requirement which states that $(y <> 0)$.

$return\ x = z \rightarrow (forall\ y : nat, lt\ y\ z \rightarrow y <> 0 \rightarrow nat) \rightarrow nat$

As in the previous version, the function f is only needed to be made explicit in the last branch where the actual work is happening on the right hand side. Consequently, there is the place where the axiom should be used as the last input element of the function f.

$$
\begin{aligned}
&Definition\ log\_comp2\_part\_adapted\ (x : nat)\ (domain : x <> 0) :=\\
&match\ x\ as\ z\ return\ x = z \rightarrow (forall\ y : nat, lt\ y\ z \rightarrow y <> 0 \rightarrow nat) \rightarrow nat\ with\\
&\mid 0 \Rightarrow fun\ g \Rightarrow fun\ f \Rightarrow False\_rec\ nat\ (domain\ g)\\
&\mid S\ 0 \Rightarrow fun\ g \Rightarrow fun\ f \Rightarrow 0\\
&\mid S\ (S\ m) \Rightarrow fun\ g \Rightarrow\\
&\quad fun\ f : (forall\ y : nat, lt\ y\ (S\ (S\ m)) \rightarrow y <> 0 \rightarrow nat) \Rightarrow\\
&\qquad (S\ (f\ (div2\ (S\ (S\ m)))\ (div2\_lt\ (S\ m))\ (non\_zero\ m)))\\
&end\ (refl\_equal\ x) :\\
&: forall\ x : nat, x <> 0 \rightarrow (forall\ y : nat, y < x \rightarrow y <> 0 \rightarrow nat) \rightarrow nat
\end{aligned}
$$

As one may observe, the resulted type of this new function is closer to the intended type except for the position of $(x <> 0)$ which is desired to be a part of the output result B. Therefore, we define an auxiliary function which does the replacement:

$$
\begin{aligned}
&Definition\ log\_comp2\_part\_final\ (x : nat)\ (\phi : forall\ y : nat, y < x \rightarrow y <> 0 \rightarrow nat)\ (dom : x <> 0)\\
&:= log\_comp2\_part\_adapted\ x\ dom\ \phi.\\
&log\_comp2\_part\_final : forall\ x : nat, (forall\ y : nat, y < x \rightarrow y <> 0 \rightarrow nat) \rightarrow x <> 0 \rightarrow nat
\end{aligned}
$$

The function $log\_comp2\_part\_final$ has the desired type and can be used in the definition of $well\_founded\_induction$.

$$
\begin{aligned}
&Definition\ log2\_final :=\\
&\quad (well\_founded\_induction\ lt\_wf\ (fun\ y : nat \Rightarrow y <> 0 \rightarrow nat)\ log\_comp2\_part\_final).
\end{aligned}
$$

Note that a minor change needs to be carried out on the type of function which was responsible for relating the input type to the output type. This adjustment $(y <> 0 \rightarrow nat)$ complies with

the result type according to the new computation function(log_comp2_part_final).

Last but not least, the extracted OCAML code, is according to the expectation according to both solutions:

```
(**val log2_comp_part : nat → nat**)
let rec log2_part = function
  | 0 → assert false (*absurd case*)
  | S n → (match n with
    | 0 → 0
    | S m → S (log2_part (div2 (S (S m)))))
```

3.3 Ad-hoc Predicate Method

3.3.1 Description of the Method

This method belongs to the second category of approaches as introduced in the introduction chapter (4.1). In their work, Bove and Capretta [BC05], introduce an inductive accessibility predicate to tackle the problems of general recursive functions. They take the Martin-Löf's type theory as the bases. This ad-hoc predicate specifies the domain of input values for which the function terminates. The function is then defined by structural recursion on the proof that the input value satisfies this predicate. Because the new function definition is structurally recursive, it fits in the context of constructive type theory. We will refer to this predicate as the *inductive domain predicate* further on.
By means of this predicate, the three main issues can be addressed.

1. partiality: by limiting the definition of the inductive domain to the values for which a function can be defined, one can deal with partiality.
2. general recursion: given the fact that the function is defined by structural recursion on the proof that input values satisfy the inductive domain predicate, there is no notion of general recursion in the functions' definition. In fact, the goal is to mimic the general recursion definition of the functional program.
3. termination: no involvement of general recursion in the function definition implies that the function terminates due to the structural recursive nature of the definition.

3.3.2 Application of the Method

To illustrate this method, we use our running examples as defined in introduction part [2.5].

Running Example 1: log2
In order to apply the ad-hoc predicate method, one needs to define the inductive domain predicate on the permissible input values first. Considering the definition of the function in OCAML, the following observations can be made:

- the function is undefined for 0.
- if the input value is 1, than the function terminates with the output value 0.

- for input values greater than 1 (of the form "$S(S\ p)$"), the function terminates provided that it terminates on "$S(div2\ p)$".

This amounts to an inductive definition of the domain predicate expressing on which values of natural numbers, the $log2$ function is defined. Accordingly, the Dpow2 domain can be Accordingly, the log_domain can be defined by means of the two rules depicted figure [3.2].

$$\frac{}{log\_domain\ 1} \qquad \frac{log\_domain(S(div2\ p))}{log\_domain(S(S\ p))}$$

Figure 3.2: Domain rules for $log2$

Based on these two rules, the inductive domain of $log2$ can be expressed as follows:

```
Inductive log_domain : nat → Set :=
  log_domain_1 : log_domain 1
| log_domain_2 :
  forall (p : nat), log_domain (S (div2 p)) → log_domain (S (S p)).
```

By defining the log_domain as an inductive type, the partiality of $log2$ is enforced due to the fact that there exists no constructor for the input value 0 in the domain. Now the function definition can be given with a second argument which is the proof that the first argument belongs to the domain of the function. The function definition is then given by pattern matching on the constructors of the inductive predicate, log_domain.

```
Fixpoint log2 (x : nat) (h : log_domain x) {struct h} : nat :=
match h with
| log_domain_1 ⇒ 0
| log_domain_2 p h1 ⇒ S (log2 (S (div2 p)) h1)
end
   : forall x : nat, log_domain x → nat
```

Pattern matching on $(log\_domain\, x)$ results in two cases (two domain constructors should be considered). In the first case the input value is 1. Considering the OCaml version of $log2$, we know that the returned value should be 0. In the second case where the input value is greater than 1, we have that $(S(div2\ p))$ satisfies the relation log_domain (with h1 as a proof of this). According to the OCaml version, the recursion is continued on $(S(div2\ p))$. In the Coq version of this function we need to provide a proof that $log\_domain\ (S(div2\ p))$ holds which is exactly the type of h1.

Running Example 2: twoPowN
The second example considers a generally recursive total function. Considering the definition of the function in OCaml, the following observations can be stated:

- for the input value 0, the function terminates with the output value 0.
- for the input values greater than 0 (of the form "$S\ p$"), there are two cases:
  - if the input value is even, than the function terminates on "$S\ p$" provided that it terminates on "$div2(S\ p)$".
  - if the input value is odd, than the function terminates on "$S\ p$" provided that it terminates on "p".

This amounts to an inductive definition of the domain predicate expressing on which values of natural numbers, the *twoPowN* function terminates. Accordingly, the Dpow2 domain can be defined by means of the three rules depicted figure [3.3].

$$\frac{}{Dpow2\ 0} \qquad \frac{even(S\ p) \qquad Dpow2(div2(S\ p))}{Dpow2(S\ p)} \qquad \frac{odd(S\ p) \qquad Dpow2\ p}{Dpow2(S\ p)}$$

Figure 3.3: Domain rules for *twoPowN*

Based on these three rules, the inductive predicate for the domain of *twoPowN* can be expressed as follows:

```
Inductive Dpow2 : nat → Set :=
  Dpow2_0 : Dpow2 0
| Dpow2_S_even :
  forall (p : nat), even (S p) → Dpow2 (div2 (S p)) → Dpow2 (S p)
| Dpow2_S_odd :
  forall (p : nat), odd (S p) → Dpow2 p → Dpow2 (S p).
```

The function definition is given by pattern matching on the constructors of the inductive domain predicate Dpow2. A second argument is needed which is the proof that the first argument belongs to the domain of the function.

```
Fixpoint twoPowN (n : nat) (h : Dpow2 n) {struct h} : nat :=
match h with
| Dpow2_0 ⇒ 1
| Dpow2_S_even p h1 h2 ⇒ square (twoPowN (div2 (S p)) h2)
| Dpow2_S_odd p h1 h2  ⇒ 2 * (twoPowN p h2)
end
   : forall n : nat, Dpow2 n → nat
```

Pattern matching on ($Dpow2\ n$) results in three cases (three domain constructors have to be considered). In the first case where the input value is 0, the OCaml version of *twoPow*2 indicates that the returned value should be a 1. In the second case where the input value is greater than 0, there are two possibilities namely the input value could be even or odd. Considering the case where the input value is even, we have that $even(S\ p)$ holds (with $h1$ as a proof of this) and that ($div2(S\ p)$) satisfies the relation $Dpow2$ (with $h2$ as a proof of this). Considering the OCaml version of this function, the recursion is in this case continued on $div2(S\ p)$. In Coq version of this function we need to provide a proof that $Dpow2(div2(S\ p))$ holds, which is exactly the type of $h2$. The same reasoning can be followed for the other case where the input value is odd.

Choosing the suitable inductive domain predicate leads to a well-founded recursion which is structurally recursive over the proofs that the input satisfies the inductive predicate. The extracted OCaml code of our first running example is:

```
(**val log : nat → log_domain → nat**)
let rec log x = function
  | Log_domain_1 → 0
  | Log_domain_2 (p, h1) → S (log (S (div2 p)) h1)
```

Considering the original OCaml code, this is not quite the code we expected. As mentioned earlier, this method is originally based on another type theory (Martin-Löf's type theory) which is

different from the underlying type theory of Coq (Cic) in the sense that there is no distinction made between the sorts `Set` and `Prop` (observe the result type of the inductive domain predicates in the previous examples). This results in an extracted code that deviates from the expectation, simply because the `Set` sort is relevant from the computational point of view while the elements of sort `Prop` are relevant from reasoning point of view and hence are omitted from the extracted code. The Coq's extraction mechanism [2.3.2] carries therefore only the elements of `Set` sort to the extracted code. A variant of the ad-hoc predicate method as suggested by Yves Bertot and Pierre Castéran [BC04] addresses this issue by adapting the ad-hoc predicate method towards Cic. The next paragraph explains this adaptation.

3.3.3 Ad-hoc predicate method in CIC

3.3.3.1 Description of the Method

In order to adapt the ad-hoc predicate method such that it can be used satisfactorily in Cic type theory, the result type of the inductive domain predicate should be of the `Prop` sort. However, using the `Prop` sort restricts the pattern matching (this time on proofs) which was the key concept to the ad-hoc predicate method in order to obtain the structurally recursive definitions. In order to solve this compatibility problem, Bertot and Castéran introduce so called *inversion theorems* for each recursive call. As its name implies, the inversion theorem states that the only way to create a term of an inductive type is via the type's constructors; in other words the proof argument for the recursive call can be deduced from the initial proof argument. By isolating the pattern matching steps by means of these inversion theorems, the pattern matching can be carried out on the main function input rather than the constructors of the inductive domain predicate. However, the partiality problem remains an issue because pattern matching on the main argument needs to deal with all input values rather than those which are contained in the domain of the function. In order to comply with the exhaustive pattern matching requirement, one needs to prove theorems which state that certain input values are excluded from the domain, which is also a form of inversion. Hence, there is more work involved to adapt this method to Cic in comparison with the original version:

1. Inversion theorems need to be specified along with a correctness proof for each recursive call.
2. for each element which is excluded from the domain of a partial function, a separate theorem needs to be formulated along with its correctness proof.

Generally, when we have an arbitrary inductive predicate like $Inductive\, P\ :\ A\ \rightarrow\ Prop$, we define the smallest predicate on A that is closed under the given constructors. For instance for the *log_domain* (see [3.3.2]), there is no pattern on the right hand side of the inductive definition that matches *log_domain* 0, so we have $\neg$*log_domain* 0.

Likewise, the only way to create $log\_domain\, S(S\, p))$ is by using the constructor *log_domain_2*, so if we know $log\_domain(S(S\, p))$ then we know also $log\_domain(S(div2\, p))$.

3.3.3.2 Application of the Method

To illustrate the Bertot and Castéran version of the ad-hoc predicate method, we use the first running example.

Running Example 1: log2
The inductive domain predicate remains almost the same, except for the result type which should be of the sort `Prop`.

```
log_domain : nat → Prop

Inductive log_domain : nat → Prop :=
  log_domain_1 : log_domain 1
| log_domain_2 :
  forall (p : nat), log_domain (S (div2 p)) → log_domain (S (S p)).
```

Because Coq controls the pattern matching upon exhaustiveness, the values for which the function is not defined like 0 in case of $log2$, need to be covered by means of a theorem. For the $log2$ function this amounts to a theorem which proves that 0 doesn't belong to the domain. This theorem states that all of the elements of the domain are unequal to 0. This is achieved by excluding 0 from the domain definition (i.e. there is no constructor for 0 in *log_domain*). We refer to this theorem as *Partiality Theorem* further on.

Theorem 3.3.1. *Partiality Theorem*

```
Theorem log_domain_non_0 : forall (x : nat), log_domain x → (x <> 0).
```

The last required ingredient are the inversion theorems. In this example, we have to deal with a single recursive call, consequently we need to formulate just one inversion theorem. We need to formulate that when x is of the form "$S(S\ p)$", and x is an element of the domain, then "$S(div2\ p)$" is also an element of the domain. For $log2$ the *inversion theorem* amounts to:

Theorem 3.3.2. *Inversion theorem*

```
Theorem log_domain_inv :
forall (x p : nat), log_domain x → x = S (S p) → log_domain (S (div2 p)).
```

Finally the function definition can be given. Unlike the ad-hoc predicate method, the pattern matching is now carried out on the first argument$(x : nat)$. Note that the recursion is still on the proof that x is in the domain.

```
Fixpoint log2 (x : nat) (h : log_domain x) {struct h} :=
match x as y return x = y → nat with
| 0     ⇒ fun h' ⇒
          False_rec nat (log_domain_non_0 x h h') : (0 = x → nat)
| S 0   ⇒ fun h' ⇒
          0 : (S 0 = x → nat)
| S (S p)  ⇒ fun h' ⇒
          S (log2 (S (div2 p)) (log_domain_inv x p h h')) : (S (S p) = x → nat)
end (refl_equal x)
   : forall x : nat, log_domain x → nat
```

Note that in the above definition, the type of the right hand sides is not the same everywhere, as it is often the case when dealing with dependent types. Consequently, the elimination predicate needs to be specified explicitly because in this case the type can't be synthesized. Hence we need to specify the predicate after *return* explicitly. The *return* construct is used whenever the result

type can't be inferred from the type right hand sides and hence needs to be enforced. For clarity, we have given the precise type for each case. The actual result type of the match structure turns out to be $x = x \rightarrow nat$ which is not the expected type. Providing the match with an extra argument $(equal\_refl\ x)$ results in the desired result type which is type nat in this case.

eq_refl
$\quad : forall\ (A : Type)\ (x : A), x = x$

The type of $False\_rec$ (see [2.6.2]) in the above definition is given as:

$False\_rec$
$\quad : forall\ P : Set, False \rightarrow P$

For the above function, the right hand side of the first branch simplifies to :

$(log\_domain\_non\_0\ x\ h\ h') \rightsquigarrow False$
and hence
$False\_rec\ nat\ (log\_domain\_non\_0\ x\ h\ h') \rightsquigarrow nat$

Finally, the extracted code in OCAML is as expected:

```
(**val log2 : nat → nat**)
let rec log2 = function
  | 0 → assert false (*absurd case*)
  | S n → (match n with
    | 0 → 0
    | S p → S (log2 (S (div2 p))))
```

Running Example 2: twoPowN
As for the previous example, the inductive domain predicate remains almost the same(see 3.3.2), except for the result type which should be of the sort `Prop` i.e.

$$DPow2 : nat \rightarrow Prop$$

There is no partiality involved in this example and hence we can go further with defining the inversion theorems. In this example, we have to deal two recursive calls, consequently we need to formulate two inversion theorem, one for even values and one for odd values. For even values, we need to formulate that when x is of the form "$S\ p$", and x is an even element of the domain, then "$(div2\ (S\ p))$" is also an element of the domain. For values of the even branch *inversion theorem* amounts to:

Theorem 3.3.3. *Inversion theorem (even)*

```
Lemma Dpow2_even_inv :
forall (x p : nat), even (S p) → Dpow2 x → x = S p → Dpow2 (div2 (S p)).
```

For odd values, we need to formulate that when x is of the form "$S\ p$", and x is an odd element of the domain, then "q" is also an element of the domain. For values of the odd branch *inversion theorem* amounts to:

Theorem 3.3.4. *Inversion theorem (odd))*

```
Lemma Dpow2_odd_inv :
forall (x q : nat), odd (S q) → Dpow2 x → x = S q → Dpow2 q.
```

Note that while the even brach has to deal with general recursive calls, the recursive calls of odd brach are structural recursive.

Finally the function definition can be given. Unlike the ad-hoc predicate method, the pattern matching is now carried out on the first argument($x\ :\ nat$). Note that the recursion is still on the proof that x is in the domain.

```
Fixpoint twoPowN (x : nat) (h : Dpow2 x) {struct h} :=
match x as y return x = y → nat with
| 0    ⇒ fun h' ⇒ 1
| S p ⇒ match (even_odd_dec (S p)) with
          | left h1 ⇒ fun h' ⇒ (square (twoPowN (div2 (S p)) (Dpow2_even_inv x p h1 h h')))
          | right h1 ⇒ fun h' ⇒ (2 * (twoPowN p (Dpow2_odd_inv x p h1 h h')))
        end end (refl_equal x).
```

Note that in the above definition, the type of the right hand sides is not the same everywhere again. Hence the resulted type needs to be specified as a predicate after *return* explicitly. For clarity, we have given the precise type for each case. The actual result type of the match structure is again $x = x \rightarrow nat$ which is not the expected type. Providing the match with an extra argument $(equal\_refl\ x)$ results in the desired result type which is type *nat* in this case.

Finally, the extracted code in OCAML is as expected:

```
(**val twoPowN : nat → nat**)
let rec twoPowN = function
  | 0 → S 0
  | S p →
    (match even_odd_dec (S p) with
       | Left → square (twoPowN (div2 (S p)))
       | Right → mult (S (S 0)) (twoPowN p))
```

3.4 Function Framework

3.4.1 Description of the Method

This method belongs to the second category of approaches (4.1). The FUNCTION framework due to Forest et al. [GBR06] is based on the observation that inductive relations provide a convenient tool to describe mathematical functions by their graph. The FUNCTION framework offers the user sufficient ingredients to reason about the recursive functions in the COQ proof assistant. More specifically, it provides the user with the following:

- induction principles: to prove properties about the function's output.
- inversion principles: to deduce the possible function's input knowing the output.
- fixpoint equation: to unfold the function definitions in proofs.

This framework was initially implemented as a stand alone tool. Currently it is integrated in COQ by means of the experimental command *Function*. From a description of a function, a graph representation of the given function's pseudo code is generated as an inductive relation. This graph relation is generated by following the function's definition step by step i.e. a constructor is defined for each branch of the pattern matching. Furthermore, in case of general recursive functions, a measure or a well-founded order is provided which yields proof obligations which need to be discharged by the user in order to accomplish the termination proof. Feeding the tool with the function $f : A \to B$, a function $\bar{f}$ is generated along with all the principles needed to reason about $\bar{f}$ (see figure [3.4]). More precisely, the graph relation is proved to represent a function which is by construction equal to the original function definition. Using the extraction mechanism of COQ, the resulted OCAML is (almost) the same as the expected code.
It is worth mentioning that the graph relation bears some resemblance with the domain definition of the ad-hoc predicate method.

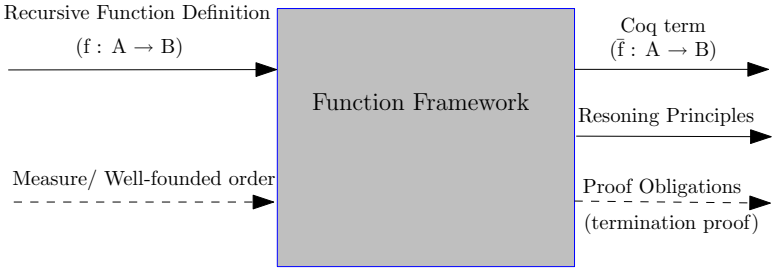

Figure 3.4: FUNCTION Framework

The main shortcoming of this framework is its inability to deal with dependent types and hence partiality. However, the two other main issues are covered in this framework:

1. general recursion: by enabling the user to define a measure function or a well-founded order, the decreasing behavior of the argument to the recursive call can be verified.
2. termination: by imposing the usage of measure function or well-founded ordering in case of general recursion and providing the proof obligations accordingly, one can prove the termination of these functions after discharging the provided proof obligations.

There are several internal steps involved to achieve the function definition along with the required principles:

Step 1: Defining an inductive relation $\bar{f}$, which is the graph of the intended function $f : A \to B$.

Step 2: Providing an auxiliary function which encodes that the function f satisfies its graph relation.

$$f\_rich : A \to \{y : B \mid \bar{f}\ x\ y\}$$

Step 3: Defining f by removing the logical part of f_rich, reducing its type to $A \to B$.

Step 4: Proving that $\bar{f}$ defines a function in the usual mathematical sense i.e.

$$\Pi x\, y_1\, y_2.\ \bar{f}\ x\, y_1 \to \bar{f}\ x\, y_2 \to y_1 = y_2$$

Step 5: Finally, from the definition of $\bar{f}$ a fixpoint equation for f is obtained and from the induction principle of $\bar{f}$ an induction principle for f is derived and from the inversion principles of $\bar{f}$ the inversion principles for f are derived.

All the steps named above are automated by means of the FUNCTION framework. The user provides the pseudo code of the original function f along with a well_founded order for general recursive functions, the framework generates then the assosiated principles to the function, suitable for reasoning purposes. In order to clarify these steps, in the sequel we recite these steps by means of the second running example.

3.4.2 Application of the Method

Running Example 2: twoPowN
First of all, a function definition along with a measure function needs to be provided by the user. The latter is due to the general recursive nature of twoPowN.

```
Function twoPowN (n : nat) {measure id n} : nat :=
match n with
| 0 → S 0
| S p →
  (match even_odd_dec p with
     | Left → mult (S (S 0)) (twoPowN p)
     | Right → square (twoPowN (div2 (S p))))
end
   : nat → nat
```

Providing this definition, the underlying FUNCTION command performs the following steps internally.

Step 1: Defining the graph of twoPowN

First of all it generates the graph relation corresponding to the definition automatically, using one constructor for each branch of the pattern matching. For twoPowN this graph relation is defined as follows:

```
Inductive twoPowN_rel : nat → nat → Prop
  pow2_0 : twoPowN_rel 0 1
| pow2_even :
  forall q res, (even_odd_dec (S q)) = left _ →
    twoPowN_rel (div2 (S q)) res → twoPowN_rel (S q) (square res)
| pow2_odd :
  forall q res, (even_odd_dec (S q)) = right _ →
    twoPowN_rel q res → twoPowN_rel (S q) (2 * res)
```

Whereby the *pow*2_0 provides the base case, whereas the *pow*2_*even* and *pow*2_*odd* deal with the cases where argument is even, respectively odd.

Step 2: Auxiliary function enriched with dependent types

Using this graph relation, a Coq function ($twoPowN : nat \rightarrow nat$) is generated which specifies that the function satisfies its graph relation:

$$twoPowN\_rel : \Pi n.twoPowN\_rel\ n\ (twoPowN\ n)$$

This auxiliary function can be defied as:

```
Definition twoPowN_type (n : nat) := { twoPowN_n : nat | twoPowN_rel n twoPowN_n }.
```

There are two techniques to achieve obtain this auxiliary function.

- the first technique which is inspired by the converging iteration method of Balaa and Bertot (see section [3.2]), uses a well-founded induction principle to define an auxiliary function by using *well_founded_induction* from the standard library. Recall that defining a function by means of the converging iteration method has involved two steps i.e the computational step and the actual definition. In this case the computational step can be given as follows:

  ```
  Definition twoPowN_wf : forall x, (forall y, y < x → twoPowN_type y) → twoPowN_type x.
  ```

  After which the actual definition can be provided as:

  ```
  Definition twoPowN_rich := well_founded_induction lt_wf (fun x ⇒ twoPowN_type x) twoPowN_wf.
  ```

- the second technique which is inspired by the Ad-hoc predicate method of Bove and Capretta (see section [3.3]), extracts the ad-hoc predicate from the graph relation by projection. Depending on whether the predicate is in `Set` or `Prop`, the exact definition of the inductive domain predicate *Dpow*2 (definition [3.3.2]) is produced by inspection of the graph relation. If the predicate lives in the `Prop` sort, the corresponding inversion lemmas are generated first to justify recursive calls, as explained in Section [3.3].

Step 3: Eliminating the dependent types

In order to define the *twoPowN* the logical parts of the dependent type information of *twoPowN_rich* is eliminated, thus making it a function of the type $nat \rightarrow nat$.

```
Definition twoPowN (n : nat) : nat := let (f, _) := twoPowN_rich n in f.
```

Step 4: Proving *twoPowN_rel* is functional

The next step is to prove that *twoPowN_rel* is functional i.e to each value in the domain it associates only one value in the co-domain. Fortunately, since the definition of the graph relation *twoPowN_rel* follows the function's definition, and because the branches of the match are mutually exclusive, the constructors of the graph relation are also mutually exclusive.

$$\Pi\ x\ y_1\ y_2.\ twoPowN\_rel\ x\ y_1 \rightarrow twoPowN\_rel\ x\ y_2 \rightarrow y_1 = y_2$$

Step 5: Fixpoint equation and Induction/Inversion principle

At this point the principles can be derived automatically. The crucial fact that the *twoPowN_rel* relation is functional, provides the possibility to obtain the fixpoint definition for *twoPowN* from it.

- fixpoint equation: fixpoint equation is obtained by reasoning on *twoPowN_rel* (for each constructor individually), following the definition of the pseudo-code. For instance for the even case the following is proved:

  $$pow2\_odd : forall\ q, (even\_odd\_dec\ (S\ q)) = right\ \_ \rightarrow twoPowN\ (S\ q) = (2 * (twoPowN\ q))$$

- induction principles: for all inductive definitions, Coq automatically generates an induction principle. Using the induction principle of the graph relation *twoPowN_rel*, the induction principle for *twoPowN* is obtained by changing the one provided by Coq slightly by removing all occurrences of *twoPowN_rel* and replacing them by values of recursive calls of *twoPowN*.

- inversion principles: an hypothesis of the form $e' = twoPowN\ e$ is first transformed into its graph relation $twoPowN\_rel\ e\ e'$, then the Coq tactic inversion is used to discriminate some incompatible cases, and again all generated hypotheses dealing with graph relation *twoPowN_rel* are replaced with the equivalent form dealing with *twoPowN*.

Having demonstrated the internal steps, we take a look at the result of the whole from the user point of view. As mentioned earlier, the user needs only to provide the pseudo code of the function definition together with the measure function. As a result of the measure function, the user is exposed with the following proof obligations required for the termination proof:

```
============================
  forall n p : nat,
  n = S p →
  forall anonymous : even p, even_odd_dec p = in_left → id p < id (S p)


============================
  forall n p : nat,
n = S p →
forall anonymous : odd p,
even_odd_dec p = in_right → id (div2 (S p)) < id (S p)
```

After discharging the proof obligations required for the termination proof, the automatically generated principles are defined: twoPowN_tcc, twoPowN_terminate, twoPowN_ind, twoPowN_rec, twoPowN_rect, R-twoPowN_correct, R-twoPowN_complete, twoPowN and twoPowN_equation the types of the most interesting ones are:

- R_twoPowN_correct (Soundness):

$$\textit{forall } n \ \textit{res} : \textit{nat}, \textit{res} = \textit{twoPowN } n \rightarrow \textit{R\_twoPowN } n \ \textit{res}$$

  where ($\textit{R\_twoPowN} :: \textit{nat} \rightarrow \textit{nat} \rightarrow \textit{Set}$) is the graph relation. The soundness principle states that for all input values n and output values res, the graph relation holds.

- R_twoPowN_complete(Completeness) :

$$: \textit{forall } n \ \textit{res} : \textit{nat}, \textit{R\_twoPowN } n \ \textit{res} \rightarrow \textit{res} = \textit{twoPowN } n$$

  The completeness principle states that for all input values n and output values res if the graph relation holds then the function $\textit{twoPowN}$ is applicable on input value n and the result of this application will be res.

- twoPowN_terminate (termination condition):

$$\begin{aligned}
&\textit{forall } n : \textit{nat}, \{ v : \textit{nat} \mid \\
&\quad \textit{exists } p : \textit{nat}, \\
&\quad\quad \textit{forall } k : \textit{nat}, p < k \rightarrow \\
&\quad\quad\quad \textit{forall def} : \textit{nat} \rightarrow \textit{nat}, \textit{iter } (\textit{nat} \rightarrow \textit{nat}) \ k \ \textit{twoPowN_F def } n = v \}
\end{aligned}$$

  Which is the termination requirement and in natural language states that: for each input value n, there is an output value v such that for all iteration upper bounds k, if the function $\textit{twoPowN\_F}$ iterates p times on n and doesn't exceed this upper bound then the outcome value is always v. def stands for the default value.

- twoPowN_ind (The induction principle):

$$\begin{aligned}
&\textit{forall } P : \textit{nat} \rightarrow \textit{nat} \rightarrow \textit{Prop}, \\
&\quad (\textit{forall } n : \textit{nat}, n = 0 \rightarrow P \ 0 \ 1) \rightarrow \\
&\quad\quad (\textit{forall } n \ p : \textit{nat}, n = S \ p \rightarrow \\
&\quad\quad\quad \textit{forall } _x : \textit{even } p, \textit{even_odd_dec } p = \textit{in_left} \rightarrow \\
&\quad\quad\quad\quad P \ p \ (\textit{twoPowN } p) \rightarrow P \ (S \ p) \ (2 * \textit{twoPowN } p)) \rightarrow \\
&\quad\quad (\textit{forall } n \ p : \textit{nat}, n = S \ p \rightarrow \\
&\quad\quad\quad \textit{forall } _x : \textit{odd } p, \textit{even_odd_dec } p = \textit{in_right} \rightarrow \\
&\quad\quad\quad\quad P \ (\textit{div2} \ (S \ p)) \ (\textit{twoPowN } (\textit{div2} \ (S \ p))) \rightarrow P \ (S \ p) \ (\textit{square } (\textit{twoPowN } (\textit{div2} \ (S \ p))))) \rightarrow \\
&\textit{forall } n : \textit{nat}, P \ n \ (\textit{twoPowN } n)
\end{aligned}$$

  Figure [3.4.2] illustrates the above induction principle in Gentzen style.

$$(\text{IB}) \quad \frac{}{\text{P } 0 \ 1}$$

$$\begin{array}{l} (\text{IH}) \\ (\text{IS}) \end{array} \quad \frac{\textit{even } p \qquad P \ p \, (\textit{twoPowN } p)}{P \ (S \ p) \, (2 * \textit{twoPowN } p)}$$

$$\begin{array}{l} (\text{IH}) \\ (\text{IS}) \end{array} \quad \frac{\textit{odd } p \qquad P \ (\textit{div2} \ (S \ p)) \quad (\textit{twoPowN } (\textit{div2} \ (S \ p)))}{P \ (S \ p) \, (\textit{power2} \, (\textit{twoPowN} \, (\textit{div2} \, (S \, p))))}$$

$$\Downarrow$$

$$\forall n. P \, n \, (\textit{twoPowN } n)$$

Finally the extracted code is as follows:

```
(**val twoPowN : nat → nat**)
let twoPowN x =
  twoPowN_terminate x
```

Which after inlining the twoPowN-terminate turns out to be our expected code:

```
(**val twoPowN : nat → nat**)
let rec twoPowN = function
  | 0 → S 0
  | S p →
    (match even_odd_dec p with
       | Left → mult (S (S 0)) (twoPowN p)
       | Right → square (twoPowN (div2 (S p))))
```

3.5 Russell Framework

3.5.1 Description of the Method

This method belongs to the first category of approaches (4.1). The RUSSELL is a language which is particularly suitable for programming functions and proving their properties using dependent types in COQ. This framework can be elaborated to define partial functions in COQ. It also provides a phase distinction between writing and proving the correctness of the functions like in the FUNCTION framework [3.4]. To this end RUSSELL uses so called *Predicate subtyping* à la PVS. This is achieved by means of extending conversion to an equivalence which equates types and subset types based on them. Concretely the coercion (denoted by $\triangleright$) is formulated as follows for a subset type $\{x : U \mid P\}$:

$$\{x : U \mid P\} \triangleright U$$

Which means that a term t of type $\{x : U \mid P\}$ will also be of type U. The projection σ_1 is left implicit (it is an implicit coercion).

The resulting type system is a weaker one in the sense that it doesn't require a term of a sub type to contain the proof components (the proof that the element satisfies the predicate P). For instance the derivation of the input type for our first running example can be expressed as follows in CIC using the ELEMENT rule:

$$\textsc{Element}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash \{x : A \,|\, P\} : Set \qquad \Gamma \vdash p : P[a/x]}{\Gamma \vdash elt\, A\, (\lambda x : A.P)\, a\, p : \{x : A \,|\, P\}}$$

$$\frac{\Gamma \vdash m : nat \quad \Gamma \vdash \{n : nat \,|\, n <> 0\} : Set \quad \overbrace{\Gamma \vdash p : m <> 0}^{\text{Proof Component}}}{\Gamma \vdash elt\ nat\ (\lambda n : nat.\, n <> 0)\, m\, p : \{n : nat \,|\, n <> 0\}}$$

Where A is instantiated with nat, a with m, x with n and P with $m <> 0$.

In RUSSELL the proof component of this rule is left out as *hole* (meta-variable) in the term where the proof should be filled in. A proof obligation will be generated for each such *hole*.

This type system enables us to write the algorithmic code only, while retaining the richness of COQ's specifications. The insertion of proof components is postponed to a later stage. Hence, the correctness proof can be carried out afterwards by means of a translation into a partial COQ derivation, where the missing parts (holes) are represented by meta-variables (existential variables). This missing parts will be on their turn instantiated with actual proofs. These *proof obligations* are generated automatically, and, once dealt with by the user, permit to construct a complete, valid COQ term. Figure [3.5] illustrates the whole idea behind RUSSELL.

The three main issues are addressed as follows in RUSSELL:

1. partiality: partiality is encoded along with the function definition using the syntactic sugar provided for representing subset types:

   $$(x : T \mid P) \equiv (x : \{\, x : T \mid P \,\})$$

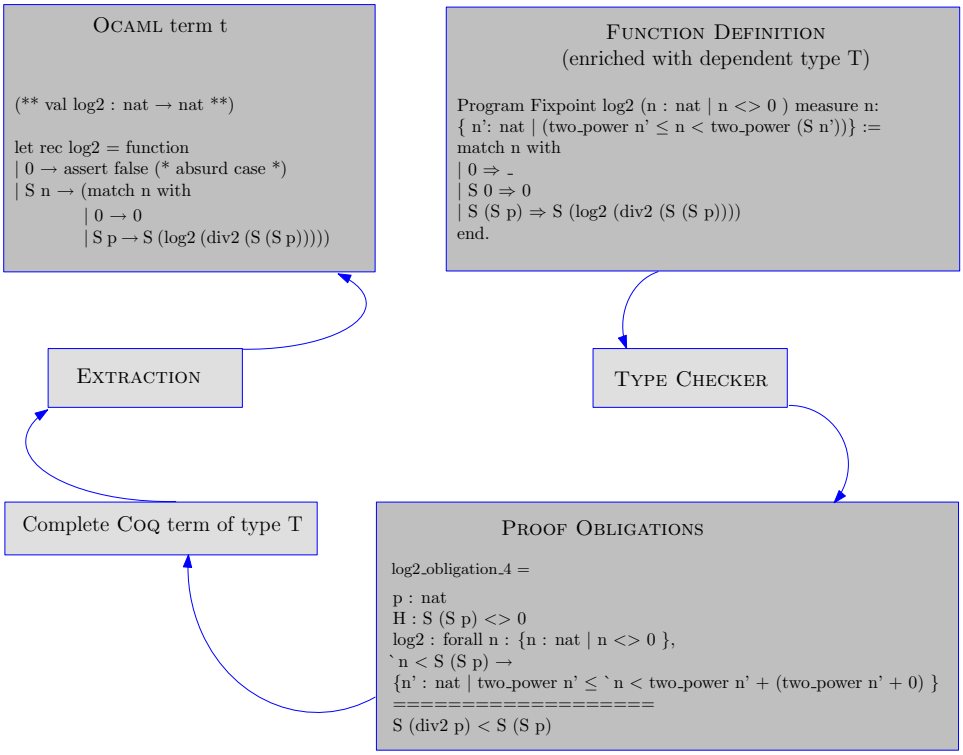

Figure 3.5: RUSSELL FRAMEWORK

Because there is an implicit coercion from $(x : T \mid P)$ to the type T, x can be used as $(x : T)$ but also as $(x : \{y : nat \mid P[y/x]\})$. Pattern matching is then done exhaustively; on cases for which the function is not supposed to provide an answer -that means whenever P does not hold- the matching branch simply provides an empty result denoted by (_) or (!) (see the example of head [3.5.3.1]).

2. general recursion: by enabling the user to provide a measure function, the decrease of the recursive argument should be proven. The PROGRAM tactic generates the corresponding proof obligations.

3. termination: the termination requirement may be specified by a measure function into a well-founded ordering. After discharging the proof obligations which state that the measure decreases at each recursive call, the termination is ensured.

The rest of this chapter is organized as follows. In section [3.5.2], the syntax and semantics of the RUSSELL type system are given. Section [3.5.3] is divided to two parts. The first subdivision [3.5.3.1], extensively discusses the differences/similarities between the generated *proof obligations* by means of the PROGRAM tactic and those one could think of given a function decorated with its specification. This investigation eases the understanding of the framework and provides more insights about the generated COQ terms in this framework. The second subdivision [3.5.3.3], is devoted the running examples are encoded in RUSSELL.

3.5.2 Russell type system

The syntax stratifies between types and terms, and these two categories will be amalgamated under the name Term hereafter.

Syntax:

$$\alpha ::= x \mid \lambda x : \tau.\, \alpha \mid \alpha\ \alpha \mid \alpha\ \tau \mid (\alpha, \alpha)_{\ \Sigma x:\tau.\tau} \mid \pi_1\ \alpha \mid \pi_2\ \alpha \mid \sigma_1\ \alpha \mid \sigma_2\ \alpha$$
$$\tau ::= x \mid \tau\ \tau \mid \tau\ \alpha \mid \lambda x : \tau.\, \tau \mid \Pi\ x : \tau.\, \tau \mid \Sigma\ x : \tau.\, \tau \mid \{x : \tau \mid \tau\} \mid \mathit{Set} \mid \mathit{Prop} \mid \mathit{Type}$$

Semantics:
The phase distinction has to be realized using the predicate subtyping of PVS. The rules for predicate subtyping in PVS can be given as follows:

$$\frac{\Gamma \vdash t : \{x : T \mid P[x]\}}{\Gamma \vdash t : T} \qquad \frac{\Gamma \vdash t : T \qquad \vdash_\Gamma P[t]}{\Gamma \vdash t : \{x : T \mid P[x]\}}$$

Figure 3.6: Predicate subtyping PVS

Which means that if we have an object of a subset type based on T, we can forget about its `Prop` part and consider it as an object of type T and furthermore in case we have an object of type T we can consider it as an object of the subset type $\{x : T \mid P[x]\}$ provided that it satisfies the predicate P. This last requirement can be considered as a type checking condition. Leaving this condition out of RUSSELL's type system results in a weaker type system which allows the users to provide the function's code without any proof statements involved in the definition. The required type checking conditions then are represented as meta-variables and are term holes which are displayed to the user as proof obligations (if not automatically proved) in a later stage.

Generally speaking the RUSSELL type system follows the same conventions as CIC, with sigma types but without universes (see figure 3.7).

The sorts $s, s_i \in S$ are elements of the set of sorts S further other restrictions are mentioned. Furthermore, the set of axioms is $\mathcal{A} = \{(\mathit{Set}, \mathit{Type}), (\mathit{Prop}, \mathit{Type}), (\mathit{Type}_i, \mathit{Type}_j \mid i < j)\}$. Finally the set of rules $\mathcal{R}$ is defined on the set of sorts with the following possible combinations: $\mathcal{R} = \{(\mathit{Set}, \mathit{Set}), (\mathit{Prop}, \mathit{Set}), (S, \mathit{Prop}), (\mathit{Type}_i, \mathit{Type}_j \mid i < j)\}$.
The dependent sums are only allowed when $(T, U : \mathit{Set})$ or $(T, U : \mathit{Prop})$. The case where $(T : \mathit{Set}, U : \mathit{Prop})$ is distinguished in the rule SUBSET. The last possible pair $(T : \mathit{Prop}, U : \mathit{Set})$ is forbidden because it corresponds to a pair where the second component contains computational information which depends on a proof of proposition U.

$$\textsc{Wf\_Empty}\ \frac{}{\vdash [\,]\,\mathbf{wf}} \qquad \textsc{Wf\_Var}\ \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A\,\mathbf{wf}}\ s \in S \wedge x \notin \Gamma$$

$$\textsc{Var}\ \frac{\vdash \Gamma\ \mathbf{wf} \quad x : A \in \Gamma}{\vdash \Gamma, x : A} \qquad \textsc{Axiom}\ \frac{\vdash \Gamma\ \mathbf{wf}}{\Gamma \vdash s_1 : s_2}\ (s_1, s_2) \in A$$

$$\textsc{Prod}\ \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash \Pi x : T.U : s_3}\ (s_1, s_2, s_3) \in R$$

$$\textsc{Abs}\ \frac{\Gamma \vdash \Pi x : T.U : s \quad \Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U} \qquad \textsc{App}\ \frac{\Gamma \vdash f : \Pi x : V.W \quad \Gamma \vdash u : V}{\Gamma \vdash (f\,u) : W[u/x]}$$

$$\textsc{Sum}\ \frac{\Gamma \vdash T : s \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash \Sigma x : T.U : s}\ s \in \{Prop, Set\}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash \Sigma x : T.U : s \quad \Gamma \vdash t : T \qquad \Gamma \vdash u : U[t/x]}{\Gamma \vdash (t, u)_{\Sigma x : T.U} : T.U}$$

$$\textsc{Pi\_1}\ \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_1 t : T} \qquad \textsc{Pi\_2}\ \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_2 t : U[\pi_1 t/x]}$$

$$\textsc{Subset}\ \frac{\Gamma \vdash A : Set \quad \Gamma, x : A \vdash P : Prop}{\Gamma \vdash \{x : A \,|\, P\} : Set}$$

$$\textsc{Element}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash \{x : A \,|\, P\} : Set \qquad \Gamma \vdash p : P[a/x]}{\Gamma \vdash elt\,A\,(\lambda x : A.P)\,a\,p : \{x : A \,|\, P\}}$$

$$\textsc{Subset\_}\sigma_1\ \frac{\Gamma \vdash t : \{x : A \,|\, P\}}{\Gamma \vdash \sigma_1 t : A} \qquad \textsc{Subset\_}\sigma_2\ \frac{\Gamma \vdash t : \{x : A \,|\, P\}}{\Gamma \vdash \sigma_2 t : P[\sigma_1 t/x]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash t : U \qquad \Gamma \vdash U \equiv_{\beta\pi} T : s}{\Gamma \vdash t : T}$$

Figure 3.7: Calculation of coercion by predicates- Russell deduction rules

3.5.2.1 Conversion Rules in Russell

In RUSSELL, the conversion rule CONV is replaced by a new subsumption rule COERCE (see figure [3.8]) which will also implement the subset equivalence.

$$\textsc{Coerce}\ \frac{\Gamma \vdash t : U \qquad \Gamma \vdash T \triangleright U : s}{\Gamma \vdash t : T}$$

$$\textsc{Subset}\ \frac{\Gamma \vdash U : Set \quad \Gamma, x : U \vdash P : Prop}{\Gamma \vdash \{x : U \mid P\} : Set}$$

Figure 3.8: RUSSELL new rules

Predicate subtyping is renamed to subset equivalence because the relation is made symmetric, contrary to usual subtyping relations. The rule CONV makes sure that $\beta\pi$-conversion is included in the subset equivalence judgment.

$$\triangleright\text{-Conv}\quad \frac{\Gamma \vdash T \equiv_{\beta\pi} U : s}{\Gamma \vdash T \triangleright U : s}$$

$$\triangleright\text{-Sym}\quad \frac{\Gamma \vdash U \triangleright T : s}{\Gamma \vdash T \triangleright U : s} \qquad \triangleright\text{-Trans}\quad \frac{\Gamma \vdash S \triangleright T : s \quad \Gamma \vdash T \triangleright U : s}{\Gamma \vdash S \triangleright U : s}$$

$$\triangleright\text{-Prod}\quad \frac{\Gamma \vdash U \triangleright T : s_1 \qquad \Gamma, x : U \vdash V \triangleright W : s_2}{\Gamma \vdash \Pi x : T.V \triangleright \Pi x : U.W : s_2}$$

$$\triangleright\text{-Sum}\quad \frac{\Gamma \vdash T \triangleright U : s \qquad \Gamma, x : T \vdash V \triangleright W : s}{\Gamma \vdash \Sigma x : T.V \triangleright \Sigma y : U.W : s} \quad s \in \{Prop, Set\}$$

$$\triangleright\text{-Subset}\quad \frac{\Gamma \vdash U \triangleright V : Set \qquad \Gamma, x : U \vdash P : Prop}{\Gamma \vdash \{x : U \,|\, P\} \triangleright V : Set}$$

$$\triangleright\text{-Proof}\quad \frac{\Gamma \vdash U \triangleright V : Set \qquad \Gamma, x : V \vdash P : Prop}{\Gamma \vdash U \triangleright \{x : V \,|\, P\} : Set}$$

Figure 3.9: Russell conversion

3.5.3 Application of the Method

As mentioned in the previous chapter, the Russell framework generates the corresponding *proof obligations* when given a function enriched with its specification. Depending on the obligations, it then attempts to solve these obligations automatically. The non-trivial obligations remain and are presented to the user.

Notations In order to present the examples, different styles are mixed together, some following the conventional Coq syntax while others are presented by means of concrete Coq syntax as introduced in section [2.2]. To make these differences clear, the headings are annotated with suitable titles: concrete Coq syntax versus raw Coq syntax.

Program Definition: Hereby the Coq syntax is extended with predicates expressing the specifications to which the functions must adhere. In general the syntax confirms with the following conventions:

Program Definition / Fixpoint $F\ (x : Ind_A \mid P_x)\ \{measure\ x\} : \{y : B \mid R\ x\ y\}$
match x *with*
$\quad \mid cons_1\ (\vec{x}_1) \Rightarrow Body_1\ (\vec{x}_1)$
$\quad \mid cons_2\ (\vec{x}_2) \Rightarrow Body_2\ (\vec{x}_2)$
$\quad \vdots$
$\quad \mid cons_n\ (\vec{x}_n) \Rightarrow Body_n\ (\vec{x}_n)$
end.

Here, the predicate P_x refines the domain elements of the function and fulfills the same role as the precondition in imperative programming. The predicate $(R\,x\,y)$ specifies the relation between input and output of the function and fulfills the same role as postcondition of the function in imperative programs. In other words, it specifies the properties which should hold for the output

of the function. Furthermore, the measure function can be added in case of general recursion. It will cause the generation of proof obligations which guarantee the decreasing character of the argument to the recursive call. Resolving these obligations will result in the proof of termination for the general recursive functions.

List of remaining Obligations (concrete Coq syntax): Typing the keyword *Obligations* provides us with a list of remaining obligations. Note that some obligations may have been solved automatically and are hence not included in this list. One can use *Check [obligation-name]* in order to view the obligations for which the correctness has been proved already.

Unsolved Obligations representation(raw Coq syntax): In order to start proving the next unsolved obligation; the keyword *Next Obligation* should be used.

Emerged obligations: The proof obligation is established by means of induction on an inductive type when the function is defined recursively. The general form of such an obligation is:
$\forall x : Ind_A \ . \ P_x \rightarrow R_x \grave{(}F_x)$
In cases where the function is non-recursive, these obligations are provided by means of case distinction on the branches of the match construct.

Created term (concrete Coq syntax): After proving the correctness of all remaining obligations, one can ask for the certified term by Coq, containing the function definition and its proofs of logical obligations, using *Print [Definition/Fixpoint-name].*

3.5.3.1 Manually vs. Automatically Generated Proof Obligations

One interesting aspect regarding generation of proof obligations is to study the correspondence between the expected *proof obligations* (as the user can write down manually given a function decorated with its desired specifications) and those generated automatically using the Russell framework. This investigation helps us to identify the differences and gain more insight into the automatically generated proof obligations.
Throughout this chapter, this comparison is carried out between the proof obligations as they are generated by Coq, using the *Program* tactic, and the ones which intuitively emerge from the given annotated function. this is realized by means of a number of simple examples which are organized as follows.

First of all the function definition, enriched with the specification, is provided. Secondly, the remaining proof obligations generated by Coq are brought into the light. These proof obligations are presented in two formats, the second of which will be presented to the user as proof obligation. Having the automatically generated proof obligations at hand, we then introduce the intuitive proof obligations one could think of when the annotated function is given. Finally a comparison is made between these two versions of the proof obligations and the differences are discussed. Last but not least the certified term definition is presented.

Example 3.5.1. *Definition of head in* Coq*:*

Given a list of items, the head function returns the first element of an non-empty list.

```
Program Definition head (l : list A | l <> [ ]) :
  { a : A | exist l' : list A, a :: l' = l } :=
match l with
  | hd :: tl ⇒ hd
  | nil      ⇒ _
end.
```

List of remaining Obligations (concrete Coq syntax):
Coq provides us with the following as the remaining proof obligation:

```
forall l : { l : list A | l <> [ ] },
  let filtered_var :=`l in
    forall (hd : A) (tl : list A),
      hd :: tl = filtered_var → exists l' : list A, hd :: l' = hd :: tl.
```

Note that here a new variable called $filtered\_var$ has been introduced and the value of `l, which is the first projection of l is assigned to it. The variable $filtered\_var$ is hence of the type $list\,A$. As the name of the variable $filtered\_var$ suggests, it concerns the filtering of the first projection of the sigma type $\{l : list\,A \mid l <> [\,]\}$.
The notation $(hd :: tl = filtered_var- >)$ means that the substitution of the first projection of the input list with $hd :: tl$ results in a result type where all occurrences of input list are substituted by $hd :: tl$.

Unsolved Obligations representation(raw Coq syntax):
The remaining proof obligation is presented to the user as follows:

```
hd : A
tl : list A
H : hd :: tl <> [ ]
============================
  exists l' : list A, hd :: l' = hd :: tl
```

Having seen the obligations as they are generated by Coq, we are now going to come up with the obligations we would think of given the definition of head.

Emerged obligations:
Because the definition of head is not recursive, the case distinction technique is applicable here. There are two cases for which the function produces an output. Expressing the proof obligation in terms of predicate logic yield:

(*To prove case (`l = input):*)

$(\forall$ l : list A, l $\neq$ [ ] $\xrightarrow{`l=input}$ ($\exists$ a : A . ($\exists$ l' : list A, a :: l' = input)))

(*case `l = nil*)
Let's denote *nil* as [ ], then for `l = [ ] then the following should hold:

$(\forall$ l : list A, l $\neq$ [ ] $\xrightarrow{`l=[]}$ ($\exists$ _ : A . ($\exists$ l' : list A, `(head([ ])) :: l' = [ ])))
– note that the assumption [ ] $\neq$ [ ] is contradictory and because "Ex Falso Quodlibet", the proof of this case is trivial.

$(case\ `l = hd :: tl)$

$(\forall$ l : list A, l $\neq$ [ ].
$(\forall$ hd : A, tl : list A $\xrightarrow{`l=hd::tl}$ $(\exists$ l' : list A, `(head(hd :: tl)) :: l' = hd :: tl)))

– note that this can be simplified using the definition of head to:

$(case\ `l = hd :: tl)$

$\forall$ l : list A, l $\neq$ [ ].
$(\forall$ hd : A, tl : list A $\xrightarrow{`l=hd::tl}$ $(\exists$ l' : list A, hd :: l' = hd :: tl)))

As one may observe, the generated proof obligation by Coq is the the same as the one we've introduced. The important observation is

The only subtle difference is that the case of empty lists is not considered by Coq among the list of remaining obligations which means that it has been solved automatically.
If we ask for it explicitly, the following is presented as the base case:

The automatically solved obligation:

```
head_obligation_2
  : forall l : { l : list A | l <> [] },
    [] = `l → { a : A | exists l' : list A, a :: l' = [] }
```

Which is the same the intuitive base case one would think of.

The generated head-term (concrete Coq syntax):
After proving the correctness, a valid Coq term is constructed for the certified definition of head. This term is a partial Coq derivation where the missing parts (holes) are instantiated with actual proofs, once they have been solved by the user.

```
head =
fun l : { l : list A | l <> [] } ⇒
let filtered_var := `l in
let branch_0 :=
  fun (hd : A) (tl : list A) (Heq_l : hd :: tl = filtered_var) ⇒
  exist (fun a : A ⇒ exists l' : list A, a :: l' = hd :: tl) hd
    (head_obligation_1 l hd tl Heq_l) in
let branch_1 := fun Heq_l : [] = filtered_var ⇒ head_obligation_2 l Heq_l in
match filtered_var as l0
  return (l0 = filtered_var → { a : A | exists l' : list A, a :: l' = l0 })
with
| nil ⇒ branch_1
| hd :: tl ⇒ branch_0 hd tl
end eq_refl
  : forall l : { l : list A | l <> [] },
    { a : A | exists l' : list A, a :: l' = l' }
```

Although the generated term includes the proof components and cause the whole to be less readable, the extracted code of head provides the user with the expected Ocaml code:

```
(**val head : a list → a**)
let head = function
  | Nil → assert false (*absurd case*)
  | Cons (hd, tl) → hd
```

The partiality of function head is obvious from the above definition. In OCAML it is encoded as asserting false as output. The focus here however is to abstract from the generated object to obtain the essential parts of it as a whole (see section [3.5.3.2]).

Example 3.5.2. *Definition of tail in* COQ*:*

Given a non-empty list, the tail function returns the list with the first element removed. Although the function tail bears commonalities with head but due to the fact that there is no partiality involved in its definition, we'll use it as an example here.

```
Program Definition tail (l : list A | l <> nil) :
{l' : list A | exists a : A, a :: l' = l} :=
match l with
| hd :: tl ⇒ tl
| nil ⇒ nil
end.
```

List of remaining Obligations (concrete Coq syntax):
COQ provides us with the following as the remaining proof obligation:

```
forall l : {l : list A | l <> []},
  let filtered_var :=`l
    forall (hd : A) (tl : list A),
      hd :: tl = filtered_var → exists a : A, a :: tl = hd :: tl.
```

Unsolved Obligations representation(raw Coq syntax):
The remaining proof obligation as presented to the user:

```
hd : A
tl : list A
H : hd :: tl <> []
============================
  exists a : A, a :: tl = hd :: tl
```

Emerged obligations:
Because the definition of tail is not recursive. The case distinction technique is applicable here again. There are two cases for which the function produces an output. Expressing the proof obligation in terms of predicate logic yield:

(*To prove case (`l = input):*)

$(\forall$ l : list A,l $\neq$ [ ] $\xrightarrow{`l=input}$ $(\exists$ l' : list A . $(\exists$ a : A, a :: l' = l)))

(*case `l = [] *)
Let's denote *nil* as [ ], then for l = [ ] then the following should hold:

$(\forall\ \mathrm{l} : \mathrm{list\ A}, \mathrm{l} \neq [\,] \xrightarrow{`l=[]} (\exists\ [\,] : \mathrm{list\ A}\ .\ (\exists\ \mathrm{a} : \mathrm{list\ A}, \mathrm{a} :: `(\mathrm{tail}\ ([\,])) = [\,])))$
– note that the assumption $[\,] \neq [\,]$ is contradictory and because "Ex Falso Quodlibet", the proof of this case is trivial.

$(case `l = hd :: tl)$

$(\forall\ \mathrm{l} : \mathrm{list\ A}, \mathrm{l} \neq [\,].$
$(\forall\ \mathrm{tl} : \mathrm{list\ A}, \mathrm{hd} : \mathrm{A} \xrightarrow{`l=hd::tl} (\exists\ \mathrm{l'} : \mathrm{list\ A}\ .\ (\exists\ \mathrm{a} : \mathrm{A}, \mathrm{a} :: `(\mathrm{tail}\ (\mathrm{hd} :: \mathrm{tl})) = \mathrm{hd} :: \mathrm{tl}))))$

– note that this can be simplified using the definition of tail to:

$(\forall\ \mathrm{l} : \mathrm{list\ A}, \mathrm{l} \neq [\,].$
$(\forall\ \mathrm{tl} : \mathrm{list\ A}, \mathrm{hd} : \mathrm{A} \xrightarrow{`l=hd::tl} (\exists\ \mathrm{l'} : \mathrm{list\ A}\ .\ (\exists\ \mathrm{a} : \mathrm{A}, \mathrm{a} :: \mathrm{tl} = \mathrm{hd} :: \mathrm{tl}))))$

Which is again as the same as the generated obligation by Coq.

The generated tail-term (concrete Coq syntax):
After proving the correctness, a valid Coq term is constructed for the certified definition of tail. This term is a partial Coq derivation where the missing parts (holes) are instantiated with actual proofs, once they have been solved by the user.

```
tail =
fun l : { l : list A | l <> [] } =>
let filtered_var :=`l in
let branch_0 :=
  fun (hd : A) (tl : list A) (Heq_l : hd :: tl = filtered_var) =>
  exist (fun l' : list A => exists a : A, a :: l' = hd :: tl) tl
    (tail_obligation_1 l hd tl Heq_l) in
let branch_1 :=
  fun Heq_l : [] = filtered_var =>
  exist (fun l' : list A => exists a : A, a :: l' = [])
    [] (tail_obligation_2 l Heq_l) in
match filtered_var as l0
  return (l0 = filtered_var -> { l' : list A | exists a : A, a :: l' = l0 })
with
| nil => branch_1
| hd :: tl => branch_0 hd tl
end eq_refl
  : forall l : { l : list A | l <> [] },
    { l' : list A | exists a : A, a :: l' = l' }
```

It is remarkable that the automated solved proof obligations of the head function and that of the tail function are not solved by the same strategy. While in head, the base case is immediately solved due to the context contradiction, the base case in the tail function is first instantiated with nil where after the contradiction is derived. This can be declared because the head function is a partial function for which the base case doesn't produce a result while for the tail function, the unit element of lists is returned.

head's base case:

```
let branch_1 := fun Heq_l : [] = filtered_var ⇒ head_obligation_2 l Heq_l in
```

tail's base case:

```
let branch_1 :=
  fun Heq_l : [] = filtered_var ⇒
  exist (fun l' : list A ⇒ exists a : A, a :: l' = [])
    [] (tail_obligation_2 l Heq_l) in
```

Nevertheless, both approaches amount to the same result (due to the contradictory assumption).

Having the examples of head and tail, now we are going to look at a more interesting case where the function definition is given using recursion.

Example 3.5.3. *The definition of eltOf in* Coq*:*

Given a list of integers and an integer number, eltOf determines whether that integer occurs in the list or not.

```
Program Fixpoint eltOf (l : list nat) (n : nat) :
{ b : bool | ((b = true) ↔ (In n l)) } :=
match l with
| nil ⇒ false
| (x :: xs) ⇒ (beq_nat x n)||(eltOf xs n)
end.
```

Where the definition of *In* from Coq's Standard is given as follows:

```
Fixpoint In (a : A) (l : list) { struct l } : Prop :=
match l with
| nil ⇒ False
| b :: m ⇒ b = a ∨ In a m
end.
```

List of remaining Obligations (concrete Coq syntax):
Coq provides us with the following remaining obligations:

```
Obligation 1 of eltOf :
forall (l : list nat) (n : nat), [] = l → (false = true ↔ In n []).
```

```
Obligation 2 of eltOf :
forall (eltOf : fix_proto
  (forall (l : list nat) (n : nat), { b : bool | b = true ↔ In n l }))
    (l : list nat) (n x : nat) (xs : list nat),
      x :: xs = l → ((beq_nat x n || (eltOf xs n)) = true) ↔
        In n (x :: xs)).
```

Where the definition of fix_proto which stands for the polymorphic identity function is given as:

$$fix\_proto = fun\ (A : Type)\ (a : A) \Rightarrow a$$
$$: forall\ A : Type, A \to A$$

The *eltOf* is of type fix_proto in *Obligation 2 of eltOf* . The reason to this will be explained later on.

Unsolved Obligations representation(raw Coq syntax):
The remaining obligations as presented to the user are:

$$n : nat$$
$$==========$$
$$false = true \leftrightarrow False$$

Note that $false : bool$ and $False : Prop$.

$$n : nat$$
$$x : nat$$
$$xs : list\ nat$$
$$x0 : bool$$
$$i : x0 = true \leftrightarrow In\ n\ xs$$
$$==========$$
$$beq\_nat\ x\ n || x0 = true \leftrightarrow x = n \vee In\ n\ xs$$

In this example, the proof obligations for both the base case and the induction step are generated. The reason is clearly that the function eltOf has a recursive definition.

Emerged obligations:
The definition of eltOf is recursive and hence the proof can be established using induction. The proof obligation is expressed as follows in terms of predicate logic:

(*to prove case (l = input)*:)

$(\forall$ l : list nat, n : nat $\xrightarrow{l=input}$ $(\exists$ b : bool , (b = true) $\leftrightarrow$ (In n input)))

(IB) assume $l = []$ than the following should hold:

$\forall$ l : list nat, n : nat $\xrightarrow{l=[]}$ $(\exists$ b : bool , (*grave* (eltOf [ ] n) = true) $\leftrightarrow$ (In n [ ]))
– which simplifies to:
$\forall$ l : list nat, n : nat $\xrightarrow{l=[]}$ $(\exists$ b : bool , (false = true) $\leftrightarrow$ (In n [ ]))
– note that this is equivalent to Obligation 1 of eltOf.

(IH) assume it holds for $(l = xs)$
(IS) prove that it holds for $(l = x :: xs)$ where $(x : nat)$:

(IH) $\forall$ l : list nat, n : nat $\xrightarrow{l=xs}$ $(\exists$ b : bool , (`(eltOf xs n) = true) $\leftrightarrow$ (In n xs))
$\rightarrow$
(IS) $\forall$ l : list nat, n : nat $\xrightarrow{l=x::xs}$ $(\exists$ b : bool , (`(eltOf (x ::xs) n) = true) $\leftrightarrow$ (In n (x :: xs)))

– note that $(`(eltOf\ (x :: xs)\ n) = true)$ can be unfolded to $(beq\_nat\ x\ n\ \|\ `(eltOf\ xs\ n) = true)$. The latter because in order to determine whether n is an element of the list $(x : xs)$, one needs

to check whether it is equal to the head of the list and otherwise the recursion should go further with the rest of the list. Hence the induction step can be rewritten as follows:

(IH) $\forall$ l : list nat, n : nat $\xrightarrow{l=xs}$ ($\exists$ b : bool , (`(eltOf xs n) = true) $\leftrightarrow$ (In n (x :: xs)))
$\rightarrow$
(IS) $\forall$ l : list nat, n : nat
$\xrightarrow{l=x::xs}$ ($\exists$ b : bool , ((beq_nat x n || (`(eltOf xs n) = true)) $\leftrightarrow$ (In n (x ::xs))))

The induction step in this case is almost equivalent to Obligation 2 of eltOf. The subtle difference is that in the Coq version the link between IH and IS seems missing (i.e. the fact that l in IH needs to be instantiated with xs in order to make conclusions on $(x :: xs)$ in induction step seems to be missing in Obligation 2 of eltOf as provided by Coq). This is however hidden in the fact that $`(eltOf\ xs\ n)$ is equal to $b$ for which the following holds: $(b = true \leftrightarrow In\ n\ xs)$ which provides us with IH for xs. More precisely, $eltOf$ is defined to be of the type $fix\_proto$ which as described earlier stands for the polymorphic identity function. Calling the $`(eltOf\ xs\ n)$ causes this fix_proto to be called with the list $(l = xs)$ and the integer n. The result of this call is a boolean b which is exactly its input value namely $`(eltOf\ xs\ n)$ because the function $fix\_proto$ is a polymorphic function which returns its input. The returned value fulfills hence the following property: $(`(eltOf\ xs\ n) = true \leftrightarrow In\ n\ xs)$.

In other words the expression $eltOf\ xs\ n$ results in the instantiation of the type

$$(forall\ (l : list\ nat)\ (n : nat), \{\, b : bool \mid b = true \leftrightarrow In\ n\ l \,\})$$

with $l = xs$ and hence the desired IH is obtained:

$$(forall\ (xs : list\ nat)\ (n : nat), \{\, `\ (eltOf\ xs\ n : bool) \mid `\ (eltOf\ xs\ n) = true \leftrightarrow In\ n\ l \,\})$$

The generated eltOf-term (concrete Coq syntax):
After proving the correctness, a valid Coq term is constructed for the certified definition of eltOf. This term is a partial Coq derivation where the missing parts (holes) are instantiated with actual proofs, once they have been solved by the user.

```
eltOf =
fix eltOf (l : list nat) (n : nat) {struct l} :
  { b : bool | b = true <-> In n l } :=
  match l as l0 return (l0 = l -> { b : bool | b = true <-> In n l0 }) with
  | nil =>
    fun Heq_l : [] = l =>
    exist (fun b : bool => b = true <-> In n []) false
      (eltOf_obligation_1 l n Heq_l)
  | x :: xs =>
    fun Heq_l : x :: xs = l =>
    exist (fun b : bool => b = true <-> In n (x :: xs))
      (beq_nat x n || `(eltOf xs n)) (eltOf_obligation_2 eltOf l n x xs Heq_l)
  end eq_refl
    : forall (l : list nat) (n : nat), { b : bool | b = true <-> In n l }
```

Looking closely to the constructed Coq terms by means of the above examples and comparing them to each other, existing of some common patterns is redolent. By abstracting from the involved details, the underlying frameworks are produced in the following section.

3.5.3.2 General patterns of the generated term

Comparing the resulting COQ terms of the previous section,the common structure among these terms can be extracted. By abstracting away from the let constructions and other details, two patterns can be produced; one in case of a non-recursive definitions and one in case of a recursive definitions. These two patterns are slightly different from each other [1].

Let the $cons_i$ refer to the ith constructor of the inductive type Ind_A and $\vec{x_i}$ to the linear factor of (distinct) variables.

Framework1: Definition The first framework is applicable to non-recursive definitions.

$$
\begin{array}{l}
\mathit{Program\ Definition\ fun\_name}\ (x : \mathit{Ind}_A \mid P_x) : \{\, y : B \mid R\ x\ y \,\} \\
\mathit{match}\ x\ \mathit{with} \\
\quad \mid \mathit{cons}_1\ (\vec{x}_1) \Rightarrow \mathit{Body}_1\ (\vec{x}_1) \\
\quad \mid \mathit{cons}_2\ (\vec{x}_2) \Rightarrow \mathit{Body}_2\ (\vec{x}_2) \\
\qquad \vdots \\
\quad \mid \mathit{cons}_n\ (\vec{x}_n) \Rightarrow \mathit{Body}_n\ (\vec{x}_n) \\
\mathit{end}.
\end{array}
$$

After discharging the proof obligations, the following term can is generated in COQ:

$$
\begin{array}{l}
\mathit{defname} = \\
\mathit{fun}\ x : \{\, x : \mathit{Ind}_A \mid P_x \,\} \Rightarrow \\
\textbf{let}\ \mathit{filtered\_var} := \grave{}x\ \textbf{in} \\
\mathit{match}\ \mathit{filtered\_var}\ \mathit{as}\ z\ \mathit{return}\ (z = \mathit{filtered\_var} \rightarrow \{\, y : B \mid R\ z\ y \,\})\ \mathit{with} \\
\quad \mid \mathit{cons}_1\ (\vec{x}_1) \Rightarrow \mathit{fun}\ H_1 : \mathit{cons}_1\ (\vec{x}_1) = z \Rightarrow (\mathit{elt}\ \_\ \_)\ (\mathit{obligation}_1\ H_1\ (\vec{x}_1)) \\
\quad \mid \mathit{cons}_2\ (\vec{x}_2) \Rightarrow \mathit{fun}\ H_2 : \mathit{cons}_2\ (\vec{x}_2) = z \Rightarrow (\mathit{elt}\ \_\ \_)\ (\mathit{obligation}_2\ H_2\ (\vec{x}_2)) \\
\qquad \vdots \\
\quad \mid \mathit{cons}_n\ (\vec{x}_n) \Rightarrow \mathit{fun}\ H_n : \mathit{cons}_n\ (\vec{x}_n) = z \Rightarrow (\mathit{elt}\ \_\ \_)\ (\mathit{obligation}_n\ H_n\ (\vec{x}_n)) \\
\mathit{end}\ \mathit{eq\_refl} : \\
\quad (\mathit{forall}\ x : \{\, x : \mathit{Ind}_A \mid P_x \,\}, \{\, y : B \mid R\ x\ y \,\}).
\end{array}
$$

Where $(elt\ \_\ \_)$ is defined as follows for each constructor $cons_i$ $(i = 0, ..., n)$:

$$
\begin{array}{l}
\mathit{elt}\ \_\ \_ \equiv\ \mathit{exist}\ (\mathit{fun}\ y : B \Rightarrow R\ (\mathit{cons}_i\ (\vec{x}_i), y)\ (\mathit{Body\_i}\ (\vec{x}_i))) : \{\, x : A \mid P\ x \,\} \\
\mathit{exist} : \Pi\ x : A.\, \Pi\ P : A \rightarrow \mathit{Prop}.\, \Pi\ q : P\ x.\, \{\, x : A \mid P\ x \,\}
\end{array}
$$

Note that the two arguments of *elt* are implicit arguments.
The free variable $filtered\_var$ is defined as the first projection of the input after which proper equalities are provided in the context of proof obligations inside the branches of the match construct.
Furthermore, once again because the type of the right hand sides is not the same everywhere, the elimination predicate needs to be specified explicitly. The *return* construct is used to produce the desired right hand side for each branch. For the i^{th} branch the type of the right hand side can be expressed as:

[1] In section 3.1.2; p.52 of his thesis [Soz08] , Sozeau introduces an interpretation function which interprets the RUSSELL terms to CIC. This interpretation function is however beyond the scope of this thesis. Here, the interpretation step is carried out by reverse engineering i.e. by recognizing the patterns in the created terms from the aforementioned examples.

$$: cons_i\ (\vec{x}_i) = filtered\_var \rightarrow \{\, y : B \mid R\ z\ y \,\}$$

As explained earlier, the match construct is provided with an extra argument (eq_refl). This results in the desired type after eliminating the equalities from the type

$$x = `x \rightarrow \{\, y : B \mid R\ x\ y \,\} \rightsquigarrow \{\, y : B \mid R\ x\ y \,\}$$

Framework2: Fixpoint The second framework is applicable to recursive definitions.

$$\begin{array}{l}
Program\ Fixpoint\ fix_name\ (x : Ind_A \mid P_x) : \{\, y : B \mid R\ x\ y \,\} \\
match\ x\ with \\
\quad \mid cons_1\ (\vec{x}_1) \Rightarrow Body_1\ (\vec{x}_1) \\
\quad \mid cons_2\ (\vec{x}_2) \Rightarrow Body_2\ (\vec{x}_2) \\
\quad\ \vdots \\
\quad \mid cons_n\ (\vec{x}_n) \Rightarrow Body_n\ (\vec{x}_n) \\
end.
\end{array}$$

After discharging the proof obligations, the following term is generated in COQ:

$$\begin{array}{l}
fix_name = \\
fix\ fix_name\ \{x : Ind_A \mid P_x\}\ \{struct\ x\} : \\
\{\, y : B \mid R\ x\ y \,\} := \\
\textbf{let}\ filtered_var := `x\ \textbf{in} \\
match\ filtered_var\ as\ z\ return\ (z = filtered_var \rightarrow \{\, y : B \mid R\ z\ y \,\})\ with \\
\quad \mid cons_1\ (\vec{x}_1) \Rightarrow fun\ H_1 : cons_1\ (\vec{x}_1) = z \Rightarrow (elt\ _\ _)\ (obligation_1\ H_1\ (\vec{x}_1)) \\
\quad \mid cons_2\ (\vec{x}_2) \Rightarrow fun\ H_2 : cons_2\ (\vec{x}_2) = z \Rightarrow (elt\ _\ _)\ (obligation_2\ H_2\ (\vec{x}_2)) \\
\quad\ \vdots \\
\quad \mid cons_n\ (\vec{x}_n) \Rightarrow fun\ H_n : cons_n\ (\vec{x}_n) = z \Rightarrow (elt\ _\ _)\ (obligation_n\ H_n\ (\vec{x}_n)) \\
end\ eq_refl\ (forall\ x : \{x : Ind_A \mid P_x\}, \{\, y : B \mid R\ x\ y \,\})
\end{array}$$

The subtle difference between these two patterns is that in case of a recursive definition, the structural induction is made explicit by means of the *struct* construct and the keyword fix. In this case the proof is given by means of induction on the inductive type whilst in the other case the proof is provided using case distinction as the proof strategy.

3.5.3.3 The Running Examples

The RUSSELL framework provides the possibility to give the function definition along with its specifications. In COQ these specifications are given as the result type of the function. Furthermore, the partiality of a function can be expressed by defining restrictions on the input arguments of the function.

Running Example 1: log2
The *log2* function can be encoded in an straightforward manner. The restrictions on the function's domain are encoded directly by means of a predicate $(n <> 0)$. Furthermore, the relation between the input and output of the function can be formulated as a predicate on the result type of the function:
$((two\_power\, n' \leqslant n < two\_power\, (S\, n')))$.

The function definition is given in COQ syntax as follows:

```
Program Fixpoint log2 (n : nat | n <> 0) {measure n} :
{n' : nat | (two_power n' ⩽ n < two_power (S n'))} :=
match n with
| 0 ⇒ _
| S 0 ⇒ 0
| S (S p) ⇒ S (log2 (div2 (S (S p))))
end.
```

Where *two_power* is defined as:

```
Fixpoint two_power (n : nat) : nat :=
match n with
| 0 ⇒ 1
| S p ⇒ 2 * two_power p
end.
```

This definition results in 5 proof obligations from which 3 are resolved automatically and 2 are displayed to the user. The remaining obligations after some simplification steps is:

```
log2_obligation_4 =
p : nat
H : S (S p) <> 0
log2 : forall n : {n : nat | n <> 0},
  `n < S (S p) →
  {n' : nat | two_power n' ⩽`n < two_power n' + (two_power n' + 0)}
============================
  S (div2 p) < S (S p)
```

Notice the wrinkle in the ointment: $(`n)$ means the first projection from the decorated subset type $\{n : nat \mid n <> 0\}$. Discharging this obligation results in a proof that the recursive call is decreasing which is not obvious in case of general recursion.

```
log2_obligation_5 =
p : nat
H : S (S p) <> 0
log2 : forall n : {n : nat | n <> 0},
  `n < S (S p) →
```

```
  { n' : nat | two_power n' <=` n < two_power n' + (two_power n' + 0) }
============================
  forall x : nat,
  two_power x <= S (div2 p) < two_power x + (two_power x + 0) ->
  two_power x + (two_power x + 0) <= S (S p) <
  two_power x + (two_power x + 0) + (two_power x + (two_power x + 0) + 0)
```

The last obligation is devoted to proving the property of the function in case of $(S(S\,p))$ For the proof script of these obligations see Appendix [D].

Running Example 2: twoPowN
The *twoPowN* function can be encoded in an straightforward manner.

```
Program Fixpoint twoPowN (n : nat) { measure n } :
{ n' : nat | 0 < n' } :=
match n with
| 0 => 1
| S p => if (even_odd_dec p) then 2 * (twoPowN p) else (square (twoPowN (div2 (S p))))
end.
```

This definition results in 6 proof obligations from which 3 is resolved automatically and 3 is displayed to the user. The remaining obligation after some simplification steps is:

```
p : nat
H : even p
twoPowN : forall n : nat, n < S p -> { n' : nat | 0 < n' }
============================
  0 <`(twoPowN p (le_n (S p))) + (`(twoPowN p (le_n (S p))) + 0)
```

Discharging this obligation results in a proof that the result of second branch is always greater than 0.

The next obligation is:

```
p : nat
  H : odd p
  twoPowN : forall n : nat, n < S p -> { n' : nat | 0 < n' }
  ============================
    0 <
    square
      (`(twoPowN match p with
          | 0 => 0
          | S n' => S (div2 n')
         end (twoPowN_obligation_4 (S p) twoPowN p eq_refl H)))
```

Which when resolved results in a proof that the result of third branch is always greater than 0.

```
p : nat
H : odd p
twoPowN : forall n : nat, n < S p -> { n' : nat | 0 < n' }
============================
  forall n : nat, even n -> S (div2 n) < S (S n)
```

Discharging this obligation results in a proof that the recursive call is decreasing which is not obvious in case of general recursion.

Chapter 4

Experimental Comparison of the Methods

4.1 Introduction

In chapter [3] current methods in the literature have been discussed and illustrated by means of the running examples. One interesting aspect to study about these methods is their mutual differences. As introduced in chapter [2], two main categories can be distinguished:

category 1: give the fully specified function i.e. define the function f such that it satisfies the following specification:

$$f : \Pi\, x :\ A.Px \ \rightarrow\ \{y\ :\ B, R(x, y)\}$$

category 2: separate function specification and properties i.e. define the function f of type $f : A \rightarrow B$ and then prove that it satisfies the following property:

$$\forall\, x :\ A\,.\,P\,x \Rightarrow\ R(x, (f\ x))$$

In this chapter, these methods are compared with each other in two ways: first a comparison based on the two categories is made and afterwards a methodological comparison is carried out based on the four major criteria and the and the emerged questions (see table [4.1]).

Function definition	RQ1:	How difficult is it to write the definition of the function?
	RQ2:	How close is it to the original functional program expressed in a practical functional programming language , where there is no restriction on recursive calls?
Function properties	RQ3:	How difficult is it to formulate and prove the properties of the functions?
Program extraction	RQ4:	Is the extracted code in accordance with our expectation?
Function executability	RQ5:	Can one execute the code in Coq sufficiently well in terms of the efficiency?

Table 4.1: The comparison criteria

Roughly speaking, these approaches differ from each other in the way they deal with general recursion and partiality. While some methods like the ad-hoc predicate method use the encoding of domains, other methods like the converging iteration method cover a larger class of terminating functions, mainly using well-founded relations.

Setup Specifications: All experiments of this thesis are carried out on a workstation consisting of four Intel ® Core2 ™CPU Q6600 processors running @ 2.4GHz and 3.7 GiB of RAM memory, running a 64-bit Linux distribution using kernel version 2.6.32.11-99.

All function definitions and their corresponding correctness proof are encoded in *ProofWeb*[1] which provides an interface for using different proof assistants, among which Coq, with underlying the most recent version of Coq from the repository: Coq trunk (12803).

4.2 Comparison of the two Categories

In chapter [3] the introduced methods are categorized. Accordingly, the Russell framework belongs to the first category while the Ad-hoc predicate method, the converging iteration method and the Function framework belong to the second category.

The most important benefit of the first category of approaches is the fact that all required ingredients including the function definition and the property to which it should adhere to are presented to the user which offers a clear overview of the function as a whole. Using the properties as types paradigm, one can specify the properties by an expressive subset types. Reasoning is then carried out from input-output correspondence. In the second category of approaches the user has solely the function definition available and all properties of the function need to be formulated and proved separately.

Generally speaking, while combining the function definition and its corresponding properties is an advantage of the first category, it can be its disadvantage at the same time in the sense that one need to anticipate all properties that a function desired to have at once by providing it with an expressive enough type. In nontrivial developments, it might be the case that there are relevant properties of the function, not necessarily known in advance, which the user needs to prove as intermediate lemmas. The separation of concerns offered by the second approach can be advantage for these kind of uncertainties about the completeness of the properties. The incomplete formulation of properties in the first case would require the user to re-define the function according to each new property, in addition to re-proving all the obligations which are previously proved.

However, the Russell framework which belongs to this first category doesn't suffer from this inconvenience. Namely, after the function is added as a certified object to the environment, one can still formulate lemmas stating additional properties about the function and prove them accordingly. As an illustration, we introduce an additional lemma for the first running example which states that the logarithmic function (in this case of base 2) is less than the linear function, for all natural numbers. The *log*2 funcion was defined as follows in the Russell framework:

$$
\begin{array}{l}
\mathit{Program\ Fixpoint\ log2}\ (n : \mathit{nat} \mid n <> 0)\ \{\mathit{measure}\ n\} : \\
\{n' : \mathit{nat} \mid ((\mathit{two\_power}\ n' \leqslant n) \wedge (n < \mathit{two\_power}\ (S\ n')))\} := \\
\mathit{match}\ n\ \mathit{with} \\
\mid 0 \Rightarrow \_ \\
\mid S\ 0 \Rightarrow 0 \\
\mid S\ (S\ p) \Rightarrow S\ (\mathit{log2}\ (\mathit{div2}\ (S\ (S\ p)))) \\
\mathit{end}.
\end{array}
$$

[1] http://proofweb.cs.ru.nl/login.php

The additional lemma can be formulated as follows:

$$Lemma\ log2\_lt : (forall\ (n : nat)\ (domain : n <> 0), (proj1\_sig\ (log2\ (exist\ \_\ n\ domain))) < n).$$

As one may observe, the input to $log2$ needs to be of the subset type. Using *exist* one can construct this subset type. Furthermore, in order to use the ordering $(<)$ both operands need to be of the type *nat*. This is achieved by using the first projection of the $log2$ function denoted as $proj1\_sig$.

The second main objection to the first category of approaches is the so called *proof carrying code* which means that the definition of the function will be mixed with its correctness proof, in particular with the termination proof statements. This is obviously an undesirable combination since it goes at the expense of expressiveness and readability of the code while it doesn't have any added value for the user. The user actually needs a certified program code. How the proof is established, is not of his concern. The RUSSELL framework is capable of dealing with this issue as well. By postponing the generation of proof obligations, the framework works around this issue and allows the user to write only the computational part of their program without requiring the proof statements during the program construction. These required proof statements will be added at a later stage when the proof obligations are discharged by the user.

Furthermore, the resulted object obtained this way is once again polluted with the undesired proof statements. Fortunately, the powerful extraction mechanism of COQ provides us eventually with the desired code without the proof statements involved i.e. since the proof statements are seen as living in the `Prop` sort, they will not appear in the final extracted code and hence the user has access to the desired certified code.

4.3 Methodological Comparison

In order to compare the methods against the criteria [4.1], each research question is investigated on all methods.

4.3.1 Function definition

Starting with the function definition, in this section the following questions are considered:

RQ1: How difficult is it to write the definition of the function?
RQ2: How close is the function definition to the original functional program expressed in, a practical functional programming language , where there is no restriction on recursive calls?

Converging Iteration Method

Giving a function definition is not straightforward by means of the converging iteration method.

First of all the user is required to define and prove a theorem on well-foundedness of the recursion. This theorem is used in the computation part of the function, where one needs to prove that the argument to the recursive call decreases. The definition is then given in two phases. The first phase of the definition is responsible for the computation part (which involves most of the work) while the second phase of it regulates the recursion by means of a standard function called $well\_founded\_induction$. Moreover, the computational part of the function definition should be provided with the appropriate return type as mentioned before. Formulating such a return type is generally a difficult task which requires advanced COQ knowledge.

We can conclude that defining a general recursive and/or a partial function in Coq by means of Converging iteration method needs a lot of expertise. In fact, the preparation in this case is limited to defining and proving the well-foundedness of the recursion. One other difficulty which is involved with defining functions in this method is the fact that the prepared proof needs to be used in the body of the computational part which makes the construction extra difficult and the resulted code less readable.

The final function definition in this method deviates totally from the version which one would have written down in a practical programming language. The presence of the theorem in the body as well as the division of the definition in two phases are the reasons of this deviation.

The Ad-hoc Predicate Method

For this method [3.3], a lot of preparation is required before one is actually able to provide the actual function definition.
In both approaches, the original method as introduced by Bove and Capretta as well as the adapted version as introduced by Bertot and Castéran, the user needs to come up with an inductive domain definition. Using this inductive domain the function definition is constructed. While the original method faces unexpected effects with the extracted code, as explained later on in section [4.3.3], the adapted version requires the user to define and prove the so called *partiality theorem* as well as the *inversion theorems*. Whether or not one is able to give the function definition subsequently, depends on the way that the correctness of these theorems has been proven which is in unconcealed violation with the principle of proof irrelevance. These theorems are used afterwards as proofs in the body of the function definition. Furthermore, the function definition should be provided with the appropriate return type because the right hand sides don't have the same type everywhere. Formulating the appropriate return type is not a trivial task and requires advanced knowledge of Coq.

Consequently, defining a general recursive and/or a partial function in Coq by means of Ad-hoc predicate method needs a lot of expertise and involves lots of preparation steps. Furthermore, the proofs need to be used in the body of the function definition. Like in the previous method this results in difficult constructable code and less readable outcome.

On top of the fact that the proof of the theorems need to be embedded in the body of the function definition, the final definition is also far from the version one would have written down in a practical functional programming language.

Function Framework

Defining a general recursive function is straightforward by means of the Function framework. The only extra ingredient which is required is a measure function. In case of general recursion, this measure function generates proof obligations which can be discharged by the user later on in order to prove the termination.

$$
\begin{array}{l}
\mathit{Function}\ F\ (x : A)\ \{\, \mathit{measure}\ x \,\} : B \\
\mathit{match}\ x\ \mathit{with} \\
\quad \mid \mathit{cons}_1\ (\vec{x}_1) \Rightarrow \mathit{Body}_1\ (\vec{x}_1) \\
\quad \mid \mathit{cons}_2\ (\vec{x}_2) \Rightarrow \mathit{Body}_2\ (\vec{x}_2) \\
\quad\ \vdots \\
\quad \mid \mathit{cons}_n\ (\vec{x}_n) \Rightarrow \mathit{Body}_n\ (\vec{x}_n) \\
\mathit{end}.
\end{array}
$$

Since the only extra ingredient to the function definition is the presence of the measure function, the final function definition is very close to the version which one would think of in a practical functional programming language.

However, the most important drawback to this method is the fact that it is incapable of dealing with dependent types and hence unable to deal with the partial functions.

Russell Framework

Finally, the Russell framework provides the user with the ability to define the functions as easily as in a practical programming language whilst giving them a specification as rich as desired.

$$
\begin{array}{l}
\mathit{Program\ Definition\ /\ Fixpoint}\ F\ (x : \mathit{Ind}_A \mid P_x)\ \{\, \mathit{measure}\ x \,\} : \{\, y : B \mid R\ x\ y \,\} \\
\mathit{match}\ x\ \mathit{with} \\
\quad \mid \mathit{cons}_1\ (\vec{x}_1) \Rightarrow \mathit{Body}_1\ (\vec{x}_1) \\
\quad \mid \mathit{cons}_2\ (\vec{x}_2) \Rightarrow \mathit{Body}_2\ (\vec{x}_2) \\
\quad\ \vdots \\
\quad \mid \mathit{cons}_n\ (\vec{x}_n) \Rightarrow \mathit{Body}_n\ (\vec{x}_n) \\
\mathit{end}.
\end{array}
$$

The definition can be decorated with the specification formulated as a predicate on the output type (R) of the function, relating the input arguments to the output arguments, and the partiality can be encoded as a predicate on the input type of the function P. These restrictions on the input and the output type result in the generation of proof obligations which, if not solved automatically, will be delegated to the user. Furthermore, to deal with general recursion, a measure function can be provided on the argument to the recursive call. After discharging the corresponding proof obligations, this will lead to the termination proof for the corresponding function. Postponing the proof obligations till after the function is defined, provides user with a framework by means of which providing the function definition is straight forward.

Conclusion 4.3.1. *Defining a general recursive function by means of the converging iteration method requires some preparation step because one has to prove the well-foundedness of the recursion. This proof is used in the body of the computational part of the definition and guarantees the termination. For the ad-hoc predicate method, this preparation step is even more involved as one has to provide an inductive domain definition and in addition the partiality and inversion theorems. Furthermore, in both methods the function definition is polluted with the proof statements which causes the definition to be more complicated and less readable. On the contrary, in both the* Function *and the* Russell *framework, the general recursive function can be defined straightforwardly. The ability to include a measure function to guarantee the termination. There is however one drawback to the* Function *framework namely its inability to deal with partial functions. Hence the* Russell *framework is the most suitable and complete framework when it comes to function definition.*

4.3.2 Function properties

Considering the function properties, two aspects are inquired:

RQ3: How difficult is it to formulate and prove the properties of the functions?

Converging Iteration Method

The converging iteration method belongs to the second category of approaches and hence the properties can be formulated and proved after the function definition is provided. These properties should be formulated by means of theorems. One of the theorems for the first running example is formulated as:

Theorem pow_log_gt :
$(forall\ (x : nat), x < two\_power\ (S\ (log2\ x)))$.

The formulation of such a theorem is not a complicated task. Proving it on the other hand requires some more effort and knowledge about the recent Coq tactics. Generally, we can state that the difficulty involved in formulating and proving such theorems is proportional to the difficulty of the function in consideration. The proofs are provided in Appendix [A].

Ad-hoc Predicate Method

The ad-hoc predicate method also belongs to the second category of approaches and hence like the converging iteration method, the properties can be formulated and proved after the function definition is provided. As an illustration, one property regarding the first running example is given:

Theorem pow_log_le :
$(forall\ (x : nat)\ (h : log\_domain\ x), two\_power\ (log2\ x\ h) \leqslant x)$.

As one may observe, the formulation of such a theorem requires a little more attention because the domain of the function plays a role in it (h in this case). Proving it also requires some more effort and knowledge about the recent Coq tactics. The proof of the above theorem in addition to other theorems related to the running examples can be found in Appendix [B].

Function Framework

Due to the fact that the Function framework is incapable of handling partiality, the only remaining issue is general recursiveness and termination. There are two kinds of properties which can be distinguished here. Proving the termination is carried out after discharging the automatically generated proof obligations. The other properties related to the function have to be formulated separately as it is the case in other methods which belong to the second category of approaches. As an illustration one of the properties related to the second running example is provided.

Theorem pow_pos :
$(forall\ (x : nat), 0 < (twoPowN\ x))$.

Again in this case the difficulty with formulating and proving properties is proportional to the function's difficulty. For proofs refer to Appendix [C].

Russell Framework

In the Russell framework, the properties to which the function should adhere to is given as a post-condition, along with the function's definition. It is worth mentioning that the body of such a function definition doesn't need to be filled in neither with proof parts nor with structural constraints. These definitions are fully accepted by Coq and the corresponding proof obligations can be resolved by the user later on. Depending on how complicated the proof obligation is, the proofs have different degree of difficulty. The most important point is that the whole Coq apparatus can be used to prove that the code meets its specification. In Appendix [D] the proofs of the proof obligations regarding the running example are provided.

Furthermore, as mentioned in the previous section [4.2], additional properties can be formulated and proved after the correctness of the given definition is ensured by discharging the proof obligations. However, the formulation of these additional properties are more involved in this case. Reviewing the additional property helps to describe this subtle difficulty better:

$$Lemma\ log2\_lt : (forall\ (n : nat)\ (domain : n <> 0), (proj1\_sig\ (log2\ (exist\ \_\ n\ domain))) < n).$$

Due to the fact that subset types are used to formulate the input and output type of the function, *log*2 is this case, the proper type needs to be constructed by means of *exist* before one is able to invoke upon the function. Similarly, the projection of result type needs to be considered instead of the subset type which the function returns as the result type. However, the fact that the most crucial function property namely its post-condition is formulated fairly easy in addition to the fact that the proof obligations concerning that main properties are generated automatically and in some cases are solved automatically, outweighs the difficulty one has to face in case of formulating the additional interesting properties.

Conclusion 4.3.2. *In all the methods, the difficulty involved in the proving crucial properties is proportional to the difficulty of the concerning function. However when it comes to the formulation of the properties, the* Russell *provides the user with a natural way to express these properties as a predicate on the output of the function. The same holds for partiality of a function, i.e. one can restrict the input type by means of a predicate on the input. These abilities are achieved by using the expressive power of subset types. For all other methods, a separate theorem needs to be provided expressing the desired properties to prove. The formulation of this theorem is proportional to the difficulty of the concerning function almost for all these three methods. In case of the Ad-hoc predicate method one has to take care of the proper domain which requires a bit more attention in formulation. One question which may arise here is about the ability of* Russell *for formulating and proving additional properties (except the ones which are included in the input/output types). We have shown that the formulation of additional properties in the* Russell *framework, although more involved, is possible.*

4.3.3 Program extraction

The extracted code is a piece of certified code which can be used in larger developments in an external target language using the `Extraction` tactic. The question is:

RQ4: Is the extracted code in accordance with our expectation?

Converging Iteration Method

There is a difference between the extracted code if we compare the version where partiality is considered and the initial version where it is not considered.

In the case where the partiality is taken into account, the extracted code satisfies the expectations. For the first running example this amounts to:

```
(**val log2_final : nat → nat**)
let rec log2_final = function
  | 0 → assert false (*absurd case*)
  | S n → (match n with
    | 0 → 0
    | S m → S (log2_final (div2 (S (S m)))))
```

Ad-hoc Predicate Method

There is a difference between the extracted code we obtain via the original method and the code which is extracted after the adaptation to CIC. While the latter corresponds exactly to the expected code, the code which is extracted from the original version deviates from the expectation. The reason of this deviation is that the original method is based on a different type theory (Martin-Löf's type theory) than the underlying type theory of COQ (CIC) in the sense that there is no distinction made between the sorts `Set` and `Prop`. This results in an extracted code that deviates from the expectation, simply because the `Set` sort is relevant from the computational point of view while the elements of sort `Prop` are relevant from reasoning point of view and hence are omitted from the extracted code. The COQ's extraction mechanism [2.3.2] carries therefore only the elements of `Set` sort to the extracted code. This is the extracted code from the original version of this method:

```
(**val log2 : nat → log_domain → nat**)
let rec log2 x = function
  | Log_domain_1 → 0
  | Log_domain_2 (p, h1) → S (log2 (S (div2 p)) h1)
```

This is obviously not even close to the expected code. Moreover, a closer look at it brings doubts about its correctness, merely due to the fact that the relation between log_domain and the input argument of the type *nat* is removed in the type information of the extracted code. Fortunately, the result of extraction after adapting the method to the CIC is as expected:

```
(**val log2 : nat → nat**)
let rec log2 = function
  | 0 → assert false (*absurd case*)
  | S n → (match n with
    | 0 → 0
    | S p → S (log2 (S (div2 p))))
```

Function Framework

The extracted code using the FUNCTION is subtly different from the expected code because it involves proof statements. The extracted code of the *twoPowN* example looks as follows at first sight:

```
(**val twoPowN : nat → nat**)
let twoPowN x =
  twoPowN_terminate x
```

It is filled in with the proof of termination. However, the problem could be solved by constant inlining and reduction of *twoPowN_terminate*.

```
Extraction Inline twoPowN_terminate.
Extraction twoPowN.
```

Afterwards, the produced code is exactly the same as the expected one.

```
(**val twoPowN : nat → nat**)
let rec twoPowN = function
  | 0 → S 0
  | S p →
    (match even_odd_dec p with
       | Left → mult (S (S 0)) (twoPowN p)
       | Right → square (twoPowN (div2 (S p))))
```

Russell Framework

There is a subtlety involved when extracting code from the certified code generated by means of the Russell framework. The result of using the standard extraction mechanism of Coq results to a piece of Ocaml code mixed up with first projections of the subset types. The extracted code initially looks different from the expected version:

```
(**val log2 : nat → nat**)
let rec log2 x =
  match proj1_sig x with
    | 0 → assert false (*absurd case*)
    | S n →
      (match n with
         | 0 → 0
         | S p → S (proj1_sig (log2 (proj1_sig (div2 (S (S p)))))))
```

However, this will result in the expected code after using the Inline command which filters out the constant definition of *proj1_sig*.

```
Extraction Inline proj1_sig.
Extraction log2.
```

```
(**val log2 : nat → nat**)
let rec log2 = function
  | 0 → assert false (*absurd case*)
  | S n → (match n with
    | 0 → 0
    | S p → S (log2 (div2 (S (S p)))))
```

Conclusion 4.3.3. *Initially, in none of the cases the generated extracted code is in accordance with the expectation. However, this slight difference can be resolved easily. In the cases of the Ad-hoc predicate method and the Converging iteration method, after the adaptation of these methods the expected code is generated. In cases of the* Function- *and the* Russell *framework the inlining of the definitions results in the expected extracted code. A considerable advantage of the last two*

frameworks over the first two methods is the fact that the extraction of ***certified code*** *is enforced in these two frameworks. In other words, in the* FUNCTION- *and the* RUSSELL *framework the function definition is accepted by the type checker only when all the pending proof obligations have been resolved, which prohibits the extraction of programs for which the termination and/or the correctness has not been proved yet. This is simply caused by the phase distinction between writing the programs and proving the correctness, while the correctness proof is required in order to add the fully certified object to the environment.*

4.3.4 Function executability

In COQ there is a possibility to execute the program using a so called (`Eval Compute in`) tactic which consecutively applies a series of reductions. This way of execution is comparable with interpreting which is by far the most inefficient way of executing. External compilers like OCAML or HASKELL are way more efficient. Hence, depending on the system limits and the executed commands, the execution can address a restricted amount of input values which are obviously different per method. In case one is about to exceed the restriction, the following message will be communicated by the system:

```
Warning: Stack overflow or segmentation fault happens when working with
large numbers in nat (observed threshold may vary from 5000 to 70000
depending on your system limits and on the command executed).
```

In this case the resulting value can be still computed but the warning states clearly that the stack is going to overflow. Once that happens no results would be produced any more and the error message Stack overflow and/or segmentation fault will be shown. Our system specifications are presented in the introduction part of this chapter [4.1].

In order to measure the execution time as well, we use `Time Eval compute in` instead. The executability is tested on each method by trial and error experiments. The maximum achieved value (to 10^2 precision) is provided accordingly. In order to illustrate that COQ is not a suitable medium for computation matters, the function *two_power* as it is implemented in COQ's standard library is compared with our implementation of *twoPowN*. The function *two_power* is defined as follows in the standard library:

$$
\begin{array}{l}
\mathit{Fixpoint\ two\_power\ (m : nat) : nat :=} \\
\quad \mathit{match\ m\ with} \\
\quad \mid 0 \Rightarrow 1 \\
\quad \mid S\ n \Rightarrow \mathit{two} * \mathit{two\_power\ n} \\
\quad \mathit{end}.
\end{array}
$$

The maximum achieved value by means of trial and error experiment for this standard function is 15. The execution and its result is given below:

$$
\begin{array}{l}
\mathit{Time\ Eval\ compute}\ \mathbf{in}\ (\mathit{two\_power}\ 15). \\
\quad = 32768 \\
\quad : \mathit{nat} \\
\mathit{Finished\ transaction}\ \mathbf{in}\ 2.\ \mathit{secs}\ (1.316082\ u, 0.020001\ s)
\end{array}
$$

The question which arise hereby is:

RQ5: Can one execute the code in COQ sufficiently well in terms of the efficiency?

Converging Iteration Method

The actual execution of function is straightforward in this case. There is no extra lemma needed to regulate the execution. For the first running examples, the results are presented for both versions of this method i.e the version introduced by Balaa and Bertot (see section [3.2.3]) and the extension suggested in this thesis where partiality is involved (see section [3.2]).

Running Example 1 : log2

The execution and its result for the first running example are shown below:

Time Eval compute **in** (*log2* 5).
  $= 2$
  $: (fun \_ : nat \Rightarrow nat)\ 5$
Finished transaction **in** 0. *secs* $(0.\ u, 0.\ s)$

In the extended version where partiality is also included, this execution amounts to a slightly different outcome, namely:

Time Eval compute **in** (*log2_final* 5).
  $= fun \_ : 5 = 0 \rightarrow False \Rightarrow 2$
  $: (fun\ y : nat \Rightarrow y <> 0 \rightarrow nat)\ 5$
Finished transaction **in** 0. *secs* $(0.004001\ u, 0.\ s)$

Which stresses that the input element is not zero.

The upper bound to the accepted values is in the order of magnitude of 10^4 in both versions of this method. The maximum achieved result by means of trial and error experiment with this upper bound indicates that the maximum in the range of 34910 is reachable. However, computation of this value takes more time in the version with partiality than the original version. The result of original version for value 34910:

Time Eval compute **in** (*log2* 34910).
  $= 15$
  $: (fun \_ : nat \Rightarrow nat)\ 34910$
Finished transaction **in** 2. *secs* $(2.592162\ u, 0.048003\ s)$

The result for value 34910 when partiality is included:

Time Eval compute **in** (*log2_final* 34910).
  $= fun \_ : 34910 = 0 \rightarrow False \Rightarrow 15$
  $: (fun\ y : nat \Rightarrow y <> 0 \rightarrow nat)\ 34910$
Finished transaction **in** 4. *secs* $(3.868241\ u, 0.036002\ s)$

Considering the definition of these two versions, the difference can be justified as the computational part of the partial version is more involved.

Running Example 2 : twoPow2

For the second running example, execution has resulted to the following:

Time Eval compute **in** (*twoPowN* 15).
  = 32768
  : (*fun* _ : *nat* ⇒ *nat*) 15
Finished transaction **in** 0. *secs* (0.556035 *u*, 0.008001 *s*)

This is clearly faster than the function *two_power* from the standard library. The maximum achieved value is in this case also 15.

Ad-hoc Predicate Method

In order to execute the defined function in terms of the Ad-hoc predicate method, an auxiliary lemma is required which proves that the input elements in the domain of the function. The correctness proof of this lemma is provided by using the constructors of the domain. For the elements which are not part of the domain such a proof can't be provided simply because there are no constructors for such elements present in the domain definition. To illustrate this, the execution of first running example on input value is given for both versions of this method i.e as introduced by Bove and Capretta (see section [3.3]) and its extension to Cic as suggested by Bertot and Castéran (see section [3.3.3.2]). Furthermore the result of execution of the second running example for this method is compared with the one which is produced by the *two_power* from the standard library.

Running Example 1 : log2

Lemma auxiliary : *log_domain* 5.
repeat constructor.
Defined.

Having this lemma the execution and its result are shown below.

Time Eval compute **in** (*log2* 5 *auxiliary*).
  = 2
  : *nat*
Finished transaction **in** 0. *secs* (0. *u*, 0. *s*)

The upper bound to the accepted values is in the order of magnitude of 10^4. In order to measure the consumed time, we have to add up the time which is required to prove the lemma with the required time for doing the actual computation. For the input value 5 the result is produced instantly. The maximum achieved result by means of trial and error experiment with this upper bound indicates that the maximum in the range of 34920 is reachable. By means of the first version of this method, proving the auxilary lemma takes 41 minutes and 62 seconds. The actual computation goes fairly fast in this version. Ofcourse we need first to prove that 15 is in the domain. That goes in exact same way as for 5 thus we give only the execution and its result here:

Time Eval compute **in** (*log2* 15 *auxiliary*).
  = 15
  : *nat*
Finished transaction **in** 2. *secs* (2.368148 *u*, 0. *s*)

In total takes the computation of 34920 about 43 minutes and 98 seconds in this version.

Because the definition of domain is not changed in the second version (example [3.3.3.2]), proving the lemma consumes the same amount of time. The actual computation goes slightly slower than the previous version:

Time Eval compute **in** (*log2* 15 *auxiliary*).
= 15
: *nat*
Finished transaction **in** 2. *secs* (2.592162 u, 0. s)

In total takes the computation of 34920 about 44 minutes and 21 seconds in this version which is 23 seconds more than in the previous version of this method.

The most time consuming part of the computation in both versions is the *repeat constructor* command which goes through the domain constructors 34919 times.

Running Example 2 : twoPow2

For the second running example, execution has resulted to the following:

Time Eval compute **in** (*twoPowN* 15 *aux*).
= 32768
: *nat*
Finished transaction **in** 0. *secs* (0.560035 u, 0.004 s)

Which is again faster than the *two_power* function. The maximum reached value is 15 as well. There is however a subtility involved in proving the correctness of the *aux* lemma in this case. One needs to choose the constructor correctly in order to prove the correctness of this lemma. The simple tactic of `repeat constructor` can't be used here (see Appendix [A]).

For the second version of this method, the execution takes more time as was the case in the first running example.

Time Eval compute **in** (*twoPowN* 15 *aux*).
= 32768
: *nat*
Finished transaction **in** 1. *secs* (0.520033 u, 0. s)

Nevertheless, the time consumption is less than in the *two_power* function. The maximum reached value is also in this case 15.

Function Framework

Given the fact that the FUCNTION framework is incapable of dealing with partiality, the correct encoding of *log*2 is not possible in this framework. However, if in order to give an indication of how fast this method operates, the hacked version of this function is encoded in the FUNCTION framework i.e. it returns 0 for input value 0.

Running Example 1 : log2

The execution of the hacked version of *log*2 in this framework goes quite fast.

Time Eval compute **in** (*log2* 34920).
  = 15
  : *nat*
Finished transaction **in** 3. *secs* (3.04019 u, 0.028002 s)

The maximum of 34920 is reached in about 3 seconds.

Running Example 2 : twoPow2

The encoding of second running example is straightforward. The execution and the achieved result are as follows:

Time Eval compute **in** (*twoPowN* 5).
  = 32
  : *nat*
Finished transaction **in** 0. *secs* (0. u, 0. s)

The maximum accessible input value for this particular function is 15. For greater values stack overflow occurs.

Time Eval compute **in** (*twoPowN* 15).
  = 32768
  : *nat*
Finished transaction **in** 0. *secs* (0.584037 u, 0.016001 s)

The maximum value of 15 is achieved instantly also in this case.

Russell Framework

In order to execute the defined function in course of the Russell framework, an auxiliary lemma needs to be proved. This lemma has to prove that the precondition predicate P holds for the input parameter. Furthermore, as the input element is of a sigma type in case of partial functions, such a term needs to be created by means of *exist* from the input term and the proof that it satisfies the predicate. For complete functions, the formulation is straightforward as we'll see in the second running example.

Running Example 1 : log2

For the first running example with input value 5 this amounts to:

Lemma aux : 5 <> 0.
auto.
Defined.

As one can observe, proving this lemma is, unlike the one in the ad-hoc predicate method, straightforward. Although for large values even this simple proof consumes time.

Furthermore, as the input element is of a sigma type, such a sigma term needs to be created by means of *exist* as the input term of the function and the first projection of that term which is simply a *nat* needs to be used in the computation:

Eval compute **in** (*proj1_sig* (*log2* (*exist* _ 5 *aux*))).

The upper bound to the accepted values is in the order of magnitude of 10^4. The maximum achieved result by means of trial and error experiment with this upper bound indicates that the maximum in the range of 32760 is reachable. Again in this case the time which is consumed to proof de lemma is added up with the time required for the actual computation. The proving of this lemma consumes 7 minutes and 27 seconds. Hereby a notable phenomena happens which I couldn't justify. The actual proof is carried out by `auto` tactic which takes only about 3 seconds. The rest of the consumed time is used by `Defined` tactic. The actual computation, takes also a while:

Eval compute **in** (*proj1_sig* (*log2* (*exist* _ 14 *aux*)))
= 14
: *nat*
Finished transaction **in** 97. *secs* (96.306019 *u*, 0.112007 *s*)

In total the execution of *log*2 consumes 8 minutes and 14 seconds. Because both ad-hoc predicate method and Russell requiring proving a lemma before the actual execution happens, we can compare them. Russell is in that sense faster than both versions of the ad-hoc predicate method.

Running Example 2 : twoPow2

For the second running example, we achieve 15 as the maximum value in less time that the version from standard library does:

Time Eval compute **in** *twoPowN* 15.
= 32768
: *nat*
Finished transaction **in** 1. *secs* (0.552034 *u*, 0.024001 *s*)

The time consumption is also in this case less than the in the original version *two_power*.

Table [4.1] summarizes the achieved results from executions of the running examples on methods.[2]

Methods	log2		twoPowN	
	Maximum	Time	Maximum	Time
CIM1	34910	0,2	15	0
CIM2	34910	0,4	15	0
Ad-hocPM1	34920	43.98	15	0
Ad-hocPM2	34920	44,21	15	0.1
Function	34920	0,3	15	0
Russell	32760	8,14	15	0.1

Figure 4.1: Overview of execution results

Conclusion 4.3.4. *When introducing the second running example, twoPowN, we have justified the choice for its implementation by its efficiency. Comparing the execution result of the standard function two_power with the execution result of twoPowN confirms our claim. In all methods, the function twoPowN returns in all the methods instantly (except for* Russell *and the extended ad-hoc predicate method where it takes 1 sec.) While the execubility is achieved by all methods, the converging iteration method stands out in efficiency. By means of ad-hoc predicate method greater values are reachable but in terms of efficiency, this method is the worst one.*

[2]The name of the methods are abbreviated.

Chapter 5

Concluding remarks

5.1 Introduction

The objective of this thesis was to investigate two categories of approaches (4.1, 4.1) by means of which the general recursion, termination and partiality issues can be resolved in the Coq proof assistant and its underlying type theory pCic. This investigation has resulted in a clear picture of the state of the art in methods which can deal with general recursion and/or partiality in Coq. In this chapter, the methods under consideration are briefly explained with regard to their development in section [5.2]. Section [5.3] summarizes the conclusions of this thesis. Finally, in section [5.4] some suggestions are presented for the future work.

5.2 Summary - Methods and their Development

Considering the development progress in course of the studied methods, one can recognize a clear hierarchy in how the development has advanced (see figure [5.1]).

Starting from the work of Bengt Nordström in 1988 [Nor88] to deal with the general recursion in Martin-Löf's type theory, Balaa and Bertot [BB00] tried to embed the idea in Coq using *well_founded_induction* in 2000. In 2002 Bove and Capretta [BC05] came with the idea to define the set of accessible elements as an inductive domain and apply recursion on the constructors of this domain. The application of their method in Coq was however problematic due to the differences in the underlying type theories. While their method was based on Martin-Löf's type theory while the underlying type system of Coq was Cic. This issue was solved by Bertot and Castéran [BC04] in 2004 by means of introducing extra lemma's and providing an inductive domain which lives in `Prop`. Till then, all the methods suffered from problems involved with the mixture of function definition and proof statements. Recall that the main objections against the composition of code and proof were:

1. the construction of such a code becomes a tedious task. From the programming methodological point of view, the programmer needs to provide (at least) the termination proof simultaneously with the program derivation.
2. because the code is polluted with the proof (which is of no concern to the execution), such code is not easily readable for a programmer.

Barthe and Forest et al. [GBR06] were the first to separate the termination proof from the actual function definition by introducing a measure function for general recursive functions in 2006.

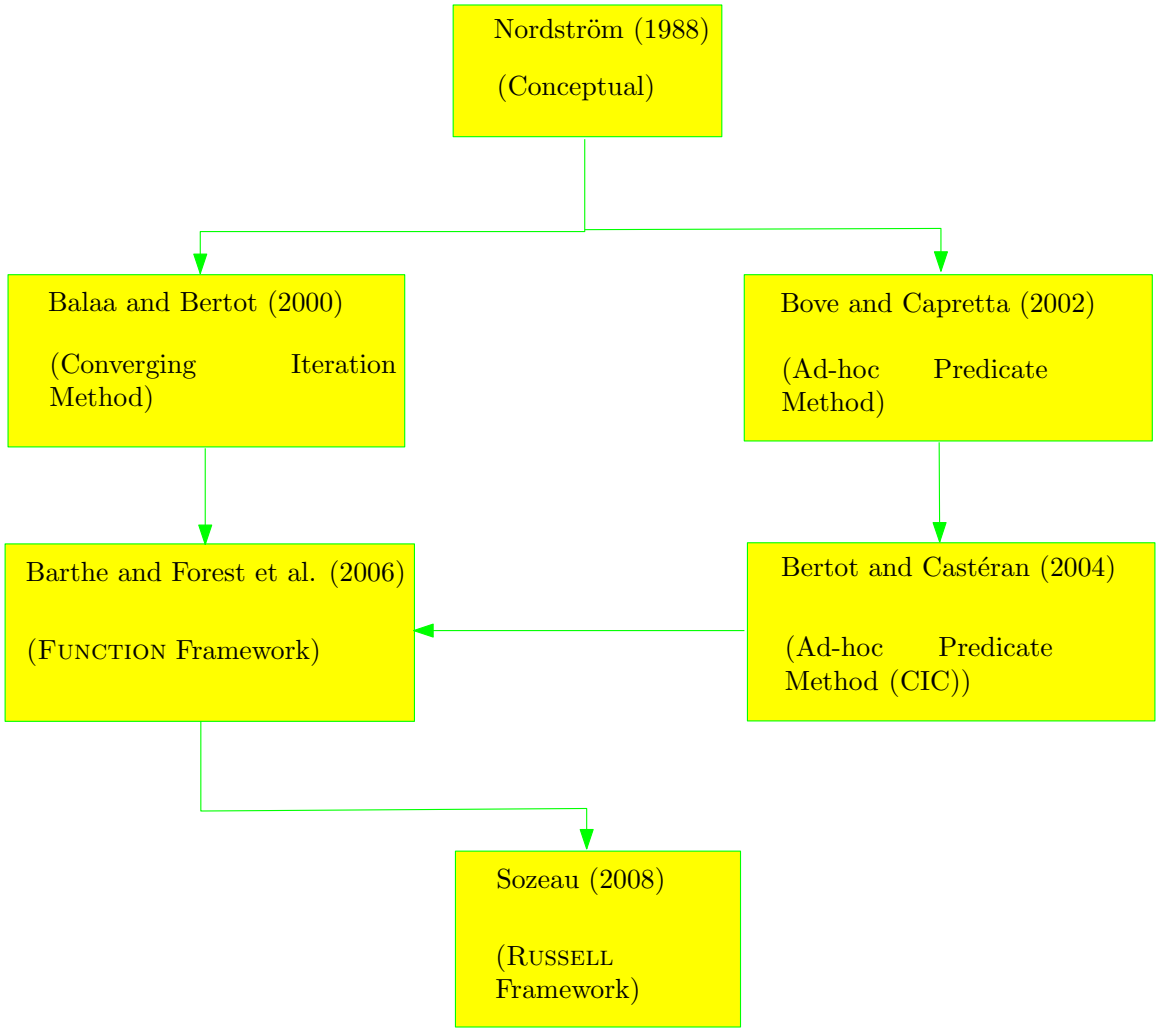

Figure 5.1: Illustration of development hierarchy

Furthermore, their FUNCTION framework provides the user with a bunch of useful principles (induction and inversion principles and fixpoint equations), suitable for reasoning about the functions. They used inductive domains (as introduced by Bove and Capretta), the well_founded induction principle and fixpoint equations (as introduced by Balaa and Bertot) and the inversion principle (as introduced by Bertot and Castéran). The most notable shortcoming of this framework was however its inability to deal with dependent types and hence the problem of partiality remains unsolved in the FUNCTION framework.

In 2008, Sozeau [Soz08] introduced the RUSSELL framework, by empowering COQ with subset types. By means of this powerful discipline, one can give the function definition enriched with a specification as expressive as desired, without having to provide the proof statements. In the RUSSELL framework the termination proof as well as the correctness proof of the properties is postponed for later time and can be given after function definition.

5.3 Conclusions

In chapter [4], the four methods have been compared based on four major criteria and the research questions are answered accordingly. Here a summary of the conclusions is provided.

RQ1. *How difficult is it to write the definition of the function?*

Conclusion 1. *Defining a general recursive function by means of the converging iteration method requires some preparation step because one has to prove the well-foundedness of the recursion. This proof is used in the body of the computational part of the definition and guarantees the termination. For the ad-hoc predicate method, this preparation step is even more involved as one has to provide an inductive domain definition and in addition the partiality and inversion theorems. Furthermore, in both methods the function definition is polluted with the proof statements which causes the definition to be more complicated and less readable. On the contrary, in both the* Function *and the* Russell *framework, the general recursive function can be defined straightforwardly. The ability to include a measure function to guarantee the termination. There is however one drawback to the* Function *framework namely its inability to deal with partial functions. Hence the* Russell *framework is the most suitable and complete framework when it comes to function definition.*

RQ2. *How close is the function definition to the original functional program expressed in a practical functional programming language , where there is no restriction on recursive calls?*

Conclusion 2. *As a consequence of Conclusion 1, it is clearly the case that the codes produced by means of the converging iteration method and the ad-hoc predicate method are not close to a practical functional programming language. The reason to this deviation is because those codes are composed of the actual function definition and proof statements. The definitions given by the* Function *and the* Russell *framework on the other hand provide the ability to define the general recursive functions as easily as in a practical functional programming language. Both frameworks achieve this by establishing a phase distinction between the writing and the proving of the functions.*

RQ3. *How difficult is it to formulate and prove the properties of the functions?*

Conclusion 3. *In all the methods, the difficulty involved in the proving crucial properties is proportional to the difficulty of the concerning function. However when it comes to the formulation of the properties, the* Russell *provides the user with a natural way to express these properties as a predicate on the output of the function. The same holds for partiality of a function, i.e. one can restrict the input type by means of a predicate on the input. These abilities are achieved by using the expressive power of subset types. For all other methods, a separate theorem needs to be provided expressing the desired properties to prove. The formulation of this theorem is proportional to the difficulty of the concerning function almost for all these three methods. In case of the Ad-hoc predicate method one has to take care of the proper domain which requires a bit more attention in formulation. One question which may arise here is about the ability of* Russell *for formulating and proving additional properties rather than the ones which are included in the input/output types. It has been shown that the formulation of additional properties in the* Russell *framework, although more involved, is possible.*

RQ4. *Is the extracted code in accordance with our expectation?*

Conclusion 4. *Initially, in none of the cases the generated extracted code is in accordance with the expectation. However, this slight difference can be resolved easily. In the cases of the Ad-hoc predicate method and the Converging iteration method, after the adaptation of these methods the expected code is generated. In cases of the* Function- *and the* Russell *framework the inlining of the definitions results in the expected extracted code. A considerable advantage of the last two frameworks over the first two methods is the fact that the extraction of* ***certified code*** *is enforced in these two frameworks. In other words, in the* Function- *and the* Russell *framework the function definition is accepted by the type checker only when all the pending proof obligations have*

been resolved, which prohibits the extraction of programs for which the termination and/or the correctness has not been proved yet. This is simply caused by the phase distinction between writing the programs and proving the correctness, while the correctness proof is required in order to add the fully certified object to the environment.

RQ5. *Can one execute the code in* Coq *sufficiently well in terms of the efficiency?*

Conclusion 5. *When introducing the second running example, twoPowN, we have justified the choice for its implementation by its efficiency. Comparing the execution result of the standard function two_power with the execution result of twoPowN confirms our claim. In all methods, the function twoPowN returns in all the methods instantly (except for* Russell *and the extended ad-hoc predicate method where it takes 1 sec.) While the execubility is achieved by all methods, the converging iteration method stands out in efficiency. By means of ad-hoc predicate method greater values are reachable but in terms of efficiency, this method is the worst one.*

5.4 Future Work

The Russell framework seems to offer rich enough possibilities to express a function with its desired type using the powerful and expressive discipline of subset types. It provides a user with a framework in which the function definition can be expressed without considering the proof elements at construction time. There is however lack of support for reasoning in this framework. The Function framework provides the user with the right principles to reason about the function (fixpoint equation, induction principle and inversion principle) once the termination of the function is ensured. In fact, these two frameworks are sharing the idea of postponing the proof till after the function has been constructed. Hence the advantages of one framework can be integrate with the other one in order to make a more suitable and user friendly framework. Because the Russell framework already covers the expressive discipline of dependent types, integrating the possibilities which Function framework offers for reasoning could be a possible way to cover the shortcomings of Russell in this area.

Furthermore, in the Russell framework, after the meta-variables have been instantiated with the actual proof statements and the complete term has been added to the environment, working with the term becomes more complicated. This includes function execution, the formulation of additional properties and so on. This is due to the presence of the sigma types in the definition of the complete term. Therefore, the user has to make an appropriate element of the sigma type (using *exist*) in those formulations each time. By advancing language technology one could work around this kind of difficulties. One last issue which is also related to the latter is the presence of the undesired first projections of sigma types in the extracted code. Automation of inlining in cases where the constants are involved in the extracted code, would help to resolve this problem as well.

Bibliography

[Acz77] Peter Aczel. An introduction to inductive definitions. In *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

[Aea98] N. I. Adams and D. H. Bartley et al. Revised[5] report on the algorithmic language Scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[BB00] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16. Springer-Verlag, 2000.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. http://www.labri.fr/publications/l3a/2004/BC04.

[BC05] A. Bove and V. Capretta. Modelling general recursion in type theory. 15(4):671–708, 2005.

[BG01] H. Barendregt and H. 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 Science Publishers B. V. and MIT Press, Amsterdam, The Netherlands, The Netherlands, 2001.

[BNS00] K. Petersson B. Nordström and J. M. Smith. Martin-Löf's type theory. In *Handbook of logic in computer science*, volume 5 of *Logic and algebraic methods*, pages 1–32, 2000.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[Coq] The Coq Proof Assistant. http://coq.inria.fr/.

[CP98] T. Coquand and H. Persson. Gröbner bases in type theory, 1998.

[ea92] S. Peyton Jones et al. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5):1, 1992.

[EMCS86] E. A. Emerson E. M. Clarke and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[Fra00] M. G. J. Franssen. *Cocktail: a tool for deriving correct programs*. PhD thesis, Eindhoven University of Technology, 2000.

[GBR06] D. Pichardie G. Barthe, J. Forest and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *In Functional and Logic Programming (FLOPS'06), LNCS 3945*, pages 114–129. Springer, 2006.

[Geu95] Herman Geuvers. The calculus of constructions and higher order logic, in the curry-howard isomorphism. In Ph. de Groote, editor, *Cahiers du Centre de logique*, volume 8, pages 139–191. Academia, Louvain-la-Neuve (Belgium), 1995.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[Lea04] X. Leroy and D. Doligez et al. The objective Caml system, version v3.08, institut national de recherche en informatique et en automatique, paris-rocquencourt, 2004. http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf.

[Let03] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Mea09] R. Milner and M. Tofte et al. *The definition of standard ML (revised)*. The MIT Press, 2009.

[Men09] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 2009.

[MKM00] J. Strother Moore M. Kaufmann and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. http://userweb.cs.utexas.edu/users/moore/acl2/.

[MW93] C. Paulin Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[Nor88] Bengt Nordström. Terminating general recursion. *BIT*, 28(3):605–619, 1988.

[PM89] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 89–104. ACM, jan 1989.

[Pnu77] A. Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, pages 46–57, 1977.

[Pol94] Robert Pollack. *The Theory of LEGO - A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, PhD thesis, Univ. of Edinburgh, 1994.

[RMDA07] F. Pichler R. Moreno Díaz and A. Quesada Arencibia, editors. *Computer Aided Systems Theory - EUROCAST 2007, 11th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 12-16, 2007, Revised Selected Papers*, volume 4739 of *Lecture Notes in Computer Science*. Springer, 2007.

[Sch09] Carsten Schürmann. The Twelf proof assistant. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009.*, volume 5674 of *Lecture Notes in Computer Science*, pages 79–83. Springer, 2009.

[Set04] Anton Setzer. Proof theory of Martin-Löf type theory – an overview. In *Mathematiques et Sciences Humaines, 42 année, n o 165:59 – 99*, pages 42–59, 2004.

[SOS92] J. M. Rushby S. Owre 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. Springer-Verlag, jun 1992. http://www.csl.sri.com/papers/cade92-pvs/.

[Soz06] Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *In Selected papers from the International Workshop on Types for Proofs and Programs (TYPES'06)*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006.

[Soz08] Matthieu Sozeau. *Un environnement pour la programmation avec types dependants*. PhD thesis, UNIVERSITE DE PARIS-SUD 11, 2008. http://mattam.org/research/publications/thesis-sozeau.pdf.

[SU06] M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., 2006.

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 1998.

[The09] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008-2009.

[TNW02] Lawrence C. Paulson T. Nipkow and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Appendices

Appendix A

Converging Iteration Method

A.1 Original version

log2

(* required libries *)

Require Import ProofWeb.
Require Import Arith.
Require Import Setoid.
Require Import Omega.
Require Import Coq.Arith.Div2.
Require Import Coq.Arith.Lt.
Require Import Arith.Mult.

(*———————— FUNCTION DEFINITION ————————*)

(* theorem which states that the argument to the recursive call is decreasing *)

Theorem div2_lt : forall $(x : nat), (lt\ (div2\ (S\ x))\ (S\ x))$.
Proof.
intuition.
Defined.

(* computational part of log2 definition *)

Definition log2_comp $(x : nat) :=$
match x return $(forall\ f : (forall\ y : nat, lt\ y\ x \to nat), nat)$ *with*
$\mid 0 \Rightarrow fun\ f : \_ \Rightarrow 0$
$\mid S\ 0 \Rightarrow fun\ f : \_ \Rightarrow 0$
$\mid S\ (S\ m) \Rightarrow fun\ f : (forall\ y : nat, lt\ y\ (S\ (S\ m)) \to nat) \Rightarrow (S\ (f\ (div2\ (S\ (S\ m)))\ (div2\_lt\ (S\ m))))$
end.

(* recursive part of the log2 definition *)

Definition log2 $:= (well\_founded\_induction\ lt\_wf\ (fun\ \_ : nat \Rightarrow nat)\ log2\_comp)$.

(*———————— PROPERTIES ————————*)

(* starting of the proofs of the properties *)

Section proof_on_log.

(* definition of 2^n from standard library; required for formulating properties *)

Fixpoint two_power $(n : nat) : nat :=$
match n with $0 \Rightarrow 1 \mid S\ p \Rightarrow 2 * two\_power\ p$ *end.*

(* hypothesis required for the proofs *)

Hypothesis mult2_div2_le $: (forall\ x : nat, 2 * div2\ x \leqslant x)$.
Hypothesis mult2_div2_gt $: (forall\ x : nat, x < 2 * (div2\ x))$.
Hypothesis div_gt $: (forall\ n : nat, 0 < n \to 0 < div2\ n)$.
Hypothesis mult2_S_2 $: (forall\ n : nat, two\_power\ (S\ n) = 2 * two\_power\ n)$.

(* fix-point equation of the log2 *)

Hypothesis log2_fix_eqn :
forall $n, log2\ n =$ *match n with*
$0 \Rightarrow 0$
$\mid 1 \Rightarrow 0$

```
  | S (S p) ⇒ S (log2 (div2 (S (S p))))
  end.

(* the property 2^(log2x) <= x for 0 < x *)

  Theorem pow_log_le :
  (forall (x : nat), (0 < x) → two_power (log2 x) ⩽ x).
  Proof.
  intros x; elim x using (well_founded_ind lt_wf).
  intros x0; case x0; auto with arith.
  intros n.
  case n; auto with arith.
  intros n0 Hrec Hlt.
  lazy beta iota zeta delta [log2 two_power]; fold two_power log2.
  rewrite (log2_fix_eqn (S (S n0))).
  rewrite mult2_S_2.
  apply le_trans with (2 * div2 (S (S n0))); auto with arith.
  Qed.

(* intermediate lemma's required for proof of the second property *)

(* induction scheme for div2 *)

  Lemma div2_ind :
  forall (P : nat → Prop),
  P 0 → P 1 → (forall n, P n → P (S (S n))) → forall n, P n.
  Proof.
  intros P H0 H1 Hstep n.
  assert (P n ∧ P (S n)).
  elim n; intuition.
  intuition.
  Qed.

(* relative value of div2n w.r.t. n *)

  Lemma div2_eq : forall n, 2 * div2 n = n ∨ 2 * div2 n + 1 = n.
  Proof.
  intros n; elim n using div2_ind; simpl; (try omega).
  intros. omega.
  Qed.

(* for all p and x, if div2 p + 1 < 2 * (2^x) then p + 2 < 2 * 2^(x+1) *)

  Lemma spec : forall p x, (S (div2 p) < 2 * two_power x) →
    (S (S p) < 2 * two_power (S x)).
  intros p x H;
  (cbv zeta iota beta delta [two_power]; fold two_power).
  elim (div2_eq p).
  intros; omega.
  intros; omega.
  Defined.


  Lemma div2_S_lt : forall p, S (div2 p) < S (S p).
  Proof.
  intro p.
```

```
  induction p; auto with arith.
  Defined.

(* hypothesis required for the proof *)

  Hypothesis pow_S : forall n : nat, 2 * two_power n = two_power (S n).

(* the property x < 2^(S(log2 x)) for 0 < x *)

  Theorem pow_log_gt :
  (forall (x : nat), x < two_power (S (log2 x))).
  Proof.
  intros x; elim x using (well_founded_ind lt_wf).
  intros x0; case x0; auto with arith.
  intros n.
  case n; auto with arith.
  intros n0 Hrec.
  lazy beta iota zeta delta [log2 two_power]; fold two_power log2.
  rewrite (log2_fix_eqn (S (S n0))).
  assert (forall p x, (S (div2 p) < 2 * two_power x) →
    (S (S p) < 2 * two_power (S x))).
  exact spec.
  assert (forall p, S (div2 p) < S (S p)).
  exact (div2_S_lt).
  apply (H n0 (log2 (div2 (S (S n0)))) (Hrec (S (div2 n0)) (H0 n0))).
  Qed.

(* end of the proofs of log2 *)

  End proof_on_log.

(*———————————— EXECUTION ————————————-*)

(* the actual execution of log2 with maximum reached value *)

  Time Eval compute in (log2 34920).

(*———————————— EXTRACTION ————————————*)

(* extraction of log2 to Ocaml *)

  Extraction log2.
```

twoPowN

```
(* required libraries *)

Require Import ProofWeb.
Require Import Arith.
Require Import Setoid.
Require Import Omega.
Require Import Coq.Arith.Div2.
Require Import Coq.Arith.Lt.
Require Import Arith.Even.

(*———————— FUNCTION DEFINITION ————————*)

(* square definition from statndard library *)

Definition square (n : nat) : nat := n * n.

(* theorem which states that the argument to the recursive call is decreasing *)

Theorem div2_lt : forall (x : nat), (lt (div2 (S x)) (S x)).
Proof.
intuition.
Defined.

(* the computational part of twoPowN *)

Definition twoPowN_comp (x : nat) :=
match x return
(forall f : (forall y : nat, lt y x → nat), nat) with
| 0 ⇒ fun f : _ ⇒ 1
| S p ⇒ fun f : (forall y : nat, lt y (S p) → nat) ⇒
                    if (even_odd_dec p)
                    then 2 * (f p (lt_n_Sn p))
                    else (square (f (div2 (S p)) (div2_lt p)))
end.

(* recursive part of the twoPowN definition *)

Definition twoPowN := (well_founded_induction lt_wf (fun _ : nat ⇒ nat) twoPowN_comp).

(*———————— PROPERTIES ————————*)

(* starting proof of log2 properties *)

Section proof_on_twoPowN.

(* hypothesis required for the proofs *)

Hypothesis square_gt : forall (n : nat), 0 < n → 0 < square n.
Hypothesis double_gt : forall (n : nat), 0 < n → 0 < 2 * n.
Hypothesis S_n : (forall n : nat, n < S n).
Hypothesis div2_S_n : (forall n : nat, 0 < n → div2 n < n).

(* the fixpoint definition of twoPowN *)

Hypothesis twoPowN_fix_eqn :
forall n, twoPowN n = match n with
                          0 ⇒ 1
```

```
                    | S p ⇒
                        if (even_odd_dec p)
                        then 2 * (f p (lt_n_Sn p))
                        else (square (f (div2 (S p)) (div2_lt p)))
```

(* the property $0 < 2^x$ *)

```
    Theorem pow_pos :
    (forall (x : nat), 0 < (twoPowN x)).
    Proof.
    intro x; elim x using (well_founded_ind lt_wf).
    intros x0; case x0.
    intro Hlt; simpl; auto with arith.
    intros n Hlt.
    lazy beta iota zeta delta [twoPowN]; fold twoPowN.
    rewrite (twoPowN_fix_eqn (S n)); case (even_odd_dec).
    intro Heven; exact (double_gt (twoPowN n) (Hlt n (S_n n))).
    intro Hodd.
    exact (square_gt (twoPowN (div2 (S n))) (Hlt (div2 (S n)) (div2_S_n (S n) (lt_0_Sn n)))).
    Qed.
```

(* end of proofs for twoPowN *)

```
End proof_on_twoPowN.
```

(*————————————— EXECUTION ——————————————-*)

(* actual exection of twoPowN *)

```
Time Eval compute in (twoPowN 15).
```

(*————————————— EXTRACTION ——————————————*)

(* inlining the definitions of the constants *)

```
Extraction Inline twoPowN_comp.
```

(* extraction of twoPowN to Ocaml *)

```
Extraction twoPowN.
```

A.2 With partiality

log2

```
(* required libraries *)

Require Import ProofWeb.
Require Import Arith.
Require Import Setoid.
Require Import Omega.
Require Import Coq.Arith.Div2.
Require Import Coq.Arith.Lt.

(*———————————— FUNCTION DEFINITION ————————————*)

(* the argument to the recursive call is decreasing *)

Theorem div2_lt : forall (x : nat), (lt (div2 (S x)) (S x)).
Proof.
intuition.
Defined.

(* computational part of log2 definition *)

(* note that due to the type of this definition well_founded_induction doesn't accept it *)

Definition log2_comp_part (x : nat) (domain : x <> 0) :=
  match x as z return x = z → (forall y : nat, lt y z → nat) → nat with
   | 0 ⇒ fun g ⇒ fun f ⇒ False_rec nat (domain g)
   | S 0 ⇒ fun g ⇒ fun f ⇒ 0
   | S (S m) ⇒ fun g ⇒
     fun f : (forall y : nat, lt y (S (S m)) → nat) ⇒
                                  (S (f (div2 (S (S m))) (div2_lt (S m))))
  end (refl_equal x)

(* new axiom for the adapted version *)

Axiom non_zero : forall x : nat, div2 (S (S x)) <> 0.

(* the extended computational part *)

(* types are adapted such that the whole would be acceptted by well_founded_induction *)

Definition log_comp2_part_adapted (x : nat) (domain : x <> 0) :=
match x as z return x = z → (forall y : nat, lt y z → y <> 0 → nat) → nat with
| 0 ⇒ fun g ⇒ fun f ⇒ False_rec nat (domain g)
| S 0 ⇒ fun g ⇒ fun f ⇒ 0
| S (S m) ⇒ fun g ⇒
  fun f : (forall y : nat, lt y (S (S m)) → y <> 0 → nat) ⇒
                    (S (f (div2 (S (S m))) (div2_lt (S m)) (non_zero m)))
end (refl_equal x).

(* the x¡¿0 should be a part of result type and hence the position of it is exchanged with the
function's result type *)

Definition log2_comp_part_final
(x : nat) (phi : forall y : nat, y < x → y <> 0 → nat) (H : x <> 0)
:= log2_comp_part_adapted x H phi.

(* finally the type of log2_comp_part_final is suitable for well_founded_induction *)
```

```
Definition log2_final :=
  (well_founded_induction lt_wf (fun y : nat ⇒ y <> 0 → nat) log2_comp_part_final).

(*———————————— PROPERTIES ————————————*)

(* starting of the proofs of the properties *)

Section proof_on_log.

(* definition of 2^n from standard library; required for formulating properties *)

Fixpoint two_power (n : nat) : nat :=
match n with 0 ⇒ 1 | S p ⇒ 2 * two_power p end.

(* hypothesis required for the proofs *)

Hypothesis mult2_div2_le : (forall x : nat, 2 * div2 x ⩽ x).

(* lemma required to define the fixpoint's first branch *)

Lemma neq_zero : forall n : nat, 0 = n → (n <> 0) → False.
Proof.
auto with arith.
Defined.

(* fix-point equation of the log2 *)

Hypothesis log2_fix_eqn :
forall n, log2_comp_part_final n = match n with
                                   0 ⇒ False_rec nat (neq_zero 0 _ _)
                                   | 1 ⇒ 0
                                   | S (S p) ⇒ S (log2_final (div2 (S (S p))))
                                   end.

(* the property 2^log2x <= x for 0 < x *)

Theorem pow_log_le :
(forall (x : nat), (0 < x) → two_power (log2 x) ⩽ x).
Proof.
intros x; elim x using (well_founded_ind lt_wf).
intros x0; case x0; auto with arith.
intros n.
case n; auto with arith.
intros n0 Hrec Hlt.
lazy beta iota zeta delta [log2 two_power]; fold two_power log2.
rewrite (log2_fix_eqn (S (S n0))).
rewrite mult2_S_2.
apply le_trans with (2 * div2 (S (S n0))); auto with arith.
Qed.

(* intermediate lemma's required for proof of the second property *)

(* induction scheme for div2 *)

Lemma div2_ind :
forall (P : nat → Prop),
P 0 → P 1 → (forall n, P n → P (S (S n))) → forall n, P n.
```

```
Proof.
intros P H0 H1 Hstep n.
assert (P n ∧ P (S n)).
elim n; intuition.
intuition.
Qed.

(* relative value of div2n w.r.t. n *)

Lemma div2_eq : forall n, 2 * div2 n = n ∨ 2 * div2 n + 1 = n.
Proof.
intros n; elim n using div2_ind; simpl; (try omega).
intros. omega.
Qed.

(* for all p and x, if div2 p + 1 < 2 * (2^x) then p + 2 < 2 * 2^(x+1) *)

Lemma spec : forall p x, (S (div2 p) < 2 * two_power x) →
  (S (S p) < 2 * two_power (S x)).
intros p x H;
(cbv zeta iota beta delta [two_power]; fold two_power).
elim (div2_eq p).
intros; omega.
intros; omega.
Defined.

(* hypothesis required for the proof *)

Hypothesis pow_S : forall n : nat, 2 * two_power n = two_power (S n).

(* lemma required for the last phase of the proof *)

Lemma div2_S_lt : forall p, S (div2 p) < S (S p).
Proof.
intro p.
induction p; auto with arith.
Defined.

(* the property x < 2^(S(log2 x)) for 0 < x *)

Theorem pow_log_gt :
(forall (x : nat), x < two_power (S (log2 x))).
Proof.
intros x; elim x using (well_founded_ind lt_wf).
intros x0; case x0; auto with arith.
intros n.
case n; auto with arith.
intros n0 Hrec.
lazy beta iota zeta delta [log2 two_power]; fold two_power log2.
rewrite (log2_fix_eqn (S (S n0))).
assert (forall p x, (S (div2 p) < 2 * two_power x) →
  (S (S p) < 2 * two_power (S x))).
exact spec.
assert (forall p, S (div2 p) < S (S p)).
exact (div2_S_lt).
```

apply (*H n0* (*log2* (*div2* (*S* (*S n0*)))) (*Hrec* (*S* (*div2 n0*)) (*H0 n0*))).
Qed.

(* end of the proofs of log2 *)

End proof_on_log.

(*———————— EXECUTION ————————————-*)

(* the actual execution *)

Time Eval compute **in** (*log2_final* 34920).

(*———————— EXTRACTION ————————————*)

(extraction of final version to Ocaml *)

Extraction log2_final.

Appendix B

Ad-hoc Predicate Method

B.1 Bove and Capretta

log2

(* required libraries *)

```
Require Export Arith.
Require Export ArithRing.
Require Export Omega.
Require Import Arith.Div2.
```

(*———————————— FUNCTION DEFINITION ————————————*)

(* the inductive domain definition *)

```
Inductive log_domain : nat → Set :=
  log_domain_1 : log_domain 1
| log_domain_2 :
  forall (p : nat), log_domain (S (div2 p)) → log_domain (S (S p)).
```

(* the partiality theorem *)

```
Theorem log_domain_non_O : forall (x : nat), log_domain x → (x <> 0).
Proof.
intros x H; case H; intros; discriminate.
Qed.
```

(* log2 fixpoint defintion *)

```
Fixpoint log2 (x : nat) (h : log_domain x) {struct h} : nat :=
match h with
| log_domain_1 ⇒ 0
| log_domain_2 p h1 ⇒ S (log2 (S (div2 p)) h1)
end.
```

(*———————————— PROPERTIES ————————————*)

(* maximal possible induction principle *)

```
Scheme log_domain_ind2 := Induction for log_domain Sort Prop.
```

(* definition of 2^n from standard library; required for formulating properties *)

```
Fixpoint two_power (n : nat) : nat :=
match n with 0 ⇒ 1 | S p ⇒ 2 * two_power p end.
```

(* starting proof of log2 properties *)

```
Section proof_on_log.
```

(* hypothesis required for the proofs *)

```
Hypothesis mult2_div2_le : (forall x : nat, 2 * div2 x ⩽ x).
```

(* the property $2^{log2\ x}$ <= x for 0 < x *)

```
Theorem pow_log_le :
(forall (x : nat) (h : log_domain x), two_power (log2 x h) ⩽ x).
Proof.
intros x h; elim h using log_domain_ind2.
simpl; auto with arith.
```

```
intros p l Hle.
lazy beta iota zeta delta [two_power log2]; fold log2 two_power.
apply le_trans with (2 * S (div2 p)). auto with arith.
exact (mult2_div2_le (S (S p))).
Qed.
```

(* intermediate lemma's required for proof of the second property *)

(* induction scheme for $div2$ *)

```
Lemma div2_ind :
forall (P : nat → Prop),
P 0 → P 1 → (forall n, P n → P (S (S n))) → forall n, P n.
Proof.
intros P H0 H1 Hstep n.
assert (P n ∧ P (S n)).
elim n; intuition.
intuition.
Qed.
```

(* relative value of $div2n$ w.r.t. n *)

```
Lemma div2_eq : forall n, 2 * div2 n = n ∨ 2 * div2 n + 1 = n.
Proof.
intros n; elim n using div2_ind; simpl; (try omega).
intros. omega.
Qed.
```

(* for all p and x, if $div2\ p + 1 < 2 * (2^x)$ then $p + 2 < 2 * 2^{x+1}$ *)

```
Lemma spec : forall p x, (S (div2 p) < 2 * two_power x) →
  (S (S p) < 2 * two_power (S x)).
intros p x H;
(cbv zeta iota beta delta [two_power]; fold two_power).
elim (div2_eq p).
intros; omega.
intros; omega.
Defined.
```

(* hypothesis required for the proof *)

```
Hypothesis pow_S : forall n : nat, 2 * two_power n = two_power (S n).
```

(* the property $x < 2^{S(log2\ x)}$ for $0 < x$ *)

```
Theorem pow_log_gt :
(forall (x : nat) (h : log_domain x), x < two_power (S (log2 x h))).
Proof.
intros x h; elim h using log_domain_ind2.
simpl; auto with arith.
intros p l Hle.
lazy beta iota zeta delta [log2 two_power]; fold log2 two_power.
assert (forall p x, (S (div2 p) < 2 * two_power x) →
  (S (S p) < 2 * two_power (S x))).
exact spec.
rewrite (pow_S (log2 (S (div2 p)) l)).
```

```
  rewrite ← pow_S in Hle.
  apply (H p (log2 (S (div2 p)) l) Hle).
  Qed.

(* end of proofs for log2 properties *)

  End proof_on_log.

(*————————————— EXECUTION ——————————————————*)

(* lemma required for execution: 5 is in the log domain *)

  Lemma aux : log_domain 5.
  repeat constructor.
  Defined.

(* actual exection of log2 *)

  Time Eval compute in (log2 5 aux).

(* the maximum reached value and its execution *)

  Lemma aux2 : log_domain 34920.
  repeat constructor.
  Defined.
  Time Eval compute in (log2 34920 aux2).

(*————————————— EXTRACTION ——————————————————*)

(* log2 extraction to Ocaml *)

  Extraction log.
```

twoPowN

```
(* required libraries *)

Require Import Arith.
Require Import ArithRing.
Require Import Omega.
Require Import Setoid.
Require Import Coq.Arith.Div2.
Require Import Coq.Arith.Even.

(*———————————— FUNCTION DEFINITION ————————————*)

(* the inductive domain definition *)

Inductive Dpow2 : nat → Set :=
  Dpow2_0 : Dpow2 0
| Dpow2_S_even :
  forall (q : nat), even (S q) → Dpow2 (div2 (S q)) → Dpow2 (S q)
| Dpow2_S_odd :
  forall (q : nat), odd (S q) → Dpow2 q → Dpow2 (S q).

(* square definition from standard library *)

Definition square (n : nat) : nat := n * n.

(* the fixpoint definition of twoPowN *)

Fixpoint twoPowN (n : nat) (h : Dpow2 n) {struct h} : nat :=
match h with
| Dpow2_0 ⇒ 1
| Dpow2_S_even p _ h2 ⇒ square (twoPowN (div2 (S p)) h2)
| Dpow2_S_odd p  _ h2 ⇒ 2 * (twoPowN p h2)
end.

(*———————————— PROPERTIES ————————————*)

(* maximal possible induction principle *)

Scheme Dpow2_ind2 := Induction for Dpow2 Sort Prop.

(* starting proof of log2 properties *)

Section proof_on_twoPowN.

(* hypothesis required for proving the properties *)

Hypothesis square_gt : forall (n : nat), 0 < n → 0 < square n.
Hypothesis gt_0 : forall n : nat, 0 < n → 0 < 2 * n.

(* the property 0 < 2^x *)

Theorem pow_pos :
(forall (x : nat) (h : Dpow2 x), 0 < (twoPowN x h)).
Proof.
intros x h; elim h using Dpow2_ind2.
simpl; auto with arith.
intros q Heven h2 Hlt.
lazy beta iota zeta delta [twoPowN]; fold twoPowN.
```

```
  exact (square_gt (twoPowN (div2 (S q)) h2) (Hlt)).
  intros q Hodd h2 Hlt.
  lazy beta iota zeta delta [twoPowN]; fold twoPowN.
  exact (gt_0 (twoPowN q h2) Hlt).
  Qed.

(* end of proof of twoPowN properties *)

  End proof_on_twoPowN.

(*———————————— EXECUTION ————————————-*)

(* lemma required for execution: 15 is in the power domain *)

  Lemma aux : (Dpow2 15).
  constructor 3; intuition.
  constructor 2; intuition.
  constructor 3; intuition.
  constructor 2; intuition.
  constructor 3; intuition.
  constructor 2; intuition.
  constructor 3; intuition.
  constructor 1.
  Defined.

(* the actual execution *)

  Time Eval compute in (twoPowN 15 aux).

(*———————————— EXTRACTION ————————————*)

(* twoPowN extraction to Ocaml *)

  Extraction twoPowN.
```

B.2 Bertot and Castéran

log2

(* required libraries *)

```
Require Export Arith.
Require Export ArithRing.
Require Export Omega.
Require Import Arith.Div2.
```

(*———————————— FUNCTION DEFINITION ————————————*)

(* the inductive domain *)

```
Inductive log_domain : nat → Prop :=
  log_domain_1 : log_domain 1
| log_domain_2 :
  forall (p : nat), log_domain (S (div2 p)) → log_domain (S (S p)).
```

(* the partiality theorem *)

```
Theorem log_domain_non_O : forall (x : nat), log_domain x → (x <> 0).
Proof.
intros x H; case H; intros; discriminate.
Qed.
```

(* the inversion theorem *)

```
Theorem log_domain_inv :
forall (x p : nat), log_domain x → x = S (S p) → log_domain (S (div2 p)).
Proof.
intros x p H; case H; (try (intros H'; discriminate H')).
intros p' H1 H2; injection H2; intros H3; rewrite ← H3; assumption.
Defined.
```

(* the fixpoint definition of log2 *)

```
Fixpoint log2 (x : nat) (h : log_domain x) {struct h} :=
match x as y return x = y → nat with
| 0 ⇒ fun h' ⇒ False_rec nat (log_domain_non_O x h h')
| S 0 ⇒ fun h' ⇒ 0
| S (S p) ⇒ fun h' ⇒ S (log2 (S (div2 p)) (log_domain_inv x p h h'))
end (refl_equal x).
```

(*———————————— PROPERTIES ————————————*)

(* maximal possible induction principle *)

```
Scheme log_domain_ind2 := Induction for log_domain Sort Prop.
```

(* definition of 2^n from standard library; required for formulating properties *)

```
Fixpoint two_power (n : nat) : nat :=
match n with 0 ⇒ 1 | S p ⇒ 2 * two_power p end.
```

(* starting proof of log2 properties *)

```
Section proof_on_log.
```

(* hypothesis required for the proofs *)

Hypothesis mult2_div2_le : (*forall* $x : nat, 2 * div2\ x \leqslant x$).

(* the property $2^{log2\ x} \leqslant x$ for $0 < x$ *)

Theorem pow_log_le :
(*forall* $(x : nat)\ (h : log\_domain\ x), two\_power\ (log2\ x\ h) \leqslant x$).
Proof.
intros x h; *elim h using log_domain_ind2*.
simpl; *auto with arith*.
intros p l Hle.
lazy beta iota zeta delta [*two_power log_domain_inv log2*]; *fold log2 two_power*.
apply le_trans with $(2 * S\ (div2\ p))$; *auto with arith*.
exact $(mult2\_div2\_le\ (S\ (S\ p)))$.
Qed.

(* intermediate lemma's required for proof of the second property *)

(* induction scheme for *div*2 *)

Lemma div2_ind :
forall $(P : nat \to Prop)$,
$P\ 0 \to P\ 1 \to (forall\ n, P\ n \to P\ (S\ (S\ n))) \to forall\ n, P\ n$.
Proof.
intros P H0 H1 Hstep n.
assert $(P\ n \wedge P\ (S\ n))$.
elim n; *intuition*.
intuition.
Qed.

(* relative value of *div*2*n* w.r.t. *n* *)

Lemma div2_eq : *forall* $n, 2 * div2\ n = n \vee 2 * div2\ n + 1 = n$.
Proof.
intros n; *elim n using div2_ind*; *simpl*; (*try omega*).
intros. *omega*.
Qed.

(* for all p and x, if $div2\ p + 1 < 2 * (2^x)$ then $p + 2 < 2 * 2^{x+1}$ *)

Lemma spec : *forall* $p\ x, (S\ (div2\ p) < 2 * two\_power\ x) \to$
$(S\ (S\ p) < 2 * two\_power\ (S\ x))$.
intros p x H;
(*cbv zeta iota beta delta* [*two_power*]; *fold two_power*).
elim (*div2_eq p*).
intros; *omega*.
intros; *omega*.
Defined.

(* hypothesis required for the proof *)

Hypothesis pow_S : *forall* $n : nat, 2 * two\_power\ n = two\_power\ (S\ n)$.

(* the property $x < 2^{S(log2\ x)}$ for $0 < x$ *)

Theorem pow_log_gt :
(*forall* $(x : nat)\ (h : log\_domain\ x), x < two\_power\ (S\ (log2\ x\ h))$).

```
Proof.
intros x h; elim h using log_domain_ind2.
simpl; auto with arith.
intros p l Hle.
lazy beta iota zeta delta [two_power]; fold two_power.
assert (forall p x, (S (div2 p) < 2 * two_power x) →
  (S (S p) < 2 * two_power (S x))).
exact spec.
rewrite (pow_S (log2 (S (S p)) (log_domain_2 p l))).
rewrite ← pow_S in Hle.
apply (H p (log2 (S (div2 p)) l) Hle).
Qed.
```

(* end of proofs for *log*2 properties *)

```
End proof_on_log.
```

(*———————————— EXECUTION ————————————*)

(* lemma required for execution: 5 is in the log domain *)

```
Lemma aux : log_domain 5.
repeat constructor.
Defined.
```

(* actual exection of log2 *)

```
Time Eval compute in (log2 5 aux).
```

(* maximum reached value and its execution *)

```
Lemma aux2 : log_domain 34920.
repeat constructor.
Defined.
Time Eval compute in (log2 34920 aux2).
```

(*———————————— EXTRACTION ————————————*)

(* log2 extraction to Ocaml *)

```
Extraction log2.
```

twoPowN

```
(* required libraries *)

Require Export Arith.
Require Export ArithRing.
Require Export Omega.
Require Import Setoid.
Require Import Coq.Arith.Div2.
Require Import Coq.Arith.Even.

(*———————————— FUNCTION DEFINITION ————————————*)

(* the inductive domain definition *)

Inductive Dpow2 : nat → Prop :=
  Dpow2_0 : Dpow2 0
| Dpow2_S_even :
  forall (q : nat), even (S q) → Dpow2 (div2 (S q)) → Dpow2 (S q)
| Dpow2_S_odd :
  forall (q : nat), odd (S q) → Dpow2 q → Dpow2 (S q).

(* the inversion theorem for even values *)

Lemma Dpow2_even_inv :
forall (x q : nat), even (S q) → Dpow2 x → x = S q → Dpow2 (div2 (S q)).
Proof.
intros x q Heven Hdom; case Hdom; (try (intros H'; discriminate H')).
intros q' Heven2 H1; intros Heq; rewrite ← Heq; assumption.
intros q' Hodd Hdom2 Heq; rewrite Heq in * | - .
assert (forall n : nat, even n → odd n → False).
exact not_even_and_odd.
destruct (H (S q)); repeat assumption.
Defined.

(* the inversion theorem for odd values *)

Lemma Dpow2_odd_inv :
forall (x q : nat), odd (S q) → Dpow2 x → x = S q → Dpow2 q.
Proof.
intros x q Heven Hdom; case Hdom; (try (intros H'; discriminate H')).
intros q' Heven2 H1; intros Heq; rewrite Heq in * | - .
assert (forall n : nat, even n → odd n → False).
exact not_even_and_odd.
destruct (H (S q)); repeat assumption.
intros q' Hodd Hdom2 Heq; rewrite Heq in * |-; injection Heq; intro Heq'.
rewrite Heq' in * | - .assumption.
Defined.

(* square definition from standard library *)

Definition square (n : nat) : nat := n * n.

(* the fixpoint definition of twoPowN *)

Fixpoint twoPowN (x : nat) (h : Dpow2 x) {struct h} :=
match x as y return x = y → nat with
| 0    ⇒ fun h' ⇒ 1
```

```
  | S p ⇒ match (even_odd_dec (S p)) with
          | left h1 ⇒ fun h' ⇒ (square (twoPowN (div2 (S p)) (Dpow2_even_inv x p h1 h h')))
          | right h1 ⇒ fun h' ⇒ (2 * (twoPowN p (Dpow2_odd_inv x p h1 h h')))
          end end (refl_equal x).

(*———————————— PROPERTIES ————————————*)

(* maximal possible induction principle *)

  Scheme Dpow2_ind2 := Induction for Dpow2 Sort Prop.

(* starting proof of twoPowN properties *)

  Section proof_on_twoPowN.

(* hypothesis required for proofs. *)

  Hypothesis square_gt : forall (n : nat), n < square n.

(* the property 0 < 2^x *)

  Theorem pow_pos :
  (forall (x : nat) (h : Dpow2 x), 0 < (twoPowN x h)).
  Proof.
  intros x h; elim h using Dpow2_ind2.
  simpl; auto with arith.
  intros q Heven h2 Hlt.
  unfold Dpow2_even_inv.
  rewrite Hlt.
  simpl.
  lazy beta iota zeta delta [Dpow2_even_inv div2 twoPowN]; fold div2 twoPowN.

(*———————————— EXECUTION ————————————-*)

(* lemma required for execution: 15 is in the power domain *)

  Lemma aux : (Dpow2 15).
  constructor 3; intuition.
  constructor 2; intuition.
  constructor 3; intuition.
  constructor 2; intuition.
  constructor 3; intuition.
  constructor 2; intuition.
  constructor 3; intuition
  constructor 1; intuition.
  Defined.

(* the actual execution *)

  Time Eval compute in (twoPowN 15 aux).

(*———————————— EXTRACTION ————————————*)

(* twoPowN extraction to Ocaml *)

  Extraction pow2.
```

Appendix C

Function Framework

twoPowN

```
(* required libraries *)

  Require Import Arith.
  Require Import Setoid.
  Require Import Omega.
  Require Import Coq.Arith.Div2.
  Require Import Arith.Even.
  Require Import Recdef.

(*———————————— FUNCTION DEFINITION ————————————*)

(* square definition from standard library *)

  Definition square (n : nat) : nat := n * n.

(* the fixpoint definition of twoPowN *)

  Function twoPowN (n : nat) {measure id n} : nat :=
  match n with
  | 0 ⇒ 1
  | S p ⇒ if (even_odd_dec p) then 2 * (twoPowN p) else (square (twoPowN (div2 (S p))))
  end.

(*———————————— PROOF OBLIGATIONS (termiation) ————————————*)


  Proof.
  intros n p Heq Heven Heven_odd_dec; auto with arith.
  intros n p Heq Hodd Heven_odd_dec; unfold id; auto with arith.
  Defined.

(*———————————— EXECUTION ————————————-*)

(* starting proof of log2 properties *)

  Section proof_on_twoPowN.

(* hypothesis required for proofs *)

  Hypothesis square_gt : forall (n : nat), 0 < n → 0 < square n.

(* the property 0 < 2^x *)

  Theorem pow_pos :
  (forall (x : nat), 0 < twoPowN x).
  Proof.
  intro x; functional induction (twoPowN x).
  auto with arith.
  omega.
  exact (square_gt (twoPowN (div2 (S p))) (IHn)).
  Qed.

(* end of proofs for log2 properties *)

  End proof_on_twoPowN.

(*———————————— EXTRACTION ————————————*)
```

(* the actual execution *)

Time Eval compute **in** (*twoPowN* 15).

(*———————— EXTRACTION ————————*)

(* inlining the constant defintions *)

Extraction Inline twoPowN_terminate.

(* extraction of twoPowN to Ocaml *)

Extraction twoPowN.

Appendix D

Russell

log2

```
(* required libraries *)

Require Import Program.
Require Import Setoid.
Require Import Arith.
Require Import Omega.
Require Import Arith.Div2.
Require Import Even.

(*———————————— FUNCTION DEFINITION ————————————*)

(* definition of two_power from standard library; required for output type *)

Fixpoint two_power (n : nat) : nat :=
match n with
| 0 ⇒ 1
| S p ⇒ 2 * two_power p
end.

(* fixpoint definition of log2 *)

Program Fixpoint log2 (n : nat | n <> 0) {measure n} :
{n' : nat | ((two_power n' ⩽ n) ∧ (n < two_power (S n')))} :=
match n with
| 0 ⇒ _
| S 0 ⇒ 0
| S (S p) ⇒ S (log2 (div2 (S (S p))))
end.

(*————————— PROOF OBLIGATIONS (termiation + correctness) —————————*)

(* list of remaining obligations *)

Obligations.

(*the first remaining obligation*)
Next Obligation.
apply lt_n_S.
case p.
simpl; intuition.
intros n; rewrite (lt_div2 (S n)).
omega. intuition.
Defined.

(* intermediate lemma's required for proving the next obligation *)

Lemma div2_ind :
forall (P : nat → Prop),
P 0 → P 1 → (forall n, P n → P (S (S n))) → forall n, P n.
Proof.
intros P H0 H1 Hstep n.
assert (P n ∧ P (S n)).
elim n; intuition.
intuition.
Qed.


Lemma div2_eq : forall n, 2 * div2 n = n ∨ 2 * div2 n + 1 = n.
Proof.
```

```
intros n; elim n using div2_ind; simpl; (try omega).
intros. omega.
Qed.


Lemma spec : forall p x, (two_power x ⩽ S (div2 p) < 2 * two_power x) →
  (two_power (S x) ⩽ S (S p) < 2 * two_power (S x)).
intros p x H;
(cbv zeta iota beta delta [two_power]; fold two_power).
elim (div2_eq p).
intros; omega.
intros; omega.
Defined.


Lemma two_times : (forall p x : nat, ((two_power x) ⩽ S (div2 p)) →
  (2 * (two_power x) ⩽ 2 * (S (div2 p)))).
intros; omega.
Defined.


Lemma plus_mul : (forall n : nat, n + n = 2 * n).
intros; simpl; omega.
Defined.


Lemma four_times :
  (forall x : nat,
    two_power x + two_power x + (two_power x + two_power x) = 2 * (two_power x + two_power x)).
intros.
rewrite ← (plus_mul); intuition.
Defined.
```

(* the second remaining obligation *)

```
Next Obligation.
simpl. elim log2. simpl.
assert (forall p x, (two_power x ⩽ S (div2 p) < 2 * two_power x) →
  (two_power (S x) ⩽ S (S p) < 2 * two_power (S x))).
exact spec.
intros.
rewrite ? (plus_0_r).
rewrite (plus_0_r) in * |- .
rewrite (four_times).
rewrite (plus_mul).
rewrite (plus_mul) in * |- .
apply (H0 p x p0).
Defined.
```

(*———————— EXECUTION ————————-*)

(* lemma required for execution: 5 is a valid input i.e. non-zero *)

```
Definition aux : 5 <> 0.
auto.
Defined.
```

(* the actual execution *)

Time Eval compute **in** (*proj1_sig* (*log2* (*exist* _ 5 *aux*))).

(* the maximum reached value and its execution *)

Definition aux : 32760 <> 0.
auto.
Defined.
Time Eval compute **in** (*proj1_sig* (*log2* (*exist* _ 32760 *aux*))).

(*———————— EXTRACTION ————————*)

(* inlining the constant defintions *)

Extraction Inline proj1_sig.

(* extraction of log2 to Ocaml *)

Extraction log2.

twoPowN

```
(* required libraries *)

  Require Import Program.
  Require Import Arith.
  Require Import Setoid.
  Require Import Omega.
  Require Import Coq.Arith.Div2.
  Require Import Arith.Even.

(*———————————— FUNCTION DEFINITION ————————————*)

(* square definition from standard library *)

  Definition square (n : nat) : nat := n * n.

(* the fixpoint definition of twoPowN *)

  Program Fixpoint twoPowN (n : nat) {measure n} :
  {n' : nat | 0 < n'} :=
  match n with
  | 0 ⇒ 1
  | S p ⇒ if (even_odd_dec p) then 2 * (twoPowN p) else (square (twoPowN (div2 (S p))))
  end.

(*————————— PROOF OBLIGATIONS (termiation + correctness) —————*)

(* list of remaining obligations *)

  Obligations.

(* the first remaining obligation *)

  Next Obligation.
  simpl; elim twoPowN; auto with arith.
  Defined.

(* the second remaining obligation *)

  Next Obligation.
  elim H; intros n IH; rewrite (even_div2); auto with arith; assumption.
  Defined.

(* hypotheis required for proving the next obligation *)

  Hypothesis square_S : forall (n : nat), square n < square (S n).

(* the third remaining obligation *)

  Next Obligation.
  destruct twoPowN; simpl; elim l; auto with arith.
  intros m Hm Hsquare; rewrite Hsquare; exact (square_S m).
  Defined.

(*———————————— EXECUTION ————————————————-*)

(* the actual execution *)

  Time Eval compute in twoPowN 15.
```

(*———————— EXTRACTION ————————*)

(* inlining the constant defintions *)

Extraction Inline proj1_sig.

(* twoPowN extraction to Ocaml *)

Extraction twoPowN.

lists

(* required libraries *)

Require Import ProofWeb.
Require Import Program.
Require Import Setoid.
Require Import Omega.
Require Import List.
Require Import Bool.
Require Import Bool.Sumbool.
Require Import Arith.

(* a variable of type set required for definitions *)

Variable A : *Set.*

(* head definition *)

Program Definition head $(l : list\ A \mid (l <> nil)) :$
$\{ a : A \mid exists\ l' : list\ A, a :: l' = l \} :=$
match l *with*
$\mid hd :: tl \Rightarrow hd$
$\mid nil \Rightarrow \_$
end.

(* the only remaining proof obligations *)

Next Obligation.
exists tl; *reflexivity.*
Qed.

(* tail definition *)

Program Definition tail $(l : list\ A \mid l <> nil) :$
$\{ l' : list\ A \mid exists\ a, (cons\ a\ l') = l \} :=$
match l *with*
$\mid hd :: tl \Rightarrow tl$
$\mid nil \Rightarrow nil$
end.

(* the only remaining proof obligations *)

Next Obligation.
exists hd; *reflexivity.*
Qed.

(* elt definition *)

Program Fixpoint elt $(l : list\ nat)\ (n : nat) :$
$\{ b : bool \mid ((b = true) <-> (In\ n\ l)) \} :=$
match l *with*
$\mid nil \Rightarrow false$
$\mid (x :: xs) \Rightarrow (beq\_nat\ x\ n) \| (elt\ xs\ n)$
end.

(* the list of remaining obligations for elt *)

Obligations.

(* the first remaining obligation *)

Next Obligation.
intuition.
Qed.

(* the last remaining obligation *)

Next Obligation.
intuition.
assert (*forall x0* : *bool*, {*x0* = *true*} + {*x0* = *false*}).
exact (*sumbool_of_bool*).
destruct (*H2 x0*).
right.
apply (*H e*).
left; *rewrite e* **in** *H1.*
assert (*forall a b* : *bool*, (*a*‖*b* = *true*) → *a* = *true* ∨ *b* = *true*).
exact (*orb_prop*).
apply (*H3* (*beq_nat x n*) *false*) **in** *H1.*
case *H1.*
apply beq_nat_true.
intro.
congruence.
assert (*forall b1 b2* : *bool*, *b1* = *true* ∨ *b2* = *true* → *b1*‖*b2* = *true*).
exact (*orb_true_intro*).
elim (*H1* (*beq_nat x n*) *x0*).
trivial.
left.
elim H2.
assert (*forall n*, *true* = *beq_nat n n*).
exact (*beq_nat_refl*).
elim (*H3 x*).
trivial.
Defined.

(* extraction of these functions to Ocaml *)

Extraction head.
Extraction tail.
Extraction elt.

List of Figures