Proof search and induction choices in AF_2 system

D. Galmiche and O. Hermann

CRIN-CNRS & INRIA-Lorraine
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex
France.
e-mail : galmiche{ohermann}@loria.fr

Abstract

We consider, in this paper, the AF_2 system with the aim to automatize the proof search process considering the possibilities to synthesize programs but also to prove some properties of the programs. In our work, we consider an analysis of program specifications to take in charge the problem of induction choice and we propose adequate induction schemas to instantiate the induction step in the proof search process. In a certain way, our analysis allows to generate a proof strategy based on a specific proof form (including an adequate induction) for different kinds of function specifications. Thus we describe a proof search procedure based on induction choices following a classification of function specifications in AF_2 w.r.t. the failure cases of given induction schemas.

1 Introduction

A lot of type-systems have been proposed since several years with a main purpose that is program extraction via the Curry-Howard isomorphism establishing a correspondence between proofs of specifications and programs. Here we consider one of these systems, namely AF_2 (second order functional arithmetic) [5, 7] such that algorithmic specifications are equations and having a programming method that ensures the correctness of programs extracted form proofs of function totality. But what about construction of proofs and programs and also efficiency of these programs ? It appears clearly that there exists a link between the choice of data types definition, and then of induction schema during the proof search, in this logical system and the efficiency of extracted programs [8]. To get the efficiency that one might expect some works have introduced new types in AF_2 as inductives types [9] or coinductive types [10] preserving the correctness of the programming method. Another point consists in trying to propose some procedures for inductive proof search that could generate efficient programs with a maximal mechanization of the proof search process.
In our work, that is the continuation of previous work on automated proof search in AF_2 based on tactics and proof plans [2, 3], we consider the initial logical system (without extensions or restrictions) as an appropriate sequent calculus and we try to automate the proof search process considering the possibilities to synthesize programs or to prove some program properties. In previous work, in a restriction of AF_2 with recursive types, [6] has proposed, for two fixed induction schemas, specific proof strategies based on specific proof forms. These strategies are correct and complete but the choice of the induction schema is made a priori by the user. Let us recall that

most inductive theorem provers rely upon the user intervention with the need for some eureka steps such as this induction schema selection [1, 4]. Moreover, expressing the properties in a second order logical framework allows the generation of new induction schemas based on specific second order substitutions.
In this paper, we consider an analysis of program specifications to take in charge this problem of induction choice and then propose adequate induction schemas to instantiate the induction step in the proof search process. In a certain way, our analysis allows to generate a proof strategy based on a specific proof form (including an adequate induction) for different kinds of function specifications. Thus, we describe here a procedure of induction choice with an original approach based on a classification of function specifications in AF_2 w.r.t. the failure cases of proof search with given induction schemas. It defines and classifies the induction positions and variables inside the specification and thus selects an induction schema for each induction position. This proposal is based on a taxonomy of function specifications w.r.t. induction choices failure cases. Compared to other works [6, 9], we do not restrict the use of some data types and then keep in this automated proof search the possibility to synthesis some efficient programs. But with this global approach we do not have directly proof forms and proof strategies that are complete w.r.t. to provability of certain specifications forms. Here the choice of the induction schema is not made a priori by the user but proposed by a procedure. Therefore we have a general proof form with four main parts that are *specification analysis* that leads to possible induction choices, *induction, rewriting* and *resolution*. If we fix some induction choices and some specifications forms, we can recover refined proof forms that are complete as in [6] and a further work would consists to start from our results and to refine them to propose new proof forms that would be complete for given specifications. In this way, we could have a powerful tool for interactive and automated proof search in AF_2 and their programming applications.
We will illustrate the interest of this procedure (and of the underlying second order intuitionistic logic) for efficient and successful proof search leading to efficient programs and also avoiding some eureka steps during properties proof search.

2 The logical framework

In this paper, we consider a second-order constructive framework, named AF_2 [7]. It is a second-order typed lambda-calculus, that is a natural deduction system for our logic with extraction of λ-terms (considered as programs) from constructive proofs.

2.1 A second order constructive logic

The formulae (or types) are constructed using the connective $\rightarrow$, the quantifier $\forall$, individual (or first order) variables x, y, z,..., predicate (or second-order) variables X, Y, Z,..., individual and function constants. The individual terms are constructed from individual and function constants. Notice that the other logical symbols are definable from $\rightarrow$ and $\forall$.
The expressions of the language are in the form $a : A$ with either one of these interpretations: a is an element of type A, or a is a proof of the proposition A. Here, we consider the second one where formulas are regarded as types and deductions as λ-terms. The construction of deductions uses the following inference rules (A, B denote formulae while Γ represents contexts, i.e., sequences of formulae and $FV(\Gamma)$ the set of free variables in the context Γ):
R1. Axiom. $x : A \vdash x : A$
R2. Abstraction rule. If $\Gamma, x : A \vdash t : B$ then $\Gamma \vdash \lambda x.t : A \rightarrow B$

R3. Application rule. If $\Gamma \vdash t : A \rightarrow B,$ and $\Gamma \vdash u : A$ then $\Gamma \vdash$ (t,u) : B
R4. Generalization rule 1. If $\Gamma \vdash t : A$ *with* $x \notin FV(\Gamma)$ then $\Gamma \vdash t: \forall x\ A$
R5. Specialization rule 1. If $\Gamma \vdash t : \forall x\ A,$ *with b individual term* then $\Gamma \vdash t : A[b/x]$
R6. Generalization rule 2. If $\Gamma \vdash t : A$ *with* $X \notin FV(\Gamma)$ then $\Gamma \vdash t : \forall X\ A$
R7. Specialization rule 2. If $\Gamma \vdash t : \forall X\ A$ *with b a formula* then $\Gamma \vdash t : A[b/ X]$
R8. Equivalence rule. If $\Gamma \vdash t : A$ then $\Delta \vdash t : A$ if Δ is obtained from Γ by permutation, extension or contraction.
R9. Equality rule. If $\Gamma \vdash t : A$ and $u = v$ then $\Gamma \vdash t : A[v/u]$.

Programming in this framework consists in giving first a formal specification (a logical proposition and some equations stating the behaviors of the intended function), constructing a proof of the given formula and then extracting code systematically from this proof with the *realizability* notion. Programs obtained this way have the essential properties of *correctness* and *termination.* This framework naturally contains induction through the data type definitions. For example, it is possible to define integers following an iterative or recursive way .
The iterative definition for integers is expressed by the following second-order formula:
$Nx \equiv \forall X\ [\forall y\ [Xy \rightarrow Xs(y)] \rightarrow [X0 \rightarrow Xx]]$ where X design a second-order variable and x, y first-order variables. The recursive definition is expressed by the following formula:
$Nx \equiv \forall X\ [\forall y\ [Ny \rightarrow Xy \rightarrow Xs(y)] \rightarrow [X0 \rightarrow Xx]].$

2.2 How to express specifications and properties

In this framework, a function f is specified by a formal specification *Spec* composed of some equations stating the behaviors of the intended function and of a formula F stating that the function is well typed. Then a n-ary function $f : T_1 \times ... \times T_n \rightarrow T$ is defined by:
- p equations of the form: $f(t_{11}, ..., t_{1n}) = u_1,$ ...... $f(t_{p1}, ..., t_{pn}) = u_p$.
- a formula that expresses its type information: $\forall z_1\ T_1 z_1 \rightarrow .... \rightarrow \forall z_n T_n z_n \rightarrow Tf(z_1, ..., z_n)$.

In fact, a data type is defined by a name T and a set of constructors composed by
- basic constructors: $c_1, .., c_p$ with c_i of type T,
- functional constructors: $cf_1, .., cf_k$ with cf_i of type $A_{i1} \times ... \times A_{is_i} \times T \times ... \times T \rightarrow T$ with A_{ij} soon defined. Moreover, the iterative definition of type T is given by the expression
$Tz \equiv \forall X(\phi_1 \rightarrow .... \rightarrow \phi_k \rightarrow Xc_1 ... \rightarrow Xc_p \rightarrow Xz$
with $\phi_i \equiv \forall a_{i1} A_{i1} a_{i1} \rightarrow ... \rightarrow a_{is_i} A_{is_i} a_{is_i} \rightarrow y_{i1} X_{i1} y_{i1} \rightarrow y_{it_i} X_{it_i} y_{it_i} \rightarrow Xcf_i(a_{i1}, ..y_{it_i})$.

Example 2.1 *We define the addition function with the following specification*
(1) $plus(0, y) = y,$
(2) $plus(s(x), y) = s(plus(x, y)),$
(3) $\forall z\ (Nz \rightarrow (\forall z_1(Nz_1 \rightarrow Nplus(z, z_1)))).$
Moreover, the iterative definition of integer data type is given by the following expression
$Nz \equiv \forall X(\forall y(Xy \rightarrow Xs(y)) \rightarrow (X0 \rightarrow Xz)).$

If we prove *(3)* in G_{AF2}, the λ-term extracted from the proof will be a program of addition. Even, we can naturally use the proof of the type formula to synthesize programs, we will consider here another formulas that express properties or conjectures on functions expressed in this calculus. Such properties have here the following general form:
$\forall z_1\ T_1 z_1 \rightarrow .... \rightarrow \forall z_n T_n z_n \rightarrow (f(g_1(z_{11}, ... z_{1r_1}), ...g_n(z_{n1}, ..., z_{nr_n})) = h(z_{11}, ...z_{nr_n})$ with $f, h, g_1, ...g_n$ are specified in the logical framework.

Example 2.2 *We can express properties as the associativity of the* addition *function by* $\forall z\ (Nz\ \rightarrow\ \forall z_1\ (Nz_1 \rightarrow\ \forall z_2\ Nz_2 \rightarrow (plus(z, plus(z_1, z_2))\ =\ plus(plus(z, z_1), z_2))))$.

3 Proof search in AF_2

There are some works devoted to AF_2 as programming with proofs language. From the proof search point of view, a main work concerns a restriction of AF_2 with recursive types [6] with, for two induction schemas, a specific proof strategy based on a specific proof form. These two strategies are correct and complete but the choice of the induction schema is made a priori by the user. Here we want to consider the initial logical system and to try to automatize such inductions choices.

3.1 A sequent calculus for AF_2

The previous system was formulated under a natural deduction formalism. Thus, a first step towards proof search methods consists in proposing an equivalent sequent calculus, denoted here G_{AF_2} that will be more adapted for automated proof search.

- Axiom:

$$\frac{}{\Gamma, x : A\ \vdash\ x : A}\ (ax)$$

- $\rightarrow$ rules:

$$\frac{\Gamma, x : A\ \vdash\ T : B}{\Gamma\ \vdash\ \lambda x.T : A \rightarrow B}\ (\rightarrow_r)* \qquad \frac{\Gamma\ \vdash\ M : A \quad \Gamma, y : B\ \vdash\ N : C}{\Gamma, x : A \rightarrow B\ \vdash\ N[y := (xM)] : C}\ (\rightarrow_l) **$$

(*) where x has no occurrence in Γ and (**) where y has no occurrence in Γ.

- $\forall_1$ rules

$$\frac{\Gamma, t : \forall x\ A, t_1 : A[y/x]\ \vdash\ M : C}{\Gamma, t : \forall x\ A\ \vdash\ M[t_1 := t] : C}\ (\forall_{1l})* \qquad \frac{\Gamma\ \vdash\ M : A}{\Gamma\ \vdash\ M : \forall x\ A}\ (\forall_{1r}) **$$

(*) where t_1 has no occurrence in Γ and $t_1 \neq t$, (**) if the first order variable x has no free occurrence in Γ.

- $\forall_2$ rules

$$\frac{\Gamma, t : \forall X\ A, t_1 : A[F/X]\ \vdash\ M : C}{\Gamma, t : \forall X\ A\ \vdash\ M[t_1 := t] : C}\ (\forall_{2l})* \qquad \frac{\Gamma\ \vdash\ M : A}{\Gamma\ \vdash\ M : \forall X\ A}\ (\forall_{2r}) **$$

(*) with $t_1 \neq t$ and t_1 has no occurrence in Γ, (**) if the second order variable X has no free occurrence in Γ

- Equality rules [1]

$$\frac{\Gamma\ \vdash\ M : A[v/u]}{\Gamma\ \vdash\ M : A}\ (=_r) \qquad \frac{\Gamma, t : A[v/u]\ \vdash\ N : C}{\Gamma, t : A\ \vdash\ N : C}\ (=_l)$$

if $u = v$ is an instance of an equation.

[1] In these rules we use the notation $A[v/u]$ to represent the substitution in the formula A of the first order term u by the first order term v

- Equivalence rules

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : B}\ (eq_r) \qquad\qquad \frac{\Gamma, t : A \vdash N : C}{\Gamma, t : B \vdash N : C}\ (eq_l)$$

if $A \equiv B$.

- Cut rule

$$\frac{\Gamma \vdash M : A \quad \Gamma, y : A \vdash N : C}{\Gamma \vdash N[y := M] : C}\ (cut)$$

where y has no occurrence in Γ.

- Structural rules

$$\frac{\Gamma, t : A, t_1 : A \vdash N : C}{\Gamma, t : A \vdash N[t_1 := t] : C}\ (cont)$$

where $t_1 \neq t$ and t_1 has no occurrence in Γ.

$$\frac{\Gamma, t : A \vdash N : C}{\Gamma, t : A, t_1 : A \vdash N : C}\ (weak) \qquad\qquad \frac{\Gamma, t : A \vdash N : C}{t : A, \Gamma \vdash N : C}\ (exch)$$

This sequent calculus is equivalent to the initial logical system, i.e.,

Theorem 3.1 $\Gamma \vdash_{AF2} M : A$ *if and only if* $\Gamma \vdash_{G_{AF2}} M : A$

Let us remark that we can naturally define, from this calculus, the notions of *tactic* and *proof plan* dedicated to this framework and thus proposed some plans to search proofs and extract programs from them [2, 3].

3.2 How to guide inductive proofs in G_{AF2}

When we want to develop inductive proofs in such a framework, some problems arise as: how to find at what place to apply induction ? on which variables to apply it ? and which induction schemas to apply ? In fact, we will restrict the form of equations such that we can find and order the places where induction is applied only with a study of the left-hand side members of the equations. Finally, it is not a significant restriction because we express the classical algorithms and functions with such a form. In this case, the first two mentioned problems have direct solutions and we will focus on the third one by looking for induction schemas to apply and for proof search strategies that can be associated to them.

3.2.1 Preliminary definitions

Definition 3.1 *Let* $t = f(t_1, ..., t_n)$ *be a term and* u *a subterm of* t*, the* place of u in t *is defined by* $pl(u,t) = i$ *if there exists* i *such that* $u = t_i$*,* $pl(u,t) = i.pl(u,t_i)$ *if* u *has an occurrence in* t_i*.*

Definition 3.2 *The* subterm u of term t in place q *is inductively defined by* $s\_term(t, 0) = t$*,* $s\_term(f(t_1, ..., t_n), i.q) = s\_term(t_i, q)$*.*

Definition 3.3 *The substitution of* z *by* t *in* $f(t_1, .., t_n)$ *at place* q*, denoted* $f(t_1, .., t_n)[t/z]_q$*, is defined by* $f(t_1, .., t_n)[t/z]_q$
$= f(t_1, ..t_{i-1}, t, t_{i+1}, ..., t_n)$*, if* $q = i$ *and* $s\_term(f(t_1, ..t_n), i) = z$*,*
$= f(t_1, ..., t_n)$*, if* $q = i$ *and* $s\_term(f(t_1, ..t_n), i) \neq z$*,*
$= f(t_1, ..t_{i-1}, t[t/z]_{q'}, t_{i+1}, ..., t_n)$*, if* $q = i.q'$*.*

We naturally extend this definition to the substitution of z by t in a formula F ($F[t/z]_q$).

3.2.2 Induction positions and induction variables

We consider here as specification a set of equations that have a restricted form and that are called *acceptable*. We do not give here formal definitions knowing that all classical functions and algorithms can be represented in such a system. In fact, an important result to notice is that the choice of places and variables to apply induction is direct in this case. Moreover, this form ensures the sufficient completeness of the function definition.

Definition 3.4 *Let us consider two places q_1 and q_2 in a term t, we have $q_1 \leq_a q_2$ if $q_2 = q_1.k$*

Definition 3.5 *A place q is an* induction position *in a term t if*
1. $s\_term(t, q) = c_T(t_1, ...t_n)$ with T data type,
2. for any i, $c_T(t_1, ...t_n)$ has no occurrence in t_i,
3. $q = inf\{p \ / \ s\_term(t, p) = c_T(t_1, ..., t_n)\}$.

Definition 3.6 *Let S be an acceptable specification of f ($g_i = d_i$, $1 \leq i \leq p$), the set of induction positions of the equation $g_i = d_i$ is $Pos_i(f) = \{q, \ q \ induction \ position \ of \ g_i\}$.*
The set of induction positions *of S is $Pos(f) = \cup_{i=1}^{p} Pos_i(f)$.*

Definition 3.7 *Let f a function with type formula $\forall z_1 T z_1 \rightarrow ......\forall z_n T z_n \rightarrow T f(z_1, ...z_n)$ with $Pos(f) = \{q_1, ..q_p\}$, the* set of induction variables *(denoted $Var(f)$) is $\{v_1, ...v_p\}$ with $v_i = z_i$ for $1 \leq q_i \leq r$ and $v_i = ?var$ (meta-variable) in the other cases.*

Thus, having a specification in AF_2 with the type formula F, we are able to determinate the list of the induction positions $pos\_list = [q_1, q_2, ..., q_n]$ and the list of the induction variables $v\_list = [v_1, v_2, .., v_n]$ with an algorithm that analyses the specification. Now we can focus our attention on the following question: which induction schemas to apply ?

3.2.3 Second order substitution

An essential feature of AF_2 is the *second-order specialization rule* which allows us to substitute a second-order variable (i.e., predicate variable) by either a predicate or a formula. This choice corresponds exactly to induction choice.
Let us start to illustrate with the addition example the importance of second order substitution and its effect on proof search success. If we want to prove *(3)* by using the iterative definition of integer data type we have $Nz \equiv \forall X(\forall y(Xy \rightarrow Xs(y)) \rightarrow (X0 \rightarrow Xz))(\equiv F)$.
To prove $\vdash \forall z \ (Nz \ \rightarrow \ (\forall z_1(Nz_1 \ \rightarrow \ Nplus(z, z_1))))$, we can apply all the possible right rules and then we have to prove $v : Nz, v_1 : Nz_1 \vdash \forall X(\forall y(Xy \rightarrow Xs(y)) \rightarrow (X0 \rightarrow Xplus(z, z_1)))$.
Therefore, we have $v : Nz, v_1 : Nz_1, f : \forall y(Xy \rightarrow Xs(y)), a : X0(\equiv \Gamma) \vdash Xplus(z, z_1)$.
We can start with $\Gamma \vdash v_1 : Nz_1$ and then develop Nz_1 to obtain
$\Gamma \vdash v_1 : \forall X_1(\forall y_1(X_1 y_1 \rightarrow X_1 s(y_1)) \rightarrow (X_1 0 \rightarrow X_1 z_1))(\equiv F_1)$.
Having to prove $\Gamma \vdash Xplus(z, z_1)$, we apply to F_1 the following second order substitution $F_1[Xplus(z, x)/X_1 x]$. In fact, we substitute a predicate $Xplus(z, \_)$ to the second order variable X_1 and obtain $\Gamma \vdash v_1 : \forall y_1(Xplus(z, y_1) \rightarrow Xplus(z, s(y_1)) \rightarrow (Xplus(z, 0) \rightarrow Xplus(z, z_1)$. It leads to an induction schema and prove, by induction on z_1, the basis case $Xplus(z, 0)$ and the induction case $\forall y_1(Xplus(z, y_1) \rightarrow Xplus(z, s(y_1)))$. Finally, the proof search is successful.

The choice of the second order substitution is essential and can lead to a success or a failure during the proof search process. For example, the substitution $F_1[Xplus(x,z_1)/X_1x]$ is not usable and the substitution $F[Xplus(x,z_1)/X_1x]$ leads to an induction schema that can be successful only if we have supplementary equations in the specification as $plus(x,0)=x$ and $plus(x,s(y))=s(plus(x,y))$. Another substitution in the previous proof $[\forall z(Nz \to Nplus(z,z_1))/X_1x]F_1$ leads to a new proof of the initial formula.

3.3 Induction schemas and proof search

The application of induction consists principally in the use of the *second order substitution rule* and the different forms of induction are derived from different substitutions. A first one consists of the maximum breakdown of the formula to prove (application of right rules) to obtain an atomic formula and is called *predicate induction* because it involves the substitution of a predicate to a second order variable. At the opposite, a second one consists of a minimum breakdown of the formula to prove and is called *formula induction* because it involves the substitution of a second order formula to a second order variable. Here we only present the rules corresponding to iteration and do not treat the recursion rules.

Definition 3.8 Predicate iteration.
A predicate iteration on the variable z at the q place consists of the application of the $(\forall_2\ l)$ rule of G_{AF2} with the following schema:

$$\frac{\Gamma, M : Tz, M : F[F_1/X_1x] \vdash N : Xf(t_1,\cdots,t_n)}{\Gamma, M : Tz, M : F \vdash N : Xf(t_1,\cdots,t_n)}\ (\forall_2\ l)$$

with X second order variable, $Tz \equiv \forall X_1(\Phi_T \to (X_1b_T \to X_1z))$, x new variable, z has a unique occurrence in $f(t_1,\cdots,t_n)$ at the q place and $F_1 \equiv Xf(t_1,\cdots,t_n)[x/z]_q$.

If we consider the previous example, we have applied, in the first case, the predicate iteration with the following instantiation

$$\frac{\Gamma, v_1 : Tz, v_1 : F[Xplus(z,x)/X_1x] \vdash t : Xplus(z,z_1)}{\Gamma, v_1 : Tz, v_1 : F \vdash t : Xplus(z,z_1)}\ (\forall_2\ l)$$

with $F \equiv \forall X_1(\forall y_1(X_1y_1 \to X_1s(y_1)) \to (X_10 \to X_1z_1))$.

Definition 3.9 Formula iteration
A formula iteration on the variable z at the q place consists of the application of the $(\forall_2\ l)$ rule of G_{AF2} with the following schema:

$$\frac{\Gamma, M : Tz, M : F[F_1/X_1x] \vdash N : F_2}{\Gamma, M : Tz, M : F \vdash N : F_2}\ (\forall_2\ l)$$

with $Tz \equiv \forall X_1(\Phi_T \to (X_1b_T \to X_1z))$, x new variable, $F_2 = ... \to Df(t_1,\cdots,t_n)$, z has a unique occurrence in $f(t_1,\cdots,t_n)$ at the q place and $F_1 \equiv F_2[x/z]_q$

If we consider the previous example, we have applied, in the second case, the formula iteration with the following instantiation

$$\frac{\Gamma, v_1 : Tz, v_1 : F[F_1/X_1x] \vdash t : Form}{\Gamma, v_1 : Tz, v_1 : F \vdash t : Form}\ (\forall_2\ l)$$

with $F \equiv \forall X_1(\forall y_1(X_1y_1 \to X_1s(y_1)) \to (X_10 \to X_1z_1))$, $F_1 \equiv \forall z(Nz \to Nplus(z,z_1))$ and $Form \equiv \forall z(Nz \to Nplus(z,z_1)$.

3.4 Inductive proof search strategies

Considering the previous induction schemas, we can define for each one a proof search strategy. Let us illustrate this point with the *predicate iteration*

Definition 3.10 *Let f be a function specified in AF_2 with F its type formula and such that $pos\_list = [q_1, q_2, .., q_n]$ and $v\_list = [v_1, v_2, ..., v_n]$, the* predicate iteration *proof strategy consists in proving F with an initial decomposition phase, n applications of* predicate iteration *on the variables $v_1, ..., v_n$ at places $q_1, ..., q_n$, followed by applications of right and left rules of the sequent calculus G_{AF2}.*

In fact, the predicate iteration is successful for the *addition* specification. It fails for the *predecessor* specification for which we have $pos\_list = [1]$ and $v\_list = [z]$. But if we consider a recursive definition of integers we obtain a success. The formula iteration fails also for the same reasons. In the case of *Ackermann* specification we have $pos\_list = [1, 2]$ and $v\_list = [z, z_1]$ and the predicate iteration proof strategy fails, even if we consider a recursive definition of integers. But we can show that the formula iteration proof leads to a success with iterative data type definition. In the case of *Fibonacci specification* the predicate iteration proof strategy fails, even with a recursive definition of integers. We have the same result for the formula iteration proof.

4 Failure cases for proof search strategies

The previous simple examples are only instances of specific specifications forms for which some proof strategies fail. In this section, we define various classes of function specifications such that a given proof strategy fails.

4.1 Functions with out-variables

Definition 4.1 *A variable y is a possibly out-variable in the term t if y has an occurrence in a subterm of t having the form $c_T(a_1, \cdots, a_r, y_1, \cdots y_s)$ with $c_T : A_1 \times \cdots A_r \times T \cdots \times T \rightarrow T$, c_T being a functional constructor associated to type T.*

Definition 4.2 *The variable y is a variable independent of f in term t if there exists a place q such that s_term(t,q) $= y$ and $\forall q' \geq_a q$, $s\_term(t, q') \neq f(\cdots)$.*

Definition 4.3 *Let f be a function defined with a system of p equations and the formula $\forall z_1\ T_1 z_1 \rightarrow \cdots \rightarrow \forall z_n\ T_n z_n \rightarrow f(z_1, \cdots, z_n)$, z_i is a out-variable for the equation e_m : $f(t_1, \cdots, t_n) = d_m$ if there exists y such that the following three conditions are verified: 1) y is a variable independent of f in d_m, 2) y is a possible out-variable of t_i, 3) the position of i is an induction position.*

The set of out-variables of the equation e_m for f will be denoted $VS_m(f)$ and the set of out-variables for f: $\cup_{i=1}^{p} VS_m(f)$ is denoted $VS(f)$.

In the case of the predecessor, in the second equation, the possible out-variable y is independent of p and thus $VS(p) = \{z\}$. If we consider the Ackermann specification, the first equation has no possible out-variable (no function constructor) and there is no independent variable of ack in the others and then $VS(ack) = \emptyset$.

Theorem 4.1 *Let f be a n-ary function such that pos_list $= [1,2,..,p]$ and $VS(f) \neq \emptyset$, let F be its type formula, if the data type definition is iterative then the proof search strategy by* predicate iteration *or by* formula iteration *applied to F fails.*

Proof 4.1 *see appendix A*

This result confirms that the predicate strategy fails for the *predecessor*.

4.2 Special recursive functions

Definition 4.4 *Let f be a function such that pos_list $= [1, 2, \cdots, p]$, it is transverse-recursive if an equation of its specification is transverse, i.e, has one of the following forms :*
(1) $f(c_1(\cdots), \cdots, c_j(\cdots), b_{j+1}, z_{j+2}, \cdots, z_n) =$
$f(c_1(\cdots), \cdots, c_{j-1}(\cdots), y_{j_i}, b_{j+1}, z_{j+2}, \cdots, h(\cdots, z_k, \cdots), \cdots, z_n)$ *or*
$f(c_1(\cdots), \cdots, c_{j-1}(\cdots), y_{j_i}, h(\cdots, b_{j+1}, \cdots), z_{j+2}, \cdots, z_n)$.
(2) $f(c_1(\cdots), \cdots, c_p(\cdots), z_{p+1}, \cdots, z_n) = f(c_1(\cdots), \cdots, c_{p-1}(\cdots), y_{p_i}, z_{p+1}, \cdots, h(\cdots, z_k, \cdots), \cdots, z_n)$
with h function soon defined.

Definition 4.5 *An equation of f is p-decreasing with $p \geq 1$ if it has one of the following forms:*
$f(c_1(\cdots), \cdots, c_j(\cdots), u_{j+1}, z_{j+2}, \cdots, z_n) = g(\cdots, f(v_1, \cdots, v_{j-1}, y_{j_m}, u_{j+1}, z_{j+2}, \cdots, z_n), \cdots)$ *or*
$f(c_1(\cdots), \cdots, c_j(\cdots), u_{j+1}, z_{j+2}, \cdots, z_n) = f(v_1, \cdots, v_{j-1}, y_{j_m}, u_{j+1}, z_{j+2}, \cdots, z_n)$
with for any i from 1 to j-1, $v_i = c_i(\cdots)$ or y_{i_k}, g defined function and p the number of v_i such that $v_i = y_{i_k}$.
A function f is a multi-decreasing function if an equation of its specification is p-decreasing with $p \geq 1$.

The decreasing set of the equation e_m for f will be denoted $MD_m(f) = \{z_k / l+1 \leq k \leq j\}$ and the decreasing set of f is $\cup_{i=1}^{p} MD_m(f)$ is denoted $MD(f)$.

Theorem 4.2 *Let f be a n-ary function such that pos_list $= [1,2,..,p]$ and $MD(f) \neq \emptyset$ and let $F \equiv \forall z_1\ T_1 z_1 \rightarrow ....\forall z_n\ T_n z_n \rightarrow Tf(z_1, ...z_n)$ be its type formula, the proof search strategy by* formula iteration *applied to F fails*

Definition 4.6 *An induction variable is* recursive *if it is an out-variable or if it belongs to $MD(f)$. The set of recursive variables for f is $Rec(f) = VS(f) \cup MD(f)$.*

Theorem 4.3 *Let f be a n-ary function such that pos_list $= [1,2,..,p]$ and $Rec(f) \neq \emptyset$ and let $F \equiv \forall z_1\ T_1 z_1 \rightarrow ....\forall z_n\ T_n z_n \rightarrow Tf(z_1, ...z_n)$ be its type formula, the proof search strategy by* formula iteration *applied to F fails*

Definition 4.7 *A function f is a multi-recursive function if an equation of its specification has one of the following forms: $f(c_1(\cdots), \cdots, c_j(\cdots), u_{j+1}, z_{j+2}, \cdots, z_n) = g(\cdots, f(t_1, \cdots, t_n), \cdots)$ or $f(c_1(\cdots), \cdots, c_j(\cdots), u_{j+1}, z_{j+2}, \cdots, z_n) = f(t_1, \cdots, t_n)$ with f having at least an occurrence in a term t_i and g defined function.*

Definition 4.8 *A function f is a special recursive function if it is transverse-recursive or multi-decreasing or multi-recursive.*

The Ackermann function is *multi-recursive* because of the equation $ack(s(x), s(y)) = ack(x, ack(s(x), y))$ and thus it is special recursive.

Theorem 4.4 *Let f be a n-ary function multi-recursive such that $pos\_list = [1, 2, .., p]$ and $v\_list = [z_1, z_2, .., z_p]$ and let $F \equiv \forall z_1\ T_1 z_1 \to ....\forall z_n\ T_n z_n \to Tf(z_1, ...z_n)$ be its type formula, the proof search strategy by* predicate iteration *applied to F fails.*

4.3 Complex recursive functions

Definition 4.9 *A function f is a complex recursive function of level 2 at position $q.i$ if there are two base cases for a same induction position, i.e., three equations of the forms :*
$f(t_1, \cdots, t_{q-1}, b_D, t_{q+1}, \cdots, t_n) = G$,
$f(t_1, \cdots, t_{q-1}, c_D(a_1, \cdots, a_r, y_1, \cdots, y_{i-1}, b_D, y_{i+1}, \cdots, y_s), t_{q+1}, \cdots, t_n) = H$,
$f(u_1, \cdots, u_{q-1}, c_D(a_1, \cdots, a_r, y_1, \cdots, y_{i-1}, c_D(a'_1, \cdots, a'_r, y'_1, \cdots, y'_s), y_{i+1}, \cdots, y_s)\ u_{q+1}, \cdots, u_n) = I$*, with at most an index j such that $u_j \neq t_j$.*

Definition 4.10 *The set of complex positions of f is defined by Comp(f) =*
{ $q.i$ such that $q \in pos\_list$ and et f is complex recursive of level 2 at the position $q.i$ }

The fibonacci function is *complex recursive* at the position 1.1

Theorem 4.5 *Let f be a n-ary function, complex recursive at level 2 at the position $q.i$, such that $pos\_list = [1, 2, .., p]$ and $v\_list = [z_1, z_2, ..., z_p]$ and let $F \equiv \forall z_1\ T_1 z_1 \to ....\forall z_n\ T_n z_n \to Tf(z_1, ...z_n)$ be its type formula, the proof search strategy by* predicate iteration *or by* formula iteration *applied to F fails*

5 Other induction schemas

In the example of the predecessor, we have a failure of the predicate iteration strategy (with iterative data types) because we have no derivation of the judgement $\Gamma, Xp(y) \vdash Xy$. But when we consider the recursive definition of integers, the type information about the out-variable y involved in the hypothesis Ny can lead to a success in the proof search. It leads to another induction schema, called *predicate recursion*, that corresponds to the definition of predicate iteration only modified with $F_1 \equiv Xf(t_1, .., t_n)[x/z]_q \wedge Xx$.
We can prove, as in the previous section, that a proof strategy based on *predicate recursion* fails in the both cases of special recursive and complex recursive functions. Thus we can apply it to the predecessor but it does not solve the other examples.
With the same approach, we can propose another induction schema, called *formula recursion*, that corresponds to the definition of formula iteration only modified with $F_1 \equiv Form[x/z]_q \wedge Dx$. With a proof strategy based on this induction schema, we can solve the addition, predecessor and Ackermann examples. But it cannot solve the failure concerning the complex recursive functions because of the impossibility to use the given equations to find a correspondence between the conclusion and the induction hypothesis. Some work have considered this problem and proposed some methods to sove it [4]. But in our work we want to consider the potentialities of the second order substitution to solve this problem and consequently we propose new induction schemas corresponding to case induction or recursion.
If we consider that a function can be at the same time complex recursive and/or special recursive

and/or can have out variables we thus can propose four new induction schemas that are *case predicate or formula / iteration or recursion*. If we apply a proof strategy based on the induction schema named case predicate iteration, we obtain a successful proof search for the fibonacci function.

6 A procedure for efficient inductive proof search

Starting from the previous results on induction, we propose here a procedure for choice of induction that take in charge some choices about finding and ordering induction positions and variables and about choosing the induction mode for each induction position.

With such a procedure we have a main goal that is to avoid the failure cases we have found, that means: to avoid predicate induction for special recursive function, to avoid iteration for a out-variable when data types are iteratively defined, to avoid formula and predicate induction for a complex induction position. In fact, we can also propose heuristics for the choices among the possible induction, if we focus on the problem on the efficiency of the extracted programs.

6.1 Principles of the procedure

The aim of this procedure is to fix some choices concerning the induction schemas. In fact, it takes the specification and gives as results the instantiations of the four following lists:

- pos_list: list of the induction positions,
- v_list: list of the induction variables,
- f_list: list of the formula inductions to apply,
- p_list: list of the predicate inductions to apply.

It corresponds to two main phases, i.e., (1) the choice and ordering of induction positions and variables, (2) the choice of the adequate induction form for each induction variable.

Let us illustrate these phases with the example of the addition. The procedure of selecting and ordering the induction positions and variables leads to $pos\_list = [2]$ and $v\_list = [z_1]$.

To instantiate f_list and p_list, we analyze the criteria studied in the previous section as

- special recursive function or not,
- recursive variables or not ($Rec(f)$ empty or not),
- complex induction positions ($Comp(f)$ empty or not),
- definition of the data types (iterative or recursive).

We obtain, with these four ordered criteria, sixteen cases to instantiate the two induction lists. For example, if f is special recursive, we reject the predicate induction and thus $p\_list = []$. Moreover, in this case, if there are no out variables, if there are complex induction positions and if the data definition is iterative, we have $f\_list = [r_1, ..., r_p]$ where $r_i = iteration$ if $Pos.j \notin Comp(f)$ and $r_i = case\_rec(j)$ if $Pos.j \in Comp(f)$

The complete procedure, including all the analysis case will be given in the full paper.

In the case of our example, $plus$ is not a special recursive function and $Rec(f) = Comp(f) = \emptyset$ and thus we have $f\_list = []$ and $p\_list = [iteration]$ for any data type definition.

6.2 Application of the procedure

It is possible to illustrate the interest of the procedure to determinate the adequate induction schemas for program synthesis in AF_2 but also for proving program property, we consider the proof search of a given conjecture or property that involves the definition of the addition, the

multiplication and a particular function f and such that automated proof search with other methods involves eureka steps and failure cases.
Let us consider at first the specification of the addition on integers. Moreover, we assume that the two following properties have been proved: $\forall z\ (Nz\ \rightarrow\ \forall z_1\ (Nz_1 \rightarrow\ \forall z_2\ Nz_2 \rightarrow (plus(z, plus(z_1, z_2))\ =\ plus(plus(z, z_1), z_2))))$, i.e., the addition function is associative and $\forall z\ (Nz\ \rightarrow\ plus(z, 0))\ =\ z$. We consider also the multiplication specification: $mult(0, y)\ =\ 0$, $mult(s(x), y)\ =\ plus(y, mult(x, y))$ and
$\forall z\ (Nz\ \rightarrow\ (\forall z_1(Nz_1\ \rightarrow\ Nmult(z, z_1))))$
Starting from the previous definition, we define the function f as follows
$f(0, y, z)\ =\ z,$
$f(s(x), y, z)\ =\ f(x, y, add(y, z))and$
$\forall z\ (Nz\ \rightarrow\ (\forall z_1(Nz_1\ \rightarrow\ \forall z_2(Nz_2\ \rightarrow\ Nf(z, z_1, z_2))))).$
We consider now the following conjecture to prove: $f(z, z_1, z_2)\ =\ plus(z_2, mult(z, z_1))$.
It means that in our logical framework we have to prove

$$\forall z\ (Nz \rightarrow\ \forall z_1(Nz_1 \rightarrow\ \forall z_2(Nz_2\ \rightarrow\ (f(z, z_1, z_2)\ =\ plus(z_2, mult(z, z_1))))))$$

We apply the induction choice procedure to the three functions $plus$, $mult$ and f and obtain the following proposal:

- for the function *plus*, it suggests to perform predicate iteration on the first variable i.e. z since this function is not a special recursive one nor a complex recursive one and has no out variables.

- for the function *mult*, it suggests to perform predicate iteration on the first variable i.e. z for the same reason.

- for the last one f, it suggests to perform formula iteration on the first variable i.e. z since f is a special recursive function.

Taking into account the result of the induction choice procedure for the three functions occurring in this conjecture and the order over the induction schemes, it proposes to prove it by formula iteration on the first variable i.e. z.
This involves a given proof form and also a proof search strategy, including this induction choice, that leads to a successful automated proof. The complete proof will be given in the full paper.

7 Concluding remarks

In a restriction of AF_2 with recursive types, [6] has proposed for two induction schemas a specific proof strategy based on a specific proof form that is correct and complete but the choice of the induction schema is made a priori by the user. In our work, we propose an analysis of the specification to take in charge this problem of induction choice and thus we propose an adequate induction schema to instantiate the induction step in the proof search process. In a certain way, our analysis allows to generate a proof strategy based on a specific proof form (including an adequate induction) for different kinds of function specifications. An important further work could be to refine our method by fixing a priori proof strategy and specification forms and thus to prove the completeness of the former w.r.t of the latter.
Most theorem provers rely upon user interaction to identify and patch failures. Our work attempts to automatize this process following an original approach. The method depends on

theoretical results of failure of induction in a second order logical framework and allows to select the induction rule in order to avoid these failure cases. In such a framework we have established a taxonomy of recursively defined functions w.r.t induction schemas failures. Our study initially dedicated to program synthesis in AF_2 appears as a nice method to prune the proof search tree in inductive proof and the use of second order substitution allows to avoid some eureka steps that appears when we develop similar proof in first order framework.
To extend these results to another conjecture forms we need also to deal with failures due to the lack of some lemmas. This problem has already been tackled in other works and would be necessarily considered in our global approach.

References

[1] A. Bundy, F. Van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1991.

[2] D. Galmiche and O. Hermann. *Automated proof and program development*, volume 35 of *IMA Conference Series*, pages 397–410. Oxford University Press, 1992.

[3] D. Galmiche and O. Hermann. SKIL: A system for programming with proofs. In *LPAR'93, International Conference on Logic Programming and Automated Reasoning, LNAI 698*, pages 348–350, St. Petersburg, Russia, July 1993.

[4] D. Hutter. Guiding induction proofs. In *10th Conference on Automated DEduction, LNAI 449*, pages 132–146, 1990.

[5] D. Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44:51–68, 1986.

[6] P. Manoury and M. Simonot. Automatizing termination proofs of recursively defined functions. *Theoretical Computer Science*, 135(2):319–343, 1994.

[7] M. Parigot. Programming with proofs : a second order type theory. In *European Symposium On Programming 88, LNCS 300*, pages 145–159, Nancy, France, March 1988. Springer Verlag.

[8] M. Parigot. On the representation of data in lambda-calculus. In *CSL '89, LNCS 440*, pages 309–321, Kaiserslautern, FRG, October 1989.

[9] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, March 1992.

[10] C. Raffali. Data types, infinity and equality in system AF2. In *7th Workshop CSL'93, LNCS 832*, pages 280–294, Swansea, UK, september 1994.

A Proof of theorem 4.1.

Theorem A.1 *Let f be a n-ary function such that $pos\_list = [1, 2, .., p]$ and $VS(f) \neq \emptyset$, let F be its type formula, if the data type definition is iterative then the proof search strategy by* predicate iteration *or by* formula iteration *applied to F fails.*

Proof A.1 *Let $z_i \in VS(f)$.*
1- *Let us consider the equation e_m having the form :*
$f(c_1(\cdots), \cdots, c_i(a_{i1}, \cdots, a_{ir_i}, y_{i1}, \cdots, y_{is_i}), \cdots, c_{m-1}(\cdots), b_m, z_{m+1}, \cdots, z_n) \ = \ y_{ij}$
We can prove that after the steps of initial decomposition and of p applications of predicate iteration *a leaf of the proof tree has the form: $\Gamma_m \ \vdash \ Xf(c_1(\cdots), \cdots, c_{m-1}(\cdots), b_m, z_{m+1}, \cdots, z_n)$ reduced by application of the $(= \ r)$ (rewriting phase) to $\Gamma_m \ \vdash \ Xy_{ij}$.*
The proof of this sequent fails because:
- no right-rule is applicable,
- there is no hypothesis of the form Xy_{ij} and thus the leaf is not an axiom,
- there is no hypothesis of the form $\Theta \ = \ \forall z \ F$ with $F \equiv ... \rightarrow Xz$ and thus the rule $(\forall_1 \ l)$ cannot be applied,
- there is no hypothesis of the form Dy_{ij} with D data types and thus $(eq \ l)$ cannot be applied.
The failure reasons are the same if the equation e_p has the form :
$f(c_1(\cdots), \cdots, c_p(a_{p1}, \cdots, a_{pr_p}, y_{p1}, \cdots, y_{ps_p}), z_{p+1}, \cdots, z_n) \ = \ y_{pj}$.

2- *Let us assume that the equation e_m has the form:*
$f(c_1(\cdots), \cdots, c_i(a_{i1}, \cdots, a_{ir_i}, y_{i1}, \cdots, y_{is_i}), \cdots, c_{m-1}(\cdots), b_m, z_{m+1}, \cdots, z_n) = \ g(t_1, \cdots, t_l)$
with $pl(y_{ij}, g(t_1, \cdots, t_l)) \ = \ q$.
We can easily prove that there exists a derivation of $\vdash F$ such that one of the leaves is:
$\Gamma_m \ \vdash \ Xf(c_1(\cdots), \cdots, c_{m-1}(\cdots), b_m, z_{m+1}, \cdots, z_n)$
that is reduced, by application of the $(= \ r)$ rule (rewriting phase), to $\Gamma_m \ \vdash \ Xg(t_1, \cdots, t_l)$
We can only apply the (cut) rule that produces the leaves $\Gamma_m \ \vdash \ Xt_i$ pour i de 1 à l.
Let us consider two cases : $q = i$ (y_{ij} has an occurrence at depth 1) or $q = i.k$ (y_{ij} has an occurrence at depth greater than 1).
In the first case, there is no complete extension of $\Gamma_m \ \vdash \ Xy_{ij}$ as in 1).
Let us assume that the proof fails if y_{ij} has an occurrence at deep $t-1$, and show that the proof fails if y_{ij} is a subterm of $g(t_1, \cdots, t_l)$ at depth 1.
If y_{ij} is a subterm of $g(t_1, \cdots, t_l)$ at depth t then y_{ij} is a subterm of t_i at depth $t-1$ and by induction hypothesis the proof of $\Gamma_m \ \vdash \ Xt_i$ fails.
The failure reasons are the same if the equation e_p has the form
$f(c_1(\cdots), \cdots, c_p(\cdots), z_{p+1}, \cdots, z_n) \ = \ g(t_1, \cdots, t_l)$ *with y_{ij} variable independent of f in $g(t_1, \cdots, t_l)$.*