The Unified Computation Laboratory, Modelling, Specifications and Tools
C M I Rattray & R G Clark (eds)

Automated proof and program development

D. Galmiche and O. Hermann
CRIN - CNRS - Université Nancy I
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex
France.

Abstract

The aim of this paper is to present proof and program development in intuitionistic logic and attempt to automating this with tactics and strategies. It is important in computer science to have correct programs and much work has been devoted to this aspect. Typically, the study of the relationships between intuitionistic logic and computer science leads to the *programming with proofs*, i.e., extracting (correct) programs from proofs. We present program and proof developments using suitable tactics or tacticals with the aim of automating the proof construction and extracting interesting and efficient programs from proofs. Proof (and program) development examples are given showing the relationships between data representations, programs and proofs. Moreover, some remarks about related work and possible future research are given.

1 Introduction

For many years a lot of work has been devoted to the study of program development in logical framework using techniques based on program transformation [17], or on theorem proving [11]. Here we consider the latter with a particular approach which is *programming with proofs* using constructive (intuitionistic) logic and its applications to the activity of programming [5].
Different theories related to this technique have been discussed in some theoretical works [8, 12]. Programming in such frameworks consists of giving a formal specification (or proposition in the system), constructing its proof and then extracting executable code systematically from this proof with notions like formulae-as-types or realizability interpretation. Some mathematical results give important properties such as correctness and termination of programs obtained in this way [10].
It appears necessary to consider formal systems adapted for developing proofs but it is not sufficient simply to have a good logical framework in which to derive programs as proofs. Some problems are yet to be solved, such as those concerned with efficiency and the corresponding relationships between proofs and programs. For instance, better proofs are not necessarily better from an algorithmic point of view and efficient algorithms cannot be expressed in the calculus we consider. To

solve such problems we have to develop elaborated versions of the logical system and to study connections between data representations, specifications, programs and proofs. The framework, at the center of the study, is based on work on second order intuitionistic logic [14]. We want to define tactics and tacticals of proofs [16] leading to efficient programs in this particular framework but with a higher level of automation and to make comparisons of proof and program construction with those of other frameworks [4].
Section 2 gives a concise presentation of the logical framework needed for the derivation of proofs and correct programs, and introducts to the language (a second order λ-calculus) and the proof rules. Section 3 recalls what programming with proofs means, in particular, in this framework, explains the importance of data representation and illustrates this by an example. Section 4 presents the notion of automation of proof developments and, more precisely, the tactics and tacticals which seem adequate for proof development. A specific method for treating proofs with recursive data structures is presented. An example is developed using these methods to show how different proof strategies correspond to different algorithms. Section 5 presents related work and possible future research in this framework; section 6 contains the conclusions to the paper.

2 The logical framework

We present here a second order typed λ-calculus, i.e, a natural deduction system for intuitionistic second order logic with extraction of λ-terms (considered as programs) from constructive proofs [14].
The terms of this λ-calculus are obtained from variables $x, y, ..z$ by a finite number of applications of the following rules:
1. if t and u are terms then $(t)u$ is a term (another notation is $ap(u,t)$).
2. if x is a variable and t is a term then $\lambda x.t$ is a term.
The notion of computation for terms is *reduction* and the λ-calculus can be transformed into a programming language by implementation of the reduction.
The formulae (or types) of the second order logic are constructed using the connective $\rightarrow$, the quantifier $\forall$, individual variables x, y, z, predicate variables of arbitrary arity X, Y, Z, predicate, function and individual constants. The individual terms are constructed from functions and individual constants. We must note that in second order (intuitionistic) logic, the logical symbols $\perp, \vee, \wedge, \neg$ and $\exists$, as well as the identity relation $=$, are definable from $\rightarrow$ and $\forall$. For instance, $A \wedge B$ is defined by the formula $\forall X((A \rightarrow (B \rightarrow X)) \rightarrow X)$.
The expressions of the language have the form $x : X$ (x is a term of type X) with the following interpretations : x is an element of type X or x is a proof of the proposition X. It is the latter that is used in this framework and formulae are considered as types and deductions as λ-terms. The construction of deductions is based on the application of axioms and inference rules. The rules of proof for intuitionistic second order logic are the following (A, B denote formulae and Γ sequences of formulae):

R1. Axiom. $x : A \vdash x : A$

R2. Abstraction rule. if $\Gamma, x : A \vdash t : B$ then $\Gamma \vdash \lambda x.t : A \to B$

R3. Application rule. if $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : A$ then $\Gamma \vdash ap(u,t) : B$

R4. Generalization rule. if $\Gamma \vdash t : A$ then $\Gamma \vdash t : \forall x A$ (x not free in Γ)

R5. Specialization rule. if $\Gamma \vdash t : \forall x A$ then $\Gamma \vdash t : A[b/x]$ (b individual term)

R6. Generalization rule 2. if $\Gamma \vdash t : A$ then $\Gamma \vdash t : \forall X A$ (X not free in Γ)

R7. Specialization rule 2. if $\Gamma \vdash t : \forall X A$ then $\Gamma \vdash t : A[B/X]$ (B formula)

R8. Equivalence rule. if $\Gamma \vdash t : A$ then $\Delta \vdash t : A$
where Δ is a sequence obtained from Γ by permutation, extension or contraction.

The system is very simple, with $\to$ the only logical symbol having an algorithmic content. The quantifications just have a conceptual role allowing descriptions of what the programs do.
From a programming point of view the expression *t is of type A* which links a term with a formula, has to be read as *the program t realizes the specification A*. Terms of the λ-calculus represent programs (in the λ-calculus machine language) whereas formulae represent specifications in a high level language. The most important properties of terms obtained from proofs are *type preservation* (the type is conserved by reduction) and *termination* (the terms obtained from proofs represent algorithms which always terminate). This system can be considered as a programming language allowing us to write exact specifications and correct programs, and λ-calculus is considered as object code into which proofs are compiled. For more details about the semantic notion of type and justification of the programming method see [10].

3 Programming in the logical framework

The study of relationships between intuitionistic logic and computer science is at the center of some theoretical work and some systems have been implemented following the paradigm *programs as proofs* in these different logical formulations [2, 8, 12]. We investigate the notion of *programming with proofs* in this second order framework and, particularly, the data types and their relationships with proof developments.

3.1 Programming with proofs

Considering the above framework the construction of programs is broken down into three principal steps:

1. Expressing data types by formulae. We extract from the second order definition of the data type a representation of the data in lambda-calculus which is very important for the implementation.
   For instance, for the type of lists of elements of type A, we introduce constructors for the type; the individual constant *nil* and a function constant *ad* (for the adjunction) and consider a formula "*x is a list*" expressing a second order definition of lists (the smallest set containing *nil* and closed for the *ad* operation): $L(A)x \equiv \forall X(\forall y \forall z(Ay \rightarrow (Xz \rightarrow Xad(y,z))) \rightarrow (Xnil \rightarrow Xx))$.

2. Expressing the specification of the program by a system of equations, semantically defining the function to be computed.
   For instance, for the concatenation of lists, we could write the equations as *conc(nil,x) = x* and *conc(ad(y,x), z) = ad(conc(y,z), x)*.

3. The program is given by a proof of the proposition saying that the function has the intended type. For instance, the proposition saying that if x and y are lists then the concatenation of x and y (*conc(x,y)*) is a list.

What is mentioned here is the general construction scheme (specification with formulae, proof construction and program extraction from this proof) adapted to this framework [14]. We know that efficient algorithms are not always expressible in the λ-calculus or that better proofs do not produce better algorithms. That is why we want to study, before possible extension of the theory, the relationships between data representation and extracted program.

3.2 Data representation

We can analyze the algorithmic properties of programs induced by the choice of the representation of data structures in λ-calculus. Here we have two classical ways to define data types: the iterative one and the recursive one define the same mathematical object but have different algorithmic content. To derive interesting proofs and algorithms we have to choose some adapted data representations. For instance, the type $L(A)x$ of lists of elements of type A is defined by the formula $L(A)x \equiv \forall X(\forall y \forall z(Ay \rightarrow (Xz \rightarrow Xad(y,z))) \rightarrow (Xnil \rightarrow Xx))$ which say that x is an element of each set X containing the empty list *nil* and closed under the binary operation *ad* of concatenation of an element of type A to an element of type X.
By proof, we can obtain the representations of the constructors in the logical framework: $\lambda f \lambda a.a$ of type $L(A)nil$ and $\lambda n \lambda v \lambda f \lambda a.ap(ap(f,n), ap(ap(v,f),a))$ of type $\forall y \forall z(Ay \rightarrow (L(A)z \rightarrow L(A)ad(y,z)))$.
This definition of type $L(A)x$ is the iterative one. Compared to other theories [12], the representation (or implementation) as λ-terms of data in the machine language is given by the definition of the type.

3.3 Proof example

We have chosen a simple example to illustrate our approach: the construction of the term *length* program which calculates the length of a list of elements of type A. To do this in the theory we have to prove that if x is a list of elements of type A then the length of x is a natural number, i.e., $\forall x L(A)x \to Il(x)$, where l defines the length of a list, with the following axioms: $l(nil) = 0$ and $l(ad(a, y)) = s(l(y))$; I is the formula for the representation of naturals.
The set of naturals, defined by iteration, is introduced by an individual constant *0*, a function constant s and by considering a formula Ix that says that x is a natural number: $Ix \equiv \forall X(\forall y(Xy \to Xs(y)) \to (X0 \to Xx))$.
A first step in the proof is to consider the iterative definition of the type list and to develop a proof by induction. For that we must consider the following hypotheses: $\varepsilon : L(A)x$, $f : \forall y(Xy \to Xs(y))$, $a : X0$; then we search for a term of type $Xl(x)$ and then prove $Xl(x)$ from the hypotheses. This proof is developed by induction on x. Consequently, we search for a proof U of the formula $Xl(nil)$ and for a proof V of the formula $\forall a \forall y(Aa \to (Xl(y) \to Xl(ad(a, y))))$.
The construction process gives $U \equiv a$ and $V \equiv \lambda d.f$ and the final term (or proof) is $\lambda\varepsilon\lambda f\lambda a.ap(ap(\varepsilon, \lambda d.f), a)$, which is the term that computes the length of a list.
Through examples like this one, we can have the impression that the construction of programs is hard and complicated (perhaps because of the derivation of a proof and a program at the same time). Moreover, to extend the possibilities of the framework, it is necessary to allow other representations of types such as recursive types to obtain new algorithms.
For instance, the definition of the recursive version of the type list is the following: y of type $L(A)x$ ($y : L(A)x$) is considered as the minimal solution K of the equation $Kx = \forall X(\forall y \forall z(Ay \to (Kz \to Xad(y, z))) \to (Xnil \to Xx))$. We will illustrate the use of such data types and their effects in proofs and discuss new possible data representations in following sections.

4 Automated proof development

In this logical framework, we are able to develop some proofs but our principal aim is to automate a maximum number of steps of a development. A first way is to break down the goal into subgoals using simple rules, but some problems such as the following appear: how to use the rules, in what order, in what direction ? Moreover, are some proofs similar ? What does being similar means ? By studying the above questions we have to characterize tactics and strategies for adequate proof derivations.

4.1 Tactics

To automate the proof development some tactics are introduced. In this system, the reasoning is goal-oriented by applying the rules of the calculus bottom-up. The

notion of *tactic* is defined as a single step to reduce a problem into subproblems (by application of some rules) and a first step to do that is the application of calculus rules. For example, by application of the generalization rule on variables, to prove ∀x.A in a context Γ it is sufficient to prove A in this context if x not a free variable in Γ. Moreover, a formula A → B can be proved by the abstraction rule in the context Γ if we can prove B in the context Γ ∪ {x:A}.
To reduce a goal into subgoals and to orient the proof search in a certain context we need other tactics (than the one issued from the R_i rules). In this particular framework, we have found tactics similar to those in other logical frameworks, like Isabelle [16], e.g.,*assume-tac* and others more specific to second order logic, e.g.,*develop*. Among these tactics we mention the following :

- tactic *assume-tac*
  From the rule R1 before searching for a proof we check that it does not appear in the context.

- tactic *add-hypo*
  The aim is to modify the context before searching for the proof. For example, to prove A → B it is sufficient to prove B in the context resulting from the addition of a proof of A in the initial context. In the same way, to prove ∀x.A or ∀X.A it is sufficient to have a proof of A.

- tactic *develop*
  Considering the length of the definition of types (data structures), we have the choice of using a formula in either a developed form or a concise form. This tactic allows the replacement of a concise formula by the actual and developed form.

- tactic *deduction*
  During the derivation of a proof it is possible that it does not directly appear in the context (then the above tactic cannot be applied), but having all the possible information we can apply the deduction tactic. The effect is to develop a possible application inside the context to obtain an extended context which may include the proof.

- tactic *rewrite*
  We name rewriting of a term *t* in a term *t'* using axiom *ax*, the substitution of the term *t* by the term *t'* by application of $ax : t = t'$. This tactic applies a possible rewriting in the current context to obtain a new context.

- tactic *lemma*
  This tactic allows the use of lemmas which have been proved already by consulting the set of proved formulas.

- tactic *substitution*
  The tactic is applied to a context, returning a new context as the result which can be used in different ways : substituting a given term for another (in the

context); substituting terms of a given scheme for terms of another scheme. This seems to be one of the most important tactics and also one of the most difficult to handle. It is important because, in proofs other than the trivial proofs that are only elementary deductions from hypotheses, it becomes essential to take the universal quantification into account. It is hard because of the diversity of substitutions which can be applied.

These tactics represent elementary steps in the proof development and they will be combined to obtain a complete proof. Such combinations lead to the notion of strategies.

4.2 Tacticals

We define the notion of tacticals (or meta-tactics) as the composition of tactics in classical way for logical frameworks [6, 15]. The tacticals are essential for such theorem provers and allow the combination of some results (of elementary proof steps) for automated proof constructions. In this second order framework, we see different proof plans depending on the data representations.
With these we obtain a partially or globally automated derivation of proof. For instance, we mention the *Repeat tactic* the effect of which is to repeat *tactic* until it is no longer applicable (*tactic* can be instantiated by the above tactics), or *tactic1 Then tactic2* which is a sequential application of two tactics. Such operators can also be applied to strategies and consequently considered as meta-strategies. This leads us to consider meta-logic to represent the framework.
For the strategies associated with data representation it is important to note that the two principal representations of a type are the recursive one and the iterative one, and that proofs of the specification are different in these two cases. Moreover, the automation aim leads to specific methods for each representation.

1. For the iterative data structures we apply sequentially, globally or partially the following tactics :

- *develop* to develop concise formulae,
- *add-hypo* to treat context,
- *substitution* to make adequate substitutions,
- *rewrite* to make possible rewritings in the context,
- *lemma* to use lemmas,
- *deduction* to make all possible deductions in the context.

This schema is available for all the iterative data (like naturals, booleans, lists or trees) and it is possible to extract explicit proof plans applicable to different data [1].

2. For the recursive data structures, we manipulate at the same time formulae in concise and developed form, and to obtain good proofs (corresponding to good algorithms), we have to know when and where to use which of these forms. That is why we propose a specific way of treating proofs using recursive data definitions

when automating proof constructions. It consists of the association of a specific operator to each data type to obtain a formulation in Martin-Löf style [12]. It is important to note that, within constructive type theory, it is possible to derive programs from the same form of specification to obtain recursive forms of programs.

4.3 Development of proofs

What we have presented in the above section concerning tactics and tacticals is illustrated by the following examples. We consider the construction of the same proof as in the section 3, using adapted tactics or strategies.
Let us recall we want to prove $\forall x L(A)x \rightarrow Il(x)$ with the above axioms in the theory: $l(nil) = 0$ and $l(ad(a, y)) = s(l(y))$.

4.3.1 Iterative proof plan

The following version consists of the application of the strategy associated with the category of **iterative** data structures i.e. successively the tactics *develop, add-hypo, substitute, rewrite and deduction.*
By *Repeat develop* we replace all the concise formulas by the developed form and then the formula to prove consists of :
$\forall X(\forall a \forall y(Aa \rightarrow (Xy \rightarrow Xad(a, y))) \rightarrow (Xnil \rightarrow Xx)) \rightarrow$
$\forall X(\forall y(Xy \rightarrow Xs(y)) \rightarrow (X0 \rightarrow Xl(x)))$.
By *Repeat add-hypo* (which constructs the context of the proof by application of rules R2, R4, R6) we have to prove $Xl(x)$ in the context:
$\varepsilon : (Aa \rightarrow (Xy \rightarrow Xad(a, y))) \rightarrow (Xnil \rightarrow Xx)$,
$f : Xy \rightarrow Xs(y)$,
$a : X0$.
By *substitution* (which applies adequate substitutions in the precedent context i.e. which replaces all expressions of the form $X\_$ except $X0$ by $Xl(\_)$) we have to prove $Xl(x)$ in the new context:
$\varepsilon : (Aa \rightarrow (Xl(y) \rightarrow Xl(ad(a, y)))) \rightarrow (Xl(nil) \rightarrow Xl(x))$,
$f : Xl(y) \rightarrow Xs(l(y))$,
$a : X0$.
By *Repeat rewrite* (which applies a possible rewriting using the axioms in the precedent context) we have to prove $Xl(x)$ in the following context:
$\varepsilon : (Aa \rightarrow (Xl(y) \rightarrow Xs(l(y)))) \rightarrow (X0 \rightarrow Xl(x))$,
$f : Xl(y) \rightarrow Xs(l(y))$
$a : X0$
By *Repeat deduction* to obtain a proof of $Xl(x)$ we have to find a proof of
1) $Aa \rightarrow (Xl(y) \rightarrow Xs(l(y)))$
2) $X0$.
For 1) we use *add-hypo* that adds $d : Aa$ to the context and by application of *deduction* we obtain $\lambda d.f$.
For 2) we use *assume-tac* to obtain $a : X0$.

then we have $ap(ap(\varepsilon, \lambda.f), a) : Xl(x)$ and $\lambda\varepsilon\lambda f\lambda a.ap(ap(\varepsilon, \lambda d.f), a)$ has the initial type and is a program to compute the length of the list.

4.3.2 Recursive proof plan

We can use the **recursive** definition of data types to derive a corresponding proof. In this case we have to manipulate, at the same time, formulae in concise and developed form, and this leads to problems if we want to automate the proof construction problem. That is why we have to find methods which obtain predefined constructions devoted to automated derivation. To do this, a possibility is to define data in Martin-Löf style and to use introduction and elimination rules to derive programs. For instance, to prove a property Xx on the lists defined recursively we have to prove the initial condition $Xnil$ and the induction step and , consequently, for every list x defined recursively Xx will be true. It is sufficient to find an operator of the form *listrec* introduced by the presentation of type $L(A)x$ in the Martin-Löf style [12]. The new data representation of $L(A)x$ is given by the rules

List-intro

$$nil \in L(A)nil$$

$$\frac{a \in A \quad z \in L(A)x}{ad(a, z) \in L(A)ad(a, x)}$$

List-elim

$$\frac{\alpha \in Xnil \quad \overset{[u \in Xy, v \in Aa]}{ap(ap(\beta, v), u) \in Xad(a, y)} \quad c \in L(A)x}{listrec(\alpha, \beta, c) \in Xx}$$

To justify this representation we have to prove the following representation and, moreover, to have the representation of the *listrec* operator as a λ-expression in the calculus.

Proposition 4.1 *If α is a proof of Xnil, β is proof of $Aa \to (Xy \to Xad(a, y))$ and c is a proof of $L(A)x$ then $listrec(\alpha, \beta, c)$ is a proof of Xx.*

> **Proof** Let us recall that listrec has the type
> $\forall X(Xnil \to \forall y \forall a(Aa \to Xy \to Xad(a, y))) \to \forall x(L(A)x \to Xx)$.
> Considering α : Xnil and $\beta : \forall y \forall a(Aa \to Xy \to Xad(a, y))$ we want to find $\gamma : \forall x(L(A)x \to Xx)$. To do that we can prove the following result: if γ satisfies the equations $\gamma(nil) = \alpha$, $\gamma(ad(a, y)) = ((\beta \text{ a})(\gamma \text{ y}))$ then $\gamma : \forall x(L(A)x \to Xx)$.
> From these equations we can find a λ-expression denoting *listrec* which has the form $\lambda\alpha\lambda\beta\lambda x.(x\rho i\rho)$ with $i = \lambda d.\alpha$ and $\rho = \lambda y\lambda r\lambda a.((\beta a)yrir)$.
> □

Remark 4.1 *The use of such a representation is simple (considering we have constructed this) and gives an automated way to derive proofs. But a constraint is to obtain such predefined operators in a systematic way for all recursive data types.*

After the derivation and the justification of the new data representation we use this new scheme (associated with recursive data structures) to develop the following construction. Let us recall that we want to prove $\forall x L(A)x \rightarrow Il(x)$.
By application of *add-hypo* it means we have to prove $Il(x)$ in the context $\varepsilon : L(A)x$.
Since we use the recursive data structure we will use the operator *listrec*.
Consequently, we have to find α, β and c such that:
$3)\alpha : Il(nil)$,
$4)\beta : \forall y \forall a(Aa \rightarrow (Il(y) \rightarrow Il(ad(a,y))))$
$5)c : L(A)x$
and we can deduce $listrec(\alpha, \beta, c) : Il(x)$.
• To obtain 3) we search for $\alpha : Il(nil)$.
By application of the tactic *rewrite* it means searching for $\alpha : I0$ and by application of *lemma* (search for the type among the types already proved) we find $\alpha \equiv 0$.
• To obtain 4) we search for $\beta : \forall y \forall a(Aa \rightarrow (Il(y) \rightarrow Il(ad(a,y))))$.
By application of *add-hypo* it means proving $Il(y) \rightarrow Il(ad(y,a))$ in the context $d : Aa$.
By application of *Repeat rewrite* it means proving $Il(y) \rightarrow Is(l(y))$ in the context $d : Aa$.
By application of *lemma* we find $s : Il(y) \rightarrow Is(l(y))$ and then $\beta \equiv \lambda.s$.
• To obtain 5) the application of *assume-tac* gives $\varepsilon \ \equiv c$.
and the complete proof is: $\lambda\varepsilon.listrec(0, \lambda d.s, \varepsilon) \ : \ L(A)x \rightarrow Il(x)$.

4.3.3 Another proof plan

We have presented the treatment of iterative and recursive data structures with specific tacticals for proof development but we can show another proof plan using the *lemma* tactic. As an example we continue to develop a proof for the proposition $\forall x(L(A)x \rightarrow Il(x))$.
By *add-hypo* it means proving $Il(x)$ in the context $\varepsilon : L(A)x$.
By induction it is sufficient to prove:
$1)Il(nil)$ (initial condition)
$2)\forall a \forall y(Aa \rightarrow (Il(y) \rightarrow Il(ad(a,y))))$. (step condition)
For 1) by *rewrite* it means finding $I0$ and by *lemma* we have $0 : I0$.
For 2) by *rewrite* it means finding a proof of $\forall a \forall y[Aa \rightarrow (Il(y) \rightarrow Is(l(y)))]$.
By *add-hypo* it is sufficient to prove $Il(y) \rightarrow Is(l(y))$ in the context $d : Aa$.
By *lemma* we find $s : Il(y) \rightarrow Is(l(y))$ and then $\lambda d.s$ has the good type.
Consequently, we obtain
$ap(ap(\varepsilon, \lambda d.s), 0) : Il(x)$ and $\lambda\varepsilon.ap(ap(\varepsilon, \lambda d.s), 0) : L(A)x \rightarrow Il(x)$.

Remark 4.2 *The tactics and tacticals presented here are implemented in a second order theorem prover written in C-Prolog and examples on different data structures* *have been developed. The development of proofs is done automatically as soon as possible using the power of unification. It is important to note that in such a framework, unification is a fundamental tool, because we have typically to prove a formula such as $a : A$ where A represents a proposition and a is unknown. It allows here applications of tactics or strategies in a goal-oriented (depth-first) research. The use of meta-logic representation and higher order unification can be a future orientation of this kind of work.*

Remark 4.3 *By these examples we have illustrated the application of tactics and tacticals. But the aim of the development of proofs is the extraction of programs from these and, if possible, of efficient programs. We have derived three different programs satisfying the same specification. We know that a sort of duality exists between iterative and recursive representations in λ-calculus and a good program is not necessarily obtained by both representations; it can be useful to develop other new representations in second order logic (even with a meaning which is not clear in our framework) and corresponding tactics or strategies. To derive another program, we need inside the theory (to preserve properties) a general induction rule as in the extended propositional version of Martin-Löf type theory [13] to derive more efficient algorithms (like, for instance, the quicksort program that uses general induction) by considering well-order on the manipulated structures. Consequently, it is necessary to extend the theory by adapted rules (without destroying the essential correctness of programs) or to change the encoding of data structures to obtain the desired algorithms.*
Moreover, there are also relationships between the specification form (particularly the axioms) and the extracted program. This aspect, connected to program transformation, is not studied here.

5 Related and future work

We consider, in this section, some future work oriented towards better automation of correct program derivation in the second order functional arithmetic. Before we can briefly present related work on topics such as theorem proving and automated program synthesis.

5.1 Related work

At first, we can mention that the work presented is included in the general topic of automatic program synthesis (by theorem proving) from formal specifications [11]. The particularity, here, is the use of the powerful second order intuitionistic logic for program synthesis involving the extraction of programs from proofs.
The general problem of proof development and its strategies appears in many logical frameworks, especially in higher order logic programming. Our system is directly connected to work on logic programming to specify program transformers or theorem provers [3, 7] but the presentation here concentrates on tactics and not on

meta-logics to implement such systems.
Considering the construction of tactics and strategies, we mention the important work of Paulson [16] on an interactive and generic theorem prover that supports a variety of logics (with propositions as rules). Our aim is to study what can be done in a given powerful logic and the genericity aspect of the theorem prover is not at the center of this work. It is normal to find some similar tactics but our approach, based on using the theorem prover to extract programs from proofs, not only for theorem proving, orients tactics elaboration in different and more specific ways.
Another important related area is Martin-Löf type theory [12]. A first connection with it appears when one notices that it can be possible to represent the logic in a framework in the Martin-Löf style and then to ensure a maximal automation. Moreover we can use this theory for program synthesis from the same form of specification as in the presented framework with a high level of automation, but to obtain recursive terms (or programs) [4]. Here we have only derived operators in a similar style. The study of relationships between the two formalisms seems very important with respect to the following points: association of adequate operators to structures, various representations of inductively defined types and extensions (preserving properties) to include general induction in the theory.

5.2 Future work

Regarding future work, this will be oriented to a better understanding of the relationships between the proofs and the corresponding algorithms and of what is a good proof in the *programming with proofs* approach. In this way we have to develop more examples of program derivations with various data structures. To extend our knowledge on tactics and strategies we have to generalize our approach and, for example, to represent the type system in a meta-language consisting of typed terms and encoded basic judgments of typed λ-calculi.
Finally, the development of other strategies concerning the inductive approach will be necessary in the system to derive more significant algorithms. We have mentioned that general induction is not in this framework as it is in constructive set theory [9] and studies of their relationships can lead to a new framework. Moreover, we have to extract explicit plans from tactics to guide particular proofs not depending on the data structures employed.

6 Conclusion

With a view to proving propositions and to extracting programs from these proofs we are concerned with the derivation of proofs in a second order intuitionistic logic framework. Some methods, through tactics and strategies, are proposed and illustrated by significant examples. But it is only the first steps towards a second order theorem prover dedicated to automatically generating programs.
For iterative data structures the development scheme always included the same tactics but they can not be globally used and we have to establish a classification of

the function operators in different categories and to define strategies for a general operational category. For the recursive data structures, we have to find predefined operators and to automate the process of operator derivation for given data structures. Moreover, it is necessary to introduce a strategy of general induction to derive efficient programs.
Our goal, consisting of the automating of the proof construction process, can be achieved but, for the moment, we have not dealt with the representation of tactics at the meta-logical level. We are exploring the use of the tactics to incorporate some extensions (without destroying the important properties of termination and correctness) or to extract more general proof plans.

References

[1] Bundy, A., The use of explicit plans to guide inductive proofs, In *Lecture Notes in Computer Science 310*, Springer-Verlag, 1988.

[2] Coquand, T., Huet, G., The Calculus of Constructions, *Information and Computation*, 76, 95-120, 1988.

[3] Felty, A., Miller, D., Specifying theorems provers in a higher-order logic programming language, In *Lecture Notes in Computer Science 310*, Springer-Verlag, 1988.

[4] Galmiche, D., The power of constructive framework for program synthesis, In *COLOG 88 - International Conference on Computer Logic*, Tallinn, Estonia, USSR, December 1988.

[5] Galmiche, D., Constructive system for automatic program synthesis, *Theoretical Computer Science*, 71(2), 227-239, 1990.

[6] Gordon, M.J., Milner, R., Wadsworth, C.P., Edinburgh LCF, *Lecture Notes in Computer Science 78*, Springer Verlag, 1979.

[7] Hannan, J., Miller, D., Uses of higher-order unification for implementing program transformers, In *Fifth International Conference and Symposium on Logic Programming*, K. Bowen and R. Kowalski (editors), MIT Press, 1988.

[8] Harper, R., Honsell, F., Plotkin, G., A framework for defining logics, In *2nd Annual Symposium on Logic in Computer Science*, Ithaca, New York, June 1987.

[9] Henson, M.C., Turner, R., A constructive set theory for program development, In *Lecture Notes in Computer Science 338*, Springer-Verlag, 329-347, 1988.

[10] Krivine, J.L., Parigot, M., Programming with proofs, In *6th Symposium on Computation Theory*, 1987. Wendisch-Rietz.

[11] Manna, Z., Waldinger, R., A deductive approach to program synthesis, *ACM Transactions on Programming Languages and Systems*, 2(1), 90-121, 1980.

[12] Martin-Löf, P., *Intuitionistic Type Theory*, Studies in Proof Theory, Lectures Notes. Bibliopolis, 1984.

[13] Nordström, B., *Terminating general recursion*, Technical Report 46, Programming Methodology Group, Göteborg University, June 1988.

[14] Parigot, M., Programming with proofs : a second order type theory, In *Lecture Notes in Computer Science 300*, Springer Verlag, 145-159, 1988.

[15] Paulson, L.C., Experience with Isabelle : A generic theorem prover, In *COLOG 88 - International Conference on Computer Logic*, Tallinn, Estonia, USSR, December 1988.

[16] Paulson, L.C., The foundation of a generic theorem prover, *Journal of Automated Reasoning*, 5, 363-397, 1989.

[17] Sato, T., Tamaki, H., Transformational logic program synthesis, In *International Conference on Fifth Generation Computer Systems*, ICOT Tokyo, Japan, 1984.