Program transformation by proof in constructive framework*

D. Galmiche

CRIN - INRIA Lorraine
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex
France.
e-mail : galmiche@loria.crin.fr

Abstract

This paper aims at presenting the notion of program transformation by proof in type theory and programming with proofs framework. In such framework, programs obtained from proofs are not always efficient and the relationships between programs and good proofs have to be studied. Automatic synthesis and verification of programs have been tackled in the constructive type theory and we are interested by the transformational approach, with some degree of automation, in such a framework. We present some techniques of proof and program transformation using proof synthesis information. A strategy of generalization by abstraction of terms is presented, with a view to deriving by constructive proofs new program versions, and we analyze the connections with data modifications and induction choices.

1 Introduction

The derivation of correct and efficient programs from formal specifications can be considered in different ways. An important one consists in using the transformation technique which is a method available in the case of functional [4] or logical programs [22] preserving semantics. The transformation methodology of program derivation is based on the idea of first writing an initial program version which corresponds to a given specification and then to derive by several transformation steps a new program version still correct and more efficient. For example, in the logical case, the transformations are semantics preserving rules like definition rule, folding and unfolding rules and their applications need to be directed by strategies [21]. These ones allow for some program classes to derive more efficient programs, and application is realized in this way via the introduction of new predicates.
In this paper, we investigate some ways of program and proof transformation in constructive logic, with the aim of automation. Recently, there has been a great interest in the development of programs in constructive framework and more precisely in programming with proofs framework [5], [9], [15], [17]. The idea of this technique is that formal specification is given first, then a proof of it is constructed and executable code is extracted from the proof systematically with the notion of realizability interpretation or the formulae-as-types one. Several systems realizing such proof compilations have been implemented in different logics to investigate the program synthesis in type theory.

*Published in Informatika'91, Theoretical Computer Science and Methods of Compilation and Program Construction, 1991

But it is not sufficient to have a good framework to derive programs as proofs. Some problems are not solved yet like those concerning 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. And, even, if we obtain good programs we want to have better versions or to reduce some redundancy. That is why we want to study the concept of program transformation in programming with proof systems. Within constructive type theory, automatic synthesis (of programs from specifications) and automatic verification (that a program meets its specification) have been tackled through different work [9], [15]. But we are also interested by automatic transformation of programs into more efficient programs, with the same specification. The approach can be the program transformation by some transformation of their synthesis proofs or by synthesizing new proofs (and consequently new programs). In this work, we investigate the proof and program transformation in constructive type theory with a view to finding adapted transformation techniques. In this way, we introduce a generalization strategy by abstraction that introduces systematically a new specification to prove and we study its applications to automated transformation within constructive framework and the relationship with data transformation and induction choices.
In section 2, we present the logical framework and section 3 gives a presentation of programming with proofs methodology and its application to program and proof derivation. Section 4 presents the concept of proof and program transformation. Section 5 introduces the technique of transformation based on generalization by abstraction with its connections with other techniques like data modification or induction. Section 6 contains some conclusions and work directions about program and proof transformation.

2 A logical framework

In this section, we present the logical framework to program with proofs, devoted to the formulation of theories as Martin-Löf's type theory [17]. Assuming some basic knowledge about constructive type theory and *propositions as types* paradigm, we give a concise summary of these concepts with a description of the general logical framework and its use to encode constructive type theory.

2.1 Constructive type theory

Martin-Löf's type theory is a constructive, higher order, typed logic and can be seen as a programming logic for functional programming language [17]. The formal system can be used to express programs, their specification and also derivations of programs. This type theory is based on the identification between *types*, *propositions* and *specifications*. The system, in a natural deduction style, has four forms of judgements: $A\ Type$ (A is a type), $A = B$ (A and B are equal types), $a \in A$ (a is an element in the type A), $a = b \in A$ (a and b are equal elements of type A). The third one has different meanings and can be read as *1) a element of type A, 2) a construction (proof) for the proposition A, 3) a solution of task or problem A, 4) a is a program with type A and 5) a is a program which satisfies specification A.*
A proposition is identified with the set of its proof objects and a specification is identified with the set of the programs satisfying the specification. For instance, the proposition $\exists x \in A.B(x)$ is interpreted as the type $\Sigma x \in A.B(x)$ which is the disjoint union of family of types. A proof of $\exists x \in A.B(x)$ consists of two things: an element a in A and a proof of $B(a)$, the elements in $\Sigma x \in A.B(x)$ consisting of pairs (a, b) where $a \in A$ and $b \in B(a)$. Moreover we have definition

of basic types like *Naturals*, *Bool* and *enumeration types*, and specific type like $Wx \in A.B(x)$ used to represent inductively defined sets or the *equality type* $(Eq(A,x,y) \equiv x =_A y)$. Here it is important to note the difference between the two equalities $x = y : A$ and $x =_A y$, the former being a *judgement* and the latter a *proposition* (or a *type*). A judgemental equality cannot be proved by induction but only by applying the rules of type theory. As example, we give the definition of type $L(A)$ (lists of elements of type A) and of type $Sexp(A)$ (s-expressions of type A) that we will use in the rest of the paper.

Representation of L(A)

$$nil \in L(A)$$

$$\frac{a \in A \quad b \in L(A)}{cons(a,b) \in L(A)}$$

$$\frac{c \in L(A) \;\; x \in C(nil) \quad \overset{[a\in A, b\in L(A), c\in C(b)]}{y(a,b,c) \in C(cons(a,b))}}{listrec(c,x,y) \in C(c)}$$

with
$listrec(nil,x,y) = x \in C(nil)$
$listrec(cons(a,b),x,y) = y(a,b,listrec(b,x,y,)) \in C(cons(a,b))$.

Representation of Sexp(A)

$$\frac{a \in A}{a \in Sexp(A)}$$

$$\frac{x \in Sexp(A) \quad y \in Sexp(A)}{snoc(x,y) \in Sexp(A)}$$

$$\frac{c \in Sexp(A) \quad \overset{[a\in A]}{x(a) \in C(a)} \quad \overset{[a\in Sexp(A), b\in Sexp(A), c\in C(a), d\in C(b)]}{y(a,b,c,d) \in C(snoc(a,b))}}{srec(c,x,y) \in C(c)}$$

with
$srec(a,x,y) = x(a) \in C(a)$
$srec(snoc(a,b),x,y) = y(a,b,srec(a,x,y),srec(b,x,y)) \in C(snoc(a,b))$.

This theory is suitable for the task of program synthesis since program is built up as the proof is constructed. All components of the program have a one to one correspondence with components of the proof but the reverse is not true because proofs contain additional information. By finding a constructive proof of the specification, we can extract a program from the proof that satisfies the input-output relation. For instance, if the specification has the form $\forall x \ \in A \ \exists \ y \in B \ Spec(x,y)$, we extract a program *prog* such that $\forall x \in A \ Spec(x, prog(x))$.
In such framework, on can reason backwards from the formula (to be proved) using the calculus which includes rules of inference for mathematical induction. The development of proofs can be automated through tactics which split the formula into subformulae, by application of refinement rules [3], [9].

Remark 2.1 *We have to mention that there are some problems considering programs as proofs. A first one is the difficulty consisting in formalization, programs to be executed having to be formalized. Thus, such identification means that the derivation of the program has to be formalized* *and informal reasoning (which can appear in programming activity) cannot be included in the process. Another problem is program termination and, for some specification, we cannot find a solution in type theory because of a lack of information. A way to treat such situations in type theory, with propositions and subsets, is an extension of the theory, for example with general recursion [19].*

2.2 A logical framework

We give here a presentation of intuitionistic type theory in a lambda calculus framework [7] like in [8], which has an important relationship with ELF [12] and Automath languages [2]. This formulation shows a presentation of logical systems in typed theory. This framework can be viewed as the implementation language of the theory.
We give here a classical presentation of the theory considering the following notations $x_1 \ldots x_n.e$ instead of $(x_1)\ldots(x_n)e$ for the abstraction and $f\ x_1 \ldots x_n$ instead of $f(x_1 \ldots x_n)$ for the application. There are four basic judgment forms in the type theory : $\alpha : U$, $\alpha = \beta : U$, $a : \alpha$, $a = b : \alpha$. The theory is a typed λ-calculus with dependent types, a ground type U_1. We give the rules of formation concerning the types and the objects.
The rules of *type formation* are:

1) $\Gamma \ \vdash \ U_1 : U$

2) $If\ \Gamma \ \vdash \ A : U_1\ then\ \Gamma \ \vdash \ A : U$

3) $If\ \Gamma \ \vdash \alpha : U,\ \Gamma, x : \alpha \vdash \ \beta : U\ then\ \Pi[x : \alpha]\beta : U.$

The rules of *object formation* are :

4) $If\ \Gamma, x : \alpha \vdash b : \beta\ then\ \lambda x : \alpha.b : \Pi[x : \alpha]\beta$

5) $If\ \Gamma \ \vdash b : \Pi[x : \alpha]\beta\ and\ \Gamma \ \vdash a : \alpha\ then\ ba : \beta[a/x]$

The *equality rules* are typed β and η conversions and correspond to:

6) $(\lambda x : \alpha.b)a \ = \ b[a/x] : \beta[a/x]$ if $a : \alpha$ and $b : \beta$.

7) $c = x.cx \ : \ [x : \alpha]\beta$ if $x : \alpha$ and $c : [x : \alpha]\beta$.

We can include a notion of polymorphism (which plays the role of second order quantification appearing in other theories [5], [20]) and we give polymorphic rules for objects and types.
The rule of *p-type formation* is:

8) $If\ \Gamma \ \vdash \alpha : U,\ \Gamma, x : \alpha \vdash \ \beta : U\ then\ \Pi\overline{[x : \alpha]}\beta : U.$

The rules of *p-object formation* are:

9) $If\ \Gamma, x : \alpha \vdash b : \beta\ then\ \lambda x : \alpha.b : \Pi\overline{[x : \alpha]}\beta$ with b not depending on x.

10) $If\ \Gamma \ \vdash b : \Pi\overline{[x : \alpha]}\beta\ and\ \Gamma \ \vdash a : \alpha\ then\ ba : \beta[a/x].$

In such a framework, a theory consists in a signature i.e. a list of type assignments to constants and a finite list $a_1 = b_1 : \beta_1, \ldots, a_m = b_m : \beta_m$ of axioms and equations. Definitions can be introduced with a judgment of the form $c = a : \alpha$ with c being a constant, an object and α their type. For more details about this logical framework, you can refer to [8], [17].
In the framework, we can represent Martin-Löf's type theory. Let us recall that, for each type forming operation T, there are four kinds of rules: the *formation rules* describing when T is a type, the *introduction rules* describing how the canonical elements of T are formed, the *elimination rules* expressing an induction principle for T and the *equality rules* formalizing computation for the induction operator. We can represent types with corresponding definitions in the theory that match with these different rules. As example, we give the representation of $L(A)$ (lists of elements of type A).

Example 2.1 *Representation of $L(A)$.*

$$L : \overline{[A : U_1]}U_1$$

$$\begin{array}{lll} nil & : & \overline{[A : U_1]}L\ A \\ cons & : & \overline{(A : U_1)}[a : A][b : L\ A]L\ A \end{array}$$

$$listrec : \overline{[A : U_1][C : [c : (L\ A)]U_1]}\ [c : L\ A](C\ nil)([a : A][b : L\ A](C\ b))(C(cons\ a\ b))(C\ c)$$

$$listrec\ nil\ x\ y\ = x\ :\ \overline{[A : U_1][C : [a : (L\ A)]U_1]}\ (C\ nil)([a : A][b : L\ A](C\ b))(Ccons\ a\ b)(C\ nil)$$

$$listrec\ (cons\ z\ t)\ x\ y\ =\ y\ z\ t\ (listrec\ t\ x\ y) : \overline{[A : U_1][C : [a : (L\ A)]U_1]}\ (C\ nil)([a : A][b : L\ A](C\ b))(Ccons\ a\ b)(C\ cons\ z\ t)$$

In the rest of the paper, we use the general presentation as in section 2.1. but the expressions of the theory can be coded in the above logical framework.

3 Programming with proofs

On the bases of the logical framework presented below, we present, with the support of a simple example, the different steps of the methodology of program construction with proofs, i.e., the specification of the problem, the construction of the proof (insisting on the mechanization aspect), the extraction of the program.

3.1 Programming methodology

We consider, at first, the specification step, i.e., how can we express the specification in our calculus. To facilitate comprehension, we use the language of type theory (that can be translated in the λ-calculus framework). We can define, as informal specification of a problem, the informal but precise description of this problem. As example, we consider a program multiplying by a natural all the elements of a natural list x.
The formal specification of a problem is a formula of the logical theory (considered as a type of the theory) and a general form of that formula is : $\forall x : T_1\ \exists y : T_2\ Spec(x, y)$ such that $x : T_1$ and $y : T_2$ represent the input and output parameters with their types. After this step, we have to prove the specification formula in the theory and extract a program. Consequently, we can then write the formal specification in the theory with the proper types or formulae (the functions *length* (length of a list) and *elem* (giving the ith element of a list) are defined) and prove it.

Proposition 3.1 $\forall x : L(nat)\forall y : nat\exists z : L(nat)(length(x) = length(z)\wedge$
$\forall i : nat(1 \leq i \leq length(x)) \Rightarrow y \times elem(i, x) = elem(i, z)).$
(it is an instance of a general formula $\forall x : L(nat)\forall y : nat\ \exists z : L(nat)G(x, y, z)$).

Proof 3.1 *By structural induction on x.*

In this case, we obtain a proof of this specification and we want to extract the program which computes z depending on x and y. An important aim would be to automatize the development of such a proof.

Remark 3.1 *We can also consider an other form of specification. When we consider a specification of the form* $\forall x : T_1 \ \exists y : T_2 \ Spec(x, y)$ *we must determine a* y *and then verify that it satisfies the relation* $Spec(x, y)$. *A transformation leads to replace the relation* $Spec(x, y)$ *by a relation* $Spec(x, f(x))$ *where* $f(x)$ *represents the searched* y *depending on. We thus obtain a formula of the form* $\forall x : T_1 \ f(x) : T_2$ *with* f *relation represented by equations in the logical theory. It can lead to other form of specification (as in [20]) consisting in the formula* $\forall x : L(nat) \ \forall y : nat \ multi(x, y) : L(nat)$ *and the equations* $multi(nil, y) = nil$ *and* $multi(cons(a, x), y) = cons(mul(a, n), multi(x, y))$.

3.2 Mechanization of proof development

From the specification we can construct a proof of the formula-specification. In this way, we adopt an oriented-by-types strategy. The goal to achieve is the proposition (or specification) we have to prove possibly considering some hypotheses.
A traditional method, to look for a proof, is to work backwards from goal to subgoals. For instance to prove $A \wedge B$ it is sufficient to prove A and B. In type theory, the goal would be $x : A \times B$ for an unknown x. As the proof proceeds, constraints upon x accumulate and when the proof is finished they determine x completely. If the type A represents the proposition to prove then the goal has the form $x : A$ for x. In proving $x : A$, each backwards step decomposes the type constructing step by step the structure of x.
The *unification* gives a flexible treatment of unknown expressions in goals and inference rules. Backward proof is natural for discovering and representing formal proofs and to automatize the proof construction process. That is why an important aim is to have adequate inference rules in the theory to use backward proof. For the mechanization of the construction process, let us define some *tactics*, methods that, applied to the satisfaction of a goal, reduce it to the satisfaction of subgoals. We then define one or several tactic(s) for each type of formulae.
A *strategy* is a combination of tactics which leads to the construction of new tactics. These tactics are applied in the constructive system by unification, with a research strategy oriented by the types (formulae), the construction of the program satisfying the goal being progressive during the application of the research strategy.
The construction process is mechanized in the global type-driven strategy. More details on proof development and its automation are given in [8],[9]. Some tactics or strategies are general and appears also in other logical frameworks (like second order type theory) but others are directly depending on the theory and its characteristics (like second order quantification) [10].

3.3 Proof and program derivation

A good example of program synthesis is the derivation of the well known sorting algorithms from a common specification [6]. We can see how algorithmic features can be reflected in proof strategy to derive proof of formal specification. At first, we present a general formulation of sorting in a mathematical notation and two different strategies of proofs. At the end, it will be important to formulate such expressions and derivations in our theory with special attention to the treatment and the representation of the data structures. The aim of this section is to show the importance of the proof development to obtain good programs.
We consider a formal specification of sorting problem, which means that for any list of naturals x there exists a natural list y such that the length of y is equal to that of x, every element occurring in x also occurs in y and y is sorted.

Proposition 3.2 *(Specification)*
$\forall x \in L(nat)\ \exists y \in L(nat)\ (\forall n \in N\ (occ(n,x) = occ(n,y))\ \wedge\ sorted(y))$
with $sorted(x) \equiv \forall i,j \in N\ (1 \leq i \leq j < length(x) \Rightarrow elem(i,x) \leq elem(j,x))$.

Remark 3.2 *From this specification, we can develop a proof introducing the definition of partially sorted list and proving connected lemmas. The program issued from this proof development is the bubble sort algorithm. From the same specification, we can develop another proof introducing a divide lemma saying that for a list x and a natural n, there exists a permutation of the list such that the resulted list is the concatenation of the list of elements less than n and of the list of elements greater than n. The proof developed in this case leads to the quicksort algorithm.*

We can see that constructive proofs of formal specification have a structure similar to that of the algorithms in satisfying specifications. A constructive proof of a specification can be considered as a description of an algorithm with its mathematical meaning. Moreover these examples show that a good proof strategy corresponds to an efficient algorithm. The strategies are defined by the introduction of adequate lemmas and definitions and we have the possibilities of developing such strategies through rules corresponding to various data treatments.
But it appears that in certain cases the extension of theory for good data representations and good proof developments is not easy and does not lead directly to most efficient programs. That is why we want to study, in this paper, the notion of program transformation by proof in type theory and the relationships with program development.

4 Program and proof transformation

In this section, we consider the notion of transformation of programs and proofs in our framework and try to define transformation strategies of programs obtained by extraction from proofs. We know that good proofs do not lead systematically to good (more efficient) programs and also efficient algorithms can not be always expressed in the logical system. Moreover the representation of data has an impact on the nature of proof development and of the program issued from it. That is why, in parallel to the study of the relationship between the data representation and proof, we are interested by transformation techniques to apply to the programs inside the logical theory and to study automatic transformation of programs by constructive proof.

4.1 Program transformation

Let us consider the following formula $\forall x \in L(A) \forall y \in L(A)\ length(append(x,y)) = length(x) + length(y)$ (I). We say that two programs are equal if they give equal results for equal arguments, that is termed extensional equality. But this equality is not sufficient from computer science point of view because we are concerned with certain intensional properties of programs (for instance, how much space a program needs or how much time it requires to be executed). It is often claimed that program clarity and program efficiency are rather incompatible and it leads to the program transformation methodology where clear but inefficient programs may be systematically transformed (by correctness preserving methods) into efficient (although probably opaque) programs. The right and left-hand sides of (I) have different intensional properties (considering for instance the data construction). Moreover if we define the proposition $\forall x \in L(A) \forall y \in L(A)\ length2(x,y) = length(x) + length(y)$, a way is to take this as the definition of *lenght2* but it does not improve calculation efficiency. Another way is to use such

proposition as an *invariant* (we should find a program for which the property always holds). We will see that is beneficial to consider this definition or invariant as *specification* that is rather hard to find (*eurekastep*) [14].
The proof transformation methodology has been introduced and applied for functional languages [4] but it can also be applied in the case of logic programs [22]. One basic idea of that methodology is the separation of efficiency requirement from the correctness requirement. The latter one is taken into account by allowing only transformation rules which are correctness preserving. The efficiency requirement is taken into account by considering heuristic techniques. The programmer is asked to derive from the initial program a more efficient version (through transformation rules) by inventing auxiliary (*eureka*) functions or predicates, whose definition may require a deep insight in the behaviour of the programs be transformed. These inventions are hard steps to be made during the program derivation process and several strategies can be adopted.
Transformational programming is a methodology for program development in which specifications or (inefficient) programs are transformed into efficient forms by correctness preserving programs transformations [4]. In our framework, we know that such transformations are essentially inductive proof of program equivalence but the advantage of this approach is that the proof and the program are derived simultaneously. Moreover, eureka definitions are fundamental to successful program improvement but there are not easy to construct although techniques corresponding to well-known programming methods exist [14]. Automation of this definition search (with a view to automating transformation process) is also a hard task we have to consider.

4.2 Transformation of recursive programs

The computational efficiency of a recursive program is directly related to the form of recursion and the way in which a program recurses on an input can be controlled by the way in which induction is used in program synthesis. It is clear that a system, devoted to program synthesis, must contain, in the form of heuristics, different induction schemas. A proof from which a program is derived will contain more information than that required for the execution. An idea is that more efficient programs can be obtained by tranforming the associated proofs, with the important point that is recursive programs are optimized by transforming the induction schema in the synthesis proofs. For instance, let us consider the transformation of a program that computes fibonacci function. To synthesize the program we can employed *course of values induction* or *stepwise induction*. Considering the former, the program extracted from a proof with c.o.v. induction is defined by the following equations: fib(0) = 1, fib(1) = 1 and fib(n+2) = fib(n+1) + fib(n). It is clear that in this case the computation cost is exponential.
We can transform the course of values inductive proof into a proof that uses stepwise induction (corresponding, with integers, to the standard mathematical induction). To do this, we introduce a new function ($mkt(x, y)$)that combines the values of the two step cases of the course of values definition and obtain the function $g(0) = mkt(1, 1)$ and $g(n) = mkt(x + y, z)$ where $mkt(x, z) = g(n)$. Finally, there is no use of original program and we obtain a more efficient program [16].

Remark 4.1 *The process of providing new functions with their definitions is known as eureka step. It consists in generating new functions that can be folded with current equations to introduce some recursion into the target program. Such process is hard to automate and relies on user interaction. It is also linked to the particular action you want to develop (tail-recursion introduction, pruning mechanisms, partial evaluation or generalisation).*

4.3 Proof transformation

The classical approaches are concerned with a direct transformation of the executable code but not with the transformation of constructive proofs. This latter approach has important advantages. A proof contains more information than the program which it specifies; it contains the thinking behind the program design. This aspect of proof transformation has been studied for equational theories [18]. The logical structure of a proof is better understood that that of a program. Moreover, the knowledge of the structure of inductive proofs are linked to the synthesis of recursive programs [1].
In the constructive type theory, the language is devoted, at the same time, to the specification, the verification and the synthesis of programs. Such features mean that synthesis and transformation can be treated uniformely, referred to the proof-program uniformity. We try to consider the transformation within the theory knowing that program transformation corresponds to inductive proof, eureka step to induction hypothesis. For a transformation performed on a synthesis proof there will be a corresponding transformation on the program. Moreover, this uniformity may prevent some complexity which can occur when the specification and implementation languages are different. Some work in type theory have considered proof transformation problem and its connexion with program transformation [16], considering a set of pruning transformation (reducing the size of the proof) after a specialization (or generalization) step.
In the following section, we want, at first, to present a generalization strategy by abstraction that can be used to introduce automatically a new definition (or specification) and to transform programs by proof. A first example illustrates the generalization strategy and a second one shows connexions between definition proofs and some induction proofs.

5 The generalization strategy

We present, in this section, a strategy based on the use of a generalization approach which is a classical technique in the field of theorem proving for deriving proofs. We know that, sometimes, general statements can have simple proofs while some instances have more complex proofs.
Considering the form of the programs extracted from proofs and having in mind that we work with *proofs as programs* paradigm, we search for a strategy that allows to transform programs within the theory, by generating new functions or predicates and their definitions with automation. Considering the functional form of the programs, it appears interesting to define an *λ-abstraction generalization strategy* that consists in generalizing variables or expressions to functions (to derive more efficient algorithms). It allows to introduce new definitions and, using these, to transform recursive definitions into tail recursive ones, to reduce redundancy of some recursive definitions or to replace some operations by more efficient ones. This strategy is a general one that can be applied in different frameworks. Techniques like λ-abstraction and generalization are used in logical and functional program transformation [21]. The aim, here, is to consider such a strategy with a view to studying proof and program transformation in constructive framework for which it is well adapted.
To present the strategy, in a not too general way, we will consider, as illustration, programs on $L(A)$ domain. But it can be applied to different classes of operators and types.
Refering to the presentation of the theory, such programs $f(x)$ extracted from proofs (by induction) have the general form $listrec(x, d, \lambda abc.e(a, b, c))$ with the associated evaluation rules. From $f(x)$ and from its third argument (corresponding to the induction step), we construct a term by generalization of this argument by $f(x)$ and by λ-abstraction. This term construction

can be made systematically for various classes of programs.

Definition 5.1 *(generalization strategy)*
Let us consider the general form $f(x) = listrec(x,d,\lambda abc.e(a,b,c))$ with $x \in L(A)$, we associate a term $g(x)$ defined by $g(x) = \lambda z \lambda y.e(z,y,f(x))$. It is obtained considering abstraction on the third argument $e(a,b,c)$ of listrec form with replacing c by $f(x)$.

Remark 5.1 *By application from $g(x)$ we can build a term of the form $h(x,z,y)$ which is $g(x)$ z y that why we use the equality $h(x,z,y) = g(x)\ z\ y$. Moreover even we are concerned by the fact that, in $f(x)$, x in of type $L(A)$, we have also to consider the type of $f(x)$ that we note B.*

After the definition of a new term by λ-abstraction generalization, we construct a new proposition, in our logical framework, using this new term. This proposition only means that it exists a term equal to the new (generalized term) and corresponding to a generalization of the first specification formula.

Proposition 5.1 $\forall x \in L(A) \forall z \in A \forall y \in L(A) \exists y_1 \in B\ y_1 = h(x,z,y)$
(assuming that f is a program of type $L(A) \rightarrow B$).

Proof 5.1 *by induction on $L(A)$.*

In fact, we have two ways to prove it. A trivial one saying that a solution is $y_1 \equiv h(x,z,y)$, considering the proposition as a definition. A second one, by induction on the type (here $L(A)$), leads us to a new term y_1 which is a new program definition that generalizes the initial program, considering the proposition as a new specification. Let us notice that the initial expression $f(x)$ is a specific instance of the $h(x,z,y)$ schema.
Consequently, by a proof of a new proposition, after a generalization by λ-abstraction we obtain a new version of the program. The application of this methodology can be automated.
We have now to study the interest of such generalization in our framework and its consequence on program development.

5.1 A first example

We consider, at first, an example about programs on lists ($L(A)$), the reverse program (denoted by rev), and show that application of the strategy leads to an accumulator version of the program. an example on lists with lists as results which is the reverse program (denoted by rev).
After deriving in our logical system the proof of the corresponding specification, we obtain the program $rev(x) \equiv listrec(x, nil, \lambda abc.append(c, cons(a, nil)))$, recursive program using the $listrec$ operator. Considering the application of the λ-abstraction strategy, we have:

Definition 5.2 *(generalization transformation)*
Let us consider the term $rev(x) = listrec(x,nil,\lambda abc.append(c,cons(a,nil)))$, we define by abstraction a term $rev_1(x) = \lambda z.append(listrec(x, nil, \lambda abc.append(c, cons(a, nil))), z)$.

We can also define $rev_2(x,z)$ as $rev_1(x)z$ (application of the λ-terms).
Considering the generalization, we have to prove the following proposition (generated systematically):

Proposition 5.2 $\forall x \in L(A) \forall z \in L(A) \exists\ y \in L(A)\ y\ = rev_2(x,z)$.

Proof 5.2 *by induction on L(A).*
a) Base step. $x = nil$.
Let z an arbitrary element of L(A), we have to prove $\exists y \in L(A) y = rev_2(nil, z)$.
We have $y = rev_2(nil,z) = rev_1(nil)\ z = append(listrec(x,nil,\lambda abc.append(c,cons(a,nil))),\ z) = append(nil,z) = z$. *So we take* $y \equiv z$.
b) Induction step. $x = cons(a,l)$ *with* $a \in A, l \in L(A)$.
Let z an arbitrary element of L(A), we have to prove $\exists y \in L(A) y = rev_2(cons(a,l), z)$ *with the induction hypothesis is* $\forall z \in L(A) \exists y \in L(A) y = rev_2(l, z)$.
Moreover, we have
$y = rev_2(cons(a,l),z) = rev_1(cons(a,l))\ z$
$= append(listrec(cons(a,l),nil,\lambda abc.append(c,cons(a,nil))),\ z)$
$= append(append(listrec(l,nil,\lambda abc.append(c,cons(a,nil))),cons(a,nil)),z)$
$= append(listrec(l,nil,\lambda abc.append(c,cons(a,nil))),append(cons(a,nil),z))$
$= append(listrec(l,nil,\lambda abc.append(c,cons(a,nil))),cons(a,z))$
$= rev_2(l,\ cons(a,z))$.
In this process we have used the fundamental property of append that is associativity.
The induction hypothesis with $cons(a, z) \in L(A)$ *gives* y_1 *such that* $y_1 = rev_2(l, cons(a, z))$.
Then we set $y \equiv y_1$.
This proof leads to the program defined by the equations: $rev_2(nil, z) = z$ *and* $rev_2(cons(a, l), z) = rev_2(l, cons(a, z))$.

Consequently we obtain $rev_2(x, z) = append(listrec(x, nil, \lambda abc.append(c, cons(a, nil))), z)$
$= listrec(x, z, \lambda abc.append(c, cons(a, z)))$
and the initial program $rev(x) = rev_2(x, nil)$ is an instance of the generalized program $rev_2(x, z)$.
This illustrates the relationship between the (intensional) structure of the proof and the (intensional) structure of the program..

Remark 5.2 *We obtain the accumulator version of the initial program. Let us recall that the technique of accumulator introduction is to define a new function in which we accumulate partial results. Moreover, the elimination rule of the type concerned (here L(A)) results in the elimination of the initial program in the extracted one.*

In type theory, we have the possibility to extracted direct efficient programs as the quicksort program (it needs extension of the kernel theory with subsets, propositions and general recursion [19] with new operators). It is not necessary to precise the form in our calculus to say that an occurence of *append* function appears in it, decreasing the efficiency. By application of the generalization strategy on this program, we can transform it by proof into an more efficient tail-recursive version, as in the previous example.

Remark 5.3 *This kind of generalization can be applied to all the recursive structures manipulated in the theory. We obtain by this way accumulator versions of programs and in this case more efficient programs. In classical transformation of logical programs we apply the same technique introducing new predicates generalizing the first version and, like in the examples, making the append function disappear.*

5.2 A second example

We consider the data type $Sexp(A)$ (defined in the section about logical framework) and want to derive a program which according to an S-expression, transforms it into left linear form. Before to give the specification, we give the following predicates definitions:

1. $llf(a) = true$
2. $llf(snoc(x, y)) \;=\; atom(x) \wedge llf(y)$

with the following definition:
$eqfr(x, y) \;\equiv\; equal(fr(x), (fr(y)))$ with fr such that $\forall x \in Sexp(A)\; fr(x) : list(A)$ defined by $srec(x, \lambda a.cons(a, nil), \lambda abcd.append(c, d))$.
First we give the formula, corresponding to the specification of the problem, as the proposition to prove.

Proposition 5.3 *(Specification)* $(\forall x \in Sexp(A))(\exists y \in Sexp(A))\; (llf(y) \wedge eqfr(x, y))$

Proof 5.3 *by induction on $Sexp(A)$.*
a) Base step. $x = a$.
We have to prove $\exists y \in Sexp(A)(llf(y) \wedge eqfr(a, y))$. So we can take $y \equiv a$.
b) Induction step. $x = snoc(x_1, x_2)$ with $x_1, x_2 \in Sexp(A)$.
We have to prove $\exists y \in Sexp(A)(llf(y) \wedge eqfr(snoc(x_1, x_2), y))$. By induction hypothesis, we have y_1 as witness for x_1 and y_2 for x_2.
Let us consider an auxiliary function j defined by
$j(x, z) = srec(x, \lambda a.snoc(a, z), \lambda abcd.snoc(a, d))$ (which means that $j(snoc(x, y), z) \;=\; snoc(x, j(y, z)$ and $j(a, s) \;=\; snoc(a, s)$). We can take $y \equiv j(y_1, y_2)$ as a solution for x.
The proof yields to the program p such that $p \equiv \lambda x.srec(\lambda a.a, \lambda abcd.j(c, d), x)$ (which means that $p(a) = a$ and $p(snoc(x_1, x_2)) = j(p(x_1), p(x_2))$).

Remark 5.4 *the program obtained is a poor program following complexity analysis. The number of data constructions required to form the tree n^2 in worst case. Are there other better solutions derivable in the theory, to have, for instance, a program linear in the number of constructions ?*

Remark 5.5 *If we consider the proposition, we have to verify that, for a given x, y is such that eqfr(x,y) and llf(y) and consequently we have two tests concerning y. We will see that the proposed generalization strategy allows, by generalizing expressions into functions, to obtain a sort of data structure transformer (generalizing a data structure into a function to manipulate it. The point is also that if y is in letf-linear form then it has the form a or $snoc(a, x)$ with the relation $j(j(a, y), z) = j(a, j(y, z))$ as consequence.*

To propose some modification for better solution, we have to analyze the proof. A first idea consists in trying to apply λ-abstraction generalization to the initial version of the program. A second one can be to modify the definition of the type $Sexp(A)$ and then to modify the induction schema of the proof for obtaining by extraction a better program.

5.2.1 Generalization strategy

Let us consider the program (named $p(x)$) extracted from the previous proof of proposition 5.3, i.e., $p(x) \equiv srec(x, \lambda a.a, \lambda abcd.j(c, d))$, by abstraction strategy we obtain the following result:

Definition 5.3 *By λ-abstraction generalization, we associate to the term srec(x, λa.a, λabcd.j(c,d)) the generalized term $p_0(x) = \lambda z.j(z, p(x))$.*

We have $p_0(x)z = j(z, p(x))$ and we define $p_1(z, x) = p_0(x)z$. Moreover with z $= a_0$, we observe that $p_1(a_0, x) = j(a_0, p(x)) = snoc(a_0, p(x))$. and now have to prove the following proposition (or specification):

Proposition 5.4 $\forall x \in Sexp(A) \forall z \in Sexp(A) \exists y \in Sexp(A)\ y = p_1(z, x)$

Proof 5.4 *(of specification)*
1. Base step. $x \equiv a$ *with* $a \in Sexp(A)$.
Let z *an arbitrary element of* $Sexp(A)$, *we search* $y \in Sexp(A)\ y = p_1(z, a)$. *We have* $p_1(z, a) = j(z, p(a)) = snoc(z, a)$. *So we can take* $y \equiv snoc(z, a)$.
2. Induction step. $x \equiv snoc(x_1, x_2)$ *with* $x_1, x_2 \in Sexp(A)$
Let z *an arbitrary element of* $Sexp(A)$, *we have to find some* y *such that* $y = p_1(z, x) = j(z, p(x))$.
By induction hypothesis, we have
(I) $\forall z \in Sexp(A) \exists y_1 \in Sexp(A)\ y_1 = p_1(z, x_1)$ *and (II)* $\forall z \in Sexp(A) \exists y_2 \in Sexp(A)\ y_2 = p_1(z, x_2)$.
Moreover, $p_1(z, snoc(x_1, x_2)) = j(z, p(snoc(x_1, x_2))) =$
$j(z, j(p(x_1), p(x_2))) = j(j(z, p(x_1)), p(x_2))$ *(associativity of* j*, which proof is given below)*
$= j(y_1, p(x_2))$ *(application of (I))*
$= y_2$ *(application of (II) with* $z \equiv y_1$*). Then we can consider* $y \equiv y_2$.
The program issued from the proof is defined by the following equations: $p_1(z, a) = snoc(z, a)$ *and* $p_1(z, snoc(x_1, x_2)) = p_1(p_1(z, x_1), x_2)$.

From this proof, we can deduce that $p(x) = \pi_2(p_1(x, a0))$ (π_2 being the second projection of $snoc(x, y)$). Moreover, considering this definition, it is easy to verify that the program p is defined by $p(a) = a$, $p(snoc(a, x)) = snoc(a, p(x))$ and $p(snoc(snoc(x, y), z)) = p(snoc(x, snoc(y, z)))$.
To complete this subsection, we have to give the proof of associativity of j, property essential to derive a better solution using the λ-abstraction property for the program p. We will see that it has some connections with the new form of induction proposed below in the second idea consisting in new data definition.

Proposition 5.5 *Considering the function* $j(x, z) = srec(x, \lambda a.snoc(a, z), \lambda abcd.snoc(a, d))$, *we have the property:* $\forall x \in Sexp(A) \forall y \in Sexp(A)\ \forall z \in Sexp(A)\ j(x, j(y, z)) = j(j(x, y), z)$ *(which means that* j *is associative).*

Proof 5.5 *by induction on Sexp(A)*
1. Base step. $x \equiv a$.
Let y *and* z *arbitrary elements of* $Sexp(A)$, *we have*
$j(x, j(y, z)) = snoc(x, j(y, z)) = j(snoc(x, y), z) = j(j(x, y), z)$

2. Induction step. $x \equiv snoc(x_1, x_2)$
Let y *and* z *arbitrary elements of* $Sexp(A)$, *we have*
$j(x, j(y, z)) = j(snoc(x_1, x_2), j(y, z))$
$= snoc(x_1, j(x_2, j(y, z))) = snoc(x_1, j(j(x_2, y), z)$ *(by induction hypothesis on* x_2*)*
$= j(snoc(x_1, j(x_2, y)), z) = j(j(snoc(x_1, x_2), y), z)$
$= j(j(x, y), z)$

Remark 5.6 *We have observed that the proposed generalization strategy allows, by generalizing expressions into functions, to obtain a sort of data structure transformer (generalizing a data structure into a function to manipulate it.*

5.2.2 Data representation modification

The example has been developed in a transformation programming way whereas we can consider program derivation, with new induction choice and consequently modification of the data representation to derive classically the new version of program [15]. Extensions or modifications can be difficult in the framework and transformational approach can help us to obtain some specific programs. Considering the proposition 5.3 we know that for an arbitrary $x \in Sexp(A)$ we search for an $y \in Sexp(A)$ that is in left-linear form. In this case, we have y with the form a or $snoc(a, x_1)$ with a atome. To extend such form to the entire type $Sexp(A)$ we need explicitely an associativity rule of *snoc*. It is sure that it consists in a creative step for a new induction choice through a new data definition.
Let us consider the following modification of the data structure definition for $Sexp_1(A)$:

Representation of $Sexp_1(A)$

$$\frac{a \in A}{a \in Sexp_1(A)}$$

$$\frac{a \in A \quad y \in Sexp_1(A)}{snoc(a, y) \in Sexp_1(A)}$$

$$\frac{snoc(x, snoc(y, z)) \in Sexp_1(A)}{snoc(snoc(x, y), z) \in Sexp_1(A)}$$

$$\frac{c \in L(A) \quad \overset{[a \in A]}{x(a) \in C(a)} \quad \overset{[a \in A, s \in Sexp_1(A), t \in C(s)]}{y(a, s, t) \in C(snoc(a, s))} \quad \overset{[a \in A, b \in Sexp_1(A), c \in Sexp_1(A), u \in C(snoc(a, snoc(b, c)))]}{z(a, b, c, u) \in C(snoc(snoc(a, b), c))}}{srec_1(c, x, y, z) \in C(c)}$$

$srec_1(a, x, y, z) = x(a) \in C(a)$
$srec_1(snoc(a, y_1), x, y, z) = y(a, y_1, srec_1(y_1, x, y, z) \in C(snoc(a, y_1))$
$srec_1(snoc(snoc(a, b), c), x, y, z) = z(a, b, c, srec_1(snoc(a, snoc(b, c))), x, y, z) \in C(snoc(a, snoc(b, c)))$.

If we consider the new definition of data $Sexp(A)$, we can prove the proposition, specification of the problem.

Proposition 5.6 *(Specification)* $(\forall x \in Sexp(A))(\exists y \in Sexp(A))\ (llf(y) \wedge eqfr(x, y))$

Proof 5.6 *(of specification)*
1. Base step. $x \equiv a$ *with* $a \in Sexp(A)$.
Let z *an arbitrary element of* $Sexp(A)$*, we search* $y \in Sexp(A)$ $llf(y) \wedge eqfr(a, y)$*. So we can take* $y \equiv a$.
2. Induction step 1. $x \equiv snoc(a, x_2)$ *with* $a, x_2 \in Sexp(A)$
We have to prove $(\exists y \in Sexp(A))\ (llf(y) \wedge eqfr(snoc(a, x_2), y))$ *with* $a \in Sexp(A)$ *and the induction hypothesis* $(\exists y \in Sexp(A))\ (llf(y) \wedge eqfr(x_2, y))$.
Let us assume that z *satisfies* $llf(z) \wedge eqfr(x_2, z)$.
We have $eqfr(snoc(a, x_2), snoc(a, z))$ *and* $llf(snoc(a, z))$*. So we take* $y \equiv snoc(a, z)$.
3. Induction step 2. $x \equiv snoc(snoc(x_1, x_2), x_3)$ *with* $x_1, x_2, x_3 \in Sexp(a)$
We have to prove $(\exists y \in Sexp(A))\ (llf(y) \wedge eqfr(snoc(snoc(x_1, x_2), x_3))$ *with* $a \in Sexp(A)$ *and*

the induction hypothesis $(\exists y_1 \in Sexp(A))$ $(llf(y_1) \wedge eqfr(snoc(x_1, snoc(x_2, x_3)), y_1)$.
Let us assume that z *is such an* y_1. *We verify that* $eqfr(snoc(x_1, snoc(x_2, x_3)), snoc(snoc(x_1, x_2), x_3))$ *and we take* $y \equiv z$.
From this proof, we obtain a program pg defined by: $pg(a) = a$, $pg(snoc(a, x)) = snoc(a, pg(x))$ *and* $pg(snoc(snoc(x, y), z)) = pg(snoc(x, snoc(y, z)))$.

We have obtain the same program following two independant ways. The former consists in applying λ-abstraction strategy and then in proving the new specification and to extract the corresponding program. The point is that the new proposition is proposed in a systematic way (automation of the eureka step). Then the proof is obtained by induction on the type using the key property of j that is associativity. The second way consists in modifiying the induction proof, after an analysis of the initial specification. This step is a creative step that is difficult to automate. Then we have to develop the proof by induction on the new data representation and to extract the issued program.

Remark 5.7 *We can also use higher order generalization [14] to derive another version in our context. For instance, from the initial program f(x) of type Sexp(A)* $\rightarrow$ *Sexp(A) we have to prove the following proposition:* $\forall x \in Sexp(A) \forall z \in Sexp(A) \rightarrow Sexp(A) \exists y \in Sexp(A).y = z(f(x))$ *and we can derive other version of the initial program. In this way we have to investigate the use of the different transformation strategies in our framework.*

Remark 5.8 *Another useful program development tactic is the technique of argument generalization that suggests taking a program of type* $X \rightarrow Y$ *and introducing one of type* $Z \rightarrow Y$ *with an embedding from* $X \rightarrow Z$. *For the* fr *function given in 5.2, it means to take* $X = Sexp(A)$ *and* $Z = L(Sexp(A))$ *with as evident embbeding the injection* $i : Sexp(A) \rightarrow L(Sexp(A))$ *defined by* $i(s) = [s]$. *We specify this with the following proposition:* $\forall x \in L(Sexp(A)) \exists y_1 \in L(A)$ $y_1 = listrec(nil, g, x)$ *where* $g(cons(x, y), z)) = append(fr(x), z)$. *If we made a L(A)-induction, we observe that the proof can not be developed because of the* fr *definition. The only possibility is to define inductively the specific type* $L(Sexp(A))$ *and then to make a proof based on L(Sexp(A))-induction. The difficulty is to find the new definition leading to a new induction choice.*

Remark 5.9 *If we consider the strategies used in the both approaches, it appears that the associative law is essential. The first example used the associativity of append, the second one the associativity of the function j and of the function snoc. It is clear that it is dependant from the examples but it can be interesting to analyze this point by studying, for example, some work about the associative law in Martin-Löf type theory [13].*

We precise now some reflexions around the associative law in type theory [13]. The associative law is a basic axiom of the semigroup and a multiplicative expression is intended to denote an element in a semigroup. The definition can be given by: $Mul(Id) ::= at((Id)) \mid Mul(Id) \bullet Mul(Id)$ where Id labels the atomic parts of the expressions. We can defined, in the type theory syntax (through introduction and elimination rules), the type Mul. Moreover, we can define the concept of *normalized* expressions as a multiplicative expression that can not be transformed with the directed associative law (or it is either atomic or the product (by $\bullet$) of an atom and a normalized expression). Then we can define the type $Norm(x)[x : Mul]$ in the type theory syntax. Moreover, the equivalence of expressions is defined by the smallest equivalence relation, compatible with $\bullet$, which make $\bullet$ associative, which means that two expressions are equivalent if they can be proved equal using only associative law and equational reasoning. We can define the

type equivalence $equiv(x, y)[x, y : Mul]$ in the logical framework. In [13], the objective is to find an algorithm that for a given multiplicative expression produces a normal equivalent expression. We can assert that any function $f(x)[x : Mul]$ which verifies the equations $f(at(a)) = at(a)$, $f(at(a) \bullet z) = at(a) \bullet f(z)$ and $f((x \bullet y) \bullet z) = f(x \bullet (y \bullet z))$ is a correct normalizer.
Let us consider the second example (5.2) and the proposition (5.3), we consider the type $Sexp(A)$ which is a multiplicative expression (with $at(a) \equiv a$ and $x \bullet y \equiv snoc(x, y)$). The program p obtained by application of generalization strategy verifies the equations below and consequently is a correct normalizer. Then this strategy allows the construction of a normalizer by a constructive proof using the associativity of j. We can remark that if we want to use the type $L(A)$ excluding the empty list (denoted by $L(A)*$), it can be defined as a multiplicative expression by the correspondance $at(a) \equiv [a]$ and $x \bullet y \equiv append(x, y)$). So it is possible to treat different types through the more general type Mul and to work in a more general way.
Further work will be devoted to a study between the application of the λ-abstraction strategy and the proof of the new specification based on a type which is multiplicative expression and the general problem to search a normalizer. The treatment of multiplicative expressions leads in this way to a new choice of induction or a new definition of data representation.
The link with such work is clear but if we consider the technique of argument generalization mentioned in remark 5.8., the definition of the new type $LS(A)$ depend on an property different of associativity and further work will be necessary for analyzing various properties.

6 Conclusion

We have presented the notion of program transformation by proof in a constructive framework. In fact, the specifications, which are the propositions obtained after the generalization strategy, were simple and they admitted proof connected to a kind of *eureka definitions* traditionally used in theorem proving. Compared to the traditional program transformation, we are more concerned by proof transformation and by using additional information of a proof, even if the difference is more in formulation in the theory and in the structure and the role of the data. The point is here an attempt of automation for new specification definitions. In our constructive framework, program transformation by proof completes the classical derivation of programs from proofs, using the information in the proof. For example, to obtain accumulator versions of programs, transformation methodology seems easier that to write the specification leading to this version. Moreover, even in a powerful framework where it is possible to use general induction and to obtain efficient program, this one can include inefficient functions like append function. The generalization allows to obtain better programs and leads to a sort of generalization of initial data structure into a function to manipulate it. In a framework where there is no trivial way to extend or modify data representation, the use of transformational methods like generalization can be interesting.
We see also that what we obtain, in constructive type theory, by modification of data definition (to introduce a new induction schema) can be obtained by application of the generalization strategy. More studies about relationships between specification forms, data representation and program construction are necessary and also about program and proof transformation in programming with proofs framework. We have to consider other frameworks (devoted to this approach) like for instance second order type theory [20] with different data representations and program developments and investigate the notion of transformation. Moreover, some connections with work on the associative law seem interesting to deepen. It appears also important to study other program generalization techniques in type theory [11] and their applications for

proof and program development.

Acknowledgements

I should like to thank Bengt Nordström that indicates me some similarities with the work on associative law developed in his group and also the anonymous referees that provided advice.

References

[1] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[2] N.G. De Bruijn. A survey of the project automath. In *To H.B. Curry: Essays on Combinatory Logic, λ-calcul and Formalism*, pages 579–606. Academic Press, 1980.

[3] A. Bundy. Automatic guidance of program synthesis proofs. In *Workshop on Automating Software Design, IJCAI 89*, pages 57–59, Palo Alto, 1989.

[4] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[5] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[6] J. Darlington. Synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[7] P. Dybjer. Inductively defined sets in martin-löf's type theory. In *Workshop on General Logic, LFCS-88-52*, Edinburgh, February 1987.

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

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

[10] D. Galmiche and O. Hermann. Automated proof and program development. In C.M. Rattray & R.G. Clark, editor, *The Unified Computation Laboratory: Modelling, Specifications and Tools*, pages 397–410. Oxford University Press, 1992.

[11] M. Hagiya. Generalization from partial parametrization in higher-order type theory. *Theoretical Computer Science*, 63:113–139, 1989.

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

[13] M. Hedberg. Normalizing the associative law - an experiment with martin-löf's type theory. Technical report, Chalmers University of Technology, Göteborg, Sweden, 1990.

[14] M.C. Henson. Higher order transformations and type simulations. *Computer Journal*, 31(6):517–524, 1988.

[15] M.C. Henson. Program development in the constructive set theory tk. *Formal Aspects of Computing*, 1:173–192, 1989.

[16] P. Madden. The specialization and transformation of constructive existence proofs. In *11th IJCAI*, pages 413–418, Detroit, Michigan, August 1989.

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

[18] T. Nipkow. Proof transformations for equational theories. In *5th IEEE Symposium on Logic in Computer Science*, pages 278–288, Philadelphia, June 1990.

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

[20] 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.

[21] A. Pettorossi and A. Skowron. The lambda abstraction strategy for program derivation. *Fundamenta Informaticae*, XII:542–561, 1989.

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