Program Transformation in Martin-Löf's Type Theory

Alberto Pardo
FG Praktische Informatik
Technische Hochschule Darmstadt
Magdalenenstr. 11c
64289 Darmstadt - Germany
pardo@isa.informatik.th-darmstadt.de

Sylvia da Rosa
Laboratorio de Ciencia de la Computación
Instituto de Computación
Universidad de la República
Montevideo - Uruguay
darosa@fing.edu.uy

1 Introduction

Program Transformation in Functional Programming [3, 5] works on the basis of equational reasoning by means of transformation rules. This equational reasoning allows either the proof of properties, relating functional programs (e.g. by structural induction on data types), or the derivation of new functions from given ones by means of case analysis and strategies.

Work on program and proof transformation in constructive logic and its representation in logical frameworks has been the subject of recent research [9, 8, 1]. In this paper we show some results in an attempt to formalize the program transformation process in Martin-Löf's theory of sets (MTS) [11] (in which the functional programs, transformations and proofs are represented). This formalization is being developed with the help of the proof-assistant ALF [7], an implementation of Martin-Löf's logical framework.

Martin-Löf's type theory allows to represent functional programs, proofs and program derivations in an uniform way. As a functional language, it only allows to define total functions built by higher-order primitive recursion. With the new version of ALF it is possible also to define noncanonical constants by computation rules that are pattern matching equations (see [6]).

From the formal representation of basic transformation rules (the classical fold/unfold system [5]), we show how proofs and program derivations in the transformational approach can be represented directly in the type theory. The usefulness of this kind of direct representation is imaginable, if one thinks about it as a transformation-checker for transformations performed outside the type theory.

The organization of this extended abstract is as follows. Section 2 presents a general formal definition of the basic transformation rules, independent from the particular equality relation. Section 3 comments how proofs and program synthesis can be represented in MTS. Section 4 discusses two possibilities on the election of equality and introduces the definition of an extensional equality using MTS with a universe. Finally, Section 5 presents some conclusions and further work.

2 Definitions and Basic Rules

The formal identification between propositions and sets arising in constructive logic (identifying also proofs and programs) allows to work with MTS as a functional programming language. Taking sets as datatypes functional programs can be developed by the introduction of new noncanonical constant definitions to the set theory. When working with ALF these definitions can be made in two different manners. One is to define each new noncanonical constant in terms of the the elimination constant (like natrec or listrec) of the set involved. This is the standard presentation of MTS. The other makes use of the pattern-matching approach included in ALF that allows the introduction of noncanonical constants defined by computation rules that are pattern-matching equations which may be recursive (having no syntactical criteria for ensuring correctness).

In the transformation approach some function definitions appear as specifications or initial program versions from which new and more efficient programs can be derived using transformations. The derivation process by transformations is developed by making use of case analysis, induction and strategies [4, 12, 13], which in general introduce auxiliary function definitions called *Eurekas*. In this way, e.g. recursive function definitions can be synthesized from given specifications.

In the transformational methodology, steps in an equational proof or program derivation are the result of applying the basic transformation rules introduced in [5] so far. A formal definition of these rules in MTS is presented below. From this a formal representation of instances of transformation strategies can be studied. The definition of the rules is given in terms of a general equality relation $\simeq$. The details about this relation, its definition and its properties is the subject of Section 4. Equations are denoted

$$h \in e \simeq_A e' \quad [x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})]$$

meaning that h is a proof of the equality between the elements e and e' in the set A under the assumptions. The set of functions between A and B is denoted $A \to B$.

2.1 Instantiation

The **Instantiation** rule introduces a substitution instance of an existing equation. Most of the time this rule is used to instantiate a given definition in a program synthesis. In the formal system of MTS there are some general rules concerning substitution, as for instance,

$$\frac{\begin{array}{l} c(x_1, \ldots, x_n) \in C(x_1, \ldots, x_n) \quad [x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})] \\ \qquad\qquad a_1 \in A_1 \\ \qquad\qquad \vdots \\ \qquad\qquad a_n \in A_n(a_1, \ldots, a_{n-1}) \end{array}}{c(a_1, \ldots, a_n) \in C(a_1, \ldots, a_n)}$$

which is the rule for simultaneous substitution in elements of n variables. In order to achieve instantiation of equations these general substitution rules of MTS can be used. Therefore, a instantiation step in a formal equational derivation will be filled with the instantiated proof-object $c(a_1, \ldots, a_n)$ of the corresponding (instantiated) equation.

2.2 Unfold

If $e_1 \simeq e_2$ and $a_1 \simeq a_2$ are equations and there is some ocurrence of a_1 in e_2, then that occurrence can be replaced by a_2 obtaining the new equation $e_1 \simeq e_2[a_1 \leftarrow a_2]$. The expression e_2 can be seen as obtained by the application of certain function e to a_1 and then, $e_2[a_1 \leftarrow a_2]$ is simply the application of e now to a_2. Putting $e_1 \simeq e_2$ in this way, **Unfold** is defined by the rule:

$$\frac{e \in A \to B \qquad h_e \in e_1 \simeq_B \mathit{apply}(e, a_1) \qquad h_a \in a_1 \simeq_A a_2}{\mathit{unfold}(e, h_a, h_e) \in e_1 \simeq_B \mathit{apply}(e, a_2)}$$

which can be derived by applying transitivity (*trans*) and extensionality (*ext*) of equality as follows:

$$\frac{h_e \in e_1 \simeq_B \mathit{apply}(e, a_1) \qquad \dfrac{e \in A \to B \qquad h_a \in a_1 \simeq_A a_2}{\mathit{ext}(e, h_a) \in \mathit{apply}(e, a_1) \simeq_B \mathit{apply}(e, a_2)}}{\mathit{trans}(h_e, \mathit{ext}(e, h_a)) \in e_1 \simeq_B \mathit{apply}(e, a_2)}$$

and then,

$$\mathit{unfold}(e, h_a, h_e) \equiv \mathit{trans}(h_e, \mathit{ext}(e, h_a))$$

2.3 Fold

If $e_1 \simeq e_2$ and $a_1 \simeq a_2$ are equations and there is some ocurrence of a_2 in e_2, then that occurrence can be replaced by a_1 obtaining the new equation $e_1 \simeq e_2[a_2 \leftarrow a_1]$. Reasoning as for **Unfold**, **Fold** can be defined by the rule:

$$\frac{e \in A \to B \qquad h_e \in e_1 \simeq_B \mathit{apply}(e, a_2) \qquad h_a \in a_1 \simeq_A a_2}{\mathit{fold}(e, h_a, h_e) \in e_1 \simeq_B \mathit{apply}(e, a_1)}$$

which can be derived using symmetry (*symm*), transitivity (*trans*) and extensionality (*ext*) of equality:

$$\mathit{fold}(e, h_a, h_e) \equiv \mathit{trans}(h_e, \mathit{ext}(e, \mathit{symm}(h_a)))$$

2.4 Lemmas and Induction Hypothesis

In the transfromational approach Lemmas and Induction Hypothesis are always equations which during derivations can be applied from left to right or in the opposite direction. This can be seen as the simple application of the rules **Fold/Unfold**. For motives of expresiveness in the derivations, appropriate constant definitions (one for each application direction) can be introduced in order to capture these special cases on the application of rules **Fold/Unfold**:

$$\begin{array}{ll} \mathit{ind\_hyp\_lr}(e, h_a, h_e) \equiv \mathit{unfold}(e, h_a, h_e) & \mathit{lemma\_lr}(e, h_a, h_e) \equiv \mathit{unfold}(e, h_a, h_e) \\ \mathit{ind\_hyp\_rl}(e, h_a, h_e) \equiv \mathit{fold}(e, h_a, h_e) & \mathit{lemma\_rl}(e, h_a, h_e) \equiv \mathit{fold}(e, h_a, h_e) \end{array}$$

where $e \in A \to B$, $h_e \in e_1 \simeq_B \mathit{apply}(e, a_2)$ and $h_a \in a_1 \simeq_A a_2$.

2.5 Abstraction

This rule allows the introduction of a new equation from an existing one, by means of the introduction of a local definition in terms of a where clause. If $e_1 \simeq e_2$ is an equation with e_2 containing the subexpression a, then the following equation can be derived: $e_1 \simeq e_2[a \leftarrow u]$ **where** $u = a$, where in e_2 the subexpression a has been replaced by the variable u and abstracted in a **where** clause. As MTS does not provide local definitions in expressions **where** clauses will be represented by the expressions they abbreviate. An expression of the form e **where** $u = a$ is really an abbreviation for $((\lambda u.e)\ a)$. So, given an equation $e_1 \simeq e_2$, if its rhs e_2 can be seen as a function e applied to an object a, then the corresponding abstraction of the subexpression a was found. Hence, **Abstraction** can be defined by the rule:

$$\frac{e \in A \to B \qquad a \in A \qquad h_1 \in e_1 \simeq_B e_2 \qquad h_2 \in e_2 \simeq_B \mathit{apply}(e, a)}{\mathit{abstr}(h_1, h_2) \in e_1 \simeq_B \mathit{apply}(e, a)}$$

that is a simple application of transitivity (*trans*) of equality:

$$\mathit{abstr}(h_1, h_2) \equiv \mathit{trans}(h_1, h_2)$$

3 Proofs and Synthesis

Properties relating functional programs can be proved using the logical aspect of MTS. In the transformational approach properties are normally equations to be asserted,

$$a \simeq_A a' \;\; [x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})]$$

where the structure of the proofs will depend on the equality being used. In general, these equational properties are proved by structural induction on the assumptions, making use of the definitions of the functions involved and applying in each step a transformation rule. In MTS these inductive equational proofs can be developed either using the elimination constants of the corresponding sets or by pattern-matching (when working with ALF). As a result, the proof-object obtained encodes the corresponding equational proof in the transformational approach.

More interesting is the way in which program synthesis can be represented. Specifications appear as function definitions of the form:

$$f(x_1, \ldots, x_n) \equiv e \in A \quad [x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})]$$

from which, again applying case analysis and induction, new program definitions can be derived. The introduction of different intermediate auxiliary *Eureka* function definitions may be necessary, according to which strategy is used. Assuming all necessary definitions are made, the synthesis of a recursive definition of the function given by a specification like above can be represented in MTS as the following predicate:

$$(\exists y \in A) f(x_1, \ldots, x_n) \simeq_A y \quad [x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})]$$

which can be proved by structural induction on the assumptions (using the corresponding elimination constants or by the pattern-matching approach in ALF), where the specification plays the role of the induction hypothesis. Each case of this inductive proof is constructed using the introduction rule for the existential quantifier:

$$\frac{a \in A \qquad B(x)\ \ Set\ \ [x \in A] \qquad b \in B(a)}{\langle a, b\rangle \in (\exists x \in A)B(x)}$$

In each inductive case a witness a is constructed (corresponding to the body of the derived recursive definition) together with the proof b that meets the specification $B(a)$, i.e. $f(v_1, \ldots, v_n) \simeq_A a$, where $v_1, \ldots, v_n$ are the correspondig values assigned to the variables $x_1, \ldots, x_n$ in that case. The proof-object b is constructed using the transformation rules, encoding, as with the proofs, the corresponding steps of the synthesis in the transformational approach. Finally, using the witness, a recursive function definition can be extracted from the whole proof.

3.1 An Example

This example shows how a simple program derivation can be performed and represented in MTS, using as equality $\simeq$ the usual intensional equality of Martin-Löf's type theory (see next section). First we present the original definitions and program derivation in a functional notation, close to the one in [3]. After that we describe their representation in MTS.

Consider the function append for lists given by the following specification:

append xs ys = foldr (λa.λas.a:as) ys xs	Spec

where : is the function cons (constructor of lists) and foldr is given by the usual definition:

foldr f e [ ]	=	e	foldr.1
foldr f e (x:xs)	=	f x (foldr f e xs)	foldr.2

A recursive definition of append can be synthesized as follows:

Case [ ]

append [ ] ys	=	foldr (λa.λas.a:as) ys [ ]	Instantiation Spec
	=	ys	Unfold foldr.1

Case (x:xs)

$$\begin{array}{rcll} \text{append (x:xs) ys} & = & \text{foldr } (\lambda\text{a}.\lambda\text{as.a:as}) \text{ ys (x:xs)} & \textsf{Instantiation Spec} \\ & = & \text{x : (foldr } (\lambda\text{a}.\lambda\text{as.a:as}) \text{ ys xs)} & \textsf{Unfold foldr.2} \\ & = & \text{x : (append xs ys)} & \textsf{Fold Spec} \end{array}$$

Let A and B be sets; the corresponding function definitions in MTS are:

$$append(xs, ys) \equiv foldr(\lambda a.\lambda as.cons(a, as), ys, xs) \in \mathrm{List}(A) \quad [xs \in \mathrm{List}(A),\ ys \in \mathrm{List}(A)]$$

$$foldr(f, e, nil) \equiv e \in B \quad [f \in A \to (B \to B),\ e \in B]$$

$$foldr(f, e, cons(x, xs)) \equiv apply(apply(f, x), foldr(f, e, xs)) \in B$$
$$[f \in A \to (B \to B),\ e \in B,\ x \in A,\ xs \in \mathrm{List}(A)]$$

simulating pattern-matching definitions as in ALF. Finally, using intensional equality the synthesis is represented by the following proposition in MTS:

$$(\exists y \in \mathrm{List}(A))\, append(xs, ys) =_{\mathrm{List}(A)} y \quad [xs \in \mathrm{List}(A),\ ys \in \mathrm{List}(A)]$$

whose proof-object in each case of the inductive proof is as follows:

Case *nil*

$$\begin{array}{l} \langle ys, \\ \quad unfold(\lambda z.z, \\ \qquad id(append(nil, ys)), \\ \qquad id(foldr(\lambda a.\lambda as.cons(a, as), ys, nil)))\rangle \ \in \ (\exists y \in \mathrm{List}(A))\, append(nil, ys) = y \end{array}$$

Case $cons(x, xs)$

$$\begin{array}{l} \langle cons(x, append(xs, ys)), \\ \quad fold(\lambda z.cons(x, z), \\ \qquad id(append(xs, ys)), \\ \qquad unfold(\lambda z.z, \\ \qquad\quad id(foldr(\lambda a.\lambda as.cons(a, as), ys, cons(x, xs))), \\ \qquad\quad id(append(cons(x, xs), ys)))))\rangle \ \in \ (\exists y \in \mathrm{List}(A))\, append(cons(x, xs), ys) = y \end{array}$$

4 Equality

The definition of the basic transformation rules does not depend on which definition of equality is being used, only on the properties of equality. In this section two possible alternatives are presented: one with the use of an intensional equality and the other using an extensional equality defined in terms of a universe. There are some common properties that can be stated for both equalities. The crucial difference is, maybe, the class of transformations that can be developed with either definitions of equality.

4.1 Intensional Equality

One possibility is to represent the transformation system (rules, proofs and program derivations) based on an intensional version of equality for the relation $\simeq$. In MTS, an intensional equality can be defined by the following family of sets:

Formation

$$\frac{A\ Set \qquad a \in A \qquad b \in A}{a =_A b\ \ Set}$$

Introduction

$$\frac{a \in A}{id(a) \in a =_A a}$$

This set is defined as the least reflexive relation among objects of the respective set. In other words, two objects are proved equal if both evaluate to the same canonical. It is possible to prove symmetry, transitivity, congruence (which in this case is the same as the extensionality property used in Section 2) and substitutivity for it (for a complete definition see [11]). Using this equality there is no restriction about the sets on which transformations can be applied. $a =_A b$ must be written instead of $a \simeq_A b$ on the general formulation of the transformation rules presented in Section 2.

4.2 Extensional Equality

Working with an intensional equality, involves a very strong dependence on definitional concepts. By an equality of this kind objects are identical when they have the same normal form, but not when they behave in the same way although corresponding to different definitions.

When comparing programs one would like to state that two programs compute the same function despite they have different definitions. An extensional equality says that if they are identical as far as their functional behaviour is concerned, then they are taken as identical. Using an intensional equality like the above, the best that can be asserted concerning the behaviour of functions is the fact that when applied to the same object they return intensionaly equal objects:

$$\mathit{apply}(f, x) =_B \mathit{apply}(g, x) \;\; [x \in A] \qquad (\dagger)$$

Of course, from this $f =_{A \to B} g$ cannot (and in general is out of interest to) be concluded. Using facts like ($\dagger$), programs can be compared and transformations can be developed. In general, in the transformational approach, ($\dagger$) is of interest when B is not a functional type (or does not contain a functional type as a component in case of a compound type like tuple); or when the proposition states a structural fact (a fact in which the contents of the objects is not a matter, only the structure)[1]. If B is a functional type, a fact like this would state that for each object $x \in A$ both applications result in functions with equal definition. As opposed to this, the common procedure is to saturate both f and g with all posible arguments and selectors until an atomic type is reached, and then on this type the equality is proved. This is one way in which equality can be stated in 'extensional' terms, but using an intensional equality to express it.

The explicit definition of a set for an extensional equality would be a more natural and expressive way to deal with the behaviour of programs. Such an equality cannot be formulated in a general form for all possible sets, like the way intensional equality is introduced. It must be given an introduction rule for each set that this equality is intended to be applied, i.e. it must be said how to prove equality for each particular set. Hence, it is necessary to state two things about sets: to be able to speak at the object level about them and to restrict the application of equality to some of them. The definition of a universe (a set U) with canonical elements coding the sets to be handled by equality is the solution for these requirements. The technique in defining a universe can be found in [11].

The universe presented here only contains codes for the sets of natural numbres, booleans, pairs, lists and functions. $\mathrm{T}(x)$ *Set* $[x \in U]$ is the family of sets which decodes the elements in the universe to the set they represent. The formation rule for the universe is simply: U *Set*. The introduction rules are:

$$\widehat{\mathrm{N}} \in U \qquad \mathrm{T}(\widehat{\mathrm{N}}) = \mathrm{N} \qquad \widehat{\mathrm{Bool}} \in U \qquad \mathrm{T}(\widehat{\mathrm{Bool}}) = \mathrm{Bool}$$

$$\frac{A \in U \quad B \in U}{A \widehat{\times} B \in U} \qquad \frac{A \in U}{\widehat{\mathrm{List}}(A) \in U} \qquad \frac{A \in U \quad B \in U}{A \widehat{\to} B \in U}$$

$$\frac{A \in U \quad B \in U}{\mathrm{T}(A \widehat{\times} B) = \mathrm{T}(A) \times \mathrm{T}(B)} \qquad \frac{A \in U}{\mathrm{T}(\widehat{\mathrm{List}}(A)) = \mathrm{List}(\mathrm{T}(A))} \qquad \frac{A \in U \quad B \in U}{\mathrm{T}(A \widehat{\to} B) = \mathrm{T}(A) \to \mathrm{T}(B)}$$

[1]For instance, when a proposition relating polymorphic functions on structures is proved by structural induction, like, $\mathit{map}\ f\ (\mathit{map}\ g\ xs) = \mathit{map}\ (f \circ g)\ xs$

A set corresponding to an extensional equality is presented here, defined as a family of sets indexed on an object of the universe and two elements. It is denoted $a \approx_A b$ and its formation rule is:

$$\frac{A \in U \qquad a \in \mathrm{T}(A) \qquad b \in \mathrm{T}(A)}{a \approx_A b \ \ Set}$$

Giving a rule for proving equality for each set is not enough. In order to obtain for this family at least the properties that one would like an (extensional) equality to meet, it is necessary to assert extra explicit properties as introduction rules: symmetry, transitivity and extensionality. It seems to be that these are the weakest conditions to be imposed. Then, the introduction rules are:

$$\frac{n \in \mathrm{N}}{n \approx_{\widehat{\mathrm{N}}} n} \qquad \frac{b \in \mathrm{Bool}}{b \approx_{\widehat{\mathrm{Bool}}} b} \qquad \frac{a \approx_A a' \qquad b \approx_B b'}{\langle a, b\rangle \approx_{A \widehat{\times} B} \langle a', b'\rangle}$$

$$\frac{A \in U}{nil \approx_{\widehat{\mathrm{List}}(A)} nil} \qquad \frac{a \approx_A a' \qquad l \approx_{\widehat{\mathrm{List}}(A)} l'}{cons(a, l) \approx_{\widehat{\mathrm{List}}(A)} cons(a', l')} \qquad \frac{apply(f, a) \approx_B apply(g, a') \ \ [a \approx_A a']}{f \approx_{A \widehat{\to} B} g}$$

$$\frac{a \approx_A b}{b \approx_A a} \qquad \frac{a \approx_A b \qquad b \approx_A c}{a \approx_A c} \qquad \frac{f \in \mathrm{T}(A \widehat{\to} B) \qquad a \approx_A a'}{apply(f, a) \approx_B apply(f, a')}$$

Excluding the rule that asserts extensionality (the last one) the rest of the rules are the definition of a Partial Equivalence Relation [10] relating objects of sets coded by the universe. The addition of the extensionality condition to the definition makes it possible to prove reflexivity and congruence for this relation:

$$\frac{A \in U \qquad a \in \mathrm{T}(A)}{a \approx_A a} \qquad \frac{f \approx_{A \widehat{\to} B} g \qquad a \approx_A a'}{apply(f, a) \approx_B apply(g, a')}$$

With this equality explicitly defined, many extensional properties about functionals can be now asserted simply using congruence. Other interesting properties are:

$$\frac{a =_{\mathrm{T}(A)} b}{a \approx_A b} \qquad \frac{apply(f, a) \approx_B apply(g, a) \ \ [a \in \mathrm{T}(A)]}{f \approx_{A \widehat{\to} B} g} \qquad \frac{f \approx_{A \widehat{\to} B} g \qquad a \in \mathrm{T}(A)}{apply(f, a) \approx_B apply(g, a)}$$

where the first one asserts an expected intuitive result relating intensional and extensional equality, the second one simplifies the task of proving equality between functions giving a rule to state it by induction and the third one is a special case of congruence. Extensional equality is not substitutive (Leibnitz's Law does not hold), counterexamples can be found, for instance, using propositions that relate functions by intensional equality.

Having defined this equality a version of the transformation rules adapted to work with can be obtained. Now, these rules are only applied to sets coded by the universe U, making it necessary to include premisses of the form $A \in U$ to them. Other changes are: set membership must be written in the form $a \in \mathrm{T}(A)$, where $A \in U$, so $e \in A \to B$, as appears in the rules, is now replaced by $e \in \mathrm{T}(A \widehat{\to} B)$; and $\simeq_A$ must be replaced by $\approx_A$.

5 Conclusions and Further Work

This work made an attempt to show how transformations can be represented and developed in a constructive type theory like MTS. That led into a discussion about the most appropriate equality to be applied in order to reason about program behaviour. Also, the question of how to define a set for extensional equality was raised.

The formal representation of the transformation rules and its application in proofs and program derivations in MTS can be very helpful in order to be used from outside the type theory as a transformation-checker. More concretely, in the way that function definitions, transformation rules, proofs and synthesis are represented in MTS, and mechanized in ALF (or in other logical frameworks), they can be used to check correctness of transformations performed in a transformation system outside MTS. In this way, definitions, proofs and synthesis could be translated into their direct representation in the type theory to be checked for correctness (i.e. using the type system of MTS, transformation-correctness would be reduced to well-typedness of the representations).

Last but not least another important (and maybe) more promising way of doing a formalization of the transformation process in MTS can be briefly commented. This consists, first, on the definition of the functional language as a new set in MTS (the set of its terms). Doing that, no restriction on the form of the functional programs would be imposed (general recursion and partial functions would be possible). Then, the operational semantics of the language, the notion of reduction and conversion can be defined. Finally, on this system it is possible to define and compare different notions of equalities (extensional, observational) as in [2], to define the transformation rules, etc..

References

[1] P. Anderson. Representing Proof Transformations for Program Optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, Nancy, France, June 1994. Available also as Rapport de recherche 2229, INRIA, Sophia-Antipolis, May 1994.

[2] S. Berardi. Extensional equality for simply typed λ-terms. Dipartimento di Informatica, Universitá di Torino, Torino, Italy, 1993.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, UK, 1988.

[4] R.S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.

[5] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *JACM*, 24(1):44–67, January 1977.

[6] T. Coquand. Pattern matching with dependent types. In *Proceedings from the Logical Frameworks Workshop at Båstad*, Sweden, June 1992.

[7] T. Coquand, B. Nordström, J.M. Smith, and B. von Sydow. Type Theory and Programming. EATCS, to appear, 1994.

[8] D. Galmiche. Program transformation by proof in constructive framework. In *Informatika'91, Theoretical Computer Science and Methods of Compilation and Program Construction*, 1991.

[9] P. Madden. Automatic Program Optimization Through Proof Transformation. In *Proceedings of the 11th International Conference on Automated Deduction, Lecture Notes in Artificial Inteligence Vol. 607*, Saratoga Springs, New York, June 1992. Springer-Verlag.

[10] J.C. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science*, Elsevier Science Publishers, 1990.

[11] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[12] Alberto Pettorossi. Derivation of programs which traverse their input data only once. *Advanced Programming Methodologies*, 30, 1989.

[13] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.