Program Development in Constructive Type Theory*

Didier GALMICHE
Centre de Recherche en Informatique de Nancy
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex
France.

Abstract

We present the program development concept in a logical framework including constructive type theory and then show how to use such theories to derive programs from proofs of formal specifications. We are interested in two important facts that are the mechanization of the proof construction and the possibility to express in the theory significiant concepts for programming (like inductively defined types and general recursion). We give here a survey on some results and problems appearing in logical frameworks devoted to the *programming with proofs* approach.

1 Introduction

There has been lately a great deal of interest in development of programs in constructive frameworks. Differents theories have been studied in this way like Martin-Löf's constructive type theory [3] [9], [18], Feferman's theory T_o and its applications [13], Calculus of Constructions [4] or variants of intuitionnistic ZF set theory [14]. Their applications and corresponding systems have been developed towards verification of proofs and, more generally, interactive development of correct programs in specific logics [12], [20]. The use (and the adaptation) of proof concepts for programming and more precisely for program synthesis [13] leads to the *programming with proofs* notion which will be illustrated in this paper.
It is a necessity in computer science to have correct programs and many works in theoretical computer science have been focused on program verification. To this end, the classical solution is to construct a logic for programs based on the external description of their behaviour but impossibility results limit this type of technics. Consequently, we should make a language based on classical logical principles available where correctness is naturally present. That is what happens in mathematics when the correctness of a proof is not a difficult problem but finding (constructing)

*Theoretical Computer Science 94, 237-259, 1992

it, is generally a bigger one. We can thus consider mathematics as a programming language to a certain extent because writing a program, satisfying a given specification, will be writing a proof of a proposition expressing this specification. In this case correctness can be verified automatically and the automatized construction of programs can also be considered.

The study of the relationships between intuitionistic logic and computer science with the underlying formulae-as-types interpretation is in center of the development of programs in logical frameworks. Moreover the classical isomorphism between typed λ-calculus and intuitionistic logic explains why the frameworks are often based on variants of typed λ-calculus. From a computing point of view, typing is important and the recent research works suggest much more powerful typing systems which are strong enough to describe specifications of programs. The notion of program development in these calculi is a general scheme consisting generally in three steps: writing a logical formula as specification of the task, constructing a proof of this formula (specification) and extracting a program from this proof. One important point is, using such approach, to obtain correct programs. Developing adapted formal systems for proofs is necessary but it is not sufficient to consider these directly as programs. Moreover, to obtain a language *to program with proofs*, it is necessary not only to be able to characterize the proofs with algorithmic information but also to know how to extract the programs.

Some works start from existing logical framework (and theorem provers) to derive programs [10], [20] and others from more pragmatic programming approach defined on adapted logical systems [26]. Research in program development using such theories has lighted a number of interesting problems. These appear as a consequence of the confrontation between the program development task and logical theories. We want to understand what being a good proof or a good type structure means with a view to obtaining theories allowing to operate with classical program and type structure. For instance, a language and the logic to express programs or nature of structure cannot be directly conform to those used in programming activity.

Here, we deal with constructive type theory [18] and want to show how to use it to derive programs (through some necessary extensions to obtain interesting programs). We can mention other theories *to program with proofs* [14], [15] confronted to similar problems (perhaps under another form but) with the same essential constraint to have extensions or reformulations that preserve properties like correctness or termination of the extracted programs. From a logical point of view, a constraint will be to have extensions formalized inside the theory.

Section 2 relates to some important aspects of constructive type theory concerning generalities about types as specifications and proofs as programs. It includes a formulation of a logical framework and the representation of type theory and particular types like naturals, lists and wellorderings. Section 3 presents programming (with proofs) in constructive systems , consisting in extraction of programs from the proofs of their formal specifications. We describe the different steps of the program development i.e. specification, proof construction and program extraction and present a way to achieve the main goal which is the mechanization of proof construction (with tactics) [22]. Finally we give examples of proof derivations using different strategies to illustrate the will and the difficulty of extensions of the theory we have to formalize. Section 4 is devoted to the representation of inductively defined types and the derivation of recursion operators for such types. Section 5 presents the problem to include general recursion in the theory. Using the example of derivation of the quicksort, we explain some ways to have general recursion and the consequences for derivation of efficient programs from formal specifications. Section 6 contains some comparisons with other connected works and the conclusion.

2 Constructive type theory

Here, we want at first to present Martin-Löf's type theory [18] (denoted by *MLTT*), underlying the identification between propositions types and specifications and the relationships between proofs and programs. Secondly we present a more general logical framework [7] and the formulation of *MLTT* in it illustrated by representation of datas like naturals or lists.

2.1 General ideas about type theory

Martin-Löf's type theory can be seen as a programming logic for functional programming language. The formal system can be used to express programs, their specification and also derivations of programs. Type theory is based on the identification between types propositions and specifications [18]. Among the four classical judgments of the theory, the important one is $a \in A$ (other notation for $\in$ is :) that can be read as 1) a is an element of type A, 2) a is a construction (proof) for the proposition A, 3) a is a solution of task or problem A, 4) a is a program with type A and 5) a is a program which satisfies the specification A.

The type structure is very strong but program construction operations can appear weak since, for example, general recursion is not directly available. Such apparent weakness can be a serious problem from the programmer's point of view (termination proved during program derivation, estimation of the number of iterations in a program ...) It seems natural to extend the type theory with operators allowing general recursion representation.

Let us consider now the connexions between propositions, types and specifications. A proposition is identified with its proof objects and a specification is identified with the programs satisfying the specification. For instance the proposition $\exists x : A.B(x)$ is interpreted as $\Sigma x : A.B(x)$ which is the disjoint union of family of types. A proof of $\exists x : A.B(x)$ consists of two things : an element a in A and a proof of $B(a)$ as elements in $\Sigma x : A.B(x)$ consisting of pairs (a, b) where a element in A and b element in $B(a)$. In the same way the proposition $\forall x : A.B(x)$ is interpreted as the type $\Pi x : A.B(x)$, cartesian product of a family of types, and a proof of $\forall x : A.B(x)$ consisting of a method by which to $x : A$ corresponds a proof of $B(x)$. Moreover we have basic types N (type of naturals), $Bool$ and enumeration types, the type $Wx : A.B(x)$ used to represent inductively defined sets and finally an univers U containing types as elements.

Considering programs as proofs and identifying specifications with propositions gives an interesting approach of problems concerning specification and programming languages. But, there are some problems to consider 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. The distinction between programs and proofs is clear if we consider the treatment of equality type. Moreover if we consider a proof of $\exists x : A.B(x)$ it is a pair (a, b) where $a : A$ and $b : B(a)$ and if we take this statement as a specification we are only interested in the first element a. For instance, a general specification of a sorting problem, considering the type $List(A)$ (noted $L(A)$) is $\forall x : L(A) \exists y : L(A).sort(x, y)$ with $sort(x, y)$ predicate that is true if y is an ordered permutation of x. If we have a solution of the problem (i.e. a program p) it consists in a function which when, applied to a list x, gives a pair (a, b), a being an ordered permutation of x and b a proof of *sort(a,x)*. A possibility is to use the subset notion $\{x : A/B(x)\}$ type of all elements a in A for which $B(a)$ is true [24]. It is like the type $\Sigma x : A.B(x)$ but with the second component removed. A correct specification for our sorting algorithm will be $\Pi x : L(A).\{y : L(A)/sort(x, y)\}$, the programs satisfying it being the set of all sorting programs.
We can, also, observe differences between proofs and programs in the repetitive constructs like *while* or *repeat loops* or *general recursion* (in functional language). In proofs such constructions are based on some kind of structural induction. Another problem is the termination of programs 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 [24], is an extension with general recursion.

2.2 A logical framework

We use, here, a presentation of intuitionnistic type theory in a lambda calculus framework [7] like in [8], which has an important connexion with ELF [12] and Automath languages [2]. This formulation shows a presentation of logical systems in typed theory.

2.2.1 Type theory framework

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 dependant types, a ground type U_1 and a rule saying that every object of type U_1 is also a type. We give the rules of formation concerning the types and the objects. The rules of type formation are :

$$U_1 : U$$

$$\frac{A : U_1}{A : U}$$

$$\frac{\alpha : U \quad \overset{(x:\alpha)}{\beta : U}}{(x : \alpha)\beta : U}$$

The rules of object formation are :

$$\frac{\overset{(x:\alpha)}{b : \beta}}{x.b : (x : \alpha)\beta}$$

$$\frac{b : (x : \alpha)\beta \quad a : \alpha}{ba : \beta[a/x]}$$

The equality rules are typed β and η conversions and correspond to $(x.b)a = b[a/x] : \beta[a/x]$ if $a : \alpha$ and $b : \beta$ with $x : \alpha$ and $c = x.cx \ : \ (x : \alpha)\beta$ if $c : (x : \alpha)\beta$. We can include a notion of polymorphism (which plays the role of second order quantification in other theories [4], [15]) and we give polymorphic rules for objects and types.
The rule of p-type formation is :

$$\frac{\alpha : U \quad \overset{(x:\alpha)}{\beta : U}}{\overline{(x : \alpha)}\beta : U}$$

The rules of object formation are

$$\frac{\overset{(x:\alpha)}{b : \beta}}{b : \overline{(x : \alpha)}\beta}$$

with b not depending on x.

$$\frac{b : \overline{(x : \alpha)}\beta \quad a : \alpha}{b : \beta[a/x]}$$

In such a framework, a theory consists in a signature i.e. a list of assignments of types to constants and a finit list $a_1 = b_1 : \beta_1, \ldots, a_m = b_m : \beta_m$ of axioms and equations. Definitions can be introduced with a judgement of the form $c = a : \alpha$ with c being a constant, an object and α their type. For further details, you can refer to [9], [18].

2.2.2 Representation of MLTT

In the framework, we can represent Martin-Löf's type theory [18]. 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 present here type definitions that will be used in the rest of the paper to illustrate the program development process.

Example 2.1 *The representation of naturals is :*

$N : U_1,\ 0 : N,\ S : (N)N,$

$natrec : \overline{(C : (N)U_1)}(C\ 0)((a : N)(Ca)C(sa))(c : N)C\ c$

$natrec\ d\ e\ 0\ = d : \overline{(C : (N)U_1)}(C\ 0)((a : N)(Ca)C(sa))C\ 0$

$$natrec\ d\ e\ (sa) = e\ a(Rec\ d\ e\ a) : \overline{(C : (N)U_1)}\\ (C\ 0)((a : N)(Ca)C(sa))(a : N)C(sa)$$

Example 2.2 *The representation of lists of type A elements is :*

$L : (A : U_1)U_1$

$nil\ :\ \overline{(A : U_1)}L\ A$

$cons\ :\ \overline{(A : U_1)}(a : A)(b : L\ A)L\ A$

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

$$listrec\ x\ y\ nil = y\ :\ \overline{(A : U_1)(C : (list\ A)U_1)}\\ (C\ nil)((a : A)(b : L\ A)(C\ b))(C(cons\ a\ b))C\ nil$$

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

Example 2.3 *The representation of well-orderings is :*

$W : (A : U_1)((A)U_1)U_1$

$sup : \overline{(A : U_1)(B : (A)U_1)}(a : A)((B\ a)W\ A\ B)W\ A\ B$

$$wrec : \overline{(A : U_1)(B : (A)U_1)(C : (W\ A\ B)U_1)}\\ ((a : A)(b : (B\ a)W\ A\ B)((v : B\ a)C(b\ v))C(sup\ a\ b))(c : W\ A\ B)C\ c$$

$wrec\ d\ (sup\ a\ b) =\ d\ a\ b\ (wrec\ d\ a\ b)\ :\ \overline{(A : U_1)(B : (A)U_1)(C : (W\ A\ B)U_1)}((a : A)(b : (B\ a)W\ A\ B)((v : B\ a)C(b\ v))C(sup\ a\ b))(a : A)(b : (B\ a)W\ A\ B)C(sup\ a\ b).$

Remark 2.1 *P. Martin-Löf has suggested to add a general fixed point operator to type theory on the level of types and a general fixed point operator on level of* *elements. Let $G : (U_1)U_1$ be a strictly positive operator, we have the following rules: $Fix\ G : U_1$, $Fix\ G = G(Fix\ G) : U_1$, $fix : (C : (FixG)U_1)(g : (X : U_1)(f : (x : X)Cx)(y : GX)C\ y)\ (z : Fix\ G)Cz$, where X subtype of $Fix\ G$ and the equality rule: $fix = g.g(fix\ g)$ (of the above type). More details for semantic justification is given in [7].*

3 Programming in constructive systems

Having a logical framework to represent theories, we are concerned, in this section, by the way to use it for program development and we present 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 evaluation and the compilation of programs.

3.1 Specification step

We define here the representation of specifications and their transformation to consider then the construction of the program satisfying a given specification. For a better understanding and readibility, we present this in a mathematical notation that can be translated directly in the logical framework.

3.1.1 First form of specification

We can define, as informal specification of a problem, the informal but precise description of this problem. For example, 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). A general form of that formula is : $\forall X : Type\ \exists Y : Type\ F(X,Y)$ such that $X : Type$ and $Y : Type$ represent the input and output parameters with their types.
We can then write the formal specification in the theory with the proper types or formulae (the fonctions *length* (length of a list) and *elem* (giving the ith element of a list) are predefined).

Proposition 3.1 $\forall X : L(nat)\ \forall Y : nat\ \exists Z : L(nat)(length(X) = length(Z) \wedge$
$\forall n : nat(1 \leq n \leq length(X)) \Rightarrow Y \times elem(I,X) = elem(I,Z))$.
(it is an instance of a general formula $\forall X : L(nat)\forall Y : nat\exists Z : L(nat)G(X,Y,Z)$).

We can, in a first attempt, prove this formula by developing the proof by structural induction on the list X.

Proof 3.1 *1.* initial step : *si $X \equiv nil$ alors $Z \equiv nil$ is a solution.*
2. induction step : *Assume that, for a list X and a natural Y, a list Z verifies the two conditions, we must then construct a list L from the list Z.*
a) for the natural a, the length of $cons(a,X)$ is equal to the length of L.

b) the nth element of L is equal to the nth element of $cons(a, X)$ multiplied by Y with the following condition : $1 \leq n \leq length(cons(a, X))$.
Then if we consider $L \equiv cons(a \times Y, Z)$ we immediately verify the conditions a) and b).

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.

3.1.2 Second form of specification

When we consider a specification of the form $\forall X : Type\ \exists Y : Type\ F(X, Y)$ we must determine a Y and then verify that it satisfies the relation $F(X, Y)$. A transformation leads to replace the relation $F(X, Y)$ by a relation $F(X, f(X))$ where $f(X)$ represents the searched Y depending on X. We thus obtain a formula of the form $\forall X : Type\ f(X) : Type$ with f relation represented by equations in the logical theory. This transformation is made considering the type of the manipulated objects and their representation in the system.
X is of type $L(nat)$, type defined as below in the theory (with $A \equiv nat$):
1) $L(A) : U$ ($L(A)$ is a type), 2) $nil : L(A)$, 3) $cons(x, y) : L(A)$ if $x : A$ and $y : L(A)$, 4) $lrec(a, b, c) : C(a)$ if $a : L(A)$, $b : C(nil)$ and $e(x, y, z) : C(cons(x, y))$ (with $x : A$, $y : L(A)$, $z : C(y)$).
2) and 3) introducing the canonical elements of $L(A)$ and 4) defining an operator of induction on the list structure satisfying the following evaluations rules: $lrec(nil, b, c) = b$ and $lrec(cons(a, x), b, c) = c(a, b, lrec(x, b, c))$.
Consequently, as a specification, we have to consider the formula $\forall X : L(nat)$ $\forall Y : nat\ multi(X, Y) : L(nat)$ with thefollowing equations: $multi(nil, y) = nil$ and $multi(cons(a, x), y) = cons(mul(a, n), multi(x, y))$.

Remark 3.1 *Summarily the specifications that we can consider have the form, either of logical proposition (or type) or of a system of axioms (or equations) system with a logical proposition that translates the functional relation between input and output types. The latter is the classical one in AF2 [20]. A transformation allows to explicit the step between the first type of specification and the second, the latter being used to develop the example of automatization.*
Moreover, the equations precise what the searched program does (taking our theory into account) but not how it does it.

3.2 Mechanization of proof construction

From the specification (of one of the previously exposed forms) we can construct a proof of the formula-specification (considering the equations in the second case). In this way, we adopt an oriented-by-types strategy. The goal to achieve is the proposition (or specification) we have to prove considering possibly some hypotheses.

3.2.1 Proof construction

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 discovering step by step the structure of x.

Unification gives a flexible treatment of unknown expressions in goals. An inference rule like $\dfrac{C_1(p) ... \ C_n(p)}{C(p)}$ specifies a way to reduce a goal $C(p)$ into subgoals $C_i(p)$. Let us suppose the goal $P(p')$ and an expression q that unifies the p expression of the conclusion of the rule : $C(q)$ is identical to $P(q)$ and instantiation reduces $P(p')$ to $C_1(q) ... \ C_n(q)$ with the unifier q that may also contain unknown terms.

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. To be convinced, it is sufficient to compare developments by top-down and bottom-up strategies.

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.

3.2.2 Example of tactics.

We are searching a program X which satisfies a goal (or specification) $G(l)$ depending on a parameter l of type list (: $L(A)$) in an environment H i.e. $G(l)$, $H \vee (l : L(A))$ the tactic associated to this case (goal depending on a list of elements of type a) reduces this goal into two subgoals and consequently the construction of the program X into the construction of two subprograms:

- to find a program d satisfying the subgoal $G(nil)$ with $H \vee (l : L(A))$
- to find a program $e(x, y, z)$ satisfying the subgoal $G(cons(x, y))$ with $H \vee (l : L(A), x : A, y : L(A), z : G(y))$.

These subproblems being solved, we know that the final program have the following form $X \equiv lrec(l, d, e(x, y, z))$, $lrec$ being an operator which formalizes an induction on the type $L(A)$ of the system.

3.2.3 Application to the example

Considering the first form of specification, the application of the tactic consists in the proof 3.1. by induction on the structure of list. The important fact is to have sufficient structure representations to automatize the proof construction.

As an example, we consider the second form of specification which is :
(1) $(\forall x : L(nat))(\forall n : nat)(multi(x,n) : L(nat))$.
(2) $multi(nil, n) = nil$.
(3) $multi(cons(a,x),n) = cons(mul(a,n), multi(x,n))$.
Considering the definition of type $L(nat)$ in the system (with $mul(x,n)$ multiplication of naturals x and n. The program defined by the specification will be obtained with the construction of the proof of (1) using the axioms (2) et (3) (or rewriting rules) during the application of the tactics. The construction process is automatized in the global type-driven strategy.
Let us introduce a constant $multi$ such that $multi : [x : L(nat)][n : nat]L(nat)$ and the proposition corresponding to (1) i.e. $(\forall x : L(nat))(\forall n : nat)L(nat)[(multi(x,n)]$.
We have to note that $A[a]$ with A type is a proposition corresponding to the expression $a : A$ and that a is a proof of $A[a]$.
With the tactics (associated to $\forall$) we obtain a solution z of the form $z = \lambda x \lambda n.z_1$ with z_1 proof of $L(nat)[(multi(x,n)](\equiv G(x) with x : L(nat))$. By application of the tactic on the types parametrized with a list (given in the construction part of the text) z_1 will have the following form $z_1 \equiv lrec(x, x_1, \lambda abc.x_2)$ with $x_1 : G(nil)$ and $x_2 : G(cons(a,b))(with\ a : nat, b : L(nat), c : G(b))$.
We must then construct x_1 and x_2 to obtain the solution and for that we consider
• $G(nil) \equiv L(nat)[multi(nil,n)]$ then $x_1 \equiv multi(nil,n) \equiv nil$ with (2).
• $G(cons(a,b)) \equiv L(nat)[multi(cons(a,b),n)]$
$\equiv L(nat)[cons(mul(a,n), multi(b,n))]$ with (3).
Moreover we have in the context: $G(b) \equiv L(nat)[multi(b,n)]$ and then by unification $c \equiv multi(b,n)$. Consequently $x_2 \equiv cons(mul(a,n),c)$ and the solution has the form: $z = \lambda x \lambda n.lrec(x, nil, \lambda abc.cons(mul(a,n),c))$ with $mul(a,n)$ function product of two naturals (which can also be obtained from specifications in the same way). We give here an untyped form but the term obtained by the system is the typed one.

Remark 3.2 *This proof follows the same scheme as in AF2 from this specific specification (formula and equations) [15] but here the terms obtained are not iterative. This example illustrates the importance of adapted inference rules (leading to tactics) in such theories.*

3.3 Examples of proof derivations

A good example of synthesis is the derivation of the well known sorting algorithms from a common specification. We have some results of such exercise using transformation techniques [5]. Here we want to take up this example in a *programming with proofs* context with a view to defining new formulations or extensions to derive efficient programs. With the two following examples, we can see how algorithmic features can be reflected in proof strategy to derive proof of formal specification.
At first we present 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 the presented framework with special attention to the treatment and the representation of the data structures.

We consider the formal specification, which means that for any list of naturals x there exists a natural list y such that 1) the length of y is equal to that of x, 2) every element occuring in x also occurs in y and 3) y is sorted.

Proposition 3.2 *(Formal specification)*
$\forall x : L(N) \exists y : L(N)(\forall n : N \ (occ(n,x) = occ(n,y)) \ \wedge \ sorted(y))$
where sorted is defined as $sorted(x) \equiv \forall i,j : N(1 \leq i \leq j < length(x) \Rightarrow elem(i,x) \leq elem(j,x))$.

3.3.1 First proof derivation

We give here an informal mathematical style description of the proof. the first step of this strategy is the following lemma

Lemma 3.1 $\forall x : L(N) \forall j : N(j \leq length(x) \Rightarrow \exists y : L(N)(\forall k : N occ(k,x) = occ(k,y) \wedge \forall i : N(1 \leq j \leq i \leq length(x) \Rightarrow elem(j,y) \leq elem(i,y))))$.

Proof 3.2 *by considering the cases* $x = nil$ *and* $x \neq nil$.

To continue we introduce a new definition of a partially sorted list.

Definition 3.1 *Let* $x \equiv x_1.x_2...x_n.nil$ *be a list of length* n *and* i *a natural,* x *is partially sorted by* i *if and only if for* $i \leq n$ *the list* $x_1.x_2..x_i.nil$ *is sorted.*
$partsorted(x,j) \equiv j \leq length(x) \Rightarrow k,l : N(1 \leq k \leq l \leq j \Rightarrow elem(k,x) \leq elem(l,x))$.

Considering this definition we have the following lemma

Lemma 3.2 *Let* x *be an arbitrary list, if* x *is partially sorted by* $length(x)$ *then* x *is sorted i.e.* $\forall x : L(N) \ (partsorted(x, length(x)) \ \Rightarrow \ sorted(x))$.

Proposition 3.3 $\forall x : L(N) \forall j : N(j \leq length(x) \Rightarrow \exists y : L(N)(partsorted(j,y) \wedge \forall i : N \forall k : N(1 \leq i \leq j \wedge j \leq k \leq length(x) \Rightarrow elem(i,y) \leq elem(k,y))))$.

Proof 3.3 *by mathematical induction on* j.

The proof of the initial specification follows from the lemma and the proposition and the program corresponding to this proof is the *bubble sort* algorithm.

3.3.2 Second proof derivation

This strategy is the same as that in [25] and leads to the *quicksort* algorithm. The central part is the divide lemma.

Lemma 3.3 *(divide lemma) Let* x *a list of naturals and* n *an arbitrary natural, there exists a permutation* σ *of elements of* x *such that* $\sigma(x) = append(les(x,n), gt(x,n))$ *with every element of* $les(x,n)$ *less than* n *and every element of* $gt(x,n)$ *equal or greater than* n*, i.e.,*
$\forall x : L(N) \forall n : N \exists y : L(N)(perm(x,y) \wedge y = append(les(x,n), gt(x,n)))$.

Proof 3.4 *by induction on the recursive structure of list (transfinite induction)*

To continue we consider a variant of the specification through the proposition above

Proposition 3.4 *Let x and x_1 be natural number lists, if length(x_1) $\leq$ length(x) then there exists y_1 permutation of x_1 that is sorted.*
$\forall x : L(N)\forall x_1 : L(N)(length(x_1) \leq length(x) \Rightarrow \exists y_1 : L(N)(perm(y_1, x) \wedge sorted(y_1)))$.

Proof 3.5 *by induction on length(x) using the divide lemma.*

We can see, from these examples, 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 seen as a description of an algorithm with its mathematical meaning. Moreover these examples show that an adapted proof strategy corresponds to an efficient algorithm.
The strategies are defined by the introduction of adequate lemmas and definitions. The aim now is to have the possibilities of developping such strategies through rules corresponding to various treatments of datas. To continue, we want to illustrate such development in the given framework to understand the difficulty of representation of data and strategies and their implication with a view to mechanizing proof construction.

4 Representing inductively defined types

A first step is to see how to represent classical datas in type theory. Recursive data structures, such as lists, natural numbers or binary trees, are important in programming and the corresponding notion in Martin-Löf's type theory is that of an inductively defined type.
In a first version of the type theory [17], there is no general scheme to have an inductive definition. This is unlike another theories like Calculus of Construction [4] where inductively generated types can be defined using second order quantification or in other systems issued of Girard's system F [11]. The commun recursive data structure introduced as a primitive is that of natural numbers. The other recursive data types are represented as powerfull well-ordering.
Regarding program development, constructive type theory should be viewed as an open framework where new type operators can be added. It is the case, for instance, for the list former added as primitive type forming operation.
There are general methods meant both for representing inductively defined types in terms of wellorderings and for adding new rules for such types. The problem to add new rules for inductively defined types is solved by a general scheme meant to introduce new type operators by giving formation and introduction rules and deriving elimination and equality rules [1]. Another approach based on a fixed point operator is also possible. We discuss the problem of representation as well-orderings and show that all inductively defined type can be represented as well orderings [23]. In this case, we have to assume extensional equality relation on types as given in [18].

4.1 Wellorderings

The most classical way to represent inductively defined types in type theory is the wellordering constructor. We begin by giving a definition of this constructor illustrated by an example about binary trees.

4.1.1 Wellorderings

With this constructor we can construct many differents sets of trees and to obtain a particular set we have to indicate ways to form trees and each of their parts.
Given B a set constructor and C index family over B we can form a wellorder $W(B,C)$ with two arguments : 1. the constructor type B 2. the index family C with the notation $W(B,C)$ *or* $(Wx : B)C(x)$. The type is defined by the following rules:
W-formation

$$\frac{B\ Type \quad \overset{(x:B)}{C(x)\ Type}}{(Wx:B)C(x)\ Type}$$

W-introduction

$$\frac{b:B \quad \overset{(z:C(b))}{c(z):(Wx:B)C(x)}}{sup(b,c):(Wx:B)C(x)}$$

W-elimination

$$\frac{c:(Wx:B)C(x) \quad [x:B,\ \overset{(v:C(y))}{y(v):W(B,C)},\ \overset{(v:C(y))}{z(v):D(y(v))}]\ d(x,y,z):D(sup(x,y))}{wrec(c,d):D(c)}$$

with $wrec(sup(b,c),d) = d(b,c,(x)wrec(c(x),d)$.
The representation of well orderings in the logical framework is given by example 2.3. ($Type$ corresponding to U_1). To understand it, we can see it as a solution of an inductive definition of a particular form. Moreover all sets defined by a fixed point set operator could be represented by a wellorder in an extensional version of the type theory.

Example 4.1 *Representation of binary trees*
The set of constructors is {leaf, node} with no selector for leaf and set of selectors for node as a two element type ({left, right}). With this, the above type is W({leaf,node}, x.case(x,leaf:{},node:{left,right})) and sup(leaf, x.case$_{\{\}}$*(x)) represents leaf and sup(node, x.case(x,left:*t'_1*,right:*t'_2*) represents node(*t_1*,*t_2*) if* t'_i *represents* t_i *(i = 1,2).*
Moreover the representation of binary trees labelled by naturals has the following form: the set of constructors is {leaf }+N, the selector B(inl(leaf)) is the empty type and B(inr(n)) is {left,right }. With this, the corresponding type is W({leaf

$\}+N$, x.when(x,y{},y.{left,right })) and leaf is represented by sup(inl(leaf),case$_{\{\}}$ and node(n,t1, t2) by sup(inr(n),x.case(x,t'_1,t'_2) where t'_i represents t_i.
For instance, if we want to define the function adding all numbers in a binary tree we have to prove the proposition $\forall A$: Bintree(add(A) : N) with the axioms add(leaf) = 0 and add(node(n,t_1,t_2)) = n + add(t_1) + add(t_2). By proof, we obtain the following term : $add = w.wrec(w, x.y.z.when(x, u.0, u.u + z(left) + z(right)))$.

4.1.2 Another representation

The well-orderings in type theory are introduced by inductive definition. A version of $Wx : A.B(x)$ can be obtained by taking the least fixed point of the type operator $X.\Sigma x : A.(Bx \to X)$ [7]. Using the framework notation, for instance, natural numbers can be represented by $Wx : N_2.B_N x$ where $B_N 0_2 = N_o$, $B_N 1_2 = N_1$. These representations have to be isomorphic to the primitive type.
For example, $Wx : N.B_N x \simeq N$ where $A.B.A \simeq B$ means $A.B.(\exists f : A \to B)(\exists g : B \to A)\ ((\forall x : A)(Ap\ g(Ap\ f\ x)) =_A x)\ \wedge\ (\forall y : B)(Ap\ f(Ap\ g\ y)) =_B y))$. The correspondance is between 0 and $sup\ 0_2(\lambda R_o)$ and between $s\ a$ and $sup\ 1_2(\lambda x.a')$ where a corresponds to a'. The fact that this correspondance is an isomorphism depends on the extensional equality of functions [7].

Theorem 4.1 *For any strictly positive type operator G, we can find an $A : U_1$ and a family $B : (A)U_1$ such that if $X : U_1$ then $GX \simeq \Sigma x : A.(Bx \to X)$*

Remark 4.1 *These representations indicate that, for a given type, we may have different ways to reach it with some difficulties and constraints. It is not clear what representation is the best for our approach. Moreover, it is possible to consider generalization of well-order type constructor for a better specification [23].*

4.2 Deriving recursion operators

For each inductively defined set there is a principle of primitive recursion. For example, with naturals, we have ordinary primitive recursion and the *natrec* operator of type theory, with lists we have primitive list recursion and *listrec* operator and for well-orderings we have transfinite recursion and *wrec* operator.
We assume that G (positive type operator) is in the subclass. We have to determine its constructors with their types and the introduction rules for *Fix G* and the elimination rule for *Fix G*(and consequently the recursion operator and its type).
The recursion operator and elimination rule are obtained from the general elimination rule. We find a selector (or pattern matching function) $selec^G$ with n+2 arguments such that substitution of $selec^G d_1 \dots d_n$ for g then the premise of general elimination rule will be satisfied. The recursion operator for *Fix G* is $rec^{FixG} d_1 \dots d_n z = fix(sel^G d_1 \dots d_n) z$.

Example 4.2 *For instance, if natural numbers are defined as $N = Fix.X.N_1 + X$ we have two constructors: $0 = i0_1 : N$ and $s = \underline{j : (N)N}$ and the recursion operator $R = d_1 d_2.fix(f.D(R_1 d_1)(x_1.d_2 x_1(f x_1))) : \overline{(C : (N)U_1})(d_1 : C0)(d_2 : (x_1 :$*

$N)(Cx_1)(C(sx_1))(z : N)Cz$ *considering the selector for* $X.N_1 + X$ *of the form* $natcases = d_1.d_2.f.D(R_1 d_1)(x_1.d_2 x_1(f x_1))$*. This can be generalized to finitary induction.*

4.2.1 Parametrized types

Moreover we can represent parametrized inductively defined types like lists (generated by operator $X.N_1 + A \times X$) and binary trees (generated by operator $X.A + X \times X$) where A is a parameter. Then we get the following terms for $L(A)$ defined by $A.X.N_1 + A \times X$:

$nil = i0_1 : \overline{(A : U_1)(X : U_1)} N_1 + A \times X$

$cons = a.x.j < a, x >: \overline{(A : U_1)(X : U_1)}(A)(X).N_1 + A \times X$

$\underline{listcases = d_1.d_2.f.D}(R_1 d_1)(E(a_1.x_1.d_2 a_1 x_1(f x_1))) : \overline{(A : U_1)(X : U_1)(C_1 : (X)U_1)}$ $(C_2(N_1 + A \times X)U_1)(d_1 : C_2(nil))(d_2 : (a_1 : A)(x_1 : X)(C_1 x_1)C_2(consa_1\ x_1))(f : (x : X)C_1 x)(y : N_1 + A \times X)C_2 y$

We have similar definitions for binary trees defined by $A.X.A + X \times X$.

Remark 4.2 *The operators D,* R_1 *and E are the elimination operators of some classical types in type theory. Moreover, we have obtained a new representation of structures like labelled binary trees under a form that can be used to automatize the proof construction. But the difficulty is to generate such expressions for our structures and after to choose a better one. This illustrates the ability to have different representations for data types in the theory. It is the same case in second order intuitionistic type theory where classical representation for a structure is the iterative and the recursive ones [15].*

Different approaches to the introduction of inductive sets in type theory have been proposed using the well order set constructor, introducing a fixed point set constructor [7] or introducing a method to add new rules [1]. The first is not general enough to express all inductive sets. The second is quite complicated and the third is not sufficient for the purpose because it is not formalized inside the theory formalism and we want to have set constructor for inductive sets with reasoning about it using theory rules. More than the choice of representation of datas, it seems important to formalize programming concepts like general recursion.

5 Recursion

In Martin-Löf's type theory, the only iterating constructs are primitive recursion over defined inductive types. But we can allow a general recursion operator in type theory extended with propositions and such that addition does not distroy the property that all well-typed programs terminate. We present here the way from primitive recursion to general recursion [19]. For a better understanding we take the naturals as example and choose the general formulation with inference rules and classical functional notation. But the formulation in the theory presented in section 2 is not difficult to obtain.

5.1 Primitive Recursion

We consider the rule for natural induction in type theory
Natural Induction

$$\frac{p : N \;\; d : C(0) \;\; e(x,y) : C(suc(x))[x : N, y : C(x)]}{natrec(p,d,e) : C(p)}$$

The rule can be read in the following way : the conclusion $natrec(p,d,e) : C(p)$ can be drawn if $p : N$, $d : C(0)$ and if $e(x,y) : C(suc(x))$ under the assumptions that $x : N$ and $y : C(x)$.
We can rewrite the rule as

$$\frac{p : N \;\; d : C(0) \;\; e(x) : C(x) \rightarrow C(suc(x))[x : N]}{natrec(p,d,e) : C(p)}$$

So the problem $C(p)$ is solved by $natrec(p,d,e)$ if d solves $C(0)$ and $e(x)$ function taking a solution of the problem $C(x)$ to a solution of $C(suc(x))$.
The computation rule for primitive recursion operator natrec : if the value of p is 0 then value of $natrec(p,d,e)$ is value of d. if the value of p is $suc(a)$ then the value of $natrec(p,d,e)$ is the value of $e(a, natrec(a,d,e))$.

5.2 Course-of value recursion

In course-of-value recursion, we want a function $e(x)$ which takes a solution of all the problems $C(0)$, $C(suc(0))$, ..., $C(x)$ to a solution of the $C(suc(x))$.
The requirement of the function $e(x)$ can be expressed as $e(x,y) : C(suc(x))[x : N \; y(z) : C(z)[z \leq x \; true]]$ and we obtain the rule

Course-of-values induction 1

$$\frac{p : N \;\; d : C(0) \;\; e(x,y) : C(suc(x))[x : N, y(z) : C(z)[z \leq x \; true]]}{covrec(p,d,e) : C(p)}$$

where $covrec$ is a new primitive constant computed in the following way :
If the value of p is 0 then the value of $covrec(p,d,e)$ is d, if this value is $suc(a)$, then the value of $covrec(p,d,e)$ is $e(a, \lambda z.covrec(z,d,e)$. $e(a,y) : C(suc(a))$ if y is a function such that $y(z) : C(z)$ for $z \leq a$ and $\lambda z.covrec(z,d,e)$ is such a function since $covrec(a,d,e) : C(a)$.

Remark 5.1 *We can simplify this rule of course-of-values induction if we instead have a function $e(x)$ which takes a solution of all problems strictly smaller than x to a solution of $C(x)$ and consequently drop the second premise.*

Course-of-values-induction 2

$$\frac{p : N \;\; e(x,y) : C(x)[x : N, y(z) : C(z)[z < x \; true]]}{rec'(p,e) : C(p)}$$

the value of $rec'(p,e)$ is computed by computing the value of $e(p, \lambda z rec'(z,e))$. We get the rule for complete induction if we take away the constructions, i.e.,

Complete induction

$$\frac{p : N \;\; C(x) \; true[x : N, \;\; C(z) \; true \; [z < x \; true]]}{C(p) \; true}$$

5.3 General recursion and well-founded induction

Nothing in the course-of-value induction rule is particular to the type of natural numbers and the reason why the rule works is that this type is well-founded. So, we can generalize the rule to arbitrary set A well founded by a relation $<_A$ [19] and to express that A is well founded by $<_A$ we use the proposition $wellfounded(A, <_A)$. For further details, refer to the part about the set of accessible elements.

Recursion rule

$$\frac{wellfounded(A, <_A) \;\; p : A \;\; e(x,y) : C(x)[x : A, y(z) : C(z)[z <_A x]]}{rec(e,p) : C(p)}$$

Where the value of $rec(e,p)$ is the value of $e(p, rec(e))$.

Remark 5.2 *By defining $Y(e,p)$ by $Y(e,p) =_{def} rec(\lambda xy.e(y,x), p)$, we get a binary fix-point operator. Consequently we have $Y(e,p) \equiv rec(\lambda xy.e(y,x), p)$. $Y(e,p)$ $= \lambda xy.e(y,x).(p, rec(\lambda xy.e(y,x))) \equiv e(x,y)[x := p, y := rec(\lambda xy.e(y,x))]$ (simultaneous substitution) $\equiv \;\; e(rec(\lambda xy.e(y,x)), p) \equiv e(Y(e), p)$.*

Remark 5.3 *About reasoning on recursive functions in type theory, Smith [25] has introduced quicksort within type theory by deriving a new principle of primitive recursion. It is defined by w.f. recursion over a w.f. relation $\prec$. Each recursive call includes an explicit termination argument involving $\prec$ and quicksort terminates because the length of the list is smaller in each recursive call.*

5.3.1 Example: the quicksort algorithm

At first we consider the recursive equations for the quicksort algorithm on L(N):
qsort(nil) = nil and *qsort(cons(a,s)) = append(qsort(le(s,a)),cons(a,qsort(gt(s,a))))* with $le(s,a)$ (resp. $gt(s,a)$) list obtained from s by taking away all elements greater than (resp. smaller than or equal to) a.
Moreover, we consider the *listcases* constant defined by: $listcases(p,d,e)$ has the value of d if the value of p is nil and has the value of $e(a,s)$ if the value of p is $cons(a,s)$. Let us recall that if $f(z) = e(z,f)$ recursion equation then it can be solved by defining f as $rec(e)$. This way, we have $qsort(p)$ = *listcases(p,nil,a.s.append(qsort(le(s,a)),cons(a,qsort(gt(s,a))))*
and consequently we have the following by definition equality :
$qsort =_{def} rec\ x.y.listcases(x, nil, a.s.append(y(le(s,a)), cons(a, y(gt(s,a)))))$.
To prove the termination of this program we consider that $qsort : L(N) \rightarrow L(N)$. It means proving that *listcases(x,nil,a.s.append(y(le(s,a)),cons(a,y(gt(s,a))))) : L(N)* with $x : L(N)$ and $y(z) : L(N)$ [$z : L(N)$, $z \prec x$] for a well-founded relation $\prec$.

Proof 5.1 *by induction over x.*
1. The base case is trivial.
2. x = cons(a,s) and we have to prove that:
append(y(le(s,a)),cons(a,y(gt(s,a)))) : N that holds if y(le(s,a)) : N and gt(s,a) : N (because append : L(N) × L(N) → L(N).
It is sufficient to find a wellordering ≺ on L(N) such that le(s,a) ≺ cons(a,s) and gt(a,s) ≺ cons(a,s) by induction assumption. We can define ≺ as $x \prec y =_{def}$ *length(x)* $\prec_N$ *length(y) that is well founded. But to prove these facts it is necessary to use formal system with more rules about type equality.*

Remark 5.4 *Even it is apparent that a relation is well founded, proving this may be difficult and well founded relations are most easily constructed from simpler ones, using rules that preserve the well-founded property. Some rules can be described for well-founded relations [21].*

5.3.2 The type of accessible elements

To include general recursion inside the theory, B. Nordström has proposed a new type in the theory [19]. Let $<$ be a binary relation on a set A (i.e, $<(x,y)$ proposition for $x, y : A$).
The set $Acc(A,<)$ of accessible elements of $<$ in A is the set of elements $a < A$ such that there is no infinite descending sequence $a > a_0 > \ldots > a_n$. The set $Acc(A,<)$ is called the well-founded part of $<$ and the set A is well ordered by $<$ if $A = Acc(A,<)$ We give here a description of the set $Acc(A,<)$. The element $a : A$ is accessible if all elements smaller than a are accessible.
We formalize this by introduction rule for $Acc(A,<)$. We also give the other formation, elimination and equality rules.

Acc-intro

$$\frac{a : A \quad y : Acc(A,<)\ [y : A, y < A]}{a : Acc(A,<)}$$

Acc-formation

$$\frac{A\ set \quad x < y\ prop\ [x : A, y : A]}{Acc(A,<) set}$$

Acc-elimination

$$\frac{p : Acc(A,<)\ e(x,y) : C(x)\ [x : Acc(A,<), y(z) : C(z)\ [z : A, z < x]]}{rec(e,p) : C(p)}$$

where $rec(e,p) = e(p, rec(e)) : C(p)$.

The elimination rule corresponds to the recursion rule above if A is well-founded by $<$ due to the fact that A is well-founded if it is equal to the set $Acc(A,<)$.

Remark 5.5 *By extension with the rec-operator for the program forming operators, we can replace all operators for primitive recursion with operators for pattern matching. As an example we can consider the primitive recursion operator for lists. For that we consider the definition of listcases and define listrec as: listrec(p,d,e)* $=_{def}$ *rec(x.y.listcases(x,d,z.t.e(z,y(t)))). Moreover, all iterating constructs in type theory can be reduced to pattern matching and the general recursion operator rec.*

5.4 Program development examples

In this section we derive programs using the general recursion and more precisely the *Acc* type. These examples are extracted from [16] and are significiant concerning the program construction..

5.4.1 Example 1

We discuss here about sorting algorithms of lists of naturals.
The general specification is $\forall x : L(N)\{y : L(N)/sorted(y) \wedge perm(x,y)\}$ which expresses that y is an ordered permutation of x. To construct a sorting algorithm we use the Acc-elimination rule with $<_l$ as well-founded relation on $L(N)$: $a <_l b \Leftrightarrow le(a) <_N le(b) \quad a, b : L(N)$, le denoting the length function (we have to prove that this relation is well-founded).
Moreover we need the following terms (or functions):
$s : (\forall x : \{l : L(N)/le(l) \geq 2\}\{y : L(N) \times L(N)/$
$\quad split(y, y_1.y_2.perm(append(y_1,y_2), x) \ \wedge \ y_1 <_l x \wedge \ y_2 <_l x\}$.

$m : (\forall a : \{x : L(N)/sorted(x)\})(\forall b : \{x : L(N)/sorted(x)\})$
$\{x : L(N)/sorted(x) \wedge perm(x, append(a, b))\}$.
$leq : (\forall x : N)(\forall y : N)((x \leq y) \vee (x \not\leq y)$.
We need also some properties about *perm* to derive the programs.
1. $perm(x, y) \wedge perm(y, z) \Rightarrow perm(x, z)$ for $x, y, z : L(N)$
2. $perm(x, y) \wedge perm(x', y') \Rightarrow perm(append(x, x'), append(y, y'))$ $x, x', y, y' : L(N)$
Consequently, we can obtained by proof a correct sorting algorithm :
$\lambda x.rec(x, l.y.when(leq(le(l), 1), l, split(s(l), a.b.m(y(a), y(b)))))$.
We observe that the efficiency of the algorithm depends on the form of s and two different implementations of s lead to two different algorithms.
For instance, if $s \equiv x.(hd\ x, tl\ x)$ we obtain *insertion sort* and if
$s \equiv$ *x.rec(x,l.sl.listrec(l,(nil,nil),a.x'.z.listrec(x',([a],nil),b.x".z)split(sl (x"),x.y.(a:y,b:x)))))* we obtain the *merge sort*.

Remark 5.6 *To obtain other sorting algorithms we have to change the auxiliairy functions that fix the style of the derivation. It corresponds to defining new filter functions for the synthesis [5]. What is not clear is the way to automatize derivations using these new filters. Moreover, to derive for instance the quicksort, a method can be to derive it from equations using the listcases constant.*
What can we say about efficiency of quicksort in type theory ? The questions dealing with efficiency are embarassing because programs are executed under a lambda interpretor with lazy evaluation. Ideally sort recursion equation should be executed directly as rewrite rules (representing here a terminating computation).

5.4.2 Example 2

An other example is to determine the greatest common divisor of two non-zero naturals.
We observe the importance of properties (of the programs) and the difficulty to found (and to prove) well-founded relations [16].
The specification of this problem can be written in the theory as : $(\forall p : N^+ \times N^+)$ $\{z : N/z\ gcd\ p\}$ where $z\ gcd(x, y) \equiv z\ div\ x \wedge z\ div\ y \wedge (\forall n : N)(n\ div\ x \wedge n\ div\ y \Rightarrow n \leq_N z)$ and $z\ div\ x \equiv (\exists c : N)(x = c * z)$.
We have the well-known properties (which have to be proved)
1) $x\ gcd(x, x)$
2) $z\ gcd(x, y - x) \;\Rightarrow\; z\ gcd(x, y)$
3) $z\ gcd(x - y, x) \;\Rightarrow\; z\ gcd(x, y)$
Considering these we use the well-founded induction (with Acc rules) with the following well-founded relation: $< \equiv (x, y).(x.1 + x.2) <_N (y.1 + y.2)$ and the obtained program is : $p = rec(p, q.g.split(q, (a.b\ when(a = b, a,\ when(a < b, g((a, b - a)), g((a - b, b)))))))$.

Remark 5.7 *Type theory seems adequate to work with well-founded recursion and induction and it is necessary to look forward to using it for computer proofs. But we can mention some problems. The ability to represent propositions as types allows a compact logical system but some uninteresting propositional constructions can*

complicate the computational ones. For instance, treating the proposition $x \prec x$ as a type can be complicated, and even the final term can be computed without computations of elements of $x \prec x$ [21].
The alternative of [6] is to reason about programs in an untyped logical theory of propositions where propositions are primitive and with types as predicate. Here the fixed-point Y exists and can define recursive functions. Well founded induction is required but Y takes the place of well-founded recursion.

Remark 5.8 *An inductive type constructor has been added to MLTT [19] and can be used to derive general recursion. We have already the W type constructor to defined inductive types but it is not enough and we can say that Acc-type complements W-type in nearly the same way as subtypes complement the sum types [24] but with a necessary relaxation of the unicity of types property. The different steps towards these extensions are very interesting to study because the problems of formulation of general concepts usable in programming also appear in other theories [14], [20] under different formulations.*
An important point is to have a theory that can integrate many general aspects of program derivation. Such a theory has to have flexible typing and the addition (or the representation) of new operators or rules has to preserve the basic properties of the logical framework (like correctness or termination). Under this condition we can have possibilities for a better understanding of what a good proof is from an algorithmic point of view.

6 Conclusion

We have presented some notions for program development in constructive type theory. To use proofs in logics for programming, it is necessary to study the relationships between proofs and programs and to define what the program development activity is in such logical frameworks. The way to derive interesting programs is to have adapted rules in the theory the addition of which preserves essential properties of the framework like correctness and termination and that can be expressed in the language. Including some rules like general recursion (which is helpful for development for efficient programs) is not very easy and representing these ones in the framework as inference rules (or equivalent) is a problem appearing in all theories devoted to this approach. But this effort is an obligation with a view to mechanizing proof and program construction through elaborations of tactics and strategies. We have given here a survey on results about type theory and program derivation. Comparisons with another frameworks around data representations and mechanization aspects could be interesting for a better understanding of the program development process in type theory based systems. An important point is to develop computer systems on these theories for a better understanding of the obliged extensions to derive programs. The use of such framework to program transformation seems also interesting and would be developed.

References

[1] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In *Workshop on General Logic, LFCS-88-52*, Edinburgh, February 1987.

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

[3] R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

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

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

[6] P. Dybjer. Program Verification in a Logical Theory of Constructions. In *Functional Programming Language and Computer Architecture, LNCS 201*, pages 334–349, 1985.

[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. Formalisation de la théorie intuitionniste des types : Contribution à l'étude de la démonstration en programmation. Thèse de troisième cycle, Université Paris VI.1985

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

[10] D. Galmiche. A programming with proofs system based on type theory. In *Workshop on Computer-aided Development of Proofs, Theories, Programs and Circuits*, Madrid, Spain, July 1989.

[11] J.Y. Girard. The system F of variables types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[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] S. Hayashi. PX : a system extracting programs from proofs. In *IFIP Conference on Formal Description of Programming Concepts*, pages 399–424, 1986.

[14] M.C. Henson and R. Turner. A constructive set theory for program development. In *FST-TCS Conference, LNCS 338*, pages 329–347, Pune, India, December 1988.

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

[16] G. Malcolm and E. Saaman. Well-founded recursion in type theory. Technical report, Subfaculteit Wiskunde en Informatica, Rijksuniversiteit Groningen, 1988.

[17] P. Martin-Löf. Constructive mathematics and computer programming. In *6th Congress for Logic, Methodology and Philosophy of Science*, pages 153–175. North-Holland, 1982.

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

[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.

[21] L.C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

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

[23] K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science, LNCS 389*, pages 128–140, Manchester, UK, September 1989.

[24] A. Salvesen and J.M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Third Annual Symposium on Logic in Computer Science*, pages 384–391, Edinburgh, Scotland, July 1988.

[25] J.M. Smith. The identification of propositions and types in Martin-Löf's type theory. In *Foundations of Computation Theory*, pages 445–456, 1983.

[26] M. Weber. Introduction and formal description of a program development calculus. Technical report, GMD Karlsruhe, June 1987.