Inductive Families

Peter Dybjer

November 26, 1997

Abstract

A general formulation of inductive and recursive definitions in Martin-Löf's type theory is presented. It extends Backhouse's 'Do-It-Yourself Type Theory' to include inductive definitions of families of sets and definitions of functions by recursion on the way elements of such sets are generated. The formulation is in natural deduction and is intended to be a natural generalization to type theory of Martin-Löf's theory of iterated inductive definitions in predicate logic.

Formal criteria are given for correct formation and introduction rules of a new set former capturing definition by strictly positive, iterated, generalized induction. Moreover, there is an inversion principle for deriving elimination and equality rules from the formation and introduction rules. Finally, there is an alternative schematic presentation of definition by recursion.

The resulting theory is a flexible and powerful language for programming and constructive mathematics. We hint at the wealth of possible applications by showing several basic examples: predicate logic, generalized induction, and a formalization of the untyped lambda calculus.

1 Introduction

The inference rules of Martin-Löf's type theory can be separated into three parts:

- general rules;
- rules for ordinary set formers;
- rules for universes.

Typically the second part includes the set formers Π, Σ, $+$, I, N_n, N, W and *List* (Martin-Löf [26]), but it is often remarked that this collection can be extended when there is a need for it. However, the desire for such extensions is so common that general principles need to be layed down which ensure their correctness. There are two possibilities:

- using a general purpose construction which is part of the theory;
- giving external criteria for correct extensions of the theory.

[0]Author's address: Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden. Email: peterd@cs.chalmers.se. A shorter preliminary version of this paper appeared under the title 'An inversion principle for Martin-Löf's type theory' in the Proceedings of the Workshop on Programming Logic in Båstad, May 1989, Programming Methodology Group Report 54, Chalmers University of Technology and the University of Göteborg, pages 177-190.

An example of a general purpose construction is *impredicative quantification*, which is part of system F, the calculus of constructions [6], and related systems.

This construction is not part of Martin-Löf's type theory, which is *predicative*. But in the extensional version [26] *wellorderings* W can be used instead. It can for example be proved [11] that for any strictly positive set operator Φ built up by constants, variables, $+$, $\times$, and $\rightarrow$, there is a set A and a family of sets B over A, such that

$$\Phi(X) \cong \Sigma_{x:A} X^{B(x)}.$$

Since $W_{x:A}B(x)$ satisfies the isomorphism

$$X \cong \Sigma_{x:A} X^{B(x)}$$

we are justified in using it as a representation for the set generated inductively by Φ. But this method does not work in the intensional version of type theory given by Martin-Löf in 1986 [25] (see Nordström, Petersson, and Smith [29]), since it makes use of 'extensional isomorphisms', such as $N_1 \cong X^{N_0}$ and $X \cong X^{N_1}$.

Another possibility is to add *fixed point operators* to type theory. But the formulations of Mendler [27] has the drawback that it needs a notion of subtype and therefore require fundamental changes of the theory. A new formulation which does not assume a notion of subtype has been proposed by Coquand and Paulin [7].

The second possibility is the topic of this paper: to specify a scheme which determines correct extensions of a theory. Thus we do not deal with a fixed theory but an *open* theory. But note that what we consider fixed and open is a matter of convention. We can make the theory closed by formalizing rules for sequents

$$T; \Gamma \vdash \mathcal{J},$$

where a judgement $\mathcal{J}$ is made in a current theory T as well as in a context of assumptions Γ, and where there are formal rules for a correct current theory T as well as for a correct context Γ. (Compare the presentation of the rules for the Edinburgh LF system in Harper, Honsell, and Plotkin [17], which contain rules for correct *signatures* as well as for contexts. Signatures play a similar rule to our current theories, but cannot contain definitional equalities. Moreover, correctness of a signature is only a form of type correctness which does not require that the constants are interpreted in terms of inductive definitions.)

The main point here is that *Martin-Löf's type theory is a theory of inductive definitions formulated in natural deduction.* Each set former (logical constant) is defined inductively by its introduction rules. The elimination rule expresses a principle of definition by recursion (proof by induction). Equality rules express how these definitions are eliminated (proofs are normalized).

First we specify what it means to be a correct definition of a set former by giving formal criteria for the formation and introduction rules. Then we show how such a definition determines the elimination and equality rules by a so called *inversion principle*. We also give an alternative formulation where recursive definitions are presented schematically.

The scheme is for *monomorphic, intensional* type theory and covers *strictly positive, iterated, generalized induction*. All the ordinary set formers Π, Σ, $+$, I, N_n, N, W and *List* follow this scheme. This does not cover all forms of intuitionistically meaningful induction, and thus not all ways of forming sets in Martin-Löf's type theory. An example is the definition of a universe à la Tarski U [26] which is defined simultaneously with a decoding function T (a family which maps an element in U to the set it codes). T appears negatively in the U-introduction rule

$$\frac{a : U \qquad \begin{array}{c}(x : T(a)) \\ b(x) : U\end{array}}{\pi(a,b) : U},$$

for example. (It is however possible to extend the scheme in a natural way to include universes and similar constructions, see Dybjer [14].)

The present scheme is very close to a scheme developed independently by Thierry Coquand and Christine Paulin [7, 30]. They have arrived at essentially the same combinatorial structure, but develop it in the context of an impredicative system, so there are some differences in the type structure. The resulting system 'the Calculus of Inductive Constructions' is the basis of Coq [10] - a system for interactive proof in an extension of the Calculus of Constructions [6]. I am very grateful to Thierry Coquand and Christine Paulin for many interesting discussions on the topic of inductive definitions in type theory.

The possibility of developing a general formulation of inductive definitions in type theory was already mentioned in an early paper on type theory by Martin-Löf [22, page 10]:

> 'The type N is just the prime example of a type introduced by an *ordinary inductive definition*. However, it seems preferable to treat this special case rather than to give a necessarily much more complicated general formulation which would include $(\Sigma \in A)B(x)$, $A + B$, N_n and N as special cases. See Martin-Löf 1971 [21] for a general formulation of inductive definitions in the language of ordinary first order predicate logic.'

The present scheme generalizes a scheme given by Backhouse [2, 3] under the name 'Do-It-Yourself Type Theory'. The existence of a scheme relies on Schroeder-Heister's notion of rule of higher level [33] which made it possible for Martin-Löf to formulate the rules for Π so as to conform to the pattern for the other set formers. (Note that Π was not mentioned in the quotation above.) Schroeder-Heister [34] also formulated a scheme for Martin-Löf's logical theory which is close to Backhouse's scheme for type theory.

The present scheme subsumes both Martin-Löf's scheme for the intuitionistic theory of iterated inductive definitions mentioned in the quotation above [21] (translated to the language of type theory under the propositions-as-sets principle) and Backhouse's scheme. It generalizes Backhouse's scheme [2] in several ways:

- The main point is that *families of sets (predicates and relations)* such as the identity relation I and the finite sets N_n (viewed as a family of sets over the natural numbers N) are covered by the scheme.

- Definitions may depend on earlier definitions (*iterated induction*). For example the definition of the set of ordinals of the second number class $\mathcal{O}$ depends on N having been defined earlier.

- Sets may be defined by *simultaneous induction.* (Examples of simultaneous induction were given in Backhouse, Chisholm, Malcolm, and Saaman [3].)

- General *parameters*, which can be elements as well as (families of) sets are allowed. Thus the generalized well-founded trees of Petersson and Synek [32] are covered by the scheme. Another nice application of having a general notion of parameter is a new definition of the equality relation. The usual definition of equality in type theory is as the least reflexive relation on a given set, see Martin-Löf [24]. An alternative definition due to Christine Paulin, which is also an instance of the scheme, is to define it as a unary predicate 'to be equal to a' given a set A and an element $a : A$. The elimination rule for this new equality predicate is the rule of substitution, provided proof objects are suppressed.

We do not discuss quotients of sets arising from stipulating *definitional equalities* between elements of sets $a = b : P$ as in Backhouse, Chisholm, Malcolm, and Saaman [3]. On the other hand it is easy to introduce quotients arising from *propositional equalities* as in section 5.3 where the untyped λ-calculus with β-conversion is presented.

There is an accompanying paper [13], which presents essentially the same scheme. The purpose of that paper is to prove consistency by constructing a set-theoretic model. The purpose of this paper is to show examples and hint at the wealth of possible applications. There are also some differences in the formulation of the scheme. The present scheme uses Martin-Löf's theory of logical types [1], whereas the theory of the other paper is formulated without this in the style of Martin-Löf's earlier presentations [23, 24, 26]. Another difference is that the present paper shows explicitly how to derive elimination and equality rules for new set formers. Moreover, the class of admissible *schematic* recursive definitions has been extended here.

The contents of the paper are the following. In section 2 we present the theory of logical types, which is the underlying framework for the inductive and recursive definitions. In section 3 we present the scheme for inductive definitions. We illustrate the scheme by giving three simple instances: natural numbers, lists, and lists of a certain length. In section 4 we present recursive definitions schematically. In section 5 we give some more examples. First we show how predicate logic is subsumed by the scheme. Then we discuss generalized induction: the well-orderings and the well-founded part of a relation. Finally, we show a formalization of the untyped λ-calculus which can be used for proving metamathematical properties about it. In section 6 we extend the scheme to simultaneous inductive definitions. In section 7 we finish by giving some references which use the present theory for more extensive examples.

2 The theory of logical types

We make use of *the theory of logical types (logical framework)* which was presented by Martin-Löf in 1986. (See Nordström, Petersson, and Smith [29] for a presentation of this theory and a general introduction to Martin-Löf's type theory.) This is a typed $\lambda\beta\eta$-calculus with dependent function types written $(a:\sigma)\tau[a]$[2] (or $(\sigma)\tau$ when τ does not depend on a); a special base type *set*; and a rule which states that each set A, that is, each object of the type *set*, is also a base type. Application is written $p(q)$ and abstraction is written $(a)p[a]$. Ordinary conventions about parentheses apply. Repeated application may be written $p(q_1,\ldots,q_n)$ instead of $p(q_1)\cdots(q_n)$ and repeated abstraction may be written $(a_1,\ldots,a_n)p[a_1,\ldots,a_n]$ instead of $(a_1)\cdots(a_n)p[a_1,\ldots,a_n]$.

The theory is formulated à la Martin-Löf's type theory with four forms of judgements: σ type, $p:\sigma$, $\sigma=\tau$, and $p=q:\sigma$. Judgements are made under lists of assumptions of the form $(a_1:\sigma_1,\ldots,a_n:\sigma_n[a_1,\ldots,a_{n-1}])$.

We formulate the rules in natural deduction. In addition to rules of assumption and substitution there are the following rules.

Rules of type formation:

$$set\ type,$$

$$\frac{A : set}{A\ type},$$

$$\frac{\alpha\ type \qquad \begin{matrix}(a:\alpha)\\ \tau[a]\ type\end{matrix}}{(a:\alpha)\tau[a]\ type}.$$

[1]It can therefore directly be used as a discipline for introducing new constants in the ALF system (developed by Augustsson, Coquand, Magnusson and Nordström), an implementation of Martin-Löf's theory of logical types

[2]The notation $\tau[a]$ explicitly indicates that the expression τ may depend on a

The rules of object formation:

$$\frac{\begin{array}{c}(a:\sigma)\\ p[a]:\tau[a]\end{array}}{(a)p[a]:(a:\sigma)\tau[a]},$$

$$\frac{p:(a:\sigma)\tau[a] \qquad q:\sigma}{p(q):\tau[q]}.$$

The equality rules are typed β- and η-conversion:

$$\frac{q:\sigma \qquad \begin{array}{c}(a:\sigma)\\ p[a]:\tau[a]\end{array}}{((a)p[a])(q)=p[q]:\tau[q]},$$

$$\frac{p:(a:\sigma)\tau[a]}{p=(a)(p(a)):(a:\sigma)\tau[a]}.$$

Moreover, equality is an equivalence relation and we may everywhere substitute equals for equals:

$$\frac{\sigma\ type}{\sigma=\sigma}, \qquad \frac{p:\sigma}{p=p:\sigma},$$

$$\frac{\sigma=\sigma'}{\sigma'=\sigma}, \qquad \frac{p=p':\sigma}{p'=p:\sigma},$$

$$\frac{\sigma=\sigma' \qquad \sigma'=\sigma''}{\sigma=\sigma''}, \qquad \frac{a=a':\sigma \qquad a'=a'':\sigma}{a=a'':\sigma},$$

$$\frac{p:\sigma \qquad \sigma=\sigma'}{p:\sigma'}, \qquad \frac{p=p':\sigma \qquad \sigma=\sigma'}{p=p':\sigma'},$$

$$\frac{A=A':set}{A=A'},$$

$$\frac{\sigma=\sigma' \qquad \begin{array}{c}(a:\sigma)\\ \tau[a]=\tau'[a]\end{array}}{(a:\sigma)\tau[a]=(a:\sigma')\tau'[a]},$$

$$\frac{\begin{array}{c}(a:\sigma)\\ p[a]=p'[a]:\tau[a]\end{array}}{(a)p[a]=(a)p'[a]:(a:\sigma)\tau[a]},$$

$$\frac{p=p':(a:\sigma)\tau[a] \qquad q=q':\sigma}{p(q)=p'(q'):\tau[q]}.$$

A *theory* in the framework sense is specified by giving a finite sequence of typings of fresh constants $c:\sigma$ and a finite sequence of equalities between terms $p=q:\sigma$.

When the framework is used for formalizing Martin-Löf's type theory, then the rules of the framework are the general rules. Formation, introduction, and elimination rules for the set formers are given by typing fresh constants. For example, the universe introduction rule above would be given by

$$\pi:(a:U)(b:(x:T(a))U)U$$

Equality rules are given by equalities between terms.

We shall need to distinguish 'set-like' types (*s-types*) from other types, since we wish that the rules for Π (with F (*funsplit*) as the constant in the elimination rule, see the preface of

Martin-Löf [26]) to be instances of the scheme. An s-type is either a set or a types of functions from an s-type to an s-types. (Another possible name for s-type is *purely functional type*.)

We use de Bruijn's telescope notation [9] and write $(a :: \sigma)$ as an abbreviation of the sequence $(a_1 : \sigma_1) \cdots (a_n : \sigma_n)$ and refer to σ as a 'sequence of types'.

3 A scheme for inductive definitions

In this section I shall present the scheme for introducing a new set former. Firstly, there is the general form of formation and introduction rules. Secondly, there is the inversion principle for deriving elimination and equality rules from the formation and introduction rules.

The scheme is illustrated by presenting the rules for the inductive *sets* N of natural numbers and $List(A)$ of lists with elements in the set A, and the inductive *family* $List'(A)$ of lists with elements in the set A which is indexed by the length of the list.

In general the definition of a set former P may depend on other set formers defined previously. Let T stand for the theory which contains the formation, introduction, elimination, and equality rules for these previously defined set formers. (In section 4 we propose to let the theory be open with respect to recursive definitions as well as to inductive definitions. In that case, T stands for the theory which contains formation and introduction rules for previous set formers defined inductively and the typing and equality rules for functions defined by P-recursion.)

In the scheme below we will state requirements such as 'σ is a sequence of types' which means that 'σ_1 is a type in T, ..., σ_n is a type in T under the assumptions $a_1 : \sigma_1$, ..., $a_{n-1} : \sigma_{n-1}$'.

3.1 Formation rule

3.1.1 Scheme

There is one formation rule (unless we have simultaneous induction, see section 6). It has the form

$$
\begin{array}{ll}
P : & (A :: \sigma) \\
 & (a :: \alpha[A]) \\
 & set,
\end{array}
$$

where

- σ is a sequence of types;
- $\alpha[A]$ is a sequence of s-types under the assumptions $A :: \sigma$.

We refer to A as the *parameters* of P. Observe that the premises $A :: \sigma$ appear first in all rules for P: they are global to the definition. When P_A is thought of as a family of sets, then a are the *indices*. (P_A is the same as $P(A)$. We write some arguments in index position to improve readability.)

3.1.2 Examples

Natural numbers. The formation rule is

$$N : set.$$

There are no parameters and no indices.

Lists. The formation rule is

$$List : (A : set)set.$$

The set A is a parameter and will appear as the first premise of all the rules for lists. There are no indices.

Lists of a certain length. The formation rule is

$$\begin{array}{ll} List' : & (A : set) \\ & (a : N) \\ & set. \end{array}$$

The set A is a parameter. N is the index set. This definition depends on the natural numbers N having been defined before.

3.2 Introduction rules

3.2.1 Scheme

There are finitely many introduction rules for each set former. Each introduction rule has the form

$$\begin{array}{ll} \mathit{intro} : & (A :: \sigma) \\ & (b :: \beta[A]) \\ & (u :: \gamma[A, b]) \\ & P_A(p[A, b]), \end{array}$$

where

- $\beta[A]$ is a sequence of s-types under the assumptions $A :: \sigma$;
- each $\gamma_i[A, b]$ has the form

  $$\begin{array}{l} (x :: \xi_i[A, b]) \\ P_A(p_i[A, b, x]), \end{array}$$

  where

  - $\xi_i[A, b]$ is a sequence of s-types under the assumptions $A :: \sigma$ and $b :: \beta[A]$
  - and $p_i[A, b, x] :: \alpha[A]$ under the assumptions $A :: \sigma$, $b :: \beta[A]$ and $x :: \xi_i[A, b]$;
- $p[A, b] :: \alpha[A]$ under the assumptions $A :: \sigma$ and $b :: \beta[A]$.

We refer to b as the *non-recursive* and u as the *recursive* arguments of the constructor *intro*. ξ_i is to make provision for *generalized induction*. Since P cannot occur in ξ_i the induction is *strictly positive*. If all ξ_i are empty then we have *ordinary induction*.

3.2.2 Examples

Natural numbers. The first introduction rule is

$$0 : N.$$

There are neither non-recursive nor recursive premises.

The second introduction rule is

$$s : (u : N)N.$$

There is only a recursive premise which is ordinary.

Lists. The first introduction rule is

$$\begin{array}{ll} nil: & (A : set) \\ & List_A. \end{array}$$

There are neither recursive nor non-recursive premises.

The second introduction rule is

$$\begin{array}{ll} cons: & (A : set) \\ & (b : A) \\ & (u : List_A) \\ & List_A. \end{array}$$

There is one non-recursive premise $b : A$ and one ordinary recursive premise $u : List_A$.

Lists of a certain length. The first introduction rule is

$$\begin{array}{ll} nil': & (A : set) \\ & List'_A(0). \end{array}$$

There are neither recursive nor non-recursive premises. It is distinguished from the corresponding rule for $List_A$ by the presence of the index 0 in the conclusion.

The second introduction rule is

$$\begin{array}{ll} cons': & (A : set) \\ & (b_1 : N) \\ & (b_2 : A) \\ & (u : List'_A(b_1)) \\ & List'_A(s(b_1)). \end{array}$$

There are two non-recursive premises $b_1 : N$ and $b_2 : A$ and one ordinary recursive premise $u : List'_A(b_1)$. The rule is also distinguished from the rule for $List_A$ by the presence of the index b_1 in the recursive premise and the index $s(b_1)$ in the conclusion.

3.3 Elimination rule

3.3.1 Scheme

Given the formation and introduction rules for P we can derive the following elimination rule:

$$\begin{array}{lll} elim: & (A :: \sigma) & \\ & (C: & (a :: \alpha[A]) \\ & & (c : P_A(a)) \\ & & set) \\ & (e :: \epsilon[A]) & \\ & (a :: \alpha[A]) & \\ & (c : P_A(a)) & \\ & C(a, c). & \end{array}$$

$c : P_A(a)$ is called the *major premise* and $e :: \epsilon[A]$ are called the *minor premises*. There is one minor premise for each constructor. The type $\epsilon_j[A]$ in the minor premise corresponding to the constructor *intro* is

$$\begin{array}{l} (b :: \beta[A]) \\ (u :: \gamma[A, b]) \\ (v :: \delta[A, b]) \\ C(p[A, b], intro_A(b, u)). \end{array}$$

$\delta[A, b]$ has the same length as $\gamma[A, b]$ and $\delta_i[A, b]$ is

$$\begin{array}{l}(x :: \xi_i[A, b]) \\ C(p_i[A, b, x], u_i(x)).\end{array}$$

When the elimination rule is viewed as an induction rule then each minor premise is a case of the induction. If *intro* has no recursive argument, then it is a base case, otherwise it is a step case. $C(p_i[A, b, x], u_i(x))$ is the (generalized) induction hypothesis and $u_i(x)$ are the predecessors of $intro_A(b, u)$.

3.3.2 Examples

Natural numbers. From the information in the formation and the introduction rules for lists we can derive the elimination rule (the rule of primitive recursion or mathematical induction)

$$\begin{array}{rll} nrec: & (C : (c : N)set) & \\ & (e_1 : C(0) & \\ & (e_2 : & (u : N) \\ & & (v : C(u)) \\ & & C(s(u))) \\ & (c : N) & \\ & C(c). & \end{array}$$

Lists. Here we can derive the elimination rule (the rule of list recursion or list induction):

$$\begin{array}{rll} listrec: & (A : set) & \\ & (C : (c : List_A)set) & \\ & (e_1 : C(nil_A)) & \\ & (e_2 : & (b : A) \\ & & (u : List_A) \\ & & (v : C(u)) \\ & & C(cons_A(b, u))) \\ & (c : List_A(a)) & \\ & C(c). & \end{array}$$

Lists of a certain length. Here we can derive the elimination rule

$$\begin{array}{rll} listrec': & (A : set) & \\ & (C : (a : N)(c : List'_A(a))set) & \\ & (e_1 : C(0, nil'_A)) & \\ & (e_2 : & (b_1 : N) \\ & & (b_2 : A) \\ & & (u : List'_A(b_1)) \\ & & (v : C(b_1, u)) \\ & & C(s(b_1), cons'_A(b_1, b_2, u))) \\ & (a : N) & \\ & (c : List'_A(a)) & \\ & C(a, c). & \end{array}$$

3.4 Equality rules

3.4.1 Scheme

There is one equality rule for each introduction rule. The one corresponding to the constructor *intro* is

$$(A,C,e,b,u)elim_{A,C}(e,p[A,b],intro_A(b,u)) = (A,C,e,b,u)e_j(b,u,v): \begin{array}{l}(A :: \sigma)\\ (C: \begin{array}[t]{l}(a :: \alpha[A])\\ (c : P_A(a))\\ set)\end{array}\\ (e :: \epsilon[A])\\ (b :: \beta[A])\\ (u :: \gamma[A,b])\\ C(p[A,b], intro_A(b,u)),\end{array}$$

where v_i is

$$(x)elim_{A,C}(e,p_i[A,b,x],u_i(x)).$$

3.4.2 Examples

Natural numbers. The equality rule for the constructor 0 is

$$(C,e_1,e_2)nrec_C(e_1,e_2,0) = (C,e_1,e_2)e_1: \begin{array}{l}(C:(c:N))set)\\ (e_1 : C(0)\\ (e_2: \begin{array}[t]{l}(u:N)\\ (v:C(u))\\ C(s(u)))\end{array}\\ C(0).\end{array}$$

The equality rule for the constructor s is

$$(C,e_1,e_2,u)nrec_C(e_1,e_2,s(u)) = (C,e_1,e_2,u)e_2(u,nrec_C(e_1,e_2,u)): \begin{array}{l}(C:(c:N))set)\\ (e_1 : C(0)\\ (e_2: \begin{array}[t]{l}(u:N)\\ (v:C(u))\\ C(s(u)))\end{array}\\ (u:N)\\ C(0).\end{array}$$

Lists. The equality rule for the constructor *nil* is

$$(A,C,e_1,e_2)listrec_{A,C}(e_1,e_2,nil_A) = (A,C,e_1,e_2)e_1: \begin{array}{l}(A:set)\\ (C:(c:List_A))set)\\ (e_1 : C(nil_A)\\ (e_2: \begin{array}[t]{l}(b:A)\\ (u:List_A)\\ (v:C(u))\\ C(cons_A(b,u)))\end{array}\\ C(nil_A).\end{array}$$

The equality rule for the constructor *cons* is

$$\begin{array}{rl}
& (A, C, e_1, e_2, b, u)\,listrec_{A,C}(e_1, e_2, cons_A(b, u)) \\
= & (A, C, e_1, e_2, b, u)\,e_2(b, u, listrec_{A,C}(e_1, e_2, u)) \\
: & (A : set) \\
& (C : (c : List_A)set) \\
& (e_1 : C(nil_A) \\
& (e_2 : \begin{array}[t]{l} (b : A) \\ (u : List_A) \\ (v : C(u)) \\ C(cons_A(b, u))) \end{array} \\
& (b : A) \\
& (u : List_A) \\
& C(cons_A(b, u)),
\end{array}$$

Lists of a certain length. The equality rule for the constructor nil' is

$$(A, C, e_1, e_2)\,listrec'_{A,C}(e_1, e_2, 0, nil'_A) = (A, C, e_1, e_2)e_1 : \begin{array}[t]{l} (A : set) \\ (C : (a : N)(c : List'_A(a))set) \\ (e_1 : C(0, nil'_A)) \\ (e_2 : \begin{array}[t]{l} (b_1 : N) \\ (b_2 : A) \\ (u : List'_A(b_1)) \\ (v : C(b_1, u)) \\ C(s(b_1), cons'_A(b_1, b_2, u))) \end{array} \\ C(0, nil'{}_A). \end{array}$$

The equality rule for the constructor *cons* is

$$\begin{array}{rl}
& (A, C, e_1, e_2, b_1, b_2, u)\,listrec'{}_{A,C}(e_1, e_2, cons'_A(b_1, b_2, u)) \\
= & (A, C, e_1, e_2, b_1, b_2, u)e_2(b_1, b_2, u, listrec'{}_{A,C}(e_1, e_2, u)) \\
: & (A : set) \\
& (C : (a : N)(c : List'_A(a))set) \\
& (e_1 : C(0, nil'_A)) \\
& (e_2 : \begin{array}[t]{l} (b_1 : N) \\ (b_2 : A) \\ (u : List'_A(b_1)) \\ (v : C(b_1, u)) \\ C(s(b_1), cons'_A(b_1, b_2, u))) \end{array} \\
& (b_1 : N) \\
& (b_2 : A) \\
& (u : List'_A(b_1)) \\
& C(s(b_1), cons'_A(b_1, b_2, u))
\end{array}$$

As an application of $List'$-recursion we define a function which maps a list in $List'_A(n)$ to the corresponding list in $List_A$:

$$forgetlength = (A)listrec'_{A,(a,c)List_A}(nil_A, (b_1, b_2, u, v)\,cons_A(b_2, v)) \quad : \quad \begin{array}[t]{l} (A : set) \\ (a : N) \\ (c : List'_A(n)) \\ List_A. \end{array}$$

From this definition we can derive the recursion equations from the equality rules for *listrec'*:

$$\begin{aligned} \mathit{forgetlength}_A(0, \mathit{nil}'_A) &= \mathit{nil}_A : \mathit{List}_A, \\ (b_1, b_2, u)\mathit{forgetlength}_A(s(b_1), \mathit{cons}'_A(b_1, b_2, u)) &= \mathit{cons}_A(b_2, \mathit{forgetlength}_A(b_1, u)) : \mathit{List}_A. \end{aligned}$$

4 A scheme for recursive definitions

It would be desirable to generalize the scheme for elimination and equality rules by allowing C to be an arbitrary family of types instead of requiring that it is a family of sets. In this way the result of a function defined by P-recursion can be an object of an arbitrary type, for example, a set, and not necessarily an element of a set. The standard way of achieving this effect is to use the usual elimination rule in conjunction with Π-sets and universes.

It would be convenient to integrate this principle directly into the scheme, but we cannot simply replace *set* above by *type* in the typing of C, because we do not have *type:type*. One solution is to index the elimination rule by a family of types C, see Smith [35], but if we want to keep the notion of a finitary theory (and not extend the theory of logical types) then we must treat the theory as open (potentially infinite) with respect to recursive definitions as well as for inductive definitions. This is one argument that leads us towards the idea that it is the recursive scheme rather than the elimination rule which is the basic concept, see Dybjer [13].

Furthermore, this view is essential, and not only a matter of convenience, both for enlarging the scheme to capture universes and other *simultaneous inductive-recursive* definitions [14] and for Coquand's [5] approach to pattern matching with dependent types.

We shall now give precise criteria for schematic recursive definitions in a similar way as we did for schematic inductive definitions above. This schema will for example yield the typing rule and recursion equations of *forgetlength* as instances, so it will not be necessary to define it in terms of the elimination rule. Note also that the elimination and the equality rules themselves are instances of the schema.

The schema specifies the general form for the typing rule and recursion equations for a new function or family defined by P-recursion. As before, such a definition may depend on other constants defined previously by induction or recursion. Let as before T stand for the theory which contains the formation and introduction rules for each inductive set or family and the typing and equality rules for each recursive function or family. (A similar modification of the notion of current theory T for an inductive definition is of course needed in section 3.)

4.1 Schematic elimination rule

The typing rule for a function defined by P-recursion has the form

$$\begin{aligned} f : \quad & (B :: \tau) \\ & (a :: \alpha[Q[B]]) \\ & (c :: P_A(a)) \\ & \psi[B, a, c], \end{aligned}$$

where

- τ is a sequence of types;
- $Q[B] :: \sigma$ under the assumptions $B :: \tau$;
- $\psi[B, a, c]$ is a type under the assumptions $B :: \tau, a :: \alpha[A], c :: P_A(a)$;

and σ and α refer to the formation rule for P. We refer to B as the *parameters* of f. (Note that Q is a sequence of constants here: we can define f with arbitrary parameters, but we must instantiate the parameters of P.)

Note that an appropriate choice of τ, Q, and ψ will yield the ordinary elimination (and equality) rules given before.

As an example we can check that the typing rule for *forgetlength* is an instance of this scheme, where $\tau = set$, $Q[B] = B : \tau$, and $\psi[B, a, c] = List_B$.

4.2 Schematic equality rules

There is one equality rule for f for each introduction rule for P. The one corresponding to the constructor *intro* is

$$\begin{array}{ll}(B, a, b, u) f_B(p[Q[B], b], intro_{Q[B]}(b, u)) = (B, Q[B], b, u) e_j(b, u, v) : & (B :: \tau) \\ & (b :: \beta[Q[B]]) \\ & (u :: \gamma[Q[B], b]) \\ & \psi[B, p[Q[B], b], intro_{Q[B]}(b, u)],\end{array}$$

where v_i is

$$(x) f_B(p_i[A, b, x], u_i(x)),$$

and β, γ, p, and p_i refer to the introduction rule for *intro*.

5 More examples

5.1 Predicate logic

The logical constants $\perp = \emptyset$, $\top = 1$, $\vee = +$, $\wedge = \times$, $\supset = \rightarrow$, $\exists = \Sigma$, and $\forall = \Pi$ are examples of inductive propositions (= sets), and were thus part of Backhouse's scheme. We just point out that $\perp$ is the unique set which has no introduction rule, and show the rules for $\supset$, since the notion of s-type is essential there.

We also show rules for propositional equality, which is an inductive predicate (= family), and hence not covered by Backhouse's scheme. There are two versions. The first is the formulation by Martin-Löf [21] of an inductive relation. The second formulation by Christine Paulin is of a unary inductive predicate which can be seen as a predicative version of Leibniz equality. It is a nice example of a definition parameterized over elements of a set.

5.1.1 Implication

Formation rule.

$$\begin{array}{ll}\supset: & (A_1 : set) \\ & (A_2 : set) \\ & set.\end{array}$$

A_1 and A_2 are parameters.

Introduction rule.

$$\begin{array}{lll}\lambda : & (A_1 : set) & \\ & (A_2 : set) & \\ & (b : & (x : A_1) \\ & & A_2) \\ & A_1 \supset A_2. & \end{array}$$

This introduction rule has one non-recursive premise, the type of which is $(x : A_1)A_2$ which is an s-type but not a set.

Elimination rule.

$$\begin{aligned} F :\ & (A_1 : set) \\ & (A_2 : (x : A_1)set) \\ & (C : set) \\ & (e : (b : (x : A_1)A_2(x))C) \\ & (c : A_1 \supset A_2) \\ & C. \end{aligned}$$

This is the $\supset$-elimination rule introduced by Schroeder-Heister [34]. In the more general form, where A_2 and C are families, it becomes the Π-elimination rule introduced in the preface of Martin-Löf [26]. A synonym for F is *funsplit*.

5.1.2 Equality à la Martin-Löf

This is equality defined as the least reflexive relation [21].

Formation rule.

$$\begin{aligned} I :\ & (A : set) \\ & (a_1 : A) \\ & (a_2 : A) \\ & set. \end{aligned}$$

The set A is a parameter and the elements $a_1, a_2 : A$ are indices.

Introduction rule.

$$\begin{aligned} r :\ & (A : set) \\ & (b : A) \\ & I_A(b, b). \end{aligned}$$

Elimination rule.

$$\begin{aligned} J :\ & (A : set) \\ & (C : (a_1 : A)(a_2 : A)set) \\ & (e : (b : A)C(b, b)) \\ & (a_1 : A) \\ & (a_2 : A) \\ & C(a_1, a_2). \end{aligned}$$

5.1.3 Equality à la Paulin

This is equality defined as the least unary predicate containing a given element (a predicative version of Leibnitz equality).

Formation rule.

$$\begin{aligned} I' :\ & (A : set) \\ & (a_1 : A) \\ & (a_2 : A) \\ & set. \end{aligned}$$

The set A and the given element $a_1 : A$ are parameters, and $a_2 : A$ is an index.

Introduction rule.

$$\begin{aligned} r' :\ & (A : set) \\ & (a_1 : A) \\ & I'_{A,a_1}(a_1). \end{aligned}$$

Elimination rule.

$$\begin{aligned} J' : \quad & (A : set) \\ & (a_1 : A) \\ & (C : (a_2 : A)set) \\ & (e : C(a_1)) \\ & (a_2 : A) \\ & (c : I'_{A,a_1}(a_2)) \\ & C(a_2). \end{aligned}$$

Note that the difference only is visible in the elimination rule, which is the usual rule of equality elimination in predicate logic and simpler than equality elimination à la Martin-Löf.

5.2 Generalized induction

All definitions given so far have used ordinary induction, that is, the lists of premises ξ_i of recursive premises have been empty. We shall now give two examples of generalized induction: the wellorderings introduced by Martin-Löf [24] and the wellfounded part of a relation. Other examples are the *ordinals of the second and higher number classes*, see Martin-Löf [26] and Petersson and Synek's generalized *trees* [32].

5.2.1 Well-orderings

The well-orderings are an important instance of the scheme. It is the special case where we define an inductive *set* and where there is *one* introduction rule, which has *one* non-recursive premise $b : A_1$ and *one* recursive premise $u : (x : A_2(b))P$ (which has *one* hypothesis $x : A_2(b)$). It is thus the instance of Backhouse's scheme, where the word *several (possibly none)* has been replaced by *one*.

Backhouse [2] and Coquand and Paulin [7] allowed the inessential generalization where recursive premises may precede non-recursive ones. I prefer to put all non-recursive premises before the recursive ones, since the former cannot depend on the latter here (but the situation changes in Dybjer [14]). This restriction simplifies the presentation of the scheme and emphasizes the relationship with the well-orderings.

Formation rule.

$$\begin{aligned} W : \quad & (A_1 : set) \\ & (A_2 : (A_1)set) \\ & set. \end{aligned}$$

Introduction rule.

$$\begin{aligned} sup : \quad & (A_1 : set) \\ & (A_2 : (A_1)set) \\ & (b : A_1) \\ & (u : (x : A_2(b))W_{A_1,A_2}) \\ & W_{A_1,A_2}. \end{aligned}$$

Elimination rule.

$$\begin{array}{ll} T: & (A_1 : set) \\ & (A_2 : (A_1)set) \\ & (C : (W_{A_1,A_2})set) \\ & (e : \begin{array}[t]{l} (b : A_1) \\ (u : (x : A_2(b))W_{A_1,A_2}) \\ (v : (x : A_2(b))C(u(x))) \\ C(sup(b,u))) \end{array} \\ & (c : W_{A_1,A_2}) \\ & C(c). \end{array}$$

5.2.2 The well-founded part of a relation

If A_1 is a set and A_2 is a binary relation on that set, then $Acc_{A_1,A_2}(a)$ is true iff a is in the well-founded part of A_2.

An application of this notion in the context of type theory can be found in Nordström [28]. He suggested to add general recursion along a well-founded relation to a version of type theory in which propositions and sets are not identified and which is also extended with subset formation. Compare also the discussion in Dybjer [12].

Formation rule.

$$\begin{array}{ll} Acc: & (A_1 : set) \\ & (A_2 : (A_1)(A_1)set) \\ & (a : A_1) \\ & set. \end{array}$$

Introduction rule.

$$\begin{array}{ll} acc: & (A_1 : set) \\ & (A_2 : (A_1)(A_1)set) \\ & (b : A_1) \\ & (u : (x_1 : A_1)(x_2 : A_2(x_1,b))Acc_{A_1,A_2}(x_1)) \\ & Acc_{A_1,A_2}(b). \end{array}$$

Elimination rule.

$$\begin{array}{lll} accrec: & (A_1 : & set) \\ & (A_2 : & (A_1)(A_1)set) \\ & (C : & (a : A_1) \\ & & (c : Acc_{A_1,A_2}(a)) \\ & & set) \\ & (e : & (b : A_1) \\ & & (u : (x_1 : A_1)(x_2 : A_2(x_1,b))Acc_{A_1,A_2}(x_1)) \\ & & (v : (x_1 : A_1)(x_2 : A_2(x_1,b))C(x_1,u(x_1,x_2)) \\ & & C(b, acc_{A_1,A_2}(b,u))) \\ & (a : & A_1) \\ & (c : & Acc_{A_1,A_2}(a)) \\ & C(a,c) & . \end{array}$$

5.3 Finite sets and n-tuples

In this and the following subsection we use our theory for formalizing basic notions of the untyped λ-calculus. The formalization uses bounded de Bruijn-indices (compare Curien [8]) rather than the more common unbounded ones (see, for example, the λ-calculus theory developed by Huet [19]).

Our bounded de Bruijn-indices are elements of the inductive family of finite sets indexed by N and defined in a similar way to $List'$ above.

We have the following rules:

N'-formation:

$$N' : (N)set$$

N'-introduction:

$$\begin{aligned} 0' &: (n : N)N'(s(n)), \\ s' &: (n : N)(N'(n))N'(s(n)). \end{aligned}$$

We use the notation $1 = s(0), 2 = s(1), N_n = N'(n), 0_n = 0'(n), s_n(i) = s'(n,i)$, and $1_{s(n)} = s_{s(n)}(0_n)$. (From now on we shall use somewhat less formal notation: we often drop some parameters and A in $a = b : A$, etc.)

We also need sets of n-tuples. We could have used $List'$ for this purpose but instead we define them by recursion on the natural numbers. The typing rule is

$$Tuple : (A : set)(n : N)set,$$

where $A : set$ is a parameter and $\psi[m,n] = set$ in the scheme in section 3. We use the notation $A^n = Tuple(A,n)$. The equality rules are

$$\begin{aligned} A^0 &= \top, \\ A^{s(n)} &= A^n \times A. \end{aligned}$$

(Note that this defines *snoclists*, which have their heads at the end and are more natural for defining substitutions in the λ-calculus.)

In a similar way we define a *map*-function for n-tuples

$$map : (A, B : set)(f : (A)B)(n : N)(A^n)B^n$$

by recursion on N. $A, B : set$, and $f : (A)B$ are parameters and $\psi[A,B,f,n] = (A^n)B^n$. We use the notation $f^n = map(A,B,f,n)$. The equality rules are

$$\begin{aligned} f^0(as) &= \langle\rangle, \\ f^{s(n)}(as) &= \langle f^n(fst(as)), f(snd(as))\rangle. \end{aligned}$$

The projection function

$$\pi : (A : set)(n : N)(i : N_n)(A^n)A$$

can then be defined by N_n-recursion, where $A : set$ is a parameter and $\psi[A,n,i] = (A^n)A$. We use the notation $\pi_n^i = \pi(A,n,i)$. The equality rules are

$$\begin{aligned} \pi_{s(n)}^{0_n}(as) &= snd(as), \\ \pi_{s(n)}^{s_n(i)}(as) &= \pi_n^i(fst(as)). \end{aligned}$$

We also need to consider certain n-tuples of elements in N_m, which will be used for creating certain substitutions in the λ-calculus.

Firstly there is the sequence used for the identity substitution:

$$id : (n : N)N_n^n$$

which is defined by N-recursion

$$\begin{array}{rcl} id_0 & = & \langle\rangle, \\ id_{s(n)} & = & \langle s_n^n(id_n), 0_n\rangle. \end{array}$$

Then there is the sequence used for lifting (or thinning or projection);

$$\uparrow : (n : N)N_{s(n)}^n$$

which is defined by N-recursion

$$\begin{array}{rcl} \uparrow_0 & = & \langle\rangle, \\ \uparrow_{s(n)} & = & \langle s_{s(n)}^n(\uparrow_n), s_{s(n)}(0_n)\rangle. \end{array}$$

5.4 The untyped λ-calculus

First we define λ-terms as an inductive family indexed by N. Λ_n represents λ-terms with at most n free variables.

Λ-formation:

$$\Lambda : (N)set.$$

Λ-introduction:

$$\begin{array}{rcl} var & : & (n : N)(i : N_n)\Lambda_n, \\ \lambda & : & (n : N)(\Lambda_{s(n)})\Lambda_n, \\ ap & : & (n : N)(\Lambda_n)(\Lambda_n)\Lambda_n. \end{array}$$

The reader should check that these rules follow the scheme. Note that the index only varies in the rule for λ.

The identity combinator $\lambda x.x$ is represented by

$$\lambda_0(var_1(0_1)) : \Lambda_0,$$

and the K-combinator $\lambda xy.x$ is represented by

$$\lambda_0(\lambda_1(var_2(1_2))) : \Lambda_0.$$

Next, we shall define simultaneous substitution of terms for all free variables of a λ-term. To this end we need to use n-tuples of λ-terms, that is, *substitutions*, in Λ_m^n.

The substitution function is explicit in the sense that it is formalized explicitly in type theory, which serves as a metalanguage here. It is not explicit in the sense of Abadi, Cardelli, Curien, and Lévy [1] however, since it is not a constructor for λ-terms, but instead defined by Λ-recursion on $n : N, g : \Lambda_n$:

$$sub : (n : N)(g : \Lambda_n)(m : N)(fs : \Lambda_m^n)\Lambda_m,$$

where $\psi[n, g] = (m : N)(fs : \Lambda_m^n)\Lambda_m$.

$$\begin{array}{rcl} sub_n(var_n(i), m, fs) & = & \pi_n^i(fs), \\ sub_n(\lambda_n(g), m, fs) & = & \lambda_m(sub_{s(n)}(g, s(m), \langle lift_m^n(fs), var_{s(m)}(0_m)\rangle)), \\ sub_n(ap_n(h, f), m, fs) & = & ap_m(sub_n(h, m, fs), sub_n(f, c, fs)), \end{array}$$

where lifting

$$lift : (n : N)(\Lambda_n)\Lambda_{s(n)}$$

is defined by

$$lift_n(f) = rename(n, f, \uparrow_n),$$

and

$$rename : (n : N)(g : \Lambda_n)(m : N)(is : N_m^n)\Lambda_m$$

is simultaneous substitution of variables for free variables, that is, changes of variables in a λ-term. It is defined by Λ-recursion on $n : N, g : \Lambda_n$ with $\psi[n, g] = (m : N)(as : \Lambda_m^n)\Lambda_m$.

$$rename : (n : N)(g : \Lambda_n)(m : N)(is : N_m^n)\Lambda_m,$$

$$\begin{array}{rcl} rename_n(var_n(i), m, is) & = & var(\pi_n^i(is)), \\ rename_n(\lambda_n(g), m, is) & = & \lambda_m(rename_{s(n)}(g, s(m), \langle s_m^n(is), 0_m\rangle)), \\ rename_n(ap_n(h, f), m, fs) & = & ap_m(rename_n(h, m, fs), rename_n(f, c, fs)). \end{array}$$

Finally, we can define β-convertibility as an inductive family of relations.

$\sim$-formation:

$$\sim : (n : N)(f, f' : \Lambda_n)\Lambda_n.$$

There are no prameters: n, f, f' are all indices.

$\sim$-introduction:

$$\begin{array}{rcl} varcong & : & (n : N)(i : N_n)var_n(i) \sim_n var_n(i), \\ \xi & : & (n : N)(g, g' : \mathcal{F}_{s(n)})(g \sim_{s(n)} g')\lambda(g) \sim_n \lambda(g'), \\ apcong & : & (n : N)(h, h' : \mathcal{F}_n)(h \sim_n h')(f, f' : \mathcal{F}_n)(f \sim_n f')ap_n(h, f) \sim_n ap_n(h', f'), \\ \beta & : & (n : N)(g : \mathcal{F}_{s(n)})(f : \mathcal{F}_n)ap(\lambda_n(g), f) \sim_n sub_{s(n)}(g, n, \langle var_n^n(id_n), f\rangle), \\ trans & : & (n : N)(f, g, h : \Lambda_n)(f \sim_n g)(g \sim_n h)f \sim_n h, \\ sym & : & (n : N)(f, g : \Lambda_n)(f \sim_n g)g \sim_n f. \end{array}$$

From this we could continue and formalize proofs of theorems about the untyped λ-calculus using Λ- and $\sim$-induction, etc.

6 Simultaneous induction

The scheme above introduces one new set former at a time. It can be generalized so that finitely many are introduced simultaneously. We call such a definition a *block*.

6.1 Formation rules

6.1.1 Scheme

There is one formation rule for each set former introduced in the block. They each have the form

$$\begin{array}{rl} P_k : & (A :: \sigma) \\ & (a :: \alpha_k[A]) \\ & set, \end{array}$$

where

- σ is a sequence of types (common to all set formers in the block);
- $\alpha_k[A]$ is a sequence of s-types under the assumptions $A :: \sigma$.

6.1.2 Example: even and odd numbers

This definition presupposes N. There are no parameters. The index set is N.

Formation rules.

$$Even : (a : N)set, \qquad Odd : (a : N)set.$$

6.2 Introduction rules

6.2.1 Scheme

Each introduction rule has the form

$$\begin{array}{ll} intro : & (A :: \sigma) \\ & (b :: \beta[A]) \\ & (u :: \gamma[A, b]) \\ & P_{k\,A}(p[A, b]), \end{array}$$

where

- $\beta[A]$ is a sequence of s-types under the assumptions $A :: \sigma$;
- each $\gamma_i[A, b]$ has the form

$$\begin{array}{l} (x :: \xi_i[A, b]) \\ P_{k_i\,A}(p_i[A, b, x]), \end{array}$$

  where

  - $\xi_i[A, b]$ is a sequence of s-types under the assumptions $A :: \sigma$ and $b :: \beta[A]$,
  - and $p_i[A, b, x] :: \alpha_{k_i}$ under the assumptions $A :: \sigma$, $b :: \beta[A]$ and $x :: \xi_i[A, b]$;
- $p[A, b] :: \alpha_k$ under the assumptions $A :: \sigma$ and $b :: \beta[A]$.

6.2.2 Example

$$intro_1 : Even(0), \qquad \begin{array}{ll} intro_2 : & (b : N) \\ & (u : Even(b)) \\ & Odd(s(b)), \end{array} \qquad \begin{array}{ll} intro_3 : & (b : N) \\ & (u : Odd(b)) \\ & Even(s(b)). \end{array}$$

6.3 Elimination rules

6.3.1 Scheme

$$\begin{array}{ll} elim_l : & (A :: \sigma) \\ & (C :: \phi[A]) \\ & (e :: \epsilon[A]) \\ & (a :: \alpha_l[A]) \\ & (c : P_{l\,A}(a)) \\ & C(a, c). \end{array}$$

The length of ϕ is the number of set formers introduced in the block. $\phi_k[A]$ is

$$\begin{array}{l} (a :: \alpha_k[A]) \\ (P_{k\,A}(a)) \\ set. \end{array}$$

There is one minor premise for each constructor in *the whole block*. The type ϵ_j of the minor premise corresponding to the constructor *intro* follows the same pattern as above.

6.3.2 Example

$$
\begin{array}{llll}
evenelim: & (C_1 : (a:N)(Even(a))set) & oddelim: & (C_1 : (a:N)(Even(a))set) \\
& (C_2 : (a:N)(Odd(a))set) & & (C_2 : (a:N)(Odd(a))set) \\
& (e_1 : \ \ C_1(0, intro_1)) & & (e_1 : \ \ C_1(0, intro_1)) \\
& (e_2 : \ \ (b:N) & & (e_2 : \ \ (b:N) \\
& \qquad (u : Even(b)) & & \qquad (u : Even(b)) \\
& \qquad (v : C_1(b,u)) & & \qquad (v : C_1(b,u)) \\
& \qquad C_2(s(b), intro_2(b,u))) & & \qquad C_2(s(b), intro_2(b,u))) \\
& (e_3 : \ \ (b:N) & & (e_3 : \ \ (b:N) \\
& \qquad (u : Odd(b)) & & \qquad (u : Odd(b)) \\
& \qquad (v : C_2(b,u)) & & \qquad (v : C_2(b,u)) \\
& \qquad C_1(s(b), intro_3(b,u))) & & \qquad C_1(s(b), intro_3(b,u))) \\
& (a:N) & & (a:N) \\
& (c : Even(a)) & & (c : Odd(a)) \\
& C_1(a,c), & & C_2(a,c).
\end{array}
$$

6.4 Equality rules

Omitted.

6.5 Scheme for recursive definitions

Omitted.

6.6 More examples

Initial Many-Sorted Algebras. With simultaneous induction it is easy to see how to construct initial many sorted algebras [15]. Each sort will denote a (constant) set and each operator an introduction rule with only ordinary recursive premises. If there are equations we associate an inductively defined relation on each set. Note that the inversion principle gives us a formal structural induction principle inside type theory.

Iterated inductive definitions in predicate logic. We can also essentially interpret Martin-Löf's intuitionistic theory of iterated inductive definitions [21], provided we only allow finitely many predicate symbols in that theory. First, we define the set of individuals by letting each function symbol correspond to a constructor. Then we see that each *ordinary or generalized production* for a predicate symbol corresponds to an introduction rule for an inductive family. The *level* of a predicate symbol will determine the order in which it can be introduced and *linked* predicate symbols have to be introduced in a block.

An application of this theory is as foundation for logic programming as proposed by Hagiya and Sakurai [16].

7 Further references

Several people have used the present formulation of inductively defined sets, families of sets, and predicates in type theory for formal (and sometimes machine-assisted) program derivation and theorem proving.

One example is the normalization of if-expression, which occurs as part of Boyer and Moore's tautology-checker for propositional logic [4]. The set of if-expressions is an inductively defined set and the subset of normalized if-expressions is naturally represented in type theory as an inductively defined predicate, see Dybjer [12].

Michael Hedberg has looked at the similar but simpler example of normalization of binary trees: 'normalizing the associative law' [18]. He uses inductively defined sets and predicates in a variety of ways to show that Martin-Löf type theory can be used not only as an 'integrated logic' (based on the Curry-Howard identification), but also as an 'external logic' which can be used for verifying an externally given general recursive program. Hedberg has implemented his examples using Paulson's Isabelle-system [31].

Nora Szasz [36] has formalized the proof that Ackermann's function is not primitive recursive. The basic definition is a binary inductive family of tuples of primitive recursive functions $TPR(m,n)$, where n is the number of functions in the tuple and m is the arity of each function. Szasz has implemented her proof in the ALF-system.

Presently, there are a number of ongoing formalization projects using type theory with inductive definitions and implemented in the new version ALF-system of Coquand, Magnusson and Nordström. This version supports a powerful form of *pattern matching with dependent types* which has recently been proposed by Thierry Coquand [5]. It can be viewed as a strengthening of the schematic approach to elimination rules described in section 4. Firstly, it allows the definition of functions by case analysis on several arguments simultaneously and uses a criterion that recursive calls must refer to *structurally smaller* arguments to ensure termination. Secondly, unification is used to generate possible cases. This entails a strengthening of case analysis for inductively defined families. An example is that the proof of

$$peano4 : (I(N,0,1))\bot$$

now follows directly by pattern matching. The introduction rule for equality is reflexivity, and since this rule cannot be unified with $I(N,0,1)$ no cases are generated.

The reader is referred to Coquand's paper [5] for details.

Finally, we would like to refer to the developments using similar schemes for inductive definitions in the context of impredicative type theory, which are explored by groups at INRIA using the Coq-system [10] and in Edinburgh using the LEGO-system [20].

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *ACM Conference on Principles of Programming Languages, San Francisco*, 1990.

[2] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In *Proceedings of the Workshop on General Logic, Edinburgh, February 1987*. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1988. ECS-LFCS-88-52.

[3] R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory (part 1). *Formal Aspects of Computing*, pages 19–84, 1989.

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

[5] T. Coquand. Pattern matching with dependent types. In *Proceedings of The 1992 Workshop on Types for Proofs and Programs*, June 1992.

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

[7] T. Coquand and C. Paulin. Inductively defined types, preliminary version. In *LNCS 417, COLOG '88, International Conference on Computer Logic*. Springer-Verlag, 1990.

[8] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 1992.

[9] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, (91):189–204, 1991.

[10] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. The Coq proof assistant version 5.6, user's guide. Technical report, INRIA Rocquencourt - CNRS ENS Lyon, 1991.

[11] P. Dybjer. Inductively defined sets in Martin-Löf's type theory. In *Proceedings of the Workshop on General Logic, Edinburgh, February 1987*. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1988. ECS-LFCS-88-52.

[12] P. Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79, 1990.

[13] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[14] P. Dybjer. Universes and a general notion of simultaneous inductive-recursive definition in type theory. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[15] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*. Prentice Hall, 1978.

[16] M. Hagiya and T. Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2:59–77, 1984.

[17] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*, pages 193–204, 1987.

[18] M. Hedberg. Normalizing the associative law: An experiment with Martin-Löf's type theory. *Formal Aspects of Computing*, 3:218–252, 1991.

[19] G. Huet. Residual theory in λ-calculus: a complete Gallina development. 1993.

[20] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO (a preliminary user's manual). Technical report, University of Edinburgh, 1989.

[21] P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland, 1971.

[22] P. Martin-Löf. An intuitionistic theory of types. Unpublished report. To appear in "25 Years of Type Theory", 1972.

[23] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[24] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[25] P. Martin-Löf. The domain interpretation of type theory, lecture notes. In K. Karlsson and K. Petersson, editors, *Workshop on Semantics of Programming Languages, Abstracts and Notes*, Chalmers University of Technology and University of Göteborg, August 1983. Programming Methodology Group.

[26] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[27] P. F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, September 1987.

[28] B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.

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

[30] C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *Proceedings Typed λ-Calculus and Applications*, pages 328–245. Springer-Verlag, LNCS, March 1993.

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

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

[33] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), December 1984.

[34] P. Schroeder-Heister. Judgements of higher levels and standardized rules for logical constants in Martin-Löf's theory of logic. Unpublished paper, June 1985.

[35] J. Smith. Propositional functions and families of types. *Notre Dame Journal of Formal Logic*, 30(3):442–458, 1989.

[36] N. Szasz. A Machine Checked Proof that Ackermann's Function is not Primitive Recursive. In G. Huet and G. Plotkin, editors, *Proceedings of the Second Workshop on Logical Frameworks.* Cambridge University Press, 1992.