Inductive types in constructive languages

Bruin, Peter Johan de

Chapter 1

Introduction

This thesis circles around two themes that closely intertwine: formalized mathematical language, and inductive types. We speak about the former in section 1.1–1.6, resulting in the language description in chapter 2; inductive types are addressed in 1.7–1.10 and fill the major part of the thesis. The remaining sections of this chapter introduce some basic concepts for further use.

1.1 Mathematical language

Mathematical language is a field of interest that is shared by mathematicians and computing scientists. For mathematicians firstly as a medium to express their abstract thinking, secondly as the subject of formal analysis itself.

The computing scientist stands in between these two approaches. For the theoretical analysis of computing, (s)he needs a medium of expression just as the mathematician. But he is also fascinated by the possibility of rendering mathematical treatises accessible for computer manipulation, in order that computers may assist in creating, verifying, and transforming mathematical texts.

Here a yawning gap appears. For the mathematician excels in creative use of his language, inventing new styles of notation, new modes of reasoning. The formalist on the other hand requires a well-defined system of permitted notations and deduction steps. This may include a scheme for introducing some sort of new notation, yet time and again the language user will find good reason to step outside the provided schemes in order to attain more clarity of expression. Especially when the deduction steps themselves have to be noted down, formal calculus becomes too cumbersome for most application fields.

Our primal impetus was to work on diminishing this gap by developing a kind of universal calculus. On the one hand it would contain a sound definition of correct mathematical construction and deduction principles, complete for all practical purposes. On the other hand it should permit the user a freedom of notational definition that restricts him as little as possible.

Needless to say, this is an ideal, floating in the air, that we can hardly expect to realize on earth. Not letting ourselves be dispirited by this, we have tried to grasp the inspirations we received and to mould them into concrete form. Our second theme, inductive types, has served as a playground to gain experience in using our notational

ideas. At the same time, our treatment of inductive types contains in abstract form the various forms in which inductive types appear in other languages.

The formal realization of these ideas has often been unruly, and we apologize for the defects in our present work. Some ideas are not worked out in full detail, but there are also ideas whose formalization would require extensive elaboration and reconsideration of the language set-up.

We call the resulting language *ADAM*, as we hope virtually all of mathematics to appear among its offspring, and also in honor of the city of Amsterdam, which name abbreviates to *A'dam*.

1.2 Our approach to mathematical language

To begin with, let us note that our aim is a language that serves as a universal medium of mathematical expression, not a calculus that is directed at specific purposes such as problem solving through formal manipulation of the language expressions themselves. Rather, it should be possible to embed any specific calculus within the language. This aims in particular at *programming logics*, that describe the semantics of particular programming languages.

When offering the user notational freedom and brevity, one cannot avoid that some standard or user-defined notations may overlap. Thus, the possibility of ambiguity is inherent in our approach. Furthermore, the user should have the option to omit some details when he expects these to be obvious or reconstructible by the reader. In either case, it is the responsibility of the user (writer) to keep his text comprehensible.

Yet, the language should have a sound formal foundation, guaranteeing all results to be correct. We were drawn to use *Constructive Type Theory* (or Generalized Typed Lambda Calculus), being attracted by its elegant unified treatment of proofs and objects, and of finite and infinite products and sums. Besides the construction principles that are directly related to the basic type constructors, one needs some additional axioms to obtain a full foundation. Rather than leaving each user to establish his own foundation, we prefer to establish a fixed foundation for common use. For foundational research, one may of course study alternatives.

In shaping the notations of *ADAM*, we looked at the established notations of mathematics, making some adaptations to give them a more regular type structure. A typical example is the notation for set formation.

Example 1.1 Traditionally, one writes

$$\{\, f(x) \mid P(x) \}\tag{1.1}$$

for the set containing $f(x)$ for all x such that condition $P(x)$ holds. This notation does not indicate which of the variables occurring free in $f(x)$ and $P(x)$ are locally bound. A minor objection is that it may be necessary to look ahead to the condition $P(x)$ before one can fully understand expression $f(x)$. In the 'Eindhoven quantifier notation', introduced by E.W. Dijkstra, one writes

$$\{x : P(x) : f(x)\,\}$$

to overcome both problems. When using generalized types, the variable has to be typed; furthermore, typings and conditions are both assumptions to be treated on a par, so we move the condition to the left of the colons, where an arbitrary declaration (sequence of assumptions) may appear. In this case, we get:

$$\{x\colon A;\ P(x) :: f(x)\,\}$$

For the special case when $f(x)$ is just x, we introduce a notation similar to (1.1):

$$\{\,x\colon A \mid\colon P(x)\}\ :=\ \{x\colon A;\ P(x) :: x\,\}$$

The interesting thing about these notations for sets is that they are suited to be used for logical quantifiers and generalized constructors too. This will be described in section 2.7.

1.2.1 Defining ADAM. To define the basic syntax and simultaneously all correctness requirements of *ADAM*, we use the powerful mechanism of two-level grammar, containing Horn-clause logic, which is described in section 2.1. Ideally, this grammar mechanism should be available within *ADAM* itself, so that the user may introduce new non-conventional notations, special-purpose calculi, or other (programming) languages.

The definition of *ADAM* proceeds in the following stages:

1. Description of the language definition mechanism
2. Definition of the underlying type theory using Horn clauses
3. Definition of the abstract and concrete syntax classes and their production rules, which reduce the meaning of language constructs to type theory
4. Development of a body of useful theory and notations

Actually, our definition is not so systematic. The definition mechanism is not described in full detail, and points 3 and 4 are mingled. Some language features are defined only partially, or described merely by a suggestive example, as their full formalization would go beyond the scope of this thesis.

1.2.2 Our use of ADAM. The main body of this thesis uses both *ADAM* and informal proof notation, but it can be thought of as encoded wholly in *ADAM*. This stands in contrast with appendix D, where typed lambda calculus is used as the object of study itself, rather than as a medium of expression.

We use *ADAM* mainly to formulate principles of inductive types, to justify them and establish relationships between them. As such, *ADAM* both provides a unifying framework in which principles from many different languages can be represented, and gives a sound foundation to these principles.

1.2.3 Semantics. The semantics of *ADAM* is given by its type theory, named ATT. The rules of ATT may be regarded as a foundation for mathematics, yet for better understanding we outline an interpretation in extended set theory in section B.10. We have not studied more "mathematical" models like PER models (based on partial equivalence relations on a simple set) or categorical models [44], but we remark that finding such models may be very difficult, because of the sheer strength of ATT, transcending ZFC set theory.

1.3 Type Theory and Set Theory

The foundation of mathematics is usually sought in axiomatic set theory: all mathematical objects are assumed to exist within a single universe of sets, where each set consists of other sets.

In a formalized language, it is convenient to have all objects classified into types, in order to avoid anomalies. In such a typed language, one cannot speak about an object without specifying its type, and one may only apply operations to objects of appropriate type. For example, it does not make sense to compare two objects of different types.

A *simple* type system consists of a number of primitive types together with a number of finitary type constructors. The class of simple type expressions can be algebraically defined prior to the further language definition. A typical language is Typed Lambda Calculus, with type constructors like function space, finite cartesian product, and disjoint sum, and possibly inductive or user-defined types. It has term rewrite (or reduction) rules operating on lambda terms, not on type expressions.

A *generalized* type system contains infinitary type constructors as well. As any concrete type expression is necessarily finite, type expressions have to be parameterized with object variables. The typical type constructor is the generalized product $\prod_{x:A} B_x$ for a type A and a type B_x for any object x of type A. The product is written as $\Pi(x\colon A :: B_x)$ in this thesis. Its inhabitants are tuples that contain, for any object $x\colon A$, an object $b_x\colon B_x$. Such a tuple is written as $(x :: b_x)$ in this thesis.

Due to the generalization, object expressions may appear within type expressions. Thus, both have to be defined simultaneously, and rewriting may affect type expressions too. Type correctness (validity) and equivalence of expressions are usually inductively defined as meta-predicates (also called *judgements*) on contexts, object and type expressions. The resulting system, consisting of expressions and judgements defined by derivation rules, is called a *type theory*.

We are interested in *constructive* type theories (CTT's), which are generalized type theories based on lambda calculus in such a way that any valid expression of some type provides a mathematical construction for that type. The typical example is a disjoint sum type $B_0 + B_1$; a closed expression of this type must reduce to a canonical form containing an expression either of type B_0 or type B_1. One can even have empty types, for which there are no closed expressions.

An interesting point is that constructive type theory immediately accommodates predicate calculus. When we identify any proposition with a type that contains all proofs of that proposition, all propositional connectives and quantifiers coincide with standard type constructors. Notably, the universal quantifier proposition $\forall x\colon A.P_x$ becomes the

generalized product $\Pi(x{:}\,A :: P_x)$. The proposition is true exactly when its proof type has an inhabitant, and any valid object expression of this type provides a proof of the proposition. This is called the *propositions-as-types* principle.

If one needs higher-order quantification, i.e., propositions that quantify over all subsets of a type, one has to make a formal distinction between propositions and data types. In fact, one assumes propositions to be given *a priori*, before the hierarchy of types has been generated. This is called *impredicativity*.

To justify constructive type theory, one may seek for a set-theoretic model. However, the basic rules of CTT are so fundamental, that one may just as well consider it to constitute an alternative foundation of mathematics, which replaces axiomatic set theory. We outline two models of set theory within extended type theory in section A.5 and A.6, and a model of type theory within an extended set theory in B.10.

1.4 Related efforts

Several mathematical languages based on type theory have been developed. Note that we do not regard a bare logical derivation system, like first or higher order logic, as a mathematical language, because it provides no notation for proofs other than as a sequence of statements.

N.G. de Bruijn developed *Automath* (in many variants) [11] exactly to provide an automatically verifiable notation for definitions and proofs in any logical calculus. It has only one principle of type construction (namely generalized product), which suffices for allowing the user to axiomatically introduce any type, object, or proof constructor he would need as a so-called "primitive notion". The necessity to write down all parameters of each constructor made Automath rather unwieldy to use. The *Mathematical Vernacular* (MV) [12] was introduced to overcome this: it allowed more syntactic freedom and the possibility to omit parts of a construction. This inhibits automatic verification.

P. Martin-Löf formulated his *Intuitionistic Type Theory* (ITT) [56] in order to explicate the basic principles of intuitionistic reasoning. Its basic type structure is very much like Automath, but it includes a number of construction principles (including inductive types) that provide a sufficient logical foundation for many purposes. It does not have impredicative propositions, as this is contrary to intuitionistic philosophy. The way ITT treats the equality predicate, internalizing it by means of "equality types" (sometimes called "identity types"), generates some anomalies in the type structure. Because of this, and the omission of some type parameters, type correctness (validity) of expressions may itself require a non-trivial proof.

R.L. Constable's *Nuprl* [18] is an interactive computer implementation of a variant of ITT. It assists the user in finding valid expressions, by following the approach introduced by the automated programming logic *Edinburgh LCF* (Logic of Computable Functions) [35, 70] to employ a functional programming language, the "meta-language" (ML), which is offered to the user who may call on predefined or user-defined "tactics" that perform a goal-directed search for valid expressions. (This special-purpose language ML has developed into the general-purpose language Standard ML, SML.) Nuprl provides a notation for the search process, recursively listing all goals and subgoals.

Th. Coquand's *Calculus of Constructions* (CC) [21] is a type theory based on impredicative quantification and has been implemented in LCF-style, too. Type constructors as used in typed lambda calculus can be defined using impredicative quantification, but no induction rule can be derived inside the calculus. C. Paulin-Mohring extended CC with embedded principles for inductive types [73, 68], which were implemented in the system Coq.

A more direct style of interactive proof editing was designed by Th. Coquand and B. Nordström and implemented in Göteborg as the *ALF* proof editor [50]. Being based on ITT with inductive definitions, it allows direct manipulation through a multiple-window presentation of the current state of the proof object, which may contain "placeholders" for incomplete parts. There are windows for the current theory, the proof under construction, the current list of placeholders (or goals) with their type and context, and equational constraints on the placeholders. It features a nice syntax for recursive definition through pattern matching, described in [23].

M. Sintzoff's calculus *DEVA* [78, 85] comes closest to our goal, because of the much greater attention it pays to the readability of the resulting proof and object expressions. It offers more structuring primitives, such as a kind of labeled records with dependent field types. DEVA distinguishes between explicitly and implicitly valid expressions. The former are verifyably correct proof or object constructions, the latter "amount to developments with missing parts, e.g. incomplete proofs and tactics. They are characterized by the undecidable existence of an explication, which completes the missing parts and yields a valid expression."[1] These explications go further than the missing proofs of De Bruijn's MV, but do not give the full power of ML-tactics.

1.5 Relational calculus

Now and then we use relational notation for easy expression, and sometimes proof, of properties. Such notations are being developed by a group around Backhouse into a calculational method for deriving programs from specifications [1]. There is an essential difference between this use of relations and ours: the relational calculus employs relations to model (possibly nondeterministic) input-output behavior, making much use of relation composition, while we use relations to establish relationships between functions and other objects, using arrow composition but hardly ever relation composition. A study of inductive properties in relational calculus is given in [8].

1.6 ADAM's Type Theory

The language *ADAM* is based on a type theory, ATT, that combines the features of Martin-Löf's ITT and Coquand's CC. Thus, it has a set of basic type constructors combined with impredicative propositions, and almost all typed lambda calculi appear as subsystems. There is, however, one gap that hinders the coding of arbitrary mathematical proofs: given a constructive proof that a predicate has a *unique* solution, one

[1] Weber in [84]

cannot obtain within the calculus a term denoting that solution. To mend this, we add a stronger elimination rule for the existential quantifier, described in appendix C.

We have the following construction principles and axioms:

1. Generalized products (Π). These subsume the function space constructor.
2. Generalized sums (Σ).
3. Finite types $(0, 1, 2, \ldots)$. Combined with generalized products and sums they give finite products and sums. Many recursion constructs can already be expressed using only these and function space, but extra equational calculus is needed to express their properties.
4. A hierarchy of universes ($\mathbf{Type}_i$).
5. Impredicative propositions (**Prop**), turning $\mathbf{Type}_i$ into a *topos* [46] and yielding higher order logic.
6. Equality types, i.e., an internal equality predicate ($x =_A y$).
7. Strong existential elimination ($\exists$_elim or the description operator ι). This is our new extension, see appendix C.
8. Infinity (ω). Together with the preceding principles it allows us to construct a representation for all inductive types, as we will see in chapter 8.
9. Axiom of choice, needed because propositions are distinguished from types.
10. Finally, one may also think of adding classical propositions, for which the principle of *reductio ad absurdum* holds.

The exact rules are listed in appendix B.

1.7 Aspects of induction and recursion

In philosophy, induction stands often for the process of discovery of a general statement out of some particular cases: "In all the umpty cases we encountered we found that statement P held, which induces us to suppose that P holds in all cases." This is not what we mean by induction in this thesis.

In mathematics, induction may be understood as the production of an infinite set of things through iterated application of a fixed set of rules. One distinguishes between inductive definition and inductive proof.

Inductive definition signifies the definition of a set as the totality of all objects produced through iterated application of a fixed set of production rules.

Inductive proof signifies proving a general statement by giving proof steps that incrementally produce proofs for all individual instantiations of the statement.

Recursion stands basically for the (re-)occurrence of an object within a description of that very object. Such a description is not necessarily a valid definition of the object: it may be seen as an equation $x = f(x)$ which can have either no, a single, or multiple

solutions. Yet under suitable restrictions recursive equations have unique solutions, so that they may be used for definition. Alternatively, one can use a complete partial order so that suitable recursive equations have unique least solutions. In most applications, the object x under definition is itself a function.

The notions of induction and recursion overlap. On the one hand, any inductive type definition can be written as a recursive type equation; this is in fact what happens in most programming languages that allow such types. Inductive proofs can, within a suitable calculus with dependent types, be written as recursive dependent function definitions. On the other hand, a valid recursive function definition can be reduced to an inductive predicate definition (as a kind of set) together with an inductive proof that this predicate constitutes a function.

The phrase recursive is often taken to denote *effectively computable*. The branch of mathematics called (classical) *recursion theory* [65] deals with hierarchies of computable functions on natural numbers, and studies their complexity. Computational complexity falls outside the scope of this thesis.

1.8 Frameworks for studying induction

There are several quite different notions of inductive sets; there are many ways inductive types can be described; and there are many settings, languages, or frameworks in which one may introduce inductive types. A brief survey follows.

1.8.1 Set theory. The foundational justification for the use of induction may be found in set theory (using for example the Zermelo-Fraenkel axioms of powersets, infinity, union etc.). The simplest interpretation of an inductive set definition is that it denotes the intersection of all sets that are closed under the rules of the definition. It follows immediately that proof by induction is valid for this set, but the construction is only welldefined if one has already some domain that is closed under the rules. Otherwise, one can iterate application of the rules to get a possibly transfinite series of sets, and take its limit. Under a certain restriction this limitset is closed under the rules indeed.

1.8.2 Type theory. An alternative foundation for mathematics is provided by Constructive Type Theory in the sense of Martin-Löf [56], which is a generalization of typed lambda calculus. In these systems a principle of inductive (data-)type construction can be included as basic. Extensive use is made of dependent types; in particular the type of the result of a recursive function may usually depend on its argument value. As types can represent propositions, the recursion axiom can be used for inductive proofs as well. Thus, while classically recursion is reduced to induction, type theory reduces induction to recursion.

An advantage of dependent types is that one can easily handle constructors that have an infinite number of arguments, as opposed over the ordinary use of types in, for example, purely finitary algebraic theories.

N.P. Mendler introduced [59] an alternative recursion construct that employed a quantification over types and a subset relation on types. It is not obvious how this construct relates to the ordinary ones. We will use the *naturality property* of polymorphic

objects to show that Mendler's construct has in fact the same power as the ordinary (dependent) recursion rule, at least when we replace the subset relation by explicit mappings. (See section 6.3.)

1.8.3 Category theory. Category theory captures inductive types in a particularly simple axiom about *initial algebras*. No recursive functions with dependent types are used nor is there an induction axiom, but there is a uniqueness condition from which these may be derived. Categorical notions and the initial algebra axiom can easily be put in a type-theoretical context. In fact, the general formulation of several alternative recursion axioms, too, is most easily expressed when using categorical notions. The categorical notions do also allow a very simple generalization to *mutual* and *parametrized inductive definitions*, for each construct. There are, however, other parametrized recursion constructs possible that are sometimes easier to use.

1.8.4 Impredicative type theories. Ordinary Constructive Type Theory does not allow to form types by quantification over the class of all types, as such a principle is intuitively not well-founded, and indeed inconsistent with some other constructs of type theory. However, such impredicative quantification can be added to either simple typed lambda calculus, resulting in polymorphic lambda calculus, or to a calculus with dependent types. This is done in the *Calculus of Constructions* of Coquand [21]. It allows the construction of inductive types for which there is a recursion construct, without using extra axioms. Unfortunately one cannot derive an induction rule inside the calculus, although it may be possible to prove outside the calculus, using a generalized naturality theorem, that the induction principle does hold. One might just assume a primitive induction axiom. When *subtypes* are available one is indeed able to construct an inductive type that has an induction rule. (See subsection 10.1.1.)

1.9 Our treatment of induction and recursion

In this thesis, we shall describe inductive types as they appear or might appear in various typed languages. Using *ADAM* all the time, we start with the traditional description of inductive sets by means of well-founded relations. Then we move to the categorical framework, which will be our main tool to bring various induction principles under a common denominator.

Our treatment is separated into construction principles for inductive types, and recursion principles over inductive types. The former, discussed in chapter 5, describe for which forms of algebra signature an initial algebra does exist. The latter, discussed in chapter 6, describe the forms which a total function definition using structural recursion over an inductive type may take. These rules suggest possible language rules for including inductive types in other constructive languages. Any such rule may be taken:

- either in its full generality, if the language includes all primitives that we use in the formulation of the rule,
- or in a more restricted form. For example, instead of a generalized product $\Pi(x\colon A ::$

B_x) one might allow only finite products $B_0 \times B_1$, parametrized types B^A, and combinations of these.

In chapter 8 we construct algebras in *ADAM* that satisfy the given induction and recursion rules. This proves the relative consistency with respect to ATT of these rules. It also proves the relative consistency of other calculi with inductive types that are directly embedded in ATT, like ITT and CC.

1.10 Other kinds of inductive types

Apart from inductive sets or types as mentioned in section 1.7, there are other kinds of inductive type definitions to which we shall give some attention.

1.10.1 Co-induction. In chapter 7, we see how the categorical description of inductive types can be dualized to describe *final coalgebras*, also called *co-inductive datatypes*. These model tree structures that may be infinitely deep, while staying in a pure setting that contains only total functions and totally defined objects. All recursion constructs can be dualized too, provided the usage of dependent functions is removed. One has to add a uniqueness condition in order to preserve completeness. The set interpretation of these objects is not evident, as infinitely deep sets conflict with the foundation axiom of standard set theory, but they may interpreted as graphs (section 8.2).

1.10.2 Domain theory. An entirely distinct notion of recursion is used in programming languages. Here it is often allowed to define types and objects in terms of themselves, without significant restrictions. This may result in partial or infinite objects, where it is effectively undecidable whether some part of an object is defined or not. Domain theories, such as the theory of Complete Partial Orders, were developed to give meaning (semantics) to such recursive definitions.

In chapter 9, we will describe several rules for reasoning about partial objects. We show how partial or infinite objects can be modeled by co-inductive datatypes, and how lazy recursive object definitions, with possibly nonterminating parts, can be interpreted in this model.

1.10.3 Inductive universe formation. The intuitive justification for the set-theoretical axioms is in fact an extraordinary kind of inductive set definition itself: one where a big set is generated such that each generated element is associated with a set itself, and where the production rules may use this associated set. In section 10.3, we suggest that existence of such big sets may be presented as a general principle, and name it *inductive universe formation*.

1.11 Original contributions

The main contribution of this thesis consists first of the presentation of an alternative view on the development of formal mathematical language, secondly of the description

and generalization of principles of inductive types in constructive languages, within a coherent framework.

In the course of this work, we presented a number of constructions and proofs. We think the following ones are minor contributions of this thesis:

- The introduction of strong existential elimination into constructive type theory (appendix C)
- The equivalence proof between non-dependent Mendler recursion and initiality (theorem 6.4), using naturality
- The generalized formulation of liberal mutual recursion (paragraph 6.4.3)
- The dualization of algebras with equations (section 7.4)
- The proof that Kerkhoff's initial algebra construction can be dualized (theorem 8.3)
- The construction of recursive cpo's by means of co-induction (paragraph 9.2.3 and section 9.3), and the interpretation of recursive object definitions within this representation (section 9.4)
- The inductive model of Zermelo-Fraenkel set theory in type theory (section A.6)
- The derivation of dinaturality from naturality (section D.6)

Furthermore, we have obtained a few not very remarkable results, for which we yet do not know whether they are known in the literature. These are:

- The relation between monads and initial algebras (theorem 4.6)
- The model of set theory within type theory by means of directed graphs (section A.5)