Inductive types in constructive languages

Bruin, Peter Johan de

Chapter 11

Reflections and conclusion

In this thesis, we played and experimented with language notations, language definition, and constructive type theories, employing these in studying abstract formulations of principles for inductive types, and the relationships between these. It was not our primary aim to solve specific problems, but rather to unify different approaches and to obtain an overall perspective on them. Looking back we can make a number of reflections on the areas mentioned.

11.1 Mathematical language

We developed the language *ADAM* as a medium to express principles of inductive types. We want to make the following remarks.

11.1.1 Ambiguity. One of the characteristics of *ADAM* is the great amount of ambiguity that we allow in defining and extending the language. We found this comfortable in shaping notations and identifiers that are easy to use, but it may make automatic checking of concrete text difficult or unfeasible. However, actual writing in a formal calculus usually takes place through interactive proof editing, where the author can immediately indicate how to resolve any ambiguities, and proofs are stored in a format that represents the internal structure rather than the concrete appearance. Thus, ambiguity is not insurmountable, but it requires attention of the author not to obscure his text.

11.1.2 Generalized typing. *ADAM* is based on constructive type theory. The availability of generalized type constructors made it possible to treat many constructive calculi as sublanguages, and allowed a unified treatment of parametrization and finite and infinite products. Whether it is desirable or necessary to have a constructive type theory as the logical foundation of the language remains to be seen; we discuss this in section 11.2.

11.1.3 Proof notation. We did not define a formal proof notation, but it is clear that it should be a structured, readable representation of natural deduction style proofs, without the obligation to write down all intermediate results. The very terse proof

representation of pure type theory is generally too unwieldy to read, but may be held available to be used for small, almost evident proofs and for proofs that the reader is expected to skip.

Sometimes a linear proof style is convenient. Such a style can very well be embedded within a natural deduction framework, but it can never replace it fully. More remarks on proof notation appear in section 11.4.

11.1.4 Refinement and scope rules. Creative thought has to be given to the subject of scope rules. Often, during the construction of an object or proof, one makes definitions that one would like to extend beyond the current (sub-)proof. This conflicts with the scope rules as used in any modern programming language. Linear proofs have even more difficulty with this (see e.g. U' on page 77), because a definition made within a line of a linear proof would by ordinary rules not even extend to the following lines that lay outside the local expression.

Somewhat related is the representation of stepwise refinement. Part of a construction may be left open to be filled in later on, perhaps guided by side conditions on the construction. In appendix C we experiment a bit with "goal variables" to fill temporary gaps, which are given a value further on in the proof. These are not to be confused with the "place holders" that may be used in interactive editing [50] to temporally hold open places, for these disappear from the proof when they are filled in. Again there are scope problems, for it is not evident which identifiers that were visible at the open place may be used in its refinement.

11.1.5 Variable abstraction. We introduced a double-colon notation to be used both for simple variable abstraction $(x :: b_x)$, quantification $\forall x{:}\, A :: P_x$, families $(x{:}\, A :: b_x)$, simple case distinction on an enumerated type $(\mathsf{false} :: b_0 \mid \mathsf{true} :: b_1)$, and pattern matching $(\Box :: b \mid (x, y) +\!\!< z :: c_{xyz})$. We found it comfortable to work with and clearer than a little dot (as in $\forall x{:}\, A.P_x$) when the declarations $x{:}\, A$ take up a bit more space, especially as we can extend these with propositional assumptions and local definitions. It extends nicely to pattern matching, unlike the dot or bracket abstraction $[x]b_x$. We liked the equivalence between finite tuples $(t_0, t_1){:}\, T_0 \times T_1$ and abstractions $(i :: t_i){:}\, \Pi(i{:}\, 2 :: T_i)$.

11.2 Constructive Type Theory

11.2.1 Objections. We have used type theory as the mathematical foundation of our research. There are some problems connected with this. The first is the gap between a single-valued predicate and a term denoting the same object. We had to introduce some extra machinery to bridge it (appendix C). It arises as soon as propositions are distinguished from data types. Original Martin-Löf type theory did not make this distinction, but it is needed for higher order quantification. Our solution was satisfactory, but we had to give up the property that any closed expression of some type is reducible to head canonical form for that type, for reasons discussed in subsection C.3.2. Taken together, it removed part of the original simplicity of the propositions-as-types idea.

Secondly, the representation in type theory of equality proofs and type conversion either is rather clumsy, or just omitted from proof terms.

Thirdly, generalized type theory requires parameters to be used for instantiating polymorphic objects and for supplying proofs to operations that have conditions on their arguments. If we understand the meaning of a term to be its computational content, these parameters are superfluous and make object expressions unnecessarily complicated.

11.2.2 Universes. In this thesis, we assumed a hierarchy of universes $\mathbf{Type}_i$, but usually we did not specify in which universe we worked. Most developments could be given in any universe, and it would be desirable if the calculus supported a formalization of this, by means of some kind of "universe parameters". For example, the description of category theory should be parametrized with the universe from which the classes of objects and arrows may be chosen. Next, the theory may be applied to the big category of categories itself by instantiating it to a higher universe.

Rather than having a fixed hierarchy, universes might be formed *inside* the calculus by specifying their basic types and type constructors. The latter may either be chosen from a fixed set, or perhaps, as suggested in section 10.3, be user-defined.

A simpler and easily realisable solution is to allow the formation of the *parametrized universe* $\mathbf{Type}(\Upsilon)$ of all types generated from a family of basic types Υ. Then one could define $\mathbf{Type}_0 := \mathbf{Type}\langle\rangle$; $\mathbf{Type}_1 := \mathbf{Type}\langle\mathbf{Type}_0\rangle$, etc.

11.2.3 Alternatives. When selecting a foundation for mathematical language, I would make the following observations.

- Do not use proof information in terms. This gives unnecessary overhead and is counterintuitive for most mathematicians.
- Use classical logic by default, for most mathematicians do not care about constructivism. Constructive arguments may be specially distinguished, if needed.
- An interesting simple type theory is given by Lambek and Scott [46, p. 128]. Its class of types contains only the singleton type 1, binary products $A \times B$, infinity $\mathbb{N}$, powertypes $\mathcal{P}A$, and a type of propositions Ω.

11.3 Language definition mechanism

We used a form of two-level grammar, or Definite Clause Grammar with equations, to define part of our language *ADAM*. We find it both very elegant and powerful, as it subsumes Horn clause logic. It is really a form of logic programming with equations, but note that we regard predicates as special syntax classes rather than translating syntax classes into predicates on strings of characters, as is more common. A lot of research is going on in this area, see e.g. [24].

The mechanism can be used to define both the basic foundational theory and the concrete language with its semantics, but also search strategies for finding missing parts of proofs. It is possible that a (prototype) implementation of a language be automatically generated from the language definition, provided that the definition is set up with executability in mind. One ingredient of the definition mechanism we did not touch upon is the following.

11.3.1 Proving grammar properties. We defined the basic theory around a predicate, '$\Gamma \vdash t{:}\,T$', and the concrete language around classes like '$\mathit{Term}(\Gamma, \gamma, t, T)$'. We claimed the definition to be such that the following holds:

$$\text{Whenever one has } \mathit{Term}_{\Gamma,\gamma}(t, T), \text{ then } \Gamma \vdash t{:}\,T \ ,$$

yet we did not prove this. There is need for a formal notation for stating and proving such grammar properties, for example by checking that each production rule for *Term* corresponds to one or a few rules for $\vdash$.

Besides simple implications, one has to check well-definedness of syntactic operations, like:

$$\text{For any } \mathit{Term}\, t,\ \mathit{Subst}\, \phi, \text{ one has } \mathit{Term}\, t[\phi] \ .$$

This involves an induction on the structure of terms. Existential properties "For all x, there is a y with $p(x, y)$" can be eliminated (as is often done in automatic theorem proving) by introducing an additional syntactic operation: "For all x, $p(x, f(x))$". This transformation is called a "Skolemization". Elementary automatic theorem proving can probably check all grammar properties we need, when we provide a list of them and indicate on which variables to perform induction.

11.4 Proofs and proof notation

11.4.1 What is a proof? A (formal) proof of a statement in the basic theory is normally its derivation tree. There is no necessity to encode this tree as an object in the theory. If we have developed a mathematical language and verified that a proof formulated in this language guarantees derivability of the statement in the basic theory, then the derivation tree of such a proof can be accepted as a formal proof indeed. The textual representation of such a proof can only be accepted when there is some reasonable upper bound on the amount of computation needed to verify it.

11.4.2 Modularization. The basic theory should be embedded within a logical framework [42] for the modularization and parametrization of theories. The framework may also provide facilities for information hiding, like "Abstract Data Types".

11.4.3 Proof format. Leslie Lamport [47] has designed a format for "Structured Proofs" in natural deduction style, featuring a neat numbering scheme for referring to proof steps and assumptions, and allowing linear proofs where appropriate. Such a format may very well be used as a standard format for proofs of theorems.

11.4.4 Local structure. Proofs for distinct steps of a deduction could be given by means of an expression that gives complete combinatorial information of all derivation steps and required facts, but normally one would be satisfied with giving just hints. Checking these hints would require some proof search, which can be defined through logic programming. It may be useful for some facts and assumptions to be marked as "active", meaning that they may be used without being hinted at.

11.4.5 Equality proofs. Suppose we have (a reference to) a proof p proving an equality $a = b$ (or a sequence of such proofs). A proof of $t_a = t_b$, where t_x may be a complicated term, say $f(g(x), c)$, has a number of intermediate steps like $g(a) = g(b)$, derived by extensionality of g. A natural notation for the full proof might be to insert p for variable x in t_x, properly marked, e.g. as

$$\%f(g(\cdot\, p\, \cdot), c)\ .$$

Here, '%' marks the start of the extensionality proof format, and '$(\cdot\ \ \cdot)$' marks the insertion of ordinary proof notation.

Essentially the same format can be used for naturality proofs: given a polymorphic term

$$x\colon S[\alpha] \vdash t[x]\colon T[\alpha]$$

and a relation $R\colon A \sim B$, the extended relation (section D.3) might be noted $\%T(\cdot\, R\, \cdot)$, and given a proof p proving $(a, b) \in \%S(\cdot\, R\, \cdot)$, we would have by naturality (theorem D.1):

$$\%t(\cdot\, p\, \cdot) \text{ proving } (t[a], t[b]) \in \%T(\cdot\, R\, \cdot)\ .$$

11.5 Inductive types

We started with inductive subset definitions given by means of rule sets, well-founded relations, or monotonic operators. We went on with the description of inductive types, defined by means of construction and elimination rules or as the initial object of the category of algebras of appropriate signature.

The initial algebra approach allowed us to separate the investigation of forms of inductive type definition from the study of forms of (structural) induction and recursion over an inductive type. For an overview of these forms, we refer to the conclusions of chapter 5 (page 73) and 6 (page 83). Here we list some of the decisions to be made when including inductive types in a language definition.

- Do the inductive types appear as fixed points of functors, or as initial algebras of appropriate signatures? The first option is a special case of the second.
- Are inductive types to be generated by
  1. providing actual parameters for one of the admissible forms of signature or functor, or
  2. writing down the desired signature or functor, which has to be matched against the admissible forms, or
  3. writing down the signature or functor according to (inductive) production rules (section 5.3)?
- For families of mutually inductive types, does each type from the family have a fixed (set of) constructors, or may one have "plain" algebra signatures where the codomain of a constructor may depend on its parameters?

Similarly, for recursive function definitions:

- Are they to be given by providing actual parameters for some recursor, or by writing down the desired recursion equations where the correctness checker has to match these to the admissible forms of recursion?
- Does one allow dependent recursion, and if not, does one include a uniqueness condition?
- Does one allow liberal mutual recursion, using either equality types or syntactic equality checks?

11.6 Directions for further research

The work and ideas presented in this thesis call for further research in the following directions.

1. **Language definition method.** Research on the method of using two-level grammar to define a semi-decidable language which is formally reduced to a foundational theory:
   (a) experiment with using the method on a small language;
   (b) formally define the method.
2. **Mathematical language.**
   (a) A suitable foundation remains to be established, especially when constructive and classical reasoning are to be combined in a single system.
   (b) Develop proof notations for *ADAM*, including e.g. readable notations for structuring the argumentation, modularization, easy use of equality, better scoping rules for definitions made within a proof.
   (c) Formally define a usable subset of *ADAM*.
   (d) Write a readable manual for *ADAM*.
3. **Inductive types.**
   (a) Analyze and compare how inductive types as included in current languages follow our schemes.
   (b) Define a good concrete notation for inductive types in a general language like *ADAM*.
4. **Relational notation.**
   (a) The naturality theorem for simple types should be generalized to dependent types. This will require a non-standard interpretation of generalized type expressions, where type variables are replaced by relations as in 11.4.5.
   (b) The same interpretation may be used for replacing type variables by categorical arrow sets. Thus, one type expression can define both the object and arrow part of a category.

(c) Internalize naturality: naturality of objects in *ADAM* should be available within *ADAM* itself.