AUTM92

Some remarks on typed lambda calculi with a view to the formalization of mathematics

by R.P. Nederpelt

Author's address: Dept. of Mathematics and Computing Science, Eindhoven University of Technology, P.O.box 513, 5600 MB Eindhoven, The Netherlands. Tel. (Netherl.) 40472718 or 40472750. Telefax: area code 40, nr 455925.

Paper type: short paper.

Topic: reasoning.

Track: science.

Keywords: typed lambda calculus, generalized (polymorphic) type assignment, formalization of mathematics, Automath.

Abstract

The use of a typed lambda calculus for formalizing mathematics can be "natural" in many respects. A few syntactical adjustments can even facilitate this correspondence. The paper discusses a possible approach to these matters and makes a few proposals regarding the syntax of typed lambda calculi to be used for this purpose.

Introduction

Typed lambda calculi may be used to formalize notions and constructions in mathematics (including logic and computing science). See e.g. Howard 1980, Martin-Löf 1975, Coquand 1985 or Reynolds 1974. An early example was de Bruijn's mathematical language Automath (see de Bruijn 1970, 1980), which is essentially a typed lambda calculus provided with an appropriate "sugaring". In Automath one may render complete mathematics books, as was shown e.g. in van Benthem Jutting 1977. The checking of the correctness of such a mathematical text amounts to a number of type-checks, which can be executed mechanically (i.e. by a computer).

In this paper we describe some of the essential ideas behind Automath, which language we present in the form of a typed lambda calculus (so without the mentioned sugaring) for the sake of clarity. In passing we make a number of proposals regarding typed lambda calculi, in order to increase their usefulness as a formal language for mathematics. Some of these proposals are hardly more than notational devices; others affect the basic operations in typed lambda calculi.

1. Types

Types in lambda calculus act as domains for variables: in the lambda term $\lambda x \in A . B$ the type A is given to all free variables x occurring in B. In this section we discuss these types.

1.1. Generalized types

A type can be very simple, establishing only the functional structure of a variable (e.g. $((\alpha \to \beta) \to \beta)$, α and β being "ground types"). When we desire to formalize mathematical texts by means of a typed lambda calculus, we need a richer kind of types. It turns out to be profitable to have types of the same strength as terms, as will become clear in the sequel. For the sake of syntactical unity and transparency, we may even agree upon:

Proposal Ia: We do not distinguish syntactically between terms and types.

1.2. Function types and type-valued functions

In a number of typed lambda calculi (see e.g. Martin-Löf 1975, Coquand 1985; cf. Hindley and Seldin 1986) function types $\Pi x \in A . B$ were introduced as types for functions from A to B. It is, however, rewarding to identify function types $\Pi x \in A . B$ with type-values functions $\lambda x \in A . B$, for the following reasons:

- function types possess the same binding structure as ordinary functions,
- the natural abstraction and application rules for function types are generally quite similar to the respective rules for functions.

In this view $\lambda x \in A . B$ represents:

1) a function, sending x to B, when x and B are on "element level";
2) either a function type or a type-valued function, when x has element level and B has "type level".

Hence:

Proposal Ib: We identify function types and type-valued functions.

1.3. Higher order functions

A further extension, also very useful in this respect, is to allow higher order functions. Then $\lambda x \in A . B$ represents:

3) a second order function when x has type level and B has element level (When adopting the formulae-as-types notion - see Howard 1980 and also §2 below - this can be a valuation, or a proof of a universal statement over predicates, as well);
4) the type of such a function, when both x and B have type level.

Therefore we suggest the following uniform extension:

Proposal Ic: In a lambda term $\lambda x \in A . B$ both x and B can have either element or type level.

1.4. Type construction

With these conventions we can introduce a very general type constructor, which we call Type, acting on every term in the typed lambda calculus which ends in a bound variable. This type constructor has the simple properties:

1) Type(x) is the given type of x (so when x is bound by the λ in $\lambda x \in A . \ldots$, then Type(x) = A);
2) $\mathrm{Type}(\lambda x \in A . B) = \lambda x \in A . \mathrm{Type}(B)$;
3) $\mathrm{Type}(A(B)) = (\mathrm{Type}(A))(B)$.

2. The coding of mathematics

We shall now discuss how to "translate" mathematical notions into typed lambda calculus. A translation of a mathematical text into a formal lambda term we call a coding. For convenience we use the following notation: when S is a piece of mathematical text, then $\overline{S}$ is its coded form.

2.1. Contexts

A context is a list of variable introductions and assumptions. A variable introduction (such as 'Let x be an element of A') is written '$x \in A$' in the context. Assumptions can have the same form when we adopt the formulae-as-types notion (as we will do). Then 'Assume B' can be rephrased as 'Let y be a proof of B', which is written '$y \in B$' in a context.

Note that both variable introductions and assumptions can be coded efficiently as the head of a lambda term: $\lambda x \in A . \ldots$, $\lambda y \in B . \ldots$, etc. A coded context then becomes a compound head of such a term: $\lambda x \in A . \lambda x \in B . \ldots .$

2.2. Logic

The formulae-as-types notion can be fully exploited in a typed lambda calculus: logical operators and quantors can be naturally coded, namely in such a manner that term construction in lambda calculus reflects deduction in logic. Example: a natural code for the implication $\alpha \rightarrow \beta$ is $\lambda x \in \alpha \ . \ \beta$. Then β-reduction mimics the *modus ponens rule*: when we have a proof p of $\alpha \rightarrow \beta$ (so $p \in \lambda x \in \alpha \ . \ \beta$) and a proof q of α (so $q \in \alpha$), then $p(q) \in (\lambda x \in \alpha \ . \ \beta)(q)$. Reduction of the right hand side of the latter formula yields $[q/x]\beta$, which is β, since we may assume that β does not contain free x's. Hence $p(q) \in \beta$, so we may conclude that we have a proof - viz. p(q) - of β. (See also de Bruijn 1980, Nederpelt 1980.)

2.3. Definitions

There is an obvious way to code definitions in typed lambda calculi. Let $x_1 \in S_1, \ \ldots \ , x_n \in S_n$ be a context and y:=T a definition in this context. Then the following term codes this definition:

$$(\lambda y \in (\lambda x_1 \in \overline{S}_1 \ . \ \ldots \ . \ \lambda x_n \in \overline{S}_n \ . \ C) \ . \ D)(\lambda x_1 \in \overline{S}_1 \ . \ \ldots \ . \ \lambda x_n \in \overline{S}_n \ . \ \overline{T}) \qquad (1).$$

Here C is (a term converting to) the type of $\overline{T}$, and D is the code of the text in which the definition may be used.

In the mathematical text, application of the definition y:=T amounts to:

1) specifying appropriate instances for the context variables, i.e. $V_1, \ \ldots \ , V_n$ with $V_1 \in S_1, \ V_2 \in [V_1/x_1]S_2, \ \ldots \ , V_n \in [V_1/x_1]\ldots[V_{n-1}/x_{n-1}]S_n$;
2) replacing y by $[V_1/x_1]\ldots[V_n/x_n]T$ (2).

On the other hand, we have an analogous procedure in the coded version. When variable y occurs free in D in the above term (1), where y is followed by the n arguments $\overline{V}_1, \ \ldots \ , \overline{V}_n$ corresponding with the instances $V_1, \ \ldots \ , V_n$ mentioned, then β-reduction of (1) results in the substitution of $(\lambda x_1 \in \overline{S}_1 \ . \ \ldots \ . \ \lambda x_n \in \overline{S}_n \ . \ \overline{T})$ for that y in D. Application of the definition now amounts to n β-reduction instances, induced by:

$$(\lambda x_1 \in \overline{S}_1 \ . \ \ldots \ . \ \lambda x_n \in \overline{S}_n \ . \ \overline{T})(\overline{V}_1)\ldots(\overline{V}_n),$$

yielding term $[\overline{V}_1/x_1]\ldots[\overline{V}_n/x_n]\overline{T}$, which corresponds with (2).

In lambda term (1), subterm D can in reality be a very long term. In fact, D contains the coded version of the complete text depending on the definition y:=T. A consequence is that definiendum y and definiens T become separated over quite a distance. Besides, the natural order of the development of a mathematical theory is disturbed. Term (1) contains the coded definition divided over the beginning and end of the term, while the rest of the theory finds its place in the middle. Since these matters occur repeatedly (see also the next subsection about theorems and proofs), the coding yields the mathematical theory in what might be described as a funnel form.

One may of course object that this is unimportant, since implementation can easily cope with these difficulties. On the other hand, there is an easy way out on the pure notational level:

Proposal IIa: Write arguments in front of the functions, so write (A)B instead of B(A) for 'B applied to A', and in particular write (C)($\lambda x \in A$. B) instead of ($\lambda x \in A$. B)(C).

2.4. Theorems and proofs

Following the formulae-as-types concept, one may also interpret (1) as: in the context $x_1 \in S_1$, ... , $x_n \in S_n$, it holds that T is a proof of the theorem U (where $\overline{U} = C$), and y is a name for this proof.

It will be clear that Proposal IIa has advantages here, too: proofs are coded adjacent to the codes of the theorems they prove, and the natural order of development of a text is again preserved.

2.5. Scope of variables

In the term $\lambda x \in A$. B, term B acts as scope for x: all free x's in B are bound by the x attached to the λ. Now A can be a rather long term (e.g. the code of a theorem), which makes it harder to "find" the scope of x. A minor change in notation can be of use:

Proposal IIb: Write the type in front of the λ, i.e. write Aλx . B instead of $\lambda x \in A$. B.

2.6. Term construction

In order to facilitate the "translation" (i.e. coding) of a mathematical text into a typed lambda calculus form, it may be profitable to split terms into certain "units". By unit we mean:

1) an argument for a function;

2) the head of a lambda term.

Proposal IIc: Write (A δ)B instead of (A)B (or B(A) in the usual notation; δ being a mark for 'argument'), and write (Aλx)B instead of Aλx . B (or $\lambda x \in A$. B in the usual notation); allow delta units (A δ) and lambda units (Aλx) as syntactical entities.

Now the definition y:=T becomes (when we assume for convenience that the context is empty):

($\overline{T}$ δ)(Cλy),

and proof T,together with theorem U proved by T,has the coding:

($\overline{T}$ δ)($\overline{U}$ λy).

In both cases we may suffice with a pair of units, acting as "double heads" for a term which is in process of being developed.

Note also that a variable introduction or an assumption can be coded as the lambda unit (Cλz), and a context as the lambda unit list ($C_1\lambda x_1$)...($C_n\lambda x_n$).

2.7. Reduction

When following the above described convention regarding the coding of definitions and of theorems with their proofs, we feel a need for a different kind of (β -)reduction. For, applying a definition usually occurs only locally: when we desire to apply definition y:=T, then there is one occurrence of y that we wish to replace by T. This remark also holds for theorems. Besides, we do not want to dispose of the definition or the theorem after use, because we may need it again. (See also de Bruijn 1987.) The usual concept of β -reduction, however, is not suited to this purpose: when reducing $(\lambda x \in A \ . \ B)(C)$ one should replace by C all free occurrences of x in B, so to say in one go, and strike out the "redex part" $(\lambda x \in A \ . \ \ldots)(C)$.

Therefore we suggest:

Proposal III: Reduction is executed locally: substitutions take place for only one variable at a time; redex units $(A\ \delta)$ and $(B\ \lambda x)$ do not vanish in reduction.

When accepting this proposal, we have to extend our notion of reduction a little bit. For example, consider (in the usual notation) the term $(\lambda y \in E \ . \ \lambda x \in A \ . \ C)(D)(B)$. When the redex part $(\lambda y \in E \ . \ \ldots)(D)$ does not vanish in a reduction, then we will never attain the redex $(\lambda x \in A \ . \ [y/D]C)(B)$. A solution is to neglect "interfering" redex parts in a reduction:

Proposal III (second version): Reduction amounts to the following: when a free x is in the scope of the lambda unit $(B\ \lambda x)$ which "matches" with delta unit $(A\ \delta)$, then x may be replaced by A (with the proper variable renamings to avoid a clash of variables, as usual).

The mentioned matching of lambda units and delta units can easily be formalized.

2.8. Correctness

In checking type-correctness of a term, it is advisable to adopt a broader point of view than usual. The notion of type-correctness formalizes the "proper functionality" of a term: $(A\ \delta)B$ is only permitted as a subterm when B is some kind of "function" and A fits in the "domain" of that function. Cf. Hindley and Seldin 1985. Formally: when C codes that domain, then Type(A) should convert to C. (For details: see Nederpelt 1973.) Hence, to see whether type-correctness holds, we have to compare - i.e. establish convertibility between - pairs of terms, and this has to be done repeatedly.

When comparing two lambda terms, adopting the above described reduction convention, we have to ignore "dead" redex parts, i.e. matching units $(A\ \delta)$ and $(B\ \lambda x)$ for which there is no (more) free x in the scope of the λ mentioned. (In Nederpelt 1973 this is called β_2-reduction.) But one can also define a weeding function w, with the effect of removing all dead pairs of redex units from an expression.

in conversion.

2.9. Convertibility

There are two different options for defining convertibility of subterms B and C of a given term A: local reductions outside B and C (but inside A) may or may not have their effect on free variables in B or C during the conversion process. One of the advantages of the first, liberal option is that "global" definitions, to be found somewhere outside the subterms under consideration, may be applied inside a term during the conversion process. This is what we usually want in mathematics. Therefore we add:

Proposal IVb: When converting subterms of a given term A, one may also consider reductions in A with redex units outside these subterms.

2.10. The desire for internal definitions

When coding mathematics in the style described so far, one is obliged to write the same strings of units over and over again. This is already perceivable in term (1), in which the lambda string $(\overline{S}_1 \lambda x_1)\ldots(\overline{S}_n \lambda x_n)$ occurs twice. In practice these repetitions become not only boring, but even almost insurmountable.

We suggest it as a field for further investigations to incorporate into lambda calculus a nice "abbreviation device" for strings of units, i.e. a possibility for internal definitions of such strings. A useful attempt in this direction was made in Balsters 1986. The ideas exposed there may be elaborated in order to develop an extended typed lambda calculus which is also appropriate for "machine-oriented" formalization of mathematics. (A solution already existing is the "sugaring" present in some Automath languages; we suggest, however, to stay close to the typed lambda calculus described above, in order to keep the syntax as simple and uniform as possible.)

References

BALSTERS, H. [1986]. Lambda calculus extended with segments. Thesis, Eindhoven Univ. of Technol., Netherlands.

de BRUIJN, N.G. [1970]. The mathematical language Automath, its usage, and some of its extensions. SLNM 125, pp. 29-61.

de BRUIJN, N.G. [1980]. A survey of the project Automath. In To H.B. Curry, Essays on Logic, Lambda Calculus and Formalism. Hindley and Seldin (eds.), Academic Press, pp. 579-606.

de BRUIJN, N.G. [1987]. Generalizing Automath by means of a lambda-typed lambda calculus. To appear in Mathematical Logic and Theoretical Computer Science. Lopez-Escobar, Kusker and Smith (eds.)

COQUAND, Th. [1985]. Une théorie des constructions. Thesis, Univ. of Paris VII, France.

HINDLEY, J.R., SELDIN, J.P. [1986]. Introduction to Combinators and λ-Calculus. Cambr. Univ. Press.

HOWARD, W.A. [1980]. The formulae-as-types notion of construction. In To H.B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism. Hindley and Seldin (eds.), pp. 479-90.

van BENTHEM JUTTING, L.S. [1977]. Checking Landau's "Grundlagen" in the Automath system. Thesis, Eindhoven Univ. of Technol., Netherlands.

MARTIN-LÖF, P. [1975]. An intuitionistic theory of types: predicative part. In Logic Colloquium '73, H.E. Rose et al. (eds.). N-H, pp. 73-118.

NEDERPELT, R.P. [1973]. Strong normalization in a typed lambda calculus with lambda structured types. Thesis, Eindhoven Univ. of Technol., Netherlands.

NEDERPELT, R.P. [1980]. An approach to theorem proving on the basis of a typed lambda calculus. SLNCS 87, pp. 182-94.

REYNOLDS, J.C. [1974]. Towards a theory of type structure. SLNCS 19, pp. 408-25.