A plea for weaker frameworks

A plea for weaker frameworks

N.G. de Bruijn
Eindhoven University of Technology

Abstract

It is to be expected that logical frameworks will become more and more important in the near future, since they can set the stage for an integrated treatment of verification systems for large areas of the mathematical sciences (which may contain logic, mathematics, and mathematical constructions in general, such as computer software and even computer hardware). It seems that the moment has come to try to get to some kind of a unification of the various systems that have been proposed.

Over the years there has been the tendency to strengthen the frameworks by rules that enrich the notion of definitional equality, thus causing impurities in the backbones of those frameworks: the typed lambda calculi. In this paper a plea is made for the opposite direction: to expel those impurities from the framework, and to replace them by material in the books, where the role of definitional equality is taken over by (possibly strong) book equality.

1 Introduction

Verification systems

A verification system consists of

(i) a framework, to be called the *frame*, which defines how mathematical material (in the wide sense) can be written in the form of *books*, such that the correctness of those books is decidable by means of an algorithm (the *checker*),

(ii) a set of basic rules (*axioms*) that the user of the frame can proclaim in his books as a general basis for further work.

For several reasons (theoretical, practical, educational and sociological) it may be recommended to keep the frames as simple as possible and as weak as possible. A second recommendation is to prefer frames belonging to a hierarchy in which the various specima can be easily compared.

We shall restrict the discussion to frames that have typed lambda calculi as their backbone, an idea that is gaining field today, along with the rising popularity of the principle to treat proofs and similar constructions in exactly the same way as the more usual mathematical objects. The rules of the frame involve rules for handling lambda terms, lambda reductions, and rules about attaching a type to a term. The typing rules are strongly influenced by the notion of *definitional equality*. The latter notion is primarily based on the reductions of the lambda calculus.

The centerpiece of this paper is the typed lambda calculus $\Delta\Lambda$. It improves on the slightly simpler system Λ (Nederpelt) in a sense that is well suited for implementing definitions. Quite some attention will be given to relating members of the Automath family to $\Delta\Lambda$.

Verification systems which are not related to Automath in any way will not be discussed in this paper. Nevertheless the author believes that such systems could have similar profits from confrontation with $\Delta\Lambda$.

Strengthening the frame

Over the years there has been the tendency to put more and more power into frames, at the price of impurities of the typed lambda calculus. This already started with the early versions of Automath (AUT68 and AUT-QE, see [3], [6], [13]) which were very advanced for their time (around 1968) in the sense that the frames were light and that almost all logical and mathematical knowledge was to be developed in the books. Nevertheless there was the impurity of so-called type inclusion (see section 4). A proposal to get rid of it (see [11]) came too late for implementation in the Automath project).

Motives for strengthening

Let us list some of the motives one might consider for strengthening the frame.

(i) One has the idea that the rules of the frame can be handled automatically and that book material has to be produced essentially by hand. This explains that wherever one sees things that can be automated, one has the tendency to shift them from the book into the frame.

(ii) Mathematicians have been trained to base their work on as few axioms as possible, preferring strong axioms over lists of weaker ones. So much the nicer if one can shift axioms into the frame, where one does not see them any more!

(iii) Strengthening the frame can enlighten the burden of writing books.

(iv) It is natural to have the feeling that the way one thinks about mathematics and its implementation in a verification system should somehow be reflected in the frame.

In particular these points may mean that much of what is achieved by book equality (the kind of equality that is to be considered as a mathematical proposition and for which proofs have to be provided by the user of the system) can be shifted to definitional equality, which is based on reductions in the frame that can be handled automatically).

Candidates for admission into a frame

There are two kinds of candidates, innocent ones and others. The innocent ones are what one might call *syntactic sugar*. They deal with ways to abbreviate book material, usually in a way that corresponds to existing habits of mathematicians. One might decide to enrich the frame with rules for such forms of sugaring.

Under the heading of sugaring we have things like:

(i) Ways to handle finite strings of terms, and strings of applicators and abstractors (in the latter case we call them *telescopes*, see [17]) by means of a single identifier, just as if they were ordinary objects.

(ii) Ways to to omit pieces of input and output that can be retraced easily and uniquely by the checker.

(iii) In particular this may refer to ways to replace explicit proofs and similar constructions by hints.

Sugaring is innocent since it has no influence on the validity and the interpretation of the things we present in our verification system.

The other candidates are not necessarily innocent. They may stem from mathematical or logical insights, but once incorporated into the frame they may get applications for which they were not intended in the first place. We might even fear antinomies.

Unlike the situation in a law court, the accused is to be considered as guilty as long as its innocence has not been established. This may mean that strengthening the frame gives us quite some work, possibly of a model-theoretical nature.

The non-innocent ways to strengthen the frame all increase the number of valid definitional equalities and the number of valid typings. This does not necessarily mean increasing the number of derivable mathematical results, but it might. And some of these extensions (like (iv)) damage the purity of the typed lambda calculus backbone.

We mention

(iv) Rules for type inclusion, or, what amounts to the same thing, Π-rules (see section 4).

(v) The principle of *proof irrelevance* (cf. [13], [21]), declaring two proofs for one and the same proposition to be definitionally equal.

(vi) Shifting knowledge about natural numbers, induction and recursion into the frame.

A plea for weaker frames

When one uses arguments of economy in order to put more power into the frame, one forgets that it is neither very hard to put some automation in the task of writing books. We can invent abbreviational facilities that may result in automatic writing of parts of the books (parts that the user even does not need to see if he does not want to). In particular this may refer to handling trivial equalities like those we need when dealing with pairs and cartesian products. This need for automatic textwriting will turn up in many other situations too. It happens every now and then, when we want to economize on writing things that we consider as part of our subconcious thinking. It does not seem to be the right thing to enrich the frame at every occasion of that kind.

And needless to say the tendency to provide the frame with facilities that depend on local and personal preferences may lead to an undesirable variety of diverging frames.

Let us list a number of arguments for retreating to weaker frames.

(i) Weaker frames means fewer frames. We are still far from a general acceptance of typed lambda calulus verification systems in the scientific world, the mathematical world in particular. Maintaining a variety of competing but hardly compatible systems will not be very helpful in this respect. And it will neither be very helpful for establishing cooperation between the various groups handling verification systems.

(ii) Having to understand a verification system, one has to see the clear borderline between frame and books. The distinction is easy to understand if the frame contains everything that can not be treated in the books, and nothing else.

(iii) For many people a pure system of typed lambda calculus is already hard enough. Impurities make it harder.

(iv) As said before, the frame is not the only place where things can be done automatically. There are plenty of cases for automated or semi-automated book-writing anyway. Verification systems would hardly have a future if we would have to depend on doing all the tedious and dirty work by hand.

(v) There is no reason at all to let any kind of syntactic sugaring slip from the book into the frame. Sugaring is meant to make reading and writing easier, or better adapted to traditional notations, and that belongs entirely in the world of the book. The frame is supposed to be operated by machines which do not have the same ideas as humans if it comes to the question whether things are difficult or tedious to handle.

(vi) The pain of having to keep things book-equal that one would prefer to have definitionally equal can be somewhat relieved by the introduction of a third kind of equality: *strong book equality* (see section 6).

Hierarchy of frames

In section 1 it was recommended to limit the number of frames that we present to the world. But the world would certainly also appreciate seeing a clear hierarchical structure among the different frames we present. The hierarchical relation $F_1 \leq F_2$ should be the one that expresses that every book valid with respect to frame F_1 is also valid with respect to F_2.

The hierarchy need not necessarily be linear, but it would certainly be appreciated if the hierarchy contains a maximal element F_m with $F \leq F_m$ for all the F in the hierarchy.

The inequality $F_1 \leq F_2$ has to be kept more or less informal in the present discussion. After all, we have not given a mathematical definition of the notion "frame". We shall also use the notation $F_1 \leq F_2$ in cases where the notion of validity of a book with respect to a frame still needs some interpretation (like in the case of $\Delta\Lambda$, where the system itself does not say how a book has to be considered as a lambda term).

Implementability

An even more informal notion is the one of implementability. We write $F_1 \xrightarrow{\mathrm{i}} F_2$ if F_1 can be *implemented* in F_2. By this we mean that the essence of what F_1 can do for us can also be done in the (weaker) system F_2, provided that the books written under F_2 start with a number of suitable axioms, and that we take it for granted that some of the definitional equalities in F_1 are to be replaced by book equalities.

We should not forget that $\xrightarrow{i}$ indicates a rather weak form of implementation; one might rather call it *mimicking*. Nevertheless it might give an idea about the strength of various systems, both in a theoretical and a practical sense.

Purity of lambda calculi

We shall be informal here, *not* start by saying what a typed lambda calculus is. We just stress the essential items for purity.

(i) Definitional equality ($\stackrel{d}{=}$) should be defined by means of β-reduction only. It would probably do no harm to admit η-reduction too, but it seems hardly worth while. We do not mention α-reduction since its role can be reduced to the level of syntactic sugaring; one can assume that the frame deals with namefree lambda calculus.

(ii) We should require full unicity of types. So assuming the typing $P : Q$ and the validity of R, we have $P : R$ if and only if $Q \stackrel{d}{=} R$.

(iii) We should require that both abstraction and application commute with typing. So with the usual notation

$$\frac{\Gamma, (x : A) \ \vdash \ P(x) : Q(x)}{\Gamma \vdash \ [x : A]P(x) : [x : A]Q(x)},$$

$$\frac{\Gamma \vdash \ P : Q \quad \Gamma \vdash \ \langle a \rangle Q}{\Gamma \vdash \ \langle a \rangle P : \langle a \rangle Q}.$$

Here we followed the Automath notation for abstraction and application. Abstractors are written like $[x : A]$, where x is the bound variable and A its type, and are written in front of the term they act on. Applicators are also written in front: $\langle a \rangle P$ describes what is interpretated as a function value, where P is the function and a the argument. And, as usually, Γ stands for any context, and $\Gamma, (x : A)$ is its extension by means of the declaration of the typed variable x.

The above rule for application is not sufficient for getting all cases of $\Gamma \vdash \langle a \rangle Q$. We have to add the rule

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash \ Q \stackrel{d}{=} [x : A]R(x)}{\Gamma \vdash \ \langle a \rangle Q}.$$

2 A proposal for a hierarchy of frames

We shall discuss three frames, forming a linear hierarchy:

$$\text{AUT-QE-NTI} \;\leq\; \Delta\Lambda \;\leq\; \Lambda\Sigma.$$

The first one, AUT-QE-NTI , was designed in 1978 as an Automath language without impurities. The other members of the Automath family can be implemented in it (in the sense of section 1,subsection Implementability (see [11])). In particular we have

$$\text{PAL} \;\leq\; \text{AUT68} \;\leq\; \text{AUT-QE} \xrightarrow{\text{i}} \text{AUT-QE-NTI},$$
$$\text{AUT}\Pi \xrightarrow{\text{i}} \text{AUT-QE-NTI}.$$

For PAL, AUT68 and AUT-QE we refer to [3], [6], [13].

$\Delta\Lambda$ was designed in 1985. It is very close to the system Λ for which Nederpelt proved strong normalization ([20]). That Λ was a reformulation of the system AUT-SL ([4]) which presented complete Automath books (with extremely liberal abstraction rules) in the form of a single line. In order to do that, all definitions of the book had to be eliminated first, and therefore AUT-SL had theoretical interest only. With $\Delta\Lambda$, however, this elimination of definitions is no longer necessary (see section 5).

$\Delta\Lambda$ is strong enough for implementation (in the sense of $\xrightarrow{\text{i}}$) of Barendregt's Generalized type systems (see [2]), which in particular contains the Coquand-Huet theory of constructions. It may be interesting to compare Barendregt's [2] with the much older note [7]. The essential differences are: (i) in [7] type inclusion (cf. section 4 below) was restricted to degree 1, and (ii) in [2] there is no place for the typical Automath feature of admitting more contexts than those for which it is allowed to abstract from.

In section 7 we shall devote some attention to the system $\Lambda\Sigma$, which is intended as a very substantial extension of typed lambda calculus. But at present it should be said that the relation $\Delta\Lambda \leq \Lambda\Sigma$ is rather a program for development of $\Lambda\Sigma$ than a fact.

3 The typed lambda calculus $\Delta\Lambda$

Syntax

The difference between $\Delta\Lambda$ and Nederpelt's Λ lies in the notion of correctness, not in the syntax.

Let us agree that what we really mean is the namefree version, and that the use of names of variables is syntactic sugar (both in the description as in the use of the syntax). A complete description was given in [15] by means of trees with reference arrows (arrows from the variables at end-points of the tree to the corresponding lambdas). That reference arrow system seems to be the easiest way to describe syntax and language theory. At the stage of implementation in a checker one might decide how to implement those arrows. One might use depth references, but if the implementation is in terms of a programming language with a pointer mechanism, one might use those pointers instead. For input and output of mathematical texts, however, there is much to be said for the syntactic sugar of named variables.

In order to avoid repeating the complete formal description given in [15] we give an informal one here, with the use of names of variables. In the first round of syntax description we formulate

(i) τ is a term.
(ii) A variable is a term.
(iii) If x is a variable, if P and Q are terms, where P does not contain x, then $[x : P]Q$ is a term.
(iv) If P and Q are terms, then $\langle P\rangle Q$ is a term.

In the second round we add the restriction that a term should not contain any free variables.

Every bound variable in a term has a unique type, obtained by agreeing that in (iii) all x occurring in Q get type P.

Every term has a terminating symbol on the extreme right. If that symbol is a τ we say that the term has degree 1, and we do not define the type of the term. If the terminating symbol of the term E is a variable x, we get the *type* of E (notation $\text{typ}(E)$) if we simply replace that terminal occurrence of x by the type of x.

And we define the *degree* of a term E recursively by the rule $\text{degree}(E) = \text{degree}(\text{typ}(E)) + 1$.

Correctness

Correctness of terms depends on the notions of typing and definitional equality.

Definitional equality in $\Delta\Lambda$ is defined by means of *mini-reductions*: local β-reductions and so-called AT-removals, instead of by the global β-reduction used in Λ. Local β-reduction means that some sub-term $\langle R\rangle[x : P]Q$ is transformed into $\langle R\rangle[x : P]Q^*$, where Q^* is obtained from Q upon replacing a single one of the occurrences of x by R. It can of course be done only if there is at least one occurrence of x in Q. If there are no occurrences of x in Q we allow AT-removal: we remove the AT-pair $\langle R\rangle[x : P]$, so $\langle R\rangle[x : P]Q$ turns into Q. (A few words to explain the "AT": in the metalanguage used in [15] the "A" refers to applicators $\langle\ \rangle$, and the "T" to typed abstractors $[\,:\,]$.)

In the usual way we get the notion $\stackrel{\mathrm{d}}{=}$ of definitional equality by reflexive transitive symmetric closure of the set of mini-reductions.

For practical reasons it can be recommended to extend the set of mini-reductions by basing them on AT-*couples* instead of AT-pairs (see [15], section 4.3). As an example we mention that in $\langle R\rangle\langle Q\rangle\langle P\rangle[x:A][y:B][z:C]$ the $\langle R\rangle$ and $[z:C]$ form an AT-couple. Extending the notion of mini-reductions to AT-couples does not alter the notion of $\stackrel{\mathrm{d}}{=}$. This easily follows from the fact that applicators may jump to the right over an AT-pair, since

$$\langle Q\rangle\langle P\rangle[x:A]W \stackrel{\mathrm{d}}{=} \langle P\rangle[x:A]\langle Q\rangle W.$$

As we start from the term om the left, the Q does not contain x. Therefore definitional equality follows by full β-reduction applied to both sides.

Having the notion $\stackrel{\mathrm{d}}{=}$, we can get to the matter of correctness.

The essential point in the definition of correctness is in the situation of subterms of the form $\langle R\rangle P$. The usual conditions in typed lambda calculi, in particular in Nederpelt's Λ, are

(i) R and P are both correct,
(ii) in the finite sequence P, typ(P), typ(typ(P)), ...there is a term that is definitionally equal to a term of the form $[x:S]W$, with typ$(R) \stackrel{\mathrm{d}}{=} S$.

If a term is correct in the sense of Λ then it is also correct in the sense of $\Delta\Lambda$, but the converse is not true. $\Delta\Lambda$ is more permissive concerning the applications $\langle R\rangle P$. In $\Delta\Lambda$ it is no longer required that P is correct all by itself: the correctness check may make use of the fact that $\langle R\rangle$ is in front of this P. An example is

$$[W:[t:\tau]\tau]\ [p:\tau]\ \langle p\rangle\ [u:\tau]\ [s:\langle p\rangle W]\ \langle s\rangle\ [x:\langle u\rangle W]\ \tau.$$

Correctness of the $\langle s\rangle[x:\langle u\rangle W]\tau$ at the end requires that $\langle p\rangle W$, the type of s, is definitionally equal to $\langle u\rangle W$. According to the rules of $\Delta\Lambda$ this is the case indeed, because of the pair $\langle p\rangle[u:\tau]$ earlier in the formula. Putting

$$R=p,\quad P=[u:\tau]\ [s:\langle p\rangle W]\ \langle s\rangle\ [x:\langle u\rangle W]\ \tau,$$

we see that R is correct in the context $[W:[t:\tau]\tau][p:\tau]$, but that P is not. So in that context $\langle R\rangle P$ is incorrect in the sense of Λ. In the sense of $\Delta\Lambda$ it is correct, and therefore this example shows that $\Delta\Lambda$ is more permissive than Λ.

We shall see in section 5 how this feature enables us to interpret an Automath book as a term in $\Delta\Lambda$. The algorithm for checking the correctness of the term is an efficient algorithm for checking the correctness of the whole book.

The fact that $\Delta\Lambda$ deviates from Λ is connected to the algorithmic nature of the definition of correctness in $\Delta\Lambda$. It processes a sub-term $\langle R\rangle P$ from left to right. By the time it gets to P it makes use of the fact that $\langle R\rangle$ had been there. The definition of correctness of $\Delta\Lambda$ as given in [15],section 5.3 * will not be repeated here. Instead of repeating that algorithmic definition we shall describe an algorithm that implements it.

Algorithm for checking correctness

From the algorithmic definition of correctness in $\Delta\Lambda$ we can get to an algorithm that checks whether a given string is a correct term.

The algorithm can be described in two rounds. In the first round the algorithm runs through the term from left to right, and for every applicator it produces a context and a pair of terms for which definitional equality in that context remains to be established. The second one of the pair is always the type of a third one, and therefore our definitional equality checking can also be called *type checking.*

The first round can be completed irrespective of whether all these definitional equalities can be established or not. And we mention that this first round runs practically in linear time.

In the second round the definitional equalities of the list have to be checked. This second round does not depend on typing any more. It can be interpreted in untyped lambda calculus, and it is a matter of practice to devise an efficient strategy. The questions are decidable because of the property of strong normalization in Λ, but it is certainly a poor strategy to try to establish $\stackrel{\mathrm{d}}{=}$ by evaluation of normal forms: these can be exceedingly long.

This way to split the algorithm into two rounds is mainly intended for better understanding and for theoretical purposes. When we have to deal with very large terms (like those that represent complete mathematics books) it may be more efficient to look into every definitional equality question as soon as it comes up.

The first round of the algorithm will be presented here in Pascal style, operating on terms represented as lambda trees of the kind described in

*That definition in [15] should be corrected: the phrase "if (R, ε, U) and (R, ε, P) represent" is to be read as "if (R, ε, K) and (R, ε, U) represent".

[15], section 2.5. The trees are binary trees, with a label attached to each vertex. The tree structure is described by means of the usual vertex-to-vertex functions father, leftson and rightson. The labels will be called "pointlabels", in order to avoid confusion with the Pascal notion `label`. The binary nodes of the tree carry pointlabels A or T (related to application and typed abstraction, respectively), and the endpoints have as pointlabel either the symbol τ or a vertex (which has to be a vertex with pointlabel T). Finally a lambda tree has to satisfy what we shall call the *reference condition*. It reads: if P and N are vertices, and if the pointlabel of P is N (whence P is an end-point and N is a binary node with pointlabel T), then P is either the rightson of N or a descendant (in the sense of leftsons and rightsons) of the rightson of N.

In [15] the relation between lambda term and lambda tree is described in detail. Here we give a short sketch. For every node in the lambda tree we can consider the subtree which has that node as its root, and that subtree will correspond to a term. First consider a binary node N. Let N_1 and N_2 be its leftson and rightson. Let P, P_1, P_2 be the lambda terms corresponding to these three points. Then we have $P = \langle P_1 \rangle P_2$ if N has pointlabel A, and $P = [x : P_1]P_2$ (where x is an appropriately chosen name for a bound variable) if N has pointlabel T. We also have to describe the terms corresponding to end-points of the lambda tree. If the pointlabel is τ, the corresponding term is just τ itself. If the pointlabel of such an end-point is a vertex N (whence N is a binary node with pointlabel T, then the term coresponding to the end-point is the variable introduced in the abstractor $[x : P_1]$ which corresponds to N.

We shall provide our lambda tree with an extra vertex `fakeroot` as father of the real root of the tree. The real root is the leftson of the fakeroot; the rightson of the fakeroot plays no role. This little trick will simplify the algorithm,

The type `lab` will be used in the program for typing the pointlabels, so a `lab` is either a vertex, or A, or T, or τ.

The execution of the program will list all cases where a typecheck has to be carried out. The procedure `task(ind:integer;u,v:vertex)` reports it has to be shown that the subtree rooted by the vertex `u` is equal (in the sense of β-equivalence interpreted in the context indicated by `ind`) to the subtree rooted by `v`.

What is called "context" here is not the usual notion of an abstractor string. Here it does not necessarily contain abstractors only. Apart from those, it can have pairs consisting of an applicator followed by an abstractor. This sequence of abstractors and applicators will be described by the array `context` from 0 to `ind`. Every entry in the array is a vertex, and that vertex is always the leftson of its father. Its father's pointlabel (either A or T) shows whether that entry is an applicator or an abstractor, in the following way. Let N be one of those entries, and let P be the lambda tree corresponding to N. Then N will be associated with the applicator $\langle P\rangle$ if N's father has pointlabel A, and with an abstractor $[:P]$ if N's father has pointlabel T.

The `M` in the program represents the number of vertices of the original tree.

One of the features of the program is that it avoids typechecking work that was done previously. The integer variable `status` takes care of that. It has value 0 if it gets to a binary node for the first time (before entering the left branch), value 1 if it gets there for the second time (after a non-stop trip from an endpoint to the first binary node that is approached from the side of the leftson), but when it has value 2 the left branch of the binary node is not searched at all. The effect is that for every A-node in the lambda tree we get exactly one call of the procedure `task`. Quite another matter is that we often want to apply the program to trees that contain a large amount of repetition themselves. Avoiding duplication of work in those cases is of cause a separate problem.

As usual, the `abort` will mean that the execution of the program is to be stopped. This happens when incorrectness is established already without any typechecking (example $\langle\tau\rangle\tau$).

After all these preparations we now give the program. It is assumed that the functions `father(v:vertex):vertex`, `leftson(v:vertex):vertex`, `rightson(v:vertex):vertex`, and `pointlabel(v:vertex):lab` describe a lambda tree, so in particular the program does *not* check the reference condition.

```
var ap1:array[0..M] of vertex;ap2:array[0..M] of integer;
    ap3:array[0..M] of boolean;context:array[0..M] of vertex;
    m,ind,status:integer;loc:vertex;ba:boolean;labloc:lab;

procedure P1;
   begin m:=m+1;ap1[m]:=loc;ap2[m]:=ind;
         ap3[m]:=true;loc:=leftson(loc);
   end;
procedure P2;
   begin case labloc of
         'A': begin m:=m+1;ap1[m]:=leftson(loc);
                    ap2[m]:=status;ap3[m]:=false
              end;
         'T': begin if (not ba) then
```

```
                begin x:=ap1[m];m:=m-1;
                  if ap2[m]=1 then task(ind,x,leftson(loc));
                  ind:=ind+1;context[ind]:=x;
                end;
                ind:=ind+1;context[ind]:=leftson(loc)
              end;
        end {case labloc};
     loc:=rightson(loc);if status=1 then status:=0;
   end;
procedure P3;
   begin if labloc='tau' then abort
         else begin status:=2;loc:=leftson(labloc) end
   end;
procedure P4;
   begin loc:=ap1[m];ind:=ap2[m];m:=m-1;status:=1
   end;

begin loc:=fakeroot;m:=0;status:=0;ind:=-1;ap3[0]:=true;
  repeat
   labloc:=pointlabel(loc); ba:=ap3[m];
   if labloc='A' or labloc='T'
     then begin if status=0 then P1
                else begin if (status=1) or (not ba) then P2
                                else P4
                     end
          end
     else begin if ba then P4 else P3 end
  until loc=fakeroot;
end.
```

4 Characteristics of Automath languages

Generalities

Without trying to explain much in detail, we just indicate a number of issues in which Automath differs from some other verification systems based on typed lambda calculi.

Automath books are written as sequences of lines: definitional lines and primitive lines. Every line is written in a certain context, which is a sequence of typed variables

$$(x_1 : A_1) \cdots (x_k : A_k),$$

where each A_j is a term that may contain the previous $x_1, \cdots, x_{j-1}$. As meta-notation for contexts we use symbols like Γ, and for conxtext extensions $\Gamma, (x : A)$. There is a notion of validity of contexts, a notion of validity of terms (lambda-typed lambda terms) inside a context (meta-notation $\Gamma \vdash P$), and a notion of validity of typings inside a context (meta-notation $\Gamma \vdash P : Q$).

A definitional line (in context Γ) has the form "$f := P : Q$", where f is a new identifier, and where $\Gamma \vdash P : Q$. A primitive line has the same form, but for the fact that the term P is replaced by the symbol PN (which is not considered as a term). PN is acronym for "primitive notion".

There is a fixed set of *basic* terms (like τ, **type**, **prop**) which are said to have *degree* 1. The degree of all further valid terms is found by the rule that if $P : Q$ then degree(P) = degree(Q) + 1. And if $f := P : Q$ (or $f :=$ PN : Q) is a line then degree(f) = degree(Q) + 1. In Automath the degrees are restricted to the values 1, 2, 3, so in lines $f := P : Q$ or $f :=$ PN : Q the Q should have degree 1 or 2.

Abstractors and applicators are denoted as at the end of section 1.

Some of the Automath languages (AUT-QE , AUT-QE-NTI) admit *quasi-expressions*, which have the form of a sequence of abstractors followed by a basic term, like

$$[x_1 : A_1] \cdots [x_k : A_k] \ \mathbf{type}.$$

They are given degree 1.

Instantiation

Very typical for Automath is the fact that the rule for validity of contexts is more permissive than the rules for abstraction. Abstractors [x:**type**] are not allowed, but context extensions (x:**type**) are. A valid context Γ can be extended to a new valid context by any extension $\Gamma, (x : Q)$ provided that x is a new variable, that $\Gamma \vdash Q$ and that the degree of Q is 1 or 2. The requirements for abstraction can be more severe. Abstraction $[x : A]$ is restricted anyway to the case that A has degree 2, in AUT68 and in AUT-QE-NTI moreover by the condition that the type of A is a basic term. In PAL there are no abstractors at all.

In the Automath languages there are (like in standard mathematical language) two different devices for describing functional relationship, the *instantiation device* and the *lambda device*. The use of instantiation admits explicitly defined functions. If inside a context

$$\Gamma,\ (x_1 : A_1) \cdots (x_k : A_k)$$

we have a line $f := P : Q$ or $f := \mathrm{PN} : Q$ then in any later line (in a context Γ_1 that at least starts with Γ), we can *instantiate* f by attaching a string of sub-terms, writing $f(B_1, \ldots, B_k)$. These B_j have to be typed (in the context Γ_1) by the things we get from the corresponding A_j if we replace the x_i's by the corresponding B_i's. It has to be mentionend that the instantiation device generates new definitional equalities by means of the δ-reduction, which is essentially nothing but replacing the defined identifier f by its definition P (in case the line was $f := P : Q$).

In cases where $[x_1 : A_1] \cdots [x_k : A_k]$ are admissible abstractors, the step from f to $f(B_1, \ldots, B_k)$ corresponds to the one from F (where F is obtained from $f(x_1, \ldots, x_k)$ by k-fold lambda abstraction) to the k-fold application $\langle B_k \rangle \cdots \langle B_1 \rangle F$.

In the instantiation device the notion of function is restricted to explicitly defined functions, so it cannot describe mathematics beyond the level of the 18-th century. Nevertheless the system of "proofs as objects" enables the lambda-free Automath language PAL to use the instantiation device for true mathematical reasoning.

But what the instantiation device cannot do is to let us talk about arbitrary functions, and to *say* in the language that a function obtained by the instantiation device is a function indeed. That is what we need the lambda calculus for. We might say that the lambda calculus internalizes instantiation.

Type inclusion

Type inclusion is the point where the Automath languages AUT68 and AUT-QE deviate from pure lambda calculus. Having type inclusion is equivalent to having Π-rules like those in Zucker's AUT-Π (see [21]) and in many other systems of typed lambda calculi. Actually one can say that the notations with Π serve as sign posts for places where type inclusion has been applied. The checker never needs those indications, since it is always directly retraceable where and how type inclusion has been applied. That retraceability was the main reason why such indications were omitted in the design of Automath.

The meta-language of Π-rules seems to be more generally accepted than the one of type inclusion, and the feature of type inclusion may have been one of the reasons why in the early days of Automath logicians were inclined to dislike it. Therefore it may be the right place here to give it some attention, not just to the formal rules but also to interpretations.

The original idea about using a notation $Q : \mathbf{type}$ was that it just served as an indication that Q had degree 2, what meant that Q could be used as a type for other terms, like in $P : Q$. There is an obvious analogy between the "real" typing $P : Q$ and the notation $Q : \mathbf{type}$. They play the same role in the formation of contexts: by context extensions $(Q : \mathbf{type})$ we can introduce type variables. And they play the same role in definitional lines: just like such lines can be used for defining new objects (of degree 3), they can be used for defining types (of degree 2). And primitive lines can be used for creating primitive types (of degree 2) just like they can be used for creating primitive objects (of degree 3). All this can be achieved without saying that **type** is a term; instead of considering $Q : \mathbf{type}$ as a relation between two terms, we consider the combination ": **type**" as a kind of predicate applied to Q.

In the light of this opinion about **type** one has to interpret the AUT68 rule

$$\frac{A : \mathbf{type} \qquad \Gamma, (x : A) \vdash Q(x) : \mathbf{type}}{\Gamma \vdash [x : A]Q(x) : \mathbf{type}}.$$

It just says that $[x : A]Q(x)$ is again a valid term of degree 2.

The next step is that sometimes we want to say more about $[x : A]Q(x)$ than that it has degree 2. We sometimes want to express about some term H of degree 2 that it behaves like a term that starts with the abstractor $[x : A]$. If we have some $a : A$ we may wish to form the application $\langle a \rangle H$, and if moreover we have some $f : H$ we may want to form $\langle a \rangle f$. In order to register those possibilities, the *mock typing* $H : [x : A]\mathbf{type}$ was invented, and that extended AUT68 into AUT-QE. Note that the acronym QE in AUT-QE stands for "quasi-expressions", clearly indicating that $[x : A]\mathbf{type}$ was not considered as a term. As before, the combination ": $[x : A]\mathbf{type}$" was considered as a kind of predicate applied to H.

The mock typing $H : [x : A]\mathbf{type}$ just gave more information about H than just $H : \mathbf{type}$. Of course we took the right in AUT-QE to pass from $H : [x : A]\mathbf{type}$ to $H : \mathbf{type}$ (but not the other way round). It can be interpreted as sacrifice of information. We can see it as applying an inclusion $[x : A]\mathbf{type} \subset \mathbf{type}$, and to the step from $H : [x : A]\mathbf{type}$ to $H : \mathbf{type}$ was called *type inclusion*. It can also be applied repeatedly:

$$[x : A][y : B][z : C]\mathbf{type} \subset [x : A][y : B]\mathbf{type} \subset [x : A]\mathbf{type} \subset \mathbf{type}$$

(note that B may depend on x, and C on both x and y).

Once we start implementing AUT-QE in a lambda calculus like $\Delta\Lambda$ (cf. section 6) we of course begin to consider all the quasi-expressions as lambda terms, and the mock typings as typings of the typed lambda calculus, and that means having impurities. The type inclusion rule conflicts with the idea of unicity of types.

For dicussions where the type inclusion feature is compared to the usage of Π in AUT-Π we refer to [18] and [10].

5 AUT-QE-NTI

The rules

The syntax of AUT-QE-NTI is the same as the one of AUT-QE. The main difference is type inclusion: the acronym NTI stand for "no type inclusion".

AUT-QE-NTI handles quasi-expressions (see section 4). And all quasi-expressions can be used for typing of variables in context extensions. The general rule is that whenever Γ is a valid context, and $\Gamma \vdash Q$, where the degree of Q is 1 or 2, then $\Gamma, (x : Q)$ (with a new variable x) is a valid context. But where AUT-QE allows abstraction over all Q with degree 2, AUT-QE-NTI restricts it to the case where $Q : B$, where B is a basic term (like **type**, **prop**, or τ). So the abstraction rule is

$$\frac{\Gamma \vdash A : \mu, \quad \Gamma, (x : A) \vdash P(x) : Q(x)}{\Gamma \vdash [x : A]P(x) : [x : A]Q(x)},$$

where μ is one of the basic terms. The rules for application and instantiation are the same as for AUT-QE.

In our further discussions about AUT-QE-NTI we shall ignore the possibility to have more than one basic term, and just formulate everything for τ only.

AUT-QE-NTI books as terms in $\Delta\Lambda$

We shall explain here how a book written in AUT-QE-NTI can be transformed into a single term of $\Delta\Lambda$. The book consists of a finite sequence of lines; some are primitive lines, others are definitional lines. Corresponding to every primitive line we form a corresponding abstractor, and to every definitional line we form a pair consisting of an applicator and an abstractor. We just concatenate these abstractors and applicator-abstractor pairs in the order of the lines of the book, and we complete that sequence with a τ. That concatenation will be the term that represents the whole book.

A thing that should be looked into first, is that $\Delta\Lambda$ does not handle instantiation, and therefore the instantiations in the AUT-QE-NTI book have to be remodelled to applications. In the last line of the example given below there is a (double) case of instantiation, and from that example it will be clear how the remodelling works in general.

In [5] a notational convention was explained that can be helpful here. If the identifier f is introduced in a context of length k, then the identifier ③f will be used for what we get from f by abstracting over the last 3 pieces of the context, and the reader is suppposed to understand this without seeing a line where that new identifier ③f was introduced. With this convention we can say that the instantiation $f(U, V, W)$ is definitionally equal to the application $\langle W\rangle\langle V\rangle\langle U\rangle$ ③f.

We now describe how the abstractors and applicator-abstractor pairs are obtained from the book lines. In the case of a primitive line

$$(x_1 : A_1) \cdots (x_k : A_k) \;\vdash\; f \;:=\; \mathrm{PN} \;:\; Q$$

we create a new identifier F and take as the abstractor

$$[\, F \;:\; [x_1 : A_1] \cdots [x_k : A_k] Q \,];$$

in the case of a definitional line

$$(x_1 : A_1) \cdots (x_k : A_k) \;\vdash\; f \;:=\; P \;:\; Q$$

we form the applicator-abstractor pair

$$\langle [x_1 : A_1] \cdots [x_k : A_k] P \rangle \; [\, F \;:\; [x_1 : A_1] \cdots [x_k : A_k] Q \,].$$

The new identifier F should not be confused with the old f (with the notation mentioned above it should be ③f), only if the context was empty there is no danger for such confusion.

We give a short example. Let the book be

	$\vdash$	A	$:=$	PN	$: \tau$
	$\vdash$	c	$:=$	PN	$: A$
	$\vdash$	f	$:=$	$[y : A]c$	$: [y : A]A$
$(x : A)$	$\vdash$	g	$:=$	$\langle x\rangle f$	$: A$
	$\vdash$	B	$:=$	A	$: \tau$
$(z : \tau)$	$\vdash$	h	$:=$	$g(g(c))$	$: B$

This book transforms into the following term in $\Delta\Lambda$:

$$[A : \tau]\ [c : A]\ \langle [y : A]c\rangle\ [f : [y : A]A]\ \langle [x : A]\langle x\rangle f\rangle\ [G : [x : A]A]$$

$$\langle A\rangle\ [B : \tau]\ \langle [z : \tau]\langle\langle c\rangle G\rangle G\rangle\ [H : [z : \tau]B]\ \tau.$$

Under this translation from a book in AUT-QE-NTI into a term in $\Delta\Lambda$ we see how essential the difference between $\Delta\Lambda$ and Λ (indicated in the subsection "Correctness" of section 3) is. If in an AUT-QE-NTI book we replace some definitional line $f := P : Q$ by the primitive line $f := \mathrm{PN} : Q$, the rest of the book will often become incorrect, since the validity of the book might have depended on the definitional equality of f and P, which gets lost if we replace P by PN. But if in the definition of correctness in $\Delta\Lambda$ we would have insisted that correctness of $\langle R\rangle P$ requires correctness of P (see section 3), we would actually have required that the term in $\Delta\Lambda$ that corresponds to the AUT-QE-NTI book would remain correct after just omitting the applicator that carries the information about the definition of f, and that means the same thing as replacing the P by PN.

The example above demonstrates this for the pair $\langle A\rangle$ $[B : \tau]$. If we would omit the $\langle A\rangle$, the type of $[z : \tau]\langle\langle c\rangle G\rangle G$ would no longer be $[z : \tau]B$.

In AUT-SL (see [4]) there was the standard convention that in an application $\langle R\rangle P$ both parts P and R were required to be correct, and that made it necessary to eliminate all definitions of the book (by means of δ-reductions) before the translation into a lambda-term could begin. For that reason AUT-SL could not be more than a way to streamline Automath language theory, whereas $\Delta\Lambda$ is more than that: it is also a practical tool for working with Automath, in particular for the checker.

It is not too easy to say exactly what kind of terms in $\Delta\Lambda$ correspond to an AUT-QE-NTI book according to our translation, since *inside* a line the notion of correctness has to handle the old requirement about correctness of $\langle R\rangle P$, i.e., it has to require the correctness of both R and P. Here we might recommend the Procrustes technique: if a feature of AUT-QE-NTI does not fit in the bed $\Delta\Lambda$ then we just *make* it fit by changing the definition of AUT-QE-NTI. What we gain is the facility of admitting abbreviations which are local inside a line, or even inside a part of a term in that line. But a more important gain is the simplicity of the properties of the result of the embedding.

6 Implementing AUT-QE in AUT-QE-NTI

Introduction

AUT-QE-NTI is definitely weaker than AUT-QE and also weaker than AUT68, just because of its purity. In order to let AUT-QE-NTI enjoy the blessings of type inclusion without actually having it, we can start our book with a set of axioms that mimic the effects of type inclusion. In that way we can say that we can implement AUT-QE and AUT68 in AUT-QE-NTI.

There are two prices we have to pay:

(i) applications of what corresponds to type inclusion have to be written in the book, by means of references to those axioms, with explicit statement of the terms that have to be substituted for the parameters in these axioms,

(ii) the definitional equalities generated by type inclusion have to be replaced by book equalities.

We can try to overcome these objections by (i) providing automatic text writing facilities that does all the dirty work for us, and (ii) by distinguishing two different kinds of book equalities, a strong and a weak one, where the strong one can enjoy the decidability that the definitional equality is supposed to have (see section 6).

The axioms for Π

We now get to the axioms that mimic the type inclusion. Since they of course also mimic the role of Π, we shall use the letter Π as an identifier. Actually there is a multitude of axioms for Π, since the axioms get two parameters σ and μ that stand for basic types (like **type** or **prop**). We could do with a single set of axioms if we would handle only one basic term, or if we would introduce a facility for creating basic terms (that can be done by opening the possibility of degree 0, with a single term $*$ in it, and letting $\mu : *$ mean: "Let μ be a basic term").

As a set of axioms for Π we present

$(X:\sigma)(Q:[x:X]\mu)$	$\vdash \Pi_{\sigma\mu}$	:=PN	$:\ \mu$
$(X:\sigma)(Q:[x:X]\mu)(u:\Pi_{\sigma\mu})$	$\vdash Ax1_{\sigma\mu}$	:=PN	$:\ [x:X]\langle x\rangle Q$
$(X:\sigma)(Q:[x:X]\mu)(v:[x:X]\langle x\rangle Q)$	$\vdash Ax2_{\sigma\mu}$	:=PN	$:\ \Pi_{\sigma\mu}$

These $Ax1$ and $Ax2$ permit us to jump up and down from Q to $\Pi(X,Q)$, and that mimics the type inclusion feature without frustrating the typing rules of the lambda calculus.

We remark that in two cases we have used the "η-expansion" $[x:X]\langle x\rangle Q$ instead of Q itself. This deviates from [11], and is done here in order to avoid having to apply η-reduction in cases of multiple application of the axioms for Π.

If one wants to implement mathematics without ever having to use η-reduction, it probably suffices to pass at once to the η-expansion of Q in all those cases where $Q:[x:X]\mu$ and Q is *not* definitionally equal to a term starting with the abstractor $[x:X]$ (this happens in cases like the one above, where Q was introduced as a variable typed by $[x : X]\mu$). This remark is similar to the observation by D. van Daalen, mentioned in the discussion on η-reduction in section 4.1.1 of [19].

We have to add equality axioms, expressing that we have (in abbreviated form)

$$Ax1(Ax2(v)) \;=\; v, \quad Ax2(Ax1(u)) \;=\; u.$$

The equality here will have to be book equality.

Axioms for Σ

Another set of axioms in AUT-QE-NTI can organize the operation Σ of AUT-Π. In the context $X : \mathtt{type}$, $Q : [x : X]\mathtt{type}$ it postulates the type $\Sigma(X, Q)$, that can be imagined as being the type of all pairs p, q with $p : X$, $q : \langle p \rangle Q$. With the terminology of telescopes (see [17] it can be described as a type that enables us to replace the telescope $[x : X][y : \langle x \rangle Q]$ of length 2 by the telescope $[w : \Sigma(X, Q)]$ of length 1. Actually this set of primitives was used extensively in AUT68 under the name "OwnType" in order to treat sets (subtypes) as types.

By multiple application of Σ we can condense telescopes of length > 2 into length 1.

We can introduce Σ by means of the following axioms:

$(X : \sigma)(Q : [x : X]\mu)$	$\vdash \Sigma_{\sigma\mu}$	:=PN	$: \mu$
$(X : \sigma)(Q : [x : X]\mu)(\varphi : \Sigma_{\sigma\mu})$	$\vdash \mathrm{proj1}_{\sigma\mu}$	:=PN	$: X$
$(X : \sigma)(Q : [x : X]\mu)(\varphi : \Sigma_{\sigma\mu})$	$\vdash \mathrm{proj2}_{\sigma\mu}$	:=PN	$: \langle \mathrm{proj1}_{\sigma\mu} \rangle Q$
$(X : \sigma)(Q : [x : X]\mu)(t_1 : X)(t_2 : \langle t_1 \rangle Q)$	$\vdash \mathrm{pair}_{\sigma\mu}$	:=PN	$: \Sigma_{\sigma\mu}$

We have to add equality axioms, expressing (with book equality) that

$$\mathrm{proj}_1(\mathrm{pair}(t_1, t_2)) \;=\; t_1, \quad \mathrm{proj}_2(\mathrm{pair}(t_1, t_2)) \;=\; t_2,$$
$$\mathrm{pair}(\mathrm{proj}_1(\varphi), \mathrm{proj}_2(\varphi) \;=\; \varphi.$$

Further axioms

Needless to say we want some more axioms for dealing with standard mathematics, in particular axioms about book equality (a survey of what we need in AUT-QE can be found in [19]).

A thing that one might also like to implement is *proof irrelevance*. Proof irrelevance has been proposed as a rule for making different proofs of one and the same proposition definitionally equal (connected with the fact that mathematicians have the opinion that objects do not really depend on proofs), but that means another impurity of the typed lambda calculus. In the vein of AUT-QE-NTI it of course requires some book axioms, and the equality involved in it is a serious candidate for strong book equality (see section 6).

A further candidate for being shifted from the frame to book axioms is the matter of admitting more than 3 degrees. It is to be expected that this feature can be mimicked efficiently by means of axioms over a frame that handles only 3 degrees.

Three kinds of equality

The systems we have considered all use two kinds of equality: *definitional equality* and *book equality*. Definitional equality is handled in the frame by means of the reductions of the language. In general we expect the languages to have the property that definitional equality is decidable, whence it need not be expressed in the books. And there is never an assumption or a negation of definitional equality. On the other hand, book equality is introduced as a notion in the book, and particular book equalities can be proved, disproved or assumed in the book. In general there is no decision procedure for establishing book equality.

The effect of replacing impurities of the lambda calculus by book material is that one will have to live with the administration of many rather trivial book equalities. Let us use the term *strong book equalities* for them. Wherever we shift rules for definitional equality to the book we get such strong equalities, and, conversely, these strong equalities will be considered by others as suitable candidates for putting more definitional equalities into the frame.

As candidates for such strong book equalities we mention the equality axioms for Π and Σ in section 6.

It seems reasonable to give the strong equalities a separate status among the book equalities, in particular since much of the administrative work with these strong book equalities can be automated.

Right now this matter of strong book equality it is not more than a suggestion. The author has not acquired substantial experience with this, and such experience would be necessary for finding out whether it will be sufficiently efficient on the long run.

In order to describe some suggestions we denote ordinary book equality by IS, and strong book equality by ISSt (for types) and ISSo (for objects). Ordinary book equality is defined on any type X; if $x : X, y : X$ then $\mathrm{IS}(X, x, y)$ is a proof class. Strong book equality can be expressed between types too: If X : **type**, Y : **type**, then ISSt(X, Y) is a proof class. Moreover, we can express strong book equality between objects belonging to different but strongly book equal types. If X : **type**, Y : **type**, u : ISSt(X, Y), $x : X$, $y : Y$ then we may have ISSo(X, Y, u, x, y). A book axiom will express that ISSo(X, X, v, x, y) (where v refers to the reflexivity axiom ISSt(X, X)) implies ordinary book equality IS(X, x, y). And there have to be axioms for strong equality for function values $F(x)$ and $G(y)$ if we have strong equality between F and G as well as between x and y.

Quite something has to be done about the relation between book equality (weak or strong) and typing. We can have book equality between types. A common case is that there is a function F mapping type X into type Y, and that we have $p : X$, $q : X$, where p and q are book-equal. We of cause want to call the types $F(p)$ and $F(q)$ book-equal. But if $r : F(p)$, $s : F(q)$ then r and s cannot be compared by ordinary book equality, since they do not have the same type ($r : U$ implies $r : V$ only if U and V are definitionally equal, see section 1, subsection Purity of lambda calculi). If we do not like to extend the notion of book-equality to objects belonging to book-equal types, we will have to work with explicitly constructed bijections, which gives quite some irritating administration. But again, we can expect that such work can be efficiently automated.

We might also use strong equality for proof irrelevance. If P and Q are strongly book-equal proof classes, we may express the principle of irrelevance of proofs by postulating that whenever $p : P$ and $q : Q$ then the proofs p and q are strongly book-equal.

In [16] it was recommended to use the possibility to have two or more kinds of logic interwoven in one and the same book. In that case we have to introduce a different kind of proof classes for each kind of logic. Apart from assigning the type **prop** to a proof class, one might also handle **iprop** (for intuitionistic proofs) and **pprop** for positive (negation-free) proofs. It can be used like this: if p is some proposition then we can form Q with Q : **iprop**, and then $b : Q$ will mean that b is an intuitionistic proof for p. (In [16] it was recommended to use the word **prooftype** instead of the misleading word **prop**, and similarly **pprooftype** for negation-free logic.

In [16] this way of mixing various kinds of logic in one and the same book was recommended for treating constructions (like geometrical constructions). The idea is that we want to be able to say that something is a construction, but we do not want to allow to say that something is *not* a construction. We do not want to admit drawing conclusions from the negation of the statement that something has *not* been constructed. Whether that might ever lead to trouble is doubtful, but it is just not the kind of thing we want to have expressed.

The matter with strong equality is somewhat parallel to this. If we have some kind of definitional equality in the frame, we are unable to express its negation in the books. So if we push that piece of definitional equality into the books, in the form of strong equality, we should not be able to handle its negation either. So it is reasonable to handle strong equality in terms of positive logic.

With the notation explained above this will be expressed by

$$ISSt(X,Y) : \textbf{pprooftype}, \quad ISSo(X,Y,u,x,y) : \textbf{pprooftype}.$$

We leave it at these rather schematic remarks.

7 The project $\Lambda\Sigma$

Introduction

It was expressed already in section 2 that at present $\Lambda\Sigma$ is to be considered as the name of a project rather than as a well-defined language with a satisfactory theory. The project is to define the upper right corner in the following diagram:

$$\begin{tikzcd}
\Delta\Lambda \arrow[r] & \Lambda\Sigma \\
\lambda \arrow[u] \arrow[r] & \lambda\sigma^* \arrow[u]
\end{tikzcd}$$

In the lower left corner we have λ, which is ordinary untyped lambda calculus. The one above it is the typed lambda calculus system $\Delta\Lambda$. In the lower right corner we have a modification $\lambda\sigma^*$ of the segment calculus $\lambda\sigma$ (see section 7). The problem is to find a definition and theory of a language $\Lambda\Sigma$ that has both $\Delta\Lambda$ and $\lambda\sigma^*$ as sublanguages.

The calculus $\lambda\sigma$ was introduced in [12] as an attempt to embed abbreviations for strings and telescopes into a system that takes them seriously. Usually one considers such things as a kind of sugaring, and treats them *ad hoc*, but this kind of abbreviations is bound to accumulate: abbreviations inside abbreviations, and then it is hard to build a dependable checker.

The matter is far from easy. One of the troubles with telescopes is that they contain variables to which there are references from other places, so we can get references to variables inside a telescope (or, more generally, a segment) that is only visible in the form of an abbreviation. Moreover, in different references to one and the same telescope we may have to have different sets for the names of the variables, in order to avoid name clashes. It is obviously a situation that asks for namefree calculus.

Some language theory, like the Church-Rosser property, was developed by H. Balsters in [1]. There has not been much activity in the subject since then.

The calculus $\lambda\sigma$

We shall not *define* $\lambda\sigma$ here, but just try to give an impression of what it is about.

We use the word *segment* for any string of applicators and abstractors in (untyped) lambda calculus. Avoiding namefree notation in this and the following examples, we mention the segment

$$(1) \qquad \langle\langle x\rangle[u]\langle x\rangle u\rangle\ [z]\ \langle\langle z\rangle[t]\langle\langle t\rangle x\rangle\rangle\ [v]$$

(note that the abstractors are untyped, so we have $[x]$ instead of $[x : X]$).

We want to make a calculus that treats segments in a way similar to terms, so we also want to have *segment variables*, and we want to interpret the elimination of segment abbreviations as a kind of β-reduction. As an example we take the abbreviation of the segment $\langle\langle x\rangle[u]\langle x\rangle u\rangle\ [z]$ by the single identifier η. In section 5 we showed how to do this for terms: If the abbreviation is $f := [u]\langle v\rangle u$ then elimination of that definition in a later occurreence of f is effectuated by a local β-reduction, carried out with the applicator-abstractor pair $\langle[u]\langle v\rangle u\rangle\ [f]$. So similarly we want to use for the case of the segment

$$\langle\langle\langle x\rangle[u]\langle x\rangle u\rangle\ [z]\rangle\ [\eta],$$

and there $[\eta]$ is an abstactor that introduces the segment variable η.

But lambda calculus is more than administration of abbreviations: it also concerns variables which are *not* going to be defined like the f above. Accordingly, we will also have undefined segment variables. And since segments may always contain variables that can be referred to, it becomes necessary to have some indication about the number of such referrable variables. To make it worse, some of the variables in the abstractors of a segment can again be segment variables, and the segments to be substituted for them can contain ordinary variables to which it is possible to refer even before these segment variables are replaced by the segments that can be substituted for them.

In order to make that reference system more transparent, the suggestion was made in [12] to attach a kind of norm to the segment variables that shows the structure of the variables contained in them. That norm was called *frame*, but now that the word "frame" has got a completely different meaning it may be better to call that norm the *skeleton*.

The expressions that can be built by means of this segment calculus can sometimes be reduced (by β-reduction) to terms of the lambda calculus, but not always. The segments can begin to lead a life of their own, just like the variables in ordinary lambda calculus achieve more than abbreviations.

We know that segments, in particular telescopes, can be used to describe complex abstract mathematical notions, like the notion "group". A group is usually considered as a mathematical object, but the notion "group" is not. A particular group is represented as a string, and the statement that it represents a group is expressed (with the terminology of [17]) that this string *fits* into the telescope "group". Extensive use of telescope abbreviations for treating modern mathematics was made by Zucker (see [21]).

This segment calculus is something like category theory. The notion "group" may refer to a particular category, represented by a telescope, but if we want to say "let C be any category", then we have to express that by "let η be any telescope", and there we need the telescope variable.

So we see that the title of [12] was not very fortunate: the scope of the paper is wider than the title suggests.

Extending to a typed calculus

What we have in mind with $\Lambda\Sigma$ is primarily a language in which the terms can be reduced to terms of a typed lambda calculus like $\Delta\Lambda$. That means that all segment variables can be eliminated by β-reductions.

If we omit all type information in such a calculus, we get the fragment $\lambda\sigma^*$ of $\lambda\sigma$, containing all $\lambda\sigma$-terms which are reducible to λ-terms. With this limited concept of $\lambda\sigma^*$ one might now say that the diagram given in the beginning of this section describes what we want. This vision on $\Lambda\Sigma$ will hopefully produce a system in which we can efficiently handle accumulated segment abbreviations, both for writing and for checking.

The matter may become harder if we want more than this, like handling segment variables as hinted at in the remark on category theory at the end of the previous subsection.

Bibliography

[1] Balsters, H. (1986). Lambda calculus extended with segments. Ph.D. thesis, Eindhoven University of Technology.

[2] Barendregt, H.P. (1989). Introduction to generalised type systems. *Proceedings 3rd Italian Conference on Theoretical Computer Science*, Eds. A. Bertoni a.o.(World Scientific, Singapore).

[3] Bruijn, N.G. de (1970). The mathematical language Automath, its usage, and some of its extensions. Symposium on Automatic Demonstration (Versailles, December 1968), *Lecture Notes in Mathematics* vol. 125, pp. 29-61 (Springer Verlag).

[4] —— (1971). AUT-SL, a single line version of Automath. Report, Department of Mathematics, Eindhoven University of Technology.

[5] —— (1972). Some abbreviations in the input language for Automath. Report, Department of Mathematics, Eindhoven University of Technology.

[6] —— (1973). Automath, a language for mathematics. *Séminaire Math. Sup.* 1971, 58 p. (Les Presses de l'Université de Montréal).

[7] —— (1974). A framework for the description of a number of members of the Automath family, Memorandum 1974-08, Department of Mathematics, Eindhoven University of Technology.

[8] —— (1974). Some extensions of Automath: the AUT-4 family. Report, Department of Mathematics, Eindhoven University of Technology.

[9] —— (1976). Modifications of the 1968 version of Automath. Memorandum 1976-14, Department of Mathematics, Eindhoven University of Technology.

[10] —— (1977). Some auxiliary operators in AUT-II. Memorandum 1977-10, Department of Mathematics, Eindhoven University of Technology.

[11] —— (1978). AUT-QE without type inclusion. Memorandum 1978-04, Department of Mathematics, Eindhoven University of Technology.

[12] —— (1978). A namefree lambda calculus with facilities for internal definitions of expressions and segments. T.H. Report 78-WSK-03, Department of Mathematics, Eindhoven University of Technology.

[13] —— (1980). A survey of the project Automath. In: *To H.B. Curry: Essays in combinatory logic, lambda calculus and formalism* (ed. J.P. Seldin and J.R. Hindley), pp. 579-606 (Academic Press).

[14] —— (1984). Formalization of constructivity in Automath. In: *Papers dedicated to J.J. Seidel* (ed. P.J. de Doelder, J. de Graaf and J.H. van Lint) EUT-Report 84-WSK-03, ISSN 0167-9708, pp. 76-101. Department of Mathematics and Computing Science, Eindhoven University of Technology.

[15] —— (1987). Generalizing Automath by means of a lambda-typed lambda calculus. In: *Mathematical Logic and Theoretical Computer Science*, Lecture Notes in pure and applied mathematics, 106, (ed. D. W. Kueker, E.G.K. Lopez-Escobar, C.H. Smith) pp. 71-92 (Marcel Dekker, New York).

[16] —— (1990). The use of justification systems for integrated semantics. In: *Colog-88*,(ed. P. Martin-Löf and G. Mints). Lecture Notes in Computer Science vol 417, pp. 9-24 (Springer Verlag).

[17] ——. Telescopic mappings in typed lambda calculus. To be published in *Information and Computation.*

[18] Daalen, D.T. van (1980). *The language theory of Automath.* Ph.D. Thesis, Eindhoven University of Technology.

[19] Jutting, L.S. van Benthem (1979). *Checking Landau's "Grundlagen" in the Automath system.* Ph.D. Thesis, Eindhoven University of Technology, 1977. Mathematical Centre Tracts Nr. 83 (Mathematical Centre, Amsterdam).

[20] Nederpelt, R.P. (1973). Strong normalization in a typed lambda calculus with lambda structured types. Ph.D.thesis, Eindhoven University of Technology.

[21] Zucker, J. (1975) Formalisation of classical mathematics in Automath. In: *Colloques Internationaux du Centre National de la Recherche Scientifique* nr. 249, pp. 135-145.