Mathematics, logic and computation

Mathematics, Logic Computation

Workshop in honour of N.G. de Bruijn's 85th anniversary,
held July 4-5,
Eindhoven
The Netherlands,
in the context of ICALP 2003

Mathematics, Logic and Computation
A workshop in honour of
N.G. de Bruijn's 85th anniversary

July 4-5 2003, Eindhoven, The Netherlands

N.G. de Bruijn was born on July 9, 1918. He finished school at the age of 16, studied Mathematics at Leiden University and received his PhD on modular functions at the Free University of Amsterdam in 1943. From 1939 to 1944 N.G. de Bruijn was a full-time assistant at the Technical University of Delft. That period helped him get through a large part of the war without forced labour in Germany (Delft was in the hands of the German during the war). De Bruijn started his professional career as a researcher at the Philips Research Laboratory in Eindhoven from 1944 to 1946, then occupied a full professorship at the University of Delft from 1946 until 1952 when he moved to a professorship at the University of Amsterdam. In 1960, N.G. de Bruijn returned to Eindhoven University as a professor of Mathematics at Eindhoven University of Technology.

De Bruijn's contributions in the fields of Mathematics and Computer Science are numerous. His book on advanced asymptotic methods, North-Holland 1958, was a classic and was subsequently turned into a book by the famous Dover books series as a result. His work on combinatorics resulted in influential notions and results of which we mention the de Bruijn-sequences of 1946 and the de Bruijn-Erdos theorem of 1948. De Bruijn's famous contributions to mathematics include his work on generalized function theory, analytic number theory, optimal control, quasicrystals, the mathematical analysis of games and much more. In each area he approached, he shed a new light and was known for his originality. De Bruijn could rightly assume the motto "I did it my way" as his own motto. And when it came to automating Mathematics, he again did it his way and introduced the highly influential Automath. In the past decade he has been also working on the theories of the human brain.

Due to the varieties of contributions of de Bruijn, the workshop will concentrate on the computational aspects of Mathematics.

We are delighted to have three exceptional speakers at the workshop: Peter Aczel, Henk Barendregt and Robert Constable. We are also grateful to the excellent programme committee members: Thierry Coquand, Herman Geuvers, Fairouz Kamareddine, Jean-Louis Krivine, Michael Kohlhase and Rob Nederpelt who dealt so efficiently with the paper submissions. The eight accepted

papers have all been influenced in one way or another by the work of de Bruijn and his influence will continue for years to come.

We wish N.G. de Bruijn a long and healthy life and long live his powerful influence in our field.

Herman Geuvers and Fairouz Kamareddine

A simple canonical representation of rational numbers

Yves Bertot [1,2]

Project Lemme
INRIA Sophia Antipolis
France

Abstract

We propose to use a simple inductive type as a basis to represent the field of rational numbers. We describe the relation between this representation of numbers and the representation as fractions of non-zero natural numbers. The usual operations of comparison, multiplication, and addition are then defined in a naive way. The whole construction is used to build a model of the set of rational numbers as an ordered archimedian field. All constructions have been modeled and verified in the Coq proof assistant.

This work started as a quest to find a simple language to represent strictly positive rational numbers. It started as a reflexion on the proof part of representations of rational numbers as reduced fractions: the proof must then be a proof that the numerator and denominator are respectively prime and this proof can be viewed as a trace of Euclid's algorithm to compute the greatest common divisor of two numbers. Looking further, this trace can be used directly as a data-structure to represent rational numbers.

1 From fractions to `Qplus` and back

We propose to use a very simple formal language, which we will call Q^+, and is given by the following syntax:

$$x := 1|Nx|Dx$$

This language can easily be encoded as a recursively defined type in a functional programming language or as an inductive type in a theorem prover. For instance, the definition in Coq [5] is the following one:

```
Inductive Qplus : Set :=
```

[1] thanks to Milad Niqui and Loïc Pottier for many discussions on this subject.

[2] Yves.Bertot@sophia.inria.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

```
One : Qplus | N : Qplus -> Qplus | D : Qplus -> Qplus.
```

Being given a pair of strictly positive natural numbers (p, q), actually representing the fraction $\frac{p}{q}$, we construct the term in our language by recusively applying the following rules:

- if $p = q$ then the term associated to $\frac{p}{q}$ is 1,
- if $p > q$ then the term is Ny where y is the term associated to $\frac{p-q}{q}$ (note that $p - q > 0$),
- if $p < q$ then the term is Dy where y is the term associated to $\frac{p}{q-p}$.

This recursive technique always terminates: if there is a recursive call, then the sum of the two elements in the pair is strictly smaller than the sum of the two initial elements. Thus there is a quantity that decreases strictly at each recursive step: this ensures termination.

There may be several pairs of natural numbers representing the same rational number: in this sense the set of strictly positive rational numbers can be viewed as a quotient set obtained from a partition of the set of pairs of strictly positive rational numbers, but it turns out that the term in Q^+ constructed in this manner does not depend on the pair of numbers that was chosen. Here is another formulation of the same algorithm that shows why. Now this algorithm is described by a function c:

- $c(1) = 1$,
- if $x > 1$, $c(x) = Nc(x - 1)$,
- if $x < 1$ $c(x) = Dc(\frac{1}{\frac{1}{x}-1})$.

It is a simple computation to verify that the two algorithms perform the same steps.

Given a word w in the language Q^+, we can interpret this word as a fraction using the following recursive algorithm.

- if $w = 1$, then the fraction is $\frac{1}{1}$,
- if $w = Ny$ and y can be interpreted as the fraction $\frac{p}{q}$, then w can be interpreted as $\frac{p+q}{q}$,
- if $w = Dy$ then w can be interpreted as $\frac{p}{p+q}$.

The fraction we obtain in this manner is always reduced: the greatest common divisor of p and q is 1. This can be proved by recursion over the length of w.

- Base case: if the length is 1, then $w = 1$, the fraction is $\frac{1}{1}$, which is reduced,
- Let us suppose the length is $n + 1$, where $n \geq 1$, let us suppose any word of length n is interpreted in a reduced fraction. Now w can have one of two forms:

(i) $w = Ny$. In this case y can be interpreted in a reduced fraction $\frac{p}{q}$ and w is interpreted in $\frac{p+q}{q}$. Any divisor common to $p + q$ and q is also common

to $p+q-q$ and q: it is a divisor of p and q. Since $\frac{p}{q}$ is reduced this divisor can only be 1.

(ii) $w = Dy$. In this case we can reason symmetrically to the previous case.

Another way to present the interpretation algorithm is to view it as function returning a rational number and write as a function i. We then have the following presentation:

- $i(1) = 1$,
- $i(Ny) = 1 + i(y)$,
- $i(Dy) = \frac{1}{1+\frac{1}{i(y)}}$.

The functions c and i are clearly inverse to one another: they establish a bijection between the set of strictly positive rational numbers and the language Q^+.

2 The rationale behind rationals

At first sight, this looks like a contrived way to compute the greatest common divisor of two numbers p and q: just compute $c(\frac{p}{q})$ then interpret it as a reduced fraction $\frac{p'}{q'}$ and the greatest common divisor of p and q is the number $\frac{p}{p'} = \frac{q}{q'}$. In fact, we have not done anything else than construct a trace of the decisions made by a simplified form of the usual algorithm to compute the greatest common divisor, known as Euclid's algorithm.

When given two numbers p and q, the greatest common divisor algorithm requires that one divide p by q if $p > q$. If the remainder r is 0 then the greatest common divisor is q, otherwise one should proceed to compute compute the greatest common divisor of r and q. If $q > p$ then one should divide q by p and proceed by computing the greatest common divisor of p and r. If $p = q$ then the greatest common divisor is p.

A simplified form of this algorithm is the algorithm where one subtract q from p when $p > q$ instead of dividing (this is actually the form that was described by Euclid). In the long run, this has the same effect as division: one eventually reaches a point where either $p = q$ (which would correspond to a null remainder in the division) or subtraction has to be done in the other way.

In the simplified form, there is a three way choice that is made based on whether p is larger or smaller than q, or equal to q. The succession of choices made in the algorithm is simply what is recorded in the terms of the Q^+ language.

This is where the representation comes from: the initial motivation was to construct a datastructure to represent rational numbers, so that syntactic equality would coincide with the equality as rational numbers. The usual datastructure, where rational numbers are represented as fractions, that is, as pairs combining a natural number (for the numerator) and a strictly positive natural number (for the denominator) obviously does not fit the requirement:

the two fractions $\frac{22}{10}$ and $\frac{11}{5}$ are not syntactically equal, even though they do represent the same rational number (for this number our representation is $NNDDDD1$.

In type theory, it is usual to manipulate objects that combine data and proofs of properties satisfied by this data. Thus, to have fractions where syntactic equality is meaningful, it would be relevant to consider only reduced fractions, represented by triples where the first two elements would be the usual natural numbers, but the third element would be a proof that the greatest common denominator of the natural numbers is 1. In practice, syntactic equality between proofs is even more problematic to use, but it turned out that one could forget the first two elements, because the proof structure contains enough information to reconstruct them. Hence the idea to represent the rational numbers simply by the trace of the computation of the greatest common divisor.

Still, we could have chosen to use the trace of the computation of the greatest common divisor using the regular algorithm based on division. We will come back to this later. The motivation to take the simplified algorithm was to have all computations easily performed by structural recursion over Peano representations of natural numbers. This will be important when we describe the way Q^+ is implemented in a type-theory based theorem prover like Coq.

Because of its simplicity, this representation is not particularly efficient, when compared to fraction representations, it is still strictly more efficient than a representation where both numerator and denominator are represented as peano numbers, where $\frac{p}{q}$ is represented using $p + q$ symbols, while our representation takes less than $(p/q) + q$ symbols (no gain for natural numbers, obviously); it is probably not as efficient as a fraction representation where both numerator and denominators are represented as binary numbers, especially for rational numbers with a large integer part (or their inverse), where our representation is as inefficient as peano numbers. For a really efficient representation, continued fractions would probably be the best choice.

3 Order

If we note N' the function over rational numbers defined by:

$$N'(x) = i(N(c(x)))$$

and D' the symmetric function, it is obvious that both N' and D' are strictly monotonic functions over the positive rational numbers. Moreover, we have the two following inequalities, for any two strictly positive rational numbers x_1 and x_2:

$$N'(x_1) > 1 > D'(x_2).$$

Combining these two facts, we get the following equivalence:

$$x_1 > x_2 \Leftrightarrow c(x_1) >_{Q^+} c(x_2)$$

where the order $>_{Q^+}$ is defined by:

- for any w_1 and w_2, $Nw_1 >_{Q^+} 1 >_{Q^+} Dw_2$,
- for any w_1 and w_2, $Nw_1 >_{Q^+} Nw_2 \Leftrightarrow w_1 >_{Q^+} w_2$,
- for any w_1 and w_2, $Dw_1 >_{Q^+} Dw_2 \Leftrightarrow w_1 >_{Q^+} w_2$.

To ease notations, we shall often write $>$ for $>_{Q^+}$.

4 Primitive operations

4.1 Inversion

The symetry between numerator and denominator exhibited in the Q^+ language can be exploited to construct the inversion function. For instance, $NDN1$ is $\frac{5}{3}$, while $DND1$ is $\frac{3}{5}$, and $NN1$ is 3 while $DD1$ is $\frac{1}{3}$. We see there is a pattern.

Intuitively, the proof that p and q are relatively prime and the proof that q and p are relatively prime are the same, except that all decisions are symmetric. Thus, constructing the Q^+ representation of the inverse of the number represented by an arbitrary word in Q^+ is simply done with the following inv function:

- $inv(1) = 1$,
- $inv(Nx) = D(inv(x))$,
- $inv(Dx) = N(inv(x))$.

It is simple to prove by induction on the number of N and D that if $i(w)$ returns the fraction p/q, then $i(inv(w))$ returns the fraction q/p.

4.2 Other basic operations

We do not attempt to provide efficient implementations of addition or multiplication. An interesting, probably efficient, algorithm is presented in [8], but we only present naive implementations that use fractions as intermediary data.

We interprete words w and w' in Q^+ as reduced fractions $\frac{p}{q}$ and $\frac{p'}{q'}$, computing the result fraction in the usual manner, and then re-constructing the result word in Q^+ with the c function.

4.2.1 Addition

For addition, the result fraction is

$$\frac{(pq' + p'q)}{qq'.}$$

It is interesting to prove the following theorem:

$$1 + w = Nw$$

Here is a simple proof. If w represents the fraction $\frac{p}{q}$, then the fraction constructed for $1 + w$ is

$$\frac{(1 \times q + p \times 1)}{1 \times q} = \frac{p+q}{q}.$$

When constructing the representation for this number the comparison between numerator and denominator yields that the numerator is bigger and the resulting word is Nw' where w' is the representation of

$$\frac{(p+q-q)}{q} = \frac{p}{q},$$

that is $w' = w$.

This theorem can be used to make addition faster, by adding the integer part of rational numbers before resorting to the more complicated general solution. When adding an integer to a rational number the general solution can simply be avoided.

It is also easy to prove that addition is commutative and associative, simply because addition and multiplication are associative on natural numbers.

4.2.2 Multiplication

For multiplication, the result fraction is pp'/qq'. There is no way to be sure that this fraction is already reduced, so we really have to go the interpretation-reconstruction process. However, we can verify that 1 really acts as a neutral element for multiplication. The result fraction obtained when multiplying with 1 is

$$\frac{1 \times p'}{1 \times q'}$$

and the neutral property is simply inherited from the neutral property of 1 for the multiplication of natural numbers.

Here again, it may be interesting to compute a default approximation of the product of two rational numbers by first computing the product of their integer parts. This will give no gain when multiplying a natural number with an arbitrary rational number, because one still need to resort to the general solution to compute the multiplication of the integer with the fractional part of the other number.

Having both addition and multiplication, it is interesting to verify that we have distributivity. This is done in our formal proof, but we do not describe it in details here.

4.2.3 Subtraction

Subtracting w' from w is meaningful only when w represents a larger rational number than w', this can be checked easily thanks to the comparison procedure outlined in section 3. The result fraction is

$$\frac{(pq' - p'q)}{qq'}.$$

There is a question whether $pq'-p'q$ really is a strictly positive natural number, but this is a simple consequence of the fact that $p/q > p'/q'$ (by multiplying both sides of the inequality by qq').

Zero is not element of the set of strictly positive rational numbers, so it is not easy to express that subtraction really is the opposite of addition, still we can express it with a theorem that has the following statement:

$$\forall w, w' \in Q^+. \quad (w + w') - w' = w$$

To prove this theorem, we need to show that

$$\frac{(p''q' - p'q'')}{q''q'}$$

is the same as p/q, where p''/q'' is the reduced fraction of

$$\frac{(pq' + p'q)}{qq'},$$

that is, there exists a natural number a such that $pq' + p'q = ap''$ and $qq' = aq''$ thus the first fraction can also be written

$$\frac{a \times (p''q' - p'q'')}{aq''q'} = \frac{((pq' + p'q)q' - p'qq')}{qq'q'} = \frac{pq'^2}{qq'^2} = \frac{p}{q}.$$

5 Encoding the whole rational field

To encode the whole rational field, we need to add 0 and negative numbers. This is easily done by constructing a disjoint sum. In Coq it will be written as follows:

```
Inductive Q : Set :=
  Qpos : Qplus -> Q
| Qzero : Q
| Qneg : Qplus -> Q.
```

Generalizing inversion on this field is trivial, simply lifting the operation defined in section 4.1. Generalizing addition, multiplication, and subtraction is easily done from the basic operations for strictly positive rationals, taking care of signs almost independently of the computation of significative numbers.

For instance, when adding two positive numbers, the result is positive, and the absolute values must be added. On the other hand, when adding a positive and the negative value, then the absolute values (in Q^+) must be compared. If the absoluve value of the positive argument is larger, then the result will be positive, but the resulting absolute value is going to be the subtraction of the two values.

Of course, a null value may occur among the operands, but this is easily taken care of by expressing the properties of 0 as neutral element for addition and as absorbing element for multiplication. Taking the opposite of a rational number is a simple syntactic operation: just change the sign, when there is one.

Comparison can also be extended to the full field. Here also, it is only a matter of extending comparison for positive numbers given in section 3 with a rule of signs: negative numbers are smaller than 0, which is smaller than positive numbers. For numbers of the same sign, we just compare their absolute values, not forgetting to invert the results when the compared numbers are negative.

6 Implementing the functions in Coq

The calculus of inductive constructions, as implemented in the Coq system, provides good support for describing and proving properties of structural recursive functions. Functions of this kind are easily recognized according to a syntactic pattern when using pattern-matching constructs: recursive calls are only permitted on direct subterms of a special argument and these subterms appear as variables in a pattern.

The function c that we describe above to construct an element of Q^+ from a pair of non-zero natural numbers is not structural recursive. There are several techniques to handle functions that are not structural recursive, several of them include constructing functions that take proofs of termination as arguments [1,2]. Here we have chosen a simpler path: we add a extra artificial argument to the function, whose purpose is only to count the number of allowable recursive calls. The function we obtain has the following form:

```
Fixpoint Qplus_c [p, q, n : nat] : Qplus :=
 Cases n of
   O => One
  | (S n') =>
      Cases (minus p q) of
        O =>
          Cases (minus q p) of
            O => One
          | v => (D (Qplus_c p v n')) end
       | v => (N (Qplus_c v q n'))
      end
 end.
```

In this function we are computing the representation of p/q and the result is correct only for suitable values of n. A simple analysis of the code shows that it suffices that n is larger than the maximum of p and q.

The use of an artificial argument to the `Qplus_c` function makes that it is also defined when its semantics makes no sense. For instance, if numerator or denominator is zero, the value returned is `(D (D ...One))` or `(N (N ...One))`. When stating any proof about this function we need to check that we are talking only about meaningful uses.

For instance, we proved that the function `Qplus_c` is correct, as stated by

the following theorem (there are more theorems about addition than about maximum in Coq and we chose to use this as a lower bound of acceptable values of n). Here the fact that p and q are non-zero is ensured by the fact that they are computed by Qplus_i.

```
Theorem construct_correct:
 ∀ w : Qplus, p, q, n : nat.
 (Qplus_i w) = (p, q) → (le (plus p q) n) →
 (Qplus_c p q n) = w.
```

We can also define a Qplus_c' function that takes only the numerator and denominator of the fraction and adds them before calling Qplus_c. Thus, the fraction n/m will be represented by the term (Qplus_c' $(n)(m)$).

Defining addition and multiplication by converting terms from Q^+ to pairs of natural numbers is then an easy example of structural-recursive programming:

```
Definition Qplus_add : Qplus -> Qplus ->  Qplus :=
   [w, w' : Qplus]
    (Cases (Qplus_i w) of
     (p,q) =>
       (Cases (Qplus_i w') of
        (p',q') =>
          (Qplus_c
           (plus (mult p q') (mult p' q)) (mult q q')
           (plus (plus (mult p q') (mult p' q)) (mult q q')))
        end)
     end).
```

Thanks to the use of pure structural recursive programming, the reductions rules of the calculus of inductive constuctions can always work on closed term, and we can test our addition function on pairs of fractions.

```
Definition Qplus_c' [n,m:nat] := (Qplus_c n m (plus n m)).

Eval Compute in
  (Qplus_i (Qplus_add (Qplus_c' (5)(7))(Qplus_c' (1)(3)))).
  = ((22),(21)) : nat*nat
```

We have followed the same principles for all functions on Qplus and on Q. all functions are programmed in structural recursive way, sometimes with an extra argument to bound the recursive calls, and the functions have been made total by giving an arbitrary value when they should have been undefined.

7 Constructing the rational number field

In theorem provers, the tradition is to use a definitional approach, where new concepts are defined from old ones. In our case, we want to consider that the

natural numbers are given with the basic operations, addition, multiplication, subtraction, and comparison, the sets Q^+ and Q are defined as above, translation from words in Q^+ to pairs of natural numbers, and the definition of basic operations are also given. From this, we want to show that Q satisfies the properties of an ordered archimedian field. Thus, we have to redo a whole bunch of proofs that were simply solved in the previous section by refereeing to the set of mathematical rational numbers, which we should not be using now.

In fact, we only have to prove the 13 axioms that define an ordered archimedian field [4] (there are 14 axioms for a complete ordered archimedian field, but obviously we cannot expect completeness).

Of course, the fact that some functions are normally not total re-appears in the properties with have proved. For instance, the following property expresses that inversion is the symetric operation to multiplication, but the zero case is clearly avoided in the statement, even though our inversion function does have a value for zero.

```
Q_inv_def: ∀x : Q. x ≠ Zero →
             (Q_mult x (Q_inv x)) = (Qpos One).
```

8 Continued fractions

Readers with enough mathematical background may already have recognized simple continued fractions in the Q^+ language. When considering long sequences of the same symbol, it is possible to use natural numbers, as summarized by the following equalities:

$$N'^k x = k + x \qquad D'^k x = 1/(k + 1/x)$$

Combining these equations to analyze large words, we obtain that the word

$$N^{a_0} D^{a_1} \ldots N^{a_n} D^{a_{n+1}} 1$$

actually represents the number

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{\vdots \quad + \cfrac{1}{a_n + \cfrac{1}{a_{n+1} + 1}}}}$$

This is known as a finite simple continued fraction. In this sense we rediscover a fact that is already known: when looking for canonical representation for rational numbers, continued function can be used, as long as all the a_k's are strictly positive, except for the first one. This representation is actually used in algorithms proposed by Kornerup and Matula in [7] where the representation is also enhanced by looking at the step taken when computing the greatest common divisor, but this time when numbers are represented in binary format. The algorithms proposed in Kornerup and Matula's work are

"on-line" algorithms, which in a functional programing approach we might also want to consider as "lazy" computing algorithms.

If this construction is preferred to the other one for use in a theorem prover or in a functional programming language with recursive types, it is sensible to start by representing the rational numbers that are strictly greater than 1. In this manner, we avoid taking care of the special case for a_0 which does not need to be strictly positive. If we only represent numbers that are greater than 1, then a_0 also needs to be positive.

A rational number greater than 1 is necessarily an integer greater or equal to 2, or an integer greater or equal to 1 plus the inverse of a natural number, or a number of the form

$$a + \frac{1}{(b + \frac{1}{x})},$$

where a and b are strictly positive natural numbers, and x is a rational number greater than 1. This can be described with the following new inductive definition.

```
Inductive Qplus' : Set :=
 Nat : positive -> Qplus'
| NatInv : positive -> positive -> Qplus'
| R : positive -> positive -> Qplus'.
```

Having this subset of the field of rational numbers, it is a simple matter to add 1 and inverses of rationals greater than 1 to get all strictly positive rational numbers and to add 0 and opposites of positive rational numbers to get all rational numbers, this is done using the following inductive definition:

```
Inductive Q' : Set :=
  G1 : Qplus' -> Q'
| One' : Q'
| IG1 : Qplus' -> Q'
| Zero' : Q'
| OIG1 : Qplus' -> Q'
| OOne' :Q'
| OG1: Qplus' -> Q'.
```

In this description, `G1` is used for numbers larger than 1, `One'` is used for 1, `IG1` (the `I` stands for *inverse*) is used for numbers between 0 and 1, actually `(IG x)` represents the inverse of `(G1 x)`, `Zero'` stands for 0, `OIG1` is used for numbers between -1 and 0, actually `(OIG1 x)` represents the opposite of `(IG1 x)`, `OOne'` is used for -1, and `OG1` is used for numbers lesser than -1, actually `(OG1 x)` represents the opposite of `(G1 x)`.

Basic operations can be defined on this structure by following the guidelines given both by the interpretation of terms in `Qplus'` as finite continued fractions or as compact encodings of terms in Q^+, but this work has not been done yet.

9 Inductive proofs on rational numbers

Having an inductive structure to describe rational numbers, it can be used to guide proofs about these numbers, in the same manner as the peano structure of natural numbers guides proofs by providing the usual induction principle on these numbers. In this section, we show how this leads us into a new way of proving things, that may sometimes turn out to be more efficient.

9.1 A proof that the square root of 2 is not rational

The intuition behind this proof is that the square root of two actually is represented by the following infinite continued fraction:

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\vdots}}}$$

In other terms, if $\sqrt{2}$ were a rational number, then it would be represented by the the term:

$$\sqrt{2} = NDDN\sqrt{2}$$

This is impossible, because it leads to an infinite element in an inductive type.

Let us suppose that $\sqrt{2}$ is rational, and let us show that

$$\sqrt{2} = NDDN\sqrt{2}.$$

The square of 1 is 1 and $1 < 2$, since the square function is increasing, then $\sqrt{2}$ is necessarily of the form Nx, $N1$ is 2 and $2^2 > 2$ then $\sqrt{2}$ is necessarily of the form NDx', $ND1$ is $3/2$ and $(3/2)^2 = 9/4 > 2$, then $\sqrt{2}$ is necessarily of the form $NDDx''$, $NDD1$ is $4/3$ and $(4/3)^2 < 2$ then $\sqrt{2}$ is necessarily of the form $NDDNy$, where y represents a strictly positive rational number which we also denote y. By the definition of interpretation of N and D, we have:

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{1 + y}}.$$

Using a few algebraic transformations that are all licit because y is strictly positive, we get the following equality:

$$\sqrt{2} = \frac{3y + 4}{2y + 3}$$

After squaring both sides of the equality, multiplying by $(2y + 3)^2$ (a strictly positive number), and simplifying, we get:

$$2 = y^2$$

This proves that $y = \sqrt{2}$ and leads to the contradiction we are looking for.

The same form of reasoning applies to prove that $\sqrt{3}$ is not rational, this time using the fact that if $\sqrt{3}$ were rational, it would have to verify the fol-

lowing equality:

$$\sqrt{3} = NDN\sqrt{3}.$$

It is even possible to re-do this proof by only following the *structure* suggested by the Q^+ language, but without explicitely using the N and D constructs. Here it is:

We prove by induction on n, that there is no pair of non zero numbers p and q such that $p + q \leq n$ and $p^2 = 2q^2$. Let us take an arbirtrary n and, as induction hypothesis, let us suppose that for all $m < n$, there is no pair of non zero numbers p' and q' such that $p' + q' = m$ and $p'^2 = 2q'^2$. Let us suppose we have two non zero numbers p and q such that $p + q = n$ and $p^2 = 2q^2$. Let us prove that there is a contradiction.

Since $1^2 < 2 < 2^2$, we know that $q < p < 2q$, let $p' = p - q$, we know that $p' < q$ and since q is non zero, we have $p' < p$. We also have

(1) $(p' + q)^2 = 2q^2$.

If $q \leq 2p'$ then there exists an $x > 0$ such that $2p' = q + x$, the above equality can be transformed into:

$$(2p' + 2q)^2 = 8q^2.$$

This gives

$$9q^2 + 6qx + x^2 = 8q^2$$

and after simplification:

$$q^2 + qx + x^2 = 0.$$

This is not possible if $q > 0$.

On the other hand, if $q > 3p'$ then there exists an x such that $q = 3p' + x$ and we can simplify the equality 1 into the following one:

$$16p'^2 + 8p'x + x^2 = 18p'^2 + 12p'x + x^2$$

and after simplification

$$0 = 2p'^2 + 12p'x.$$

Again, this is not possible if $p > 0$. Thus, we know that $2p' < q < 3p'$, let q' be the non zero number such that $q = 2p' + q'$ and $q' < p$. We have

(2) $(3p' + q')^2 = 2 \times (2p' + q')^2)$

Now let p'' be the strictly positive number $p'' = p' - q'$ With this number the equation 2 becomes:

$$(3p'' + 4q')^2 = 2 \times (2p'' + 3q')^2$$

and after simplification:

$$p''^2 = 2q'^2$$

By construction $p' < q' < p$ and $q' < q$, thus $p'' + q' < n$ and by using the induction hypothesis, we can deduce that there is a contradiction. The proof is over.

If we analyze the structure of this proof, it follows directly the structure given by Q^+ and the previous proof:

(i) The decision to perform proof by induction on the sum of the numerator and denominator is guided by the fact that the function c terminates because the sum of the numerator and denominators decreases,

(ii) the introduction of the number p' corresponds to the application of N that is the first element of the segment $NDDN$ that is repeated in the continued fraction expansion,

(iii) the introduction of the number q' corresponds to the two applications of the D that occur in $NDDN$,

(iv) the introduction of the number p'' corresponds to the last N occurring in $NDDN$,

(v) the concluding use of the induction hypothesis corresponds to the remark that the continued fraction for $\sqrt{2}$ is infinite.

This proof may look a little more complicated, but we have gone to all these tedious steps to show that we have never used any other operations that multiplication, addition, and subtractions, and comparisons of natural numbers. This is important to show that this proof that square root of 2 is not rational is very simple in the amount of mathematical tools it uses. This is an important point when considered mechanized proofs, where the full extent of mathematical knowledge is rarely available. The usual proof, as proposed initially by Euclid, goes through the argument that if $p^2 = 2q^2$, then p^2 is even, then p is even, then q is even, and the fraction is not reduced. This proof usually requires that one define the concept of even numbers and then show that if the square of a number is even, this number is also even. Euclid's proof carries over to $\sqrt{3}$ only at the expense of defining the property to be a multiple of 3, and with a little efforts it also carries over to a proof that the cubic root of 2 or 3 is not rational. Proofs relying on the Q^+ structure carry easily to the proof that $\sqrt{3}$ is not rational, but they are not adapted for cubic roots.

10 Related work

Continued fraction have been used in mathematics for a long time. John Wallis, a professor at Oxford in the 17th century actually introduced the name and described them. Euler showed that simple continued fraction were in 1-1 correspondance with rational numbers. Lagrange showed that roots of quadratic equations were either rational numbers or periodic continued fractions. More recently, a french clock-maker, Achille Brocot, and the german mathematician Moritz Abraham Stern devised a technique to represent rational numbers that turns out to represent the same inductive structure as the rational numbers in Q^+ [9,3] (for an introductory presentation see [6]). Inline algorithms for

the basic operations on continued fractions have been studied by Vuillemin [10] and similar algorithms have been devised by Niqui [8] for the structures described by Stern and Brocot, which are the same as ours. Milad Niqui and the author of these lines plan to collaborate to construct the proofs that the algorithms described by Niqui compute the same values as the algorithms described naively in this work.

11 Conclusion

All proofs described in this paper have been performed using the Coq system and are available from the author on demand. These proofs include a proof that Q has a field structure and a new presentation of the proof that $\sqrt{2}$ is not rational.

We have given a quotient free representation of rational numbers. There exists several other such representations, and actually continued fractions, with which our representation is related also provide such a quotient free representation. Another example is where positive rational numbers are represented by finite lists of relative numbers, where the k^{th} element describes the power of the k^{th} prime number. Such lists may be of practical use if multiplication plays a more important role than comparison. However, the mathematical background needed to ascertain the validity of this representation is much more important than for our notation, as it relies on the fundamental theorem of arithmetics (unicity of decomposition of any natural number as a product of powers of prime numbers).

The beauty of our representation is in its simplicity. It is remarkable that the positive rational numbers, such a dense set, can be obtained from the natural numbers by virtually adding only one inductive constructor. The constructor N corresponds to the successor function of peano arithmetics, the constructor we add is simply the D constructor, which is simply presented as a symmetric to the N constructor.

Practical applications to this representation seem hard to find, mainly because the basic operations are so clumsy. We have shown that the inductive structure it gives to the set of rational numbers is well adapted to certain kinds of proofs. For instance, proofs that π is not rational may possibly be made easier thanks to this structure, since some of the known proofs rely on the fact that the rational numbers whose sum of numerator and denominator is bounded never get close enough to π. Also this presentation of rational numbers can be used as an intermediary step to prove the correctness of efficient algorithms for exact computation on rational numbers and this will be used in future collaboration with M. Niqui.

As a last remark, I would like to point out that the whole elaboration of this representation comes directly from a reflection on proof as proof objects in type-theory based theorem provers. Although all the statements given in this paper can easily be expressed in a wide variety of theorem provers,

the guideline for elaborating the data-structure is provided by a study of the structure of proofs that two numbers are relatively prime, in other words, a study of Euclid's algorithm to compute the greatest common divisor of two numbers.

References

[1] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[2] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, September 2001.

[3] Achille Brocot. Calcul des rouages par approximation, nouvelle méthode. *Revue chronométrique. Journal des horlogers, scientifique et pratique*, 3:186–194, 1861.

[4] David Delahaye and Micaela Mayero. Field: une procédure de décision pour les nombres réels en coq. In *Proceedings of JFLA'2001*. INRIA, 2001.

[5] Bruno Barras et al. *The Coq Proof Assistant Reference Manual, Version 7.3*. INRIA, `http://coq.inria.fr/doc/main.html`, oct 2002.

[6] Brian Hayes. On the teeth of wheels. *American Scientist*, 88(4):296–300, july-august 2000.

[7] Peter Kornerup and David Matula. LCF: A lexicographic binary representation of the rationals. *Journal of Universal Computer Science*, 1(7):484–503, july 1995.

[8] Milad Niqui. Exact Arithmetic on Stern-Brocot Tree. *submitted*, jan 2003.

[9] Moritz Abraham Stern. Ueber eine zahlentheoretische Funktion. *Journal für die Reine und angewandte Mathematik*, 55:193–220, 1858.

[10] Jean E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, aug 1990.

Eigenvariables, bracketing and the decidability of positive minimal intuitionistic logic

Gilles Dowek [1]

École polytechnique and INRIA,
LIX, École polytechnique,
91128 Palaiseau Cedex, France

Ying Jiang [2]

Institute of Software,
Chinese Academy of Sciences,
P.O. Box 100080 Beijing, China

Abstract

We give a new proof of a theorem of Mints that the positive fragment of minimal intuitionistic logic is decidable. The idea of the proof is to replace the eigenvariable condition by an appropriate scoping mechanism. The algorithm given by this proof seems to be more practical than that given by the original proof. A naive implementation is given at the end of the paper. Another contribution is to show that this result extends to a large class of theories, including simple type theory (higher-order logic) and second order propositional logic. We obtain this way a new proof of the decidability of inhabitation for positive types in system F.

In classical propositional logic, all the rules of sequent calculus commute with contraction and hence a sequent has a proof if and only if it has a cut-free contraction-free proof. The search space for cut-free contraction-free proofs is finite and hence classical propositional logic is decidable.

In intuitionistic propositional logic, the left rule of implication does not commute with contraction anymore and thus to remain complete when searching for a proof, we have to duplicate an implication occurring in the left part of a sequent before we decompose it. For instance, to prove the sequent $(P \Rightarrow Q) \Rightarrow P, P \Rightarrow Q \vdash Q$ it is necessary to use the proposition $P \Rightarrow Q$

[1] Email: Gilles.Dowek@polytechnique.fr
[2] Email: jy@ios.ac.cn

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

twice.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(P \Rightarrow Q) \Rightarrow P, P \Rightarrow Q, P \vdash P}\Rightarrow\text{-left}}{(P \Rightarrow Q) \Rightarrow P, P \Rightarrow Q, P \vdash Q}\Rightarrow\text{-left}}{(P \Rightarrow Q) \Rightarrow P, P \Rightarrow Q \vdash P \Rightarrow Q}\Rightarrow\text{-right}}{(P \Rightarrow Q) \Rightarrow P, P \Rightarrow Q \vdash P}\Rightarrow\text{-left}}{(P \Rightarrow Q) \Rightarrow P, P \Rightarrow Q \vdash Q}\Rightarrow\text{-left}$$

This proof yields the long normal proof-term $(y\ (x\ \lambda z : P\ (y\ z)))$ (with $x : (P \Rightarrow Q) \Rightarrow P$ and $y : P \Rightarrow Q$) where the variable y is used twice.

Thus, the decidability of intuitionistic propositional logic is not as obvious as that of classical propositional logic, and to build a decision algorithm for intuitionistic propositional logic or for inhabitation in simply typed lambda-calculus, we need either to use loop checking or to specialize sequent calculus to avoid this left rule of the implication [8,4,5].

When we extend classical propositional logic by allowing positive quantifiers (i.e. universal quantifiers at positive occurrences and existential quantifiers at negative occurrences), we need to introduce two more rules in sequent calculus: the right rule of the universal quantifier and the left rule of the existential quantifier. These rules also commute with contraction. Hence, the positive fragment of classical predicate logic also is decidable.

Of course, if we have negative quantifiers also we need to introduce two more rules: the left rule of the universal quantifier and the right rule of the existential quantifier. These rules do not commute with contraction and the decidability result does not extend. The fact that in classical predicate logic, contraction needs to be applied only before these two rules can be seen as a formulation of Herbrand's theorem.

When we extend intuitionistic propositional logic with positive quantifiers, the situation is again more complicated. For instance in the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R, P(x), Q, P(x') \vdash Q}\Rightarrow\text{-left}}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R, P(x), Q \vdash \forall x\ (P(x) \Rightarrow Q)}\forall\text{-right}, \Rightarrow\text{-right}}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R, P(x), Q \vdash R}\Rightarrow\text{-left}}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R, P(x) \vdash Q \Rightarrow R}\Rightarrow\text{-right}}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R, P(x) \vdash Q}\Rightarrow\text{-left}}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R \vdash \forall x\ (P(x) \Rightarrow Q)}\forall\text{-right}, \Rightarrow\text{-right}}{(Q \Rightarrow R) \Rightarrow Q, (\forall x\ (P(x) \Rightarrow Q)) \Rightarrow R \vdash R}\Rightarrow\text{-left}$$

we need to rename the variable x into x' when applying the right rule of the universal quantifier for the second time.

Hence the propositions that may occur in the proofs are not in a finite space and even with loop checking, proof search may fail to terminate. For instance, searching for a proof of the proposition

$$((\forall x\ (P(x) \Rightarrow Q)) \Rightarrow Q) \Rightarrow Q$$

we develop the following proof attempt where A is the proposition $(\forall x\ (P(x) \Rightarrow$

$Q)) \Rightarrow Q$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cdots}{A, P(x), P(x'), P(x'') \vdash Q}}{A, P(x), P(x') \vdash \forall x\ (P(x) \Rightarrow Q)}\ \forall\text{-right}, \Rightarrow\text{-right}}{A, P(x), P(x') \vdash Q}\ \Rightarrow\text{-left}}{A, P(x) \vdash \forall x\ (P(x) \Rightarrow Q)}\ \forall\text{-right}, \Rightarrow\text{-right}}{A, P(x) \vdash Q}\ \Rightarrow\text{-left}}{A \vdash \forall x\ (P(x) \Rightarrow Q)}\ \forall\text{-right}, \Rightarrow\text{-right}}{A \vdash Q}\ \Rightarrow\text{-left}}{\vdash A \Rightarrow Q}\ \Rightarrow\text{-right}$$

In this attempt, we accumulate propositions $P(x)$, $P(x')$, $P(x'')$, ... and loop checking fails to prune this branch.

Mints [9] proves that, in the positive fragment of intuitionistic logic, a provable sequent always has a proof with less than n variables, where n is a bound computed in function of the sequent. This way, the search space can be restricted to be finite and hence the positive fragment of intuitionistic predicate calculus is proved to be decidable.

We know that, in logic, variable names are irrelevant and that replacing named variables by another scoping mechanism, such as de Bruijn indices [1], simplifies formalisms very often. The goal of this paper is to replace the eigenvariable condition of sequent calculus, that forces to rename bound variables and to invent new variable names, by an alternative scoping mechanism.

We obtain this way an alternative decision algorithm for the positive fragment of minimal intuitionistic predicate logic, where the search space is restricted just by a loop checking mechanism, like in the propositional case. A naive implementation of this algorithm is given at the end of the paper.

1 Positive propositions

A context is a finite multiset of propositions. The set of free variables of a context $\Gamma = \{A_1, ..., A_n\}$ is defined by $FV(\Gamma) = FV(A_1) \cup ... \cup FV(A_n)$. A sequent $\Gamma \vdash A$ is a pair formed with a context and a proposition. A proposition in minimal logic is positive if all its quantifier occurrences are positive. More precisely, the set of positive and negative propositions are defined by induction as follows.

Definition 1.1 (Positive proposition)

- An atomic proposition is positive and negative,
- a proposition of the form $A \Rightarrow B$ is positive (resp. negative) if A is negative (resp. positive) and B is positive (resp. negative),
- a proposition of the form $\forall x\ A$ is positive if A is positive.

A sequent $A_1, ..., A_n \vdash B$ is positive if A_1, ..., A_n are negative and B is positive.

Proposition 1.2 *A negative proposition has the form $A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P$ where P is an atomic proposition.*

We use a cut free sequent calculus for positive propositions in minimal logic. Instead of the usual left rule for implication

$$\frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, A \Rightarrow B, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}$$

we take a more restricted rule

$$\frac{\Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P \vdash A_1 \ ... \ \Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P \vdash A_n}{\Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P \vdash P}$$

where P is an atomic proposition. This way, proofs can be directly translated to long normal proofs in natural deduction, and the proposition $A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P$ is the type of the head variable of the associated proof-term. The equivalence of this system with other presentations of minimal logic with positive quantifiers is straightforward.

Definition 1.3 (LJ, A sequent calculus for positive propositions)

$$\frac{\Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P \vdash A_1 \quad ... \quad \Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P \vdash A_n}{\Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P \vdash P} \Rightarrow\text{-left}$$

if P is atomic.

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x \ A} \forall\text{-right}$$

if x is not free in Γ.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-right}$$

2 Bracketing

Definition 2.1 (level of a variable) Let E be a positive proposition where all the bound variables are distinct. We associate to each variable x bound in E a level in E defined as follows.

- If $E = \forall x \ F$ then $level_E(x) = 1$,
- if $E = \forall y \ F$ and x is bound in F then $level_E(x) = 1 + level_F(x)$,
- if $E = F \Rightarrow G$ and x is bound in F then $level_E(x) = level_F(x)$.
- if $E = F \Rightarrow G$ and x is bound in G then $level_E(x) = level_G(x)$.

From now on, we consider a fixed closed proposition E where all bound variables are distinct. If x is a variable bound in E, we write $level(x)$ for the level of x in E, and wherever this variable may occur, we always consider its level with respect to E.

Notice that in any sub-proposition of E of the form $\forall x \ A$, all the free variables have a level strictly smaller than that of x.

Definition 2.2 (Bracketed contexts) Bracketed contexts and clusters are mutually inductively defined as follows.

- A *bracketed context* is a finite multiset of clusters,
- a *cluster* is either a proposition or of the form $[\Gamma]_l$ where Γ is a bracketed context and l is a natural number.

The intuition is that in the cluster $[\Gamma]_l$ all the variables free in Γ and of level greater than or equal to l are bound by the symbol [ ]. When we have a sequent $\Gamma \vdash P$, where all the free variables of P have a level strictly smaller than l, we can add brackets of level l to Γ yielding the equivalent sequent $[\Gamma]_l \vdash P$. This equivalence is made precise in proposition 3.3 and 3.4.

The key idea in the algorithm, used in rule $\Rightarrow$-left of definition 2.5, is that in a sequent of the form $[\Gamma]_l, \Gamma' \vdash P$ where all the free variables of P have a level strictly smaller than l, we can move the brackets from Γ to Γ', yielding the equivalent sequent $\Gamma, [\Gamma']_l \vdash P$. Applying this transformation several times, we can pull any proposition out of a bracketed context where it is hidden and put it at toplevel to used it.

Definition 2.3 (Free variables of a bracketed context) The set of free variables of a bracketed context $\Gamma = \{A_1, ..., A_n\}$ is defined by $FV(\Gamma) = FV(A_1) \cup ... \cup FV(A_n)$. The set of free variables of a cluster $[\Gamma]_l$ is defined by $FV([\Gamma]_l) = \{x \in FV(\Gamma) \mid level(x) < l\}$.

Definition 2.4 (Cleaning contexts) We consider the following terminating rules allowing to clean contexts.

If $n \leq p$

$$[\Gamma, [\Delta]_n]_p \longrightarrow [\Gamma]_p, [\Delta]_n$$

$$[\,]_n \longrightarrow \emptyset$$

$$AA \longrightarrow A$$

We present now another sequent calculus, where bracketing avoids renaming. In this sequent calculus all contexts are cleaned. Hence in a context of the form $[...[... ...[...]_{l_{i-1}}...]_{l_2}]_{l_1}$, we have $l_1 < l_2 < ... < l_{i-1}$.

Definition 2.5 (LJB, A bracketed sequent calculus)

$$\frac{\Gamma' \vdash A_1 \quad ... \quad \Gamma' \vdash A_n}{\Gamma \vdash P} \Rightarrow\text{-left}$$

where

$$\Gamma = \Gamma_1, [\Gamma_2, [...\Gamma_{i-1}, [\Gamma_i, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P]_{l_{i-1}}...]_{l_2}]_{l_1}$$

$$\Gamma' = [\Gamma_1]_{l_1}, [\Gamma_2]_{l_2}, ..., [\Gamma_{i-1}]_{l_{i-1}}, \Gamma_i, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P$$

P is atomic, $l_1 < l_2 < ... < l_{i-1}$ and for each variable x free in P, $level(x) < l_1$.

$$\frac{[\Gamma]_n \vdash A}{\Gamma \vdash \forall x\ A} \forall\text{-right}$$

where $n = level(x)$ and all the free variables of $\forall x\ A$ have a level strictly smaller than n.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-right}$$

Of course, when applying these rules bottom-up, we must clean the contexts of the premises if needed.

Example 2.6 Let us try again to prove the proposition

$$((\forall x\ (P(x) \Rightarrow Q)) \Rightarrow Q) \Rightarrow Q$$

We obtain the following proof attempt, where A is the proposition $(\forall x\ (P(x) \Rightarrow Q)) \Rightarrow Q$.

$$\begin{array}{cl}
\dots & \\ \hline
A, [P(x)]_1 \vdash \forall x\ (P(x) \Rightarrow Q) & \\ \hline
[A]_1, [P(x)]_1, P(x) \vdash Q & \Rightarrow\text{-left} \\ \hline
A, [P(x)]_1 \vdash \forall x\ (P(x) \Rightarrow Q) & \forall\text{-right}, \Rightarrow\text{-right} \\ \hline
[A]_1, P(x) \vdash Q & \Rightarrow\text{-left} \\ \hline
A \vdash \forall x\ (P(x) \Rightarrow Q) & \forall\text{-right}, \Rightarrow\text{-right} \\ \hline
A \vdash Q & \Rightarrow\text{-left} \\ \hline
\vdash A \Rightarrow Q & \Rightarrow\text{-right}
\end{array}$$

Now, instead of accumulating propositions $P(x)$, $P(x')$, $P(x'')$, ... we accumulate clusters $[P(x)]_1$, that are collapsed by context cleaning. Thus, the sequent $A, [P(x)]_1 \vdash \forall x\ (P(x) \Rightarrow Q)$ is repeated and loop checking prunes this branch.

Example 2.7 Let us try now to prove the proposition

$$((\forall x\ (P(x) \Rightarrow ((\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R) \Rightarrow R)) \Rightarrow Q) \Rightarrow Q$$

We obtain the following proof attempt, where A is the proposition $(\forall x\ (P(x) \Rightarrow ((\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R) \Rightarrow R)) \Rightarrow Q$.

$$\begin{array}{cl}
\dots & \\ \hline
A, [[P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y)]_1 \vdash \forall x\ (P(x) \Rightarrow ((\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R) \Rightarrow R) & \\ \hline
[A]_1, [[P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y)]_1, [P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y) \vdash Q & \Rightarrow\text{-left} \\ \hline
[A]_1, [[P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y)]_1, P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R \vdash \forall y\ (P(y) \Rightarrow Q) & \forall\text{-right}\ \Rightarrow\text{-right} \\ \hline
[A]_1, [[P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y)]_1, P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R \vdash R & \Rightarrow\text{-left} \\ \hline
A, [[P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y)]_1 \vdash \forall x\ (P(x) \Rightarrow ((\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R) \Rightarrow R) & \forall\text{-right}\ \Rightarrow\text{-right} \\ \hline
[A]_1, [P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R]_2, P(y) \vdash Q & \Rightarrow\text{-left} \\ \hline
[A]_1, P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R \vdash \forall y\ (P(y) \Rightarrow Q) & \forall\text{-right}\ \Rightarrow\text{-right} \\ \hline
[A]_1, P(x), (\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R \vdash R & \Rightarrow\text{-left} \\ \hline
A \vdash \forall x\ (P(x) \Rightarrow ((\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R) \Rightarrow R) & \forall\text{-right}\ \Rightarrow\text{-right} \\ \hline
A \vdash Q & \Rightarrow\text{-left} \\ \hline
\vdash A \Rightarrow Q & \Rightarrow\text{-right}
\end{array}$$

Again, loop checking prunes this branch. We can check that the other branches are pruned in the same way. Thus, the proposition is not provable.

3 Equivalence

As already said, if x is a variable of level 1 and y is a variable of level 2, then in the bracketed context $[P(x) \Rightarrow P(y)]_1, [P(x)]_1$, the two occurrences of x must be considered as distinct exactly as in the context $\exists x \exists y\ (P(x) \Rightarrow P(y)), \exists x\ P(x)$. Thus, such a context is equivalent to the context $P(x') \Rightarrow P(y'), P(x'')$.

We now introduce a flattening function Φ, mapping a bracketed context to a non bracketed one, renaming variables to fresh variables when needed. This function will be used in the proof of soundness and completeness of the system LJB.

Definition 3.1 (Flattening contexts)
$\Phi(\{A_1, ..., A_n\}) = \{\Phi A_1, ..., \Phi A_n\}$,
$\Phi A = A$ if A is a proposition,
$\Phi([\Gamma]_n) = \Phi(\Gamma\{x'_1/x_1, ..., x'_m/x_m\})$ where $x_1, ..., x_m$ are the free variables of Γ of level $\geq n$ and $x'_1, ..., x'_m$ are fresh variables.

Example 3.2 If $level(x) = 1$ and $level(y) = 2$ then

$$\Phi([P(x) \Rightarrow P(y)]_1, [P(x)]_1) = P(x') \Rightarrow P(y'), P(x'')$$

$$\Phi([[P(x) \Rightarrow P(y)]_2, P(x)]_1) = P(x') \Rightarrow P(y'), P(x')$$

Proposition 3.3 *If $\Gamma \longrightarrow \Gamma'$ for the system of definition 2.4, then for every proposition A, the sequent $\Phi\Gamma \vdash A$ is provable if and only if the sequent $\Phi\Gamma' \vdash A$ is and the sequent $\Phi\Gamma' \vdash A$ has a proof smaller than or of the same size as that of $\Phi\Gamma \vdash A$ (notice that the fresh variables used in the definition of Φ must be fresh in particular with respect to A).*

Proof. For the first rule, we just need to check that the context $\Phi([\Gamma, [\Delta]_n]_p)$ and $\Phi([\Gamma]_p, [\Delta]_n)$ are identical modulo a renaming of variables not appearing in A. For the second, we use the fact that $\Phi([\ ]_n) = \emptyset$.

For the third rule, we prove that the sequent $\Phi A, \Phi A, \Gamma \vdash B$ is provable if and only if the sequent $\Phi A, \Gamma \vdash B$ is and that $\Phi A, \Gamma \vdash B$ has a smaller proof or a proof of the same size. This is the case because the variables introduced by Φ are fresh (even if different variables are used in different applications of Φ). □

For instance, we have $[P(x)]_1, [P(x)]_1 \longrightarrow [P(x)]_1$, $\Phi([P(x)]_1, [P(x)]_1) = P(y), P(y')$ and $\Phi([P(x)]_1) = P(z)$. A proposition A is provable in one context if and only if it is provable in the other, provided it does not contain free variables among y, y', z.

Proposition 3.4 *For every proposition A, whose free variables have a level strictly smaller than n, $\Phi([\Gamma]_n, \Delta) \vdash A$ if and only if $\Phi([\Gamma]_n, [\Delta]_n) \vdash A$ (again the fresh variables used in the definition of Φ must be fresh with respect to A).*

Proof. The contexts $\Phi([\Gamma]_n, \Delta)$ and $\Phi([\Gamma]_n, [\Delta]_n)$ are identical modulo a renaming of variables not appearing in A. □

Proposition 3.5 (Soundness) *If the sequent $\vdash E$ has a derivation in the system LJB, then it has also a derivation in the system LJ.*

Proof. We prove, more generally, that if A is a sub-proposition of E, Γ a bracketed context containing only sub-propositions of E and the sequent $\Gamma \vdash A$ has a derivation in the system LJB, then the sequent $\Phi\Gamma \vdash A$ has a derivation in the system LJ. We proceed by induction on the structure of the derivation of $\Gamma \vdash A$, using proposition 3.3 to justify cleaning steps.

- If the last rule is $\Rightarrow$-right, we just apply the induction hypothesis and the $\Rightarrow$-right rule of LJ.
- If the last rule is $\forall$-right

$$\frac{[\Gamma]_n \vdash B}{\Gamma \vdash \forall x\ B}\ \forall\text{-right}$$

  by induction hypothesis we have a proof in LJ of $\Phi([\Gamma]_n) \vdash B$. The variable x, that has level n, does not occur free in $\Phi([\Gamma]_n)$, hence we can apply the $\forall$-right rule of LJ and we obtain a proof of $\Phi([\Gamma]_n) \vdash \forall x\ B$. The context $\Phi([\Gamma]_n)$ is a renaming of $\Phi\Gamma$ involving only variables of level greater than or equal to n. As no such variables occur free in $\forall x\ B$, the sequent $\Phi([\Gamma]_n) \vdash \forall x\ B$ is a renaming of $\Phi\Gamma \vdash \forall x\ B$ and we can transform this proof into one of $\Phi\Gamma \vdash \forall x\ B$.
- If the last rule is $\Rightarrow$-left

$$\frac{\Gamma' \vdash A_1 \quad \ldots \quad \Gamma' \vdash A_n}{\Gamma \vdash P}\ \Rightarrow\text{-left}$$

  where

$$\Gamma = \Gamma_1, [\Gamma_2, [\ldots \Gamma_{i-1}, [\Gamma_i, A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow P]_{l_{i-1}} \ldots]_{l_2}]_{l_1}$$

$$\Gamma' = [\Gamma_1]_{l_1}, [\Gamma_2]_{l_2}, \ldots, [\Gamma_{i-1}]_{l_{i-1}}, \Gamma_i, A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow P$$

  P atomic, $l_1 < l_2 < \ldots < l_{i-1}$ and for each variable x free in P, $level(x) < l_1$, then by induction hypothesis we have derivations in LJ of $\Phi\Gamma' \vdash A_1$, ..., $\Phi\Gamma' \vdash A_n$. Applying the $\Rightarrow$-left rule of LJ we get a proof of $\Phi\Gamma' \vdash P$. Using the proposition 3.4 and an induction on i, we get a proof of $\Phi\Gamma \vdash P$.

□

Proposition 3.6 (Completeness) *If the sequent $\vdash E$ has a derivation in the system LJ, then it has also a derivation in the system LJB.*

Proof. We prove, more generally, that if A is a sub-proposition of E, Γ a cleaned bracketed context containing only sub-propositions of E and the sequent $\Phi\Gamma \vdash A$ has a derivation in the system LJ, then the sequent $\Gamma \vdash A$ has a derivation in the system LJB. We proceed by induction on the size of the proof of $\Phi\Gamma \vdash A$.

- If the last rule is $\Rightarrow$-right

$$\frac{\Phi\Gamma, B \vdash C}{\Phi\Gamma \vdash B \Rightarrow C} \Rightarrow\text{-right}$$

then the proposition A has the form $B \Rightarrow C$ and we have a smaller proof in LJ of $\Phi\Gamma, B \vdash C$. The context $\Phi(\Gamma, B)$ is equal to $\Phi\Gamma, B$. Thus, the sequent $\Phi(\Gamma, B) \vdash C$ has a proof of the same size as that of $\Phi\Gamma, B \vdash C$ and, using proposition 3.3, the sequent $\Phi\Gamma' \vdash C$, where Γ' is the cleanning of Γ, B, has a smaller proof or a proof of the same size. We apply the induction hypothesis to this proof, we obtain a proof in LJB of $\Gamma' \vdash C$ and we conclude with the $\Rightarrow$-right rule of LJB.

- If the last rule is $\forall$-right

$$\frac{\Phi\Gamma \vdash B}{\Phi\Gamma \vdash \forall x\ B} \forall\text{-right}$$

then the proposition A has the form $\forall x\ B$ and we have a smaller proof in LJ of $\Phi\Gamma \vdash B$. The context $\Phi([\Gamma]_n)$ is a renaming of $\Phi\Gamma$ involving only variables of level greater than or equal to n. The only free variable of level n or more in B is x and this variable does not occur in $\Phi\Gamma$ (eigenvariable condition), thus this renaming does not involve variables free in B and the sequent $\Phi([\Gamma]_n) \vdash B$ is a renaming of the sequent $\Phi\Gamma \vdash B$. Thus, the sequent $\Phi([\Gamma]_n) \vdash B$ has a proof of the same size as that of $\Phi\Gamma \vdash B$ and, using proposition 3.3 the sequent $\Phi\Gamma' \vdash B$, where Γ' is the cleanning of $[\Gamma]_n$, has a smaller proof or a proof of the same size. We apply the induction hypothesis to this proof, we obtain a proof in LJB of $\Gamma' \vdash B$ and we conclude with the $\forall$-right rule of LJB.

- If the last rule is $\Rightarrow$-left

$$\frac{\Phi\Gamma \vdash A_1 \quad ... \quad \Phi\Gamma \vdash A_n}{\Phi\Gamma \vdash P} \Rightarrow\text{-left}$$

then A is an atomic proposition P, the context $\Phi\Gamma$ contains a proposition of the form $A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P$ and we have smaller proofs of $\Phi\Gamma \vdash A_1$, ..., $\Phi\Gamma \vdash A_n$. Thus, Γ contains a proposition B, of the form $U_1 \Rightarrow ... \Rightarrow U_n \Rightarrow Q$, corresponding to $A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow P$ through Φ and the context Γ has the form $\Gamma_1, [\Gamma_2, [...\Gamma_{i-1}, [\Gamma_i, U_1 \Rightarrow ... \Rightarrow U_n \Rightarrow Q]_{l_{i-1}}...]_{l_2}]_{l_1}$. As this context is clean, we have $l_1 < l_2 < ... < l_{i-1}$. Let $\Gamma' = [\Gamma_1]_{l_1}, [\Gamma_2]_{l_2}, ..., [\Gamma_{i-1}]_{l_{i-1}}, \Gamma_i, U_1 \Rightarrow ... \Rightarrow U_n \Rightarrow P$. The function Φ renames the variables of level greater than or equal to l_1 of $U_1 \Rightarrow ... \Rightarrow U_n \Rightarrow Q$ with fresh variables. The renaming of Q with fresh variables yields P that is the right hand side of the sequent, so Q is equal to P and it contains no renamed variables. Thus, $B = U_1 \Rightarrow ... \Rightarrow U_n \Rightarrow P$ and, we have $level(x) < l_1$ for each variable free in P. The sequent $\Phi\Gamma' \vdash U_1$ is a renaming of $\Phi\Gamma \vdash A_1$, ..., $\Phi\Gamma' \vdash U_n$ is a renaming of $\Phi\Gamma \vdash A_n$. Thus, $\Phi\Gamma' \vdash U_1$, ..., $\Phi\Gamma' \vdash U_n$ have proofs of of the same size as those of $\Phi\Gamma \vdash A_1$, ..., $\Phi\Gamma \vdash A_n$ and, using proposition 3.3, the sequents $\Phi\Gamma'' \vdash U_1$, ..., $\Phi\Gamma'' \vdash U_n$, where Γ'' is the cleanning of Γ', have smaller proofs or proofs of the same size. We

apply the induction hypothesis to these proofs, we obtain proofs in LJB of $\Gamma'' \vdash U_1$, ..., $\Gamma'' \vdash U_n$ and we conclude with the $\Rightarrow$-left rule of LJB.

□

4 Termination

We now prove that the system LJB is decidable. Let $\Gamma \vdash A$ be a sequent, there is only a finite number of cleaned sequents that can occur in a proof of $\Gamma \vdash A$. Indeed, as we never rename variables, all propositions are sub-propositions of $\Gamma \vdash A$ and bracketing depth is bound by the highest level of a variable in $\Gamma \vdash A$. Thus the search space is finite and LJB is decidable.

More precisely, we can prove that if a sequent has a proof then it has a non redundant proof, i.e. a proof where the same sequent does not occur twice in the same branch. Thus bottom-up search with loop checking terminates.

5 Application to simple type theory and system F

In [2] we have given a presentation of simple type theory (higher-order logic) as a theory in first-order predicate logic. We have also given a presentation of this theory in deduction modulo [3] where axioms are replaced by rewrite rules. For instance when we have a proposition $\forall x\ \varepsilon(x)$ and we substitute x by the term $\dot{\Rightarrow}(y, z)$ we have to normalize the proposition $\varepsilon(\dot{\Rightarrow}(y, z))$ yielding $\varepsilon(y) \Rightarrow \varepsilon(z)$. We have shown that simple type theory can be presented with rewrites rules only and no axioms.

When we have a theory in deduction modulo formed with a confluent and terminating rewrite system and no axiom, we can decide if a positive normal proposition is provable or not in this theory. Indeed, as we never substitute variables in a proof, normal propositions remain normal and the rewrite rules can never be used. Thus, a normal proposition is provable in this theory if and only if it is provable in predicate logic.

Thus inhabitation in the positive minimal intuitionist fragment of simple type theory is decidable.

We obtain also this way a new decidability proof for the positive fragment of system F [7], while the general inhabitation problem for system F is known to be undecidable [6].

Proposition 5.1 *The positive fragment of system F is decidable.*

Proof. To each type of system F we associate a proposition in minimal logic, with a single unary predicate ε in the line of [2].

$$\Psi(X) = \varepsilon(X)$$

$$\Psi(T \to U) = \Psi(T) \Rightarrow \Psi(U)$$

$$\Psi(\forall X\ T) = \forall X\ \Psi(T)$$

For instance $\Psi(\forall X\ (X \to X)) = \forall X\ (\varepsilon(X) \Rightarrow \varepsilon(X))$.

As variables are never substituted in the positive fragment, a positive type T is inhabited in system F if and only if the proposition $\Psi(T)$ is provable in minimal intuitionistic logic. Thus inhabitation for positive types in system F is decidable. □

6 An implementation

This decision algorithm for the positive fragment of minimal intuitionistic logic can be easily implemented. For instance, an implementation in ocaml, version 3.06, is given in figure 1. Using this implementation, we can, for example, check that the proposition

$$((\forall x\ (P(x) \Rightarrow ((\forall y\ (P(y) \Rightarrow Q)) \Rightarrow R) \Rightarrow R)) \Rightarrow Q) \Rightarrow Q$$

is not derivable

```
derivable [("x",1);("y",2)]
  (Imp(Imp(Forall("x",Imp(Atomic("P",[Var("x")]),
                Imp(Imp (Forall ("y",Imp (Atomic("P",[Var("y")]),
                                          Atomic("Q",[])))),
                                    Atomic("R",[])),
                              Atomic("R",[])))),
           Atomic("Q",[])),
      Atomic ("Q",[])));;
- : bool = false
```

It is well known that variable names are irrelevant in logic and that they can be replaced by other scoping mechanisms. We have shown in this paper that replacing the eigenvariable condition by an appropriate bracketing mechanism simplifies the decision algorithm of the positive part of minimal intuitionistic predicate logic. The generality of this bracketing mechanism still needs to be investigated.

Acknowledgments

This work is partially supported by PRA SI00-03 and NSFC F020101-69973047.

References

[1] N.G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indagationes Mathematicae*, 34, 5 (1972) pp. 381-392.

```
open List;;

type term = Var of string
          | Func of string * term list;;
type prop = Atomic of string * term list
          | Imp of prop * prop
          | Forall of string * prop;;
type elem = Prop of prop
          | Cluster of elem list * int;;

let rec maxl le l =  fold_left (fun n -> fun t -> max n (maxlt le t)) 0 l
and maxlt le t = match t with
  (Var (x))    -> assoc x le
| (Func (_,l)) -> maxl le l;;

let rec decompose p = match p with
  Atomic (s,l) -> p,[]
| Imp(a1,a2)   -> let (h,t) = decompose a2 in (h,a1::t)
| Forall _     -> failwith "negative";;

let rec red l = match l with
  a::b::l' -> if (a = b) then red (a::l') else a::(red (b::l'))
| _ -> l;;

let add a l = red (merge compare [a] l);;

let rec clus l n = let (l1,l2) = out n l
                   in if l1 = [] then l2 else add (Cluster(l1,n)) l2
and out m l = match l with
  []                  -> [],[]
| (Prop(a)::l')       -> let (l1,l2) = out m l' in (add (Prop(a)) l1,l2)
| (Cluster(c,n)::l')  -> let (l1,l2) = out m l'
                         in if (n <= m) then (l1,add (Cluster(c,n)) l2)
                                        else (add (Cluster(c,n)) l1,l2);;

let rec der seen le g  p = not (mem (g,p) seen) &&
let seen' = (g,p)::seen in match p with
  Atomic(s,l)  -> some seen' le (maxl le l) g [] p
| Imp(a,b)     -> der seen' le (add (Prop(a)) g) b
| Forall (x,a) -> let n = assoc x le in der seen' le (clus g n) a
and some seen le m g g1 a = match g with
  []                  -> false
| (Prop(p))::g'       ->
  let (h,t) = decompose p
  in ((h = a) && (for_all (der seen le (red (merge compare g g1))) t))
     || (some seen le m g' (add (Prop(p)) g1) a)
| (Cluster(l1,n))::g' ->
   ((m < n) && (some seen le m l1 (clus (red (merge compare g' g1)) n) a))
   || (some seen le m g' (add (Cluster(l1,n)) g1) a)
and derivable le p = der [] le [] p;;
```

Fig. 1. An implementation

[2] G. Dowek, Th. Hardin and C. Kirchner, HOL-lambda-sigma: an intentional first-order expression of higher-order logic, *Mathematical Structures in Computer Science*, 11 (2001) pp. 1-25.

[3] G. Dowek, Th. Hardin and C. Kirchner, Theorem proving modulo, *Journal of Automated Reasoning* (to appear).

[4] A.G. Dragalin, Mathematical intuitionism, Translations of mathematical monographs, 67, American mathematical society (1988).

[5] R. Dyckhoff, Contraction-free sequent calculi for intuitionistic logic, *The Journal of Symbolic Logic*, 57, 3 (1992) pp. 795-807.

[6] M.-H. Loeb, Embedding the predicate logic in fragments of intuitionistic logic, *The Journal of Symbolic Logic* 41, 4 (1976) pp. 705-719.

[7] Y. Jiang, Positive Types in System F, *Logic Colloquium*, Paris (2000).

[8] S.C. Kleene, Introduction to metamathematics, North-Holland (1952).

[9] G.E. Minc (G.E. Mints) Solvability of the problem of deducibility in LJ for a class of formulas not containing negative occurrences of quantifiers, *Steklov Inst.* 98 (1968), pp. 135-145.

[10] P. Urzyczyn, Inhabitation in typed lambda-calculi, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 1210 (1997) pp. 373-389.

Automath and Pure Type Systems

Fairouz Kamareddine [1]

School of Mathematical and Computational Sciences
Heriot-Watt Univ., Riccarton
Edinburgh EH14 4AS, Scotland

Twan Laan [2]

Weerdstede 45, 3431 LS Nieuwegein, The Netherlands

Rob Nederpelt [3]

Mathematics and Computing Science,
Eindhoven Univ. of Technology,
P.O.Box 513, 5600 MB Eindhoven, the Netherlands

Abstract

We study the position of the AUTOMATH systems within the framework of Pure Type Systems (PTSs). In [1,15], a rough relationship has been given between AUTOMATH and PTSs. That relationship ignores three of the most important features of AUTOMATH: *definitions*, *parameters* and Π-reduction, because at the time, PTSs did not have these features. Since, PTSs have been extended with these features and in view of this, we revisit the correspondence between AUTOMATH and PTSs. This paper gives the most accurate description of AUTOMATH as a PTS so far.

1 Introduction

The AUTOMATH systems are the first examples of proof checkers, and in this way they are predecessors of modern proof checkers like Coq [13] and Nuprl [10]. The project started in 1967 by N.G. de Bruijn:

> "it was not just meant as a technical system for verification of mathematical texts, it was rather a life style with its attitudes towards understanding, developing and teaching mathematics." ([8]; see [24] p. 201)

[1] Email: fairouz@macs.hw.ac.uk
[2] Email: twan.laan@wxs.nl
[3] Email: r.p.nederpelt@tue.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Thus, the roots of AUTOMATH are not to be found in logic or type theory, but in mathematics and the mathematical vernacular [7]. For some years, de Bruijn had been wondering what a proof of a theorem in mathematics should be like, and how its correctness can be checked. The development of computers in the sixties made him wonder whether a machine could check the proof of a mathematical theorem, provided the proof is written in a very accurate way. De Bruijn developed the language AUTOMATH for this purpose. This language is not only (according to de Bruijn [6]) "*a language which we claim to be suitable for expressing very large parts of mathematics, in such a way that the correctness of the mathematical contents is guaranteed as long as the rules of grammar are obeyed*" but also "*very close to the way mathematicians have always been writing*". The goals of the AUTOMATH project were given as:

"1. The system should be able to verify entire mathematical theories.
2. The system should remain very general, tied as little as possible to any set of rules for logic and foundations of mathematics. Such basic rules should belong to material that can be presented for verification, on the same level with things like mathematical axioms that have to be explained to the reader.
3. The way mathematical material is to be presented to the system should correspond to the usual way we write mathematics. The only things to be added should be details that are usually omitted in standard mathematics." ([8]; see [24] pp. 209–210)

Goal 1 was achieved: Van Benthem Jutting [2] translated and verified Landau's "Grundlagen der Analysis" [23] in AUTOMATH and Zucker [29] formalised classical real analysis in AUTOMATH.

As for goal 2, de Bruijn used types and a propositions as types (PAT) principle[4] that was somewhat different from Curry and Howard's [11,17].

De Bruijn spent a lot of effort on goal 3 and studied the language of mathematics in depth [7]. AUTOMATH features that helped him in goal 3 include:

- The use of books. Just like a mathematical text, AUTOMATH is written line by line. Each line may refer to definitions or results given in earlier lines.
- The use of definitions and parameters. Without definitions, expressions become too long. Also, a definition gives a name to a certain expression making it easy to remember what the use of the definiens is.

As AUTOMATH was developed independently from other developments in the world of type theory and λ-calculus, and as it invented powerful typing ideas that were later adopted in influential type systems (cf. [1]), there are many things to be explained in (and learned from) the relation between the various AUTOMATH languages and other type theories. Type theory was originally invented by Bertrand Russell to exclude the paradoxes that arose from Frege's "Begriffschrift" [14]. It was presented in 1910 in the famous "Principia Mathematica" [28] and simplified by Ramsey and Hilbert and Ackermann. In 1940, Church combined his theory of functions, the λ-calculus, with the simplified type theory resulting in the influential "simple theory of types" [9]. In 1988-1989, Berardi [4] and Terlouw [27] gave as an extension of Barendregt's work [1], a general framework for type systems, which is at the basis of the so-

[4] The first practical use of the propositions-as-types principle is found in AUTOMATH.

called Pure Type Systems (PTSs [1]). PTSs include many of the type systems that play an important role in programming languages and theorem proving.

In this paper we focus on the relation between AUTOMATH and Pure Type Systems (PTSs). Both [1] and [15] mention this relation in a few lines, but as far as we know a satisfactory explanation of the relation between AUTOMATH and PTSs is not available. Moreover, both [1] and [15] consider AUTOMATH without one of its most important mechanisms: definitions and parameters. But definitions and parameters are powerful in AUTOMATH. Even the AUTOMATH system PAL, which roughly consists of the definition system of AUTOMATH only, is able to express some simple mathematical reasoning (cf. Section 5 of [6]). According to de Bruijn [8] this is "*due to the fact that mathematicians worked with abbreviations all the time already*". Moreover, recent developments on the use of definitions and parameters in Pure Type Systems [18,26,19,20] justify renewed research on the relation between AUTOMATH and PTSs.

- In Section 2 we give a description of AUT-68, a basic AUTOMATH system.
- In Section 3 we discuss how we can transform AUT-68 into a PTS. In doing so, we notice that AUT-68 has some properties that are not usual for PTSs: • AUT-68 has η-reduction; • AUT-68 has Π-application and Π-reduction (as it does not distinguish λ and Π); • AUT-68 has a definition system; • AUT-68 has a parameter mechanism. We do not consider η-reduction as an essential feature of AUTOMATH, and focus on the definition and parameter mechanisms, which are the most characteristic type-theoretical features of AUTOMATH. In systems with Π-application, Π behaves like λ, and there is a rule of Π-reduction: $(\Pi x{:}A.B)N \to_\Pi B[x{:=}N]$. In AUTOMATH, both $\Pi x{:}A.B$ and $\lambda x{:}A.B$ are denoted by $[x{:}A]B$. It is not easy to see whether $[x{:}A]B$ represents $\lambda x{:}A.B$ or $\Pi x{:}A.B$. Fortunately, this is not a problem for AUT-68.
- In Section 4, we present a system $\lambda 68$ that is (almost) a PTS. We show that it has the usual properties of PTSs and we prove that $\lambda 68$ can be seen as AUT-68 without η-reduction, Π-application and Π-reduction.

2 Description of AUTOMATH

During the AUTOMATH-project, several AUTOMATH-languages were developed. They all have two mechanisms for describing mathematics. The first is essentially a typed λ-calculus, with the important features of λ-abstraction, λ-application and β-reduction. The second mechanism is the use of definitions and parameters. The latter is the same for most AUTOMATH-systems, and the difference between the various systems is mainly caused by the λ-calculi used. In this section we will describe the system AUT-68 [3,5,12] which not only is one of the first AUTOMATH-systems, but also a system with a relatively simple typed λ-calculus, which makes it easier to focus on the (less known) mechanism for definitions and parameters. We start with a review of PTSs.

2.1 Pure Type Systems

Definition 2.1 Let $\mathbb{V}$ be a set of variables and $\mathbb{C}$ a set of constants (both countably infinite). The set $\mathbb{T}(\mathbb{V},\mathbb{C})$ (or $\mathbb{T}$, if it is clear which sets $\mathbb{V}$ and $\mathbb{C}$ are used) of typed lambda terms with variables from $\mathbb{V}$ and constants from $\mathbb{C}$ is defined by the following abstract syntax: $\mathbb{T} ::= \mathbb{V} \mid \mathbb{C} \mid \mathbb{T}\mathbb{T} \mid \lambda\mathbb{V}{:}\mathbb{T}.\mathbb{T} \mid \Pi\mathbb{V}{:}\mathbb{T}.\mathbb{T}$.

We use x, y, z, α, β as meta-variables over $\mathbb{V}$. In examples, we sometimes want to use some specific elements of $\mathbb{V}$; we use typewriter-style to denote such specific elements. So: $\mathtt{x}$ is a specific element of $\mathbb{V}$; while x is a meta-variable *over* $\mathbb{V}$. The variables $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ are assumed to be *distinct* elements of $\mathbb{V}$ (so $\mathtt{x} \not\equiv \mathtt{y}$ etc.), while meta-variables $x, y, z, \ldots$ may refer to variables in the object language that are syntactically equal. We use $A, B, C, \ldots, a, b, \ldots$ as meta-variables over $\mathbb{T}$. $\mathrm{FV}(A)$, the set of *free variables* of A, and substitution $A[x{:=}B]$ are defined in the usual way. We use $\equiv$ to denote syntactical equality between typed lambda terms. Terms that are equal up to a change of bound variables are considered to be syntactically equal. We assume the *Barendregt Convention* [1] where bound variables are chosen to differ from free ones.

Note 1 • *We write* $AB_1 \cdots B_n$ *as shorthand for* $(\cdots((AB_1)B_2)\cdots B_n)$.

- *We write* $\pi\boldsymbol{x}{:}\boldsymbol{A}.B$, *or* $\pi_{i=1}^n x_i{:}A_i.A$, *as shorthand for* $\pi x_1{:}A_1.(\pi x_2{:}A_2.(\cdots(\pi x_n{:}A_n.A)\cdots))$; *for* $\pi \in \{\lambda, \Pi\}$
- *We use the abbreviation* $A[x_i{:=}B_i]_{i=m}^n$ *to denote* $A[x_m{:=}B_m]\cdots[x_n{:=}B_n]$. *If* $m > n$ *then* $A[x_i{:=}B_i]_{i=m}^n$ *denotes* A. *We write* $A[\boldsymbol{x}{:=}\boldsymbol{B}]$ *for* $A[x_i{:=}B_i]_{i=1}^n$.

Definition 2.2 (β-reduction) The relation $\to_\beta$ is given by the contraction rule $(\lambda x{:}A_1.A_2)B \to_\beta A_2[x{:=}B]$ and the usual compatibility. $\twoheadrightarrow_\beta$ is the smallest reflexive transitive relation that includes $\to_\beta$; $=_\beta$ is the smallest equivalence relation that includes $\to_\beta$. By $A \twoheadrightarrow_\beta^+ B$ we indicate that $A \twoheadrightarrow_\beta B$, but $A \not\equiv B$.

A term with no subterms of the form $(\lambda x{:}A_1.A_2)B$ is in *β-normal form*, or a *normal form* if no confusion arises. We write $A \to_\beta^{\mathrm{nf}} B$ (resp. $A \twoheadrightarrow_\beta^{\mathrm{nf}} B$) if $A \to_\beta B$ (resp. $A \twoheadrightarrow_\beta B$) and B is in β-normal form.

Definition 2.3 • A *specification* is a triple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$, such that $\boldsymbol{S} \subseteq \mathbb{C}$, $\boldsymbol{A} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ and $\boldsymbol{R} \subseteq \boldsymbol{S} \times \boldsymbol{S} \times \boldsymbol{S}$. The specification is *singly sorted* if $\boldsymbol{A}$ and $\boldsymbol{R}$ are (partial) function from $\boldsymbol{S} \to \boldsymbol{S}$, and $\boldsymbol{S} \times \boldsymbol{S} \to \boldsymbol{S}$ resp. We call $\boldsymbol{S}$ the set of *sorts*, $\boldsymbol{A}$ the set of *axioms*, and $\boldsymbol{R}$ the set of (Π-formation) *rules*.

- A *context* is a finite (maybe empty) list $x_1{:}A_1, \ldots, x_n{:}A_n$ (written $\boldsymbol{x}{:}\boldsymbol{A}$) of variable declarations. $\{x_1, \ldots, x_n\}$ is the *domain* $\mathrm{DOM}(\boldsymbol{x}{:}\boldsymbol{A})$ of the context. The *empty context* is denoted $\langle\rangle$. We use Γ, Δ as meta-variables for contexts.

Definition 2.4 (Pure Type Systems) Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The Pure Type System $\lambda\mathfrak{S}$ describes how judgements $\Gamma \vdash_{\mathfrak{S}} A : B$ (or $\Gamma \vdash A : B$, if it is clear which $\mathfrak{S}$ is used) can be derived. $\Gamma \vdash A : B$ states that A has type B in context Γ. The typing rules are given in Figure 1.

A context Γ is *legal* if there are A, B such that $\Gamma \vdash A : B$. A term A is

$$\text{(axiom)}\quad \langle\rangle \vdash s_1 : s_2 \qquad (s_1, s_2) \in \boldsymbol{A}$$

$$\text{(start)}\quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad x \notin \text{DOM}(\Gamma)$$

$$\text{(weak)}\quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B} \qquad x \notin \text{DOM}(\Gamma)$$

$$(\Pi)\quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3} \qquad (s_1, s_2, s_3) \in \boldsymbol{R}$$

$$(\lambda)\quad \frac{\Gamma, x{:}A \vdash b : B \qquad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)}$$

$$\text{(appl)}\quad \frac{\Gamma \vdash F : (\Pi x{:}A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$$

$$\text{(conv)}\quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$$

Fig. 1. The typing rules of PTSs

legal if there are Γ, B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.

An important class of PTSs is formed by the eight PTSs of the Barendregt Cube [1]. These systems all have $\boldsymbol{S} = \{*, \Box\}$, $\boldsymbol{A} = \{(*{:}\Box)\}$, but differ on $\boldsymbol{R}$.

2.2 Books, lines and expressions of AUTOMATH

In AUTOMATH, a mathematical text is thought of as being a series of consecutive "clauses". Each clause is expressed in AUTOMATH as a *line*. Lines are stored in so-called *books*. For writing lines and books in AUT-68 we need: • The symbol `type`; • A set $\mathcal{V}$ of variables; • A set $\mathcal{C}$ of constants; • The symbols ( ) [ ] : — , . We assume $\mathcal{V}$ and $\mathcal{C}$ are infinite, $\mathcal{V} \cap \mathcal{C} = \varnothing$ and $\texttt{type} \notin \mathcal{V} \cup \mathcal{C}$.

Definition 2.5 (Expressions) Define the set $\mathcal{E}$ of AUT-*68-expressions* by:

(variable) If $x \in \mathcal{V}$ then $x \in \mathcal{E}$.

(parameter) If $a \in \mathcal{C}$, $n \in \mathbb{N}$ ($n = 0$ is allowed) and $\Sigma_1, \ldots, \Sigma_n \in \mathcal{E}$ then $a(\Sigma_1, \ldots, \Sigma_n) \in \mathcal{E}$. We call $\Sigma_1, \ldots, \Sigma_n$ the *parameters* of $a(\Sigma_1, \ldots, \Sigma_n)$.

(abstraction) If $x \in \mathcal{V}$, $\Sigma \in \mathcal{E} \cup \{\texttt{type}\}$ and $\Omega \in \mathcal{E}$ then $[x{:}\Sigma]\Omega \in \mathcal{E}$.

(application) If $\Sigma_1, \Sigma_2 \in \mathcal{E}$ then $\langle\Sigma_2\rangle\Sigma_1 \in \mathcal{E}$.

Remark 2.6 • The AUT-68-expression $[x{:}\Sigma]\Omega$ is AUTOMATH-notation for abstraction terms. In PTS-notation one would write either $\lambda x{:}\Sigma.\Omega$ or $\Pi x{:}\Sigma.\Omega$. In a relatively simple AUTOMATH-system like AUT-68, it is easy to determine whether $\lambda x{:}\Sigma.\Omega$ or $\Pi x{:}\Sigma.\Omega$ is the correct interpretation for $[x{:}\Sigma]\Omega$. This is harder in more complex AUTOMATH-systems like AUT-QE (see Section 5).

• The AUT-68-expression $\langle\Sigma_2\rangle\Sigma_1$ is AUTOMATH-notation for the application of

the "function" Σ_1 to the "argument" Σ_2. In PTS-notation: $\Sigma_1\Sigma_2$.[5]

We define FV(A) as for PTSs adding that $\text{FV}(a(\Sigma_1,\ldots,\Sigma_n)) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \text{FV}(\Sigma_i)$. If $\Omega, \Sigma_1, \ldots, \Sigma_n$ are expressions (in $\mathcal{E}$), and $x_1, \ldots, x_n$ are distinct variables, then $\Omega[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]$ denotes the expression Ω (in $\mathcal{E}$) in which all free occurrences of $x_1,\ldots,x_n$ have simultaneously been replaced by $\Sigma_1,\ldots,\Sigma_n$. Correctness of this definition is shown by induction on the structure of Ω. We define $\mathtt{type}[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]$ as $\mathtt{type}$.

Definition 2.7 (Books/lines) An AUT-68-*book* (or *book*) is a finite list (possibly empty) of (AUT-68)-lines. If $l_1,\ldots,l_n$ are the lines of book $\mathfrak{B}$, we write $\mathfrak{B} \equiv l_1,\ldots,l_n$. An AUT-68-*line* (or *line*) is a 4-tuple $(\Gamma; k; \Sigma_1; \Sigma_2)$ where:

- Γ is a context, i.e. a finite (possibly empty) list $x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n$, where the x_is are different elements of $\mathcal{V}$ and the α_is are elements of $\mathcal{E} \cup \{\mathtt{type}\}$;
- Σ_1 can be (only): ∘ The symbol — (if $k \in \mathcal{V}$); ∘ The symbol PN (if $k \in \mathcal{C}$) (PN stands for "primitive notion"); ∘ An element of $\mathcal{E}$ (if $k \in \mathcal{C}$);
- k is an element of $\mathcal{V} \cup \mathcal{C}$; and Σ_2 is an element of $\mathcal{E} \cup \{\mathtt{type}\}$.

Remark 2.8 Three sorts of Automath-lines (see Example 2.9):

(i) $(\Gamma; k; \quad ; \Sigma_2)$ with $k \in \mathcal{V}$. This is a *variable declaration* of the variable k having type Σ_2. This does not really add a new statement to the book, but these declarations are needed to form contexts.

(ii) $(\Gamma; k; \text{PN}; \Sigma_2)$ with $k \in \mathcal{C}$. This line introduces a *primitive notion*: A constant k of type Σ_2. Constant k can act as a primitive notion (e.g., introducing the number 0, or the type of natural numbers), or as an axiom. The introduction of k is *parametrised* by the context Γ. For instance, when introducing the primitive notion of "logical conjunction", we do not use a separate primitive notion for each possible conjunction $\mathtt{and}(A,B)$. Instead, we use one primitive notion **and**, to which we can add two propositions A and B as parameters when needed to form the proposition $\mathtt{and}(A,B)$. Hence, we introduce **and** in a context $\Gamma \equiv \mathtt{x{:}prop}, \mathtt{y{:}prop}$. Given propositions A, B we can form the AUT-68-expression $\mathtt{and}(A,B)$;

(iii) $(\Gamma; k; \Sigma_1; \Sigma_2)$ with $k \in \mathcal{C}$ and $\Sigma_1 \in \mathcal{E}$. This line introduces a *definition*. The *definiendum* k is defined by the *definiens* Σ_1 and has *type* Σ_2. Definitions are parametrised like primitive notions. They help to clarify the book structure, make expression manipulations efficient, and abbreviate long expressions by a name. E.g., 7 names $\mathtt{S(S(S(S(S(S(0))))))}$.

Example 2.9 In Figure 2 we give an example of an AUTOMATH-book that introduces some elementary notions of propositional logic. We have numbered each line in the example, and use these line numbers for reference in our

[5] Note the unusual *order* of "function" Σ_1 and "argument" Σ_2. The advantages of writing $\langle\Sigma_2\rangle\Sigma_1$ instead of $\Sigma_1\Sigma_2$ are extensively discussed in [21].

∅	prop	PN	type	(1)
∅	x	—	prop	(2)
x	y	—	prop	(3)
x,y	and	PN	prop	(4)
x	proof	PN	type	(5)
x,y	px	—	proof(x)	(6)
x,y,px	py		proof(y)	(7)
x,y,px,py	and-I	PN	proof(and)	(8)
x,y	pxy		proof(and)	(9)
x,y,pxy	and-01	PN	proof(x)	(10)
x,y,pxy	and-02	PN	proof(y)	(11)
x	prx	—	proof(x)	(12)
x,prx	and-R	and-I(x,x,prx,prx)	proof(and(x,x))	(13)
x,y,pxy	and-S	and-I(y,x,and-02,and-01)	proof(and(y,x))	(14)

Fig. 2. Example of an AUTOMATH-book

comments below. To keep things clear, we have omitted the types of the variables in the context. The book consists of three parts:

- In lines 1–5 we introduce some basic material:
  1. The type `prop` (of propositions) is a primitive notion.
  2. We declare a variable `x` of type `prop`. `x` will be used in the book;
  3. We define a variable `y` of type `prop` within the context `x:prop`.
  4. Given propositions `x` and `y`, we introduce a primitive notion, the conjunction `and(x,y)` of `x` and `y`;
  5. Given a proposition `x` we introduce the type `proof(x)` of the proofs of `x` as a primitive notion.
- In lines 6 11 we show how we can construct proofs of propositions of the form `and(x, y)`, and how we can use proofs of such propositions:
  6. Given propositions `x` and `y`, we assume that we have a `px` $\in \mathcal{V}$ of type `proof(x)`. I.e., the variable `px` represents a proof of `x`;
  7. We also assume a proof `py` of `y`;
  8. Given propositions `x` and `y`, and proofs `px` and `py` of `x` and `y`, we want to conclude that `and(x,y)` holds. This is a natural deduction axiom called `and-I` (and-introduction). `and-I(x,y,px,py)` is a proof of `and(x,y)`, so of type `proof(and(x,y))`. In line 8, `proof(and)` is the type of `and-I` instead of `proof(and(x,y))`. Automath does this to keeps lines short.
  9. To express how we can use a proof of `and(x,y)`, first we introduce a

variable `pxy` that represents an arbitrary proof of `and(x,y)`;

10. As we want `x` to hold whenever `and(x,y)` holds, we introduce an axiom `and-01` (and-out, first and-elimination). Given propositions `x,y` and a proof `pxy` of the proposition `and(x,y)`, `and-01(x,y,pxy)` is a proof of `x`;
11. Similarly, we introduce an axiom `and-02` representing a proof of `y`;

• We can now derive some elementary theorems:

12. We want to derive `and(x,x)` from `x`. I.e., construct a proof of `and(x,x)` from a proof of `x`. In line 6, we introduced a variable `px` for a proof of `x` in the context `x,y`. As we do not want a second proposition `y` to occur in this theorem, we declare a new proof variable `prx`, in the context `x`;
13. We derive our theorem: The reflexivity of logical conjunction. Given a proposition `x`, and a proof `prx` of `x`, we can use the axiom `and-I` to find a proof of `and(x,x)`: we can use `and-I(x,x,px,px)` thanks to line 8. We give a name to this proof: `and-R`. If, anywhere in the sequel of the book, Σ is a proposition, and Ω is a proof of Σ, we can write `and-R`(Σ, Ω) for a proof of `and`(Σ, Σ). This is shorter, and more expressive, than `and-I`$(\Sigma, \Sigma, \Omega, \Omega)$;
14. We show `and` is symmetric: Whenever `and(x,y)` holds, we have `and(y,x)`. Given propositions `x,y` and a proof `pxy` of `and(x,y)`, we can form proofs `and-01(x,y,pxy)` of `x` and `and-02(x,y,pxy)` of `y`. We feed these proofs "in reverse order" to the axiom `and-I`: `and-I(y,x,and-02,and-01)` represents a proof of `and(y,x)`. The expressions `and-02` and `and-01` must be read as `and-02(x,y,pxy)` and `and-01(x,y,pxy)`.

2.3 Correct books

Not all books are good books. If $(\Gamma; k; \Sigma_1; \Sigma_2)$ is a line of a book $\mathfrak{B}$, the expressions Σ_1 and Σ_2 (as long as Σ_1 is not PN or $\_$, and Σ_2 is not `type`) must be well-defined, i.e. the elements of $\mathcal{V} \cup \mathcal{C}$ occurring in them must have been established (as variables, primitive notions, or defined constants) in earlier parts of $\mathfrak{B}$. The same holds for the type assignments $x_i{:}\alpha_i$ of Γ. Moreover, if Σ_1 is not PN or —, then Σ_1 must be of the same type as k, hence Σ_1 must be of type Σ_2 (within context Γ). Finally, there should be only one definition of any object in a book, so k should not occur in earlier lines. So we need notions of correctness and of typing (with respect to a book and/or a context).

We write $\mathfrak{B}; \varnothing \vdash$ OK to indicate that book $\mathfrak{B}$ is correct, and $\mathfrak{B}; \Gamma \vdash$ OK to indicate that context Γ is correct with respect to the (correct) book $\mathfrak{B}$.[6] We write $\mathfrak{B}; \Gamma \vdash \Sigma_1 : \Sigma_2$ to indicate that Σ_1 is a correct expression of type Σ_2 (or simply a correct expression) with respect to $\mathfrak{B}$ and Γ. We also say $\Sigma_1 : \Sigma_2$ is a correct *statement* with respect to $\mathfrak{B}$ and Γ. We write $\vdash_{\mathrm{AUT-68}}$ if a confusion of system arises. The following two interrelated definitions are based on [12].

Definition 2.10 (Correct books and contexts) A book $\mathfrak{B}$ and a context Γ are *correct* if $\mathfrak{B}; \Gamma \vdash$ OK can be derived with the rules below ($=_{\beta d}$ is given

[6] As the empty context will be correct with respect to any correct book, this does not lead to misunderstandings.

in Section 2.4. The rules use *correct statements* of Definition 2.11):

(axiom) $$\varnothing;\varnothing \vdash \text{OK}$$

(context ext.) $$\frac{\mathfrak{B}_1,(\Gamma;x;\quad;\alpha),\mathfrak{B}_2;\Gamma\vdash \text{OK}}{\mathfrak{B}_1,(\Gamma;x;\quad;\alpha),\mathfrak{B}_2;\Gamma,x{:}\alpha\vdash \text{OK}}$$

(book ext.: var1) $$\frac{\mathfrak{B};\Gamma\vdash \text{OK}}{\mathfrak{B},(\Gamma;x;\text{—};\mathsf{type});\varnothing\vdash \text{OK}}$$

(book ext.: var2) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_2:\mathsf{type}}{\mathfrak{B},(\Gamma;x;\text{—};\Sigma_2);\varnothing\vdash \text{OK}}$$

(book ext.: pn1) $$\frac{\mathfrak{B};\Gamma\vdash \text{OK}}{\mathfrak{B},(\Gamma;k;\text{PN};\mathsf{type});\varnothing\vdash \text{OK}}$$

(book ext.: pn2) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_2:\mathsf{type}}{\mathfrak{B},(\Gamma;k;\text{PN};\Sigma_2);\varnothing\vdash \text{OK}}$$

(book ext.: def1) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_1:\mathsf{type}}{\mathfrak{B},(\Gamma;k;\Sigma_1;\mathsf{type});\varnothing\vdash \text{OK}}$$

(book ext.: def2) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_2:\mathsf{type}\quad \mathfrak{B};\Gamma\vdash \Sigma_1:\Sigma_2'\quad \mathfrak{B};\Gamma\vdash \Sigma_2=_{\beta\text{d}}\Sigma_2'}{\mathfrak{B},(\Gamma;k;\Sigma_1;\Sigma_2);\varnothing\vdash \text{OK}}$$

In the (book ext.) rules, we assume $x\in\mathcal{V}$ and $k\in\mathcal{C}$ do not occur in $\mathfrak{B}$ or Γ.

Definition 2.11 (Correct statements) A statement $\mathfrak{B};\Gamma\vdash\Sigma:\Omega$ is *correct* if it can be derived with the rules below (the start rule uses the notions of correct context and correct book as given in Definition 2.10).

(start) $$\frac{\mathfrak{B};\Gamma_1,x{:}\alpha,\Gamma_2\vdash \text{OK}}{\mathfrak{B};\Gamma_1,x{:}\alpha,\Gamma_2\vdash x{:}\alpha}$$

$$\mathfrak{B}\equiv\mathfrak{B}_1,(x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n;b;\Omega_1;\Omega_2),\mathfrak{B}_2$$

(parameters) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_i{:}\alpha_i[x_1,\ldots,x_{i-1}{:=}\Sigma_1,\ldots,\Sigma_{i-1}](i=1,\ldots,n)}{\mathfrak{B};\Gamma\vdash b(\Sigma_1,\ldots,\Sigma_n):\Omega_2[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]}$$

(abstr.1) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_1{:}\mathsf{type}\qquad \mathfrak{B};\Gamma,x{:}\Sigma_1\vdash \Omega_1{:}\mathsf{type}}{\mathfrak{B};\Gamma\vdash[x{:}\Sigma_1]\Omega_1:\mathsf{type}}$$

(abstr.2) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_1{:}\mathsf{type}\quad \mathfrak{B};\Gamma,x{:}\Sigma_1\vdash \Omega_1{:}\mathsf{type}\quad \mathfrak{B};\Gamma,x{:}\Sigma_1\vdash \Sigma_2{:}\Omega_1}{\mathfrak{B};\Gamma\vdash[x{:}\Sigma_1]\Sigma_2:[x{:}\Sigma_1]\Omega_1}$$

(application) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma_1:[x{:}\Omega_1]\Omega_2\qquad \mathfrak{B};\Gamma\vdash \Sigma_2:\Omega_1}{\mathfrak{B};\Gamma\vdash\langle\Sigma_2\rangle\Sigma_1:\Omega_2[x{:=}\Sigma_2]}$$

(conversion) $$\frac{\mathfrak{B};\Gamma\vdash \Sigma:\Omega_1\qquad \mathfrak{B};\Gamma\vdash \Omega_2{:}\mathsf{type}\qquad \mathfrak{B};\Gamma\vdash \Omega_1=_{\beta\text{d}}\Omega_2}{\mathfrak{B};\Gamma\vdash\Sigma:\Omega_2}$$

When using the parameter rule, we assume that $\mathfrak{B};\Gamma\vdash\text{OK}$, even if $n=0$.

Lemma 2.12 *The book of Example 2.9 (see Figure 2) is correct.*

2.4 Definitional equality

We need to describe the notion $=_{\beta d}$ ("definitional equality"). This notion is based on both the definition and the abstraction/application mechanisms of AUT-68. The abstraction/application mechanism provides the well-known notion of β-equality, originating from $\langle\Sigma\rangle[x{:}\Omega_2]\Omega_1 \to_\beta \Omega_1[x{:=}\Sigma]$. We need to describe the definition mechanism of AUT-68 via the notion of d-*equality.*[7]

Definition 2.13 (d-equality) Let $\mathfrak{B};\Gamma \vdash \Sigma : \Sigma'$. We define the d-*normal form* $\mathrm{nf}_d(\Sigma)$ of Σ with respect to $\mathfrak{B}$ by induction on the length of $\mathfrak{B}$. Assume $\mathrm{nf}_d(\Sigma)$ has been defined for all $\mathfrak{B}'$ with less lines than $\mathfrak{B}$ and all correct Σ with respect to $\mathfrak{B}'$ and a context Γ. By induction on the structure of Σ:

- If Σ is a variable x, then $\mathrm{nf}_d(\Sigma) \stackrel{\mathrm{def}}{=} x$;
- Now assume $\Sigma \equiv b(\Omega_1,\ldots,\Omega_n)$, and assume that the normal forms of the Ω_is have already been defined. Determine a line $(\Delta; b; \Xi_1; \Xi_2)$ in the book $\mathfrak{B}$ (there is exactly one such line, and it is determined by b). Write $\Delta \equiv x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n$. Distinguish:
  - $\Xi_1 \equiv$ —. This case doesn't occur, as $b \in \mathcal{C}$;
  - $\Xi_1 \equiv$ PN. Then define $\mathrm{nf}_d(\Sigma) \stackrel{\mathrm{def}}{=} b(\mathrm{nf}_d(\Omega_1),\ldots,\mathrm{nf}_d(\Omega_n))$;
  - Ξ_1 is an expression. Then Ξ_1 is correct with respect to a book $\mathfrak{B}'$ that contains less lines than $\mathfrak{B}$ ($\mathfrak{B}'$ doesn't contain the line $(\Delta; b; \Xi_1; \Xi_2)$, and all lines of $\mathfrak{B}'$ are lines of $\mathfrak{B}$), and we can assume $\mathrm{nf}_d(\Xi_1)$ has already been defined. Now define $\mathrm{nf}_d(\Sigma) \stackrel{\mathrm{def}}{=} \mathrm{nf}_d(\Xi_1)[x_1,\ldots,x_n{:=}\mathrm{nf}_d(\Omega_1),\ldots,\mathrm{nf}_d(\Omega_n)]$;
- If $\Sigma \equiv [x{:}\Omega_1]\Omega_2$ then $\mathrm{nf}_d(\Sigma) \stackrel{\mathrm{def}}{=} [x{:}\mathrm{nf}_d(\Omega_1)]\mathrm{nf}_d(\Omega_2)$;
- If $\Sigma \equiv \langle\Omega_2\rangle\Omega_1$ then $\mathrm{nf}_d(\Sigma) \stackrel{\mathrm{def}}{=} \langle\mathrm{nf}_d(\Omega_2)\rangle\mathrm{nf}_d(\Omega_1)$.

Write $\Sigma_1 =_d \Sigma_2$ if $\mathrm{nf}_d(\Sigma_1) \equiv \mathrm{nf}_d(\Sigma_2)$[8] and $=_{\beta d}$ for the smallest equivalence relation containing $=_\beta$ and $=_d$.

Definition 2.14 Σ_1 and Σ_2 are called *definitionally equal* (with respect to a book $\mathfrak{B}$) if $\Sigma_1 =_{\beta d} \Sigma_2$.

Instead of Definition 2.13, d-equality can be given via a reduction relation.

Definition 2.15 (δ-reduction) Let $\mathfrak{B}$ be a book, Γ a correct context with respect to $\mathfrak{B}$, and Σ a correct expression with respect to $\mathfrak{B};\Gamma$. We define $\Sigma \to_\delta \Omega$ by the usual compatibility rules, and

[7] This definition depends on the definition of derivability $\vdash$ which in turn depends on the definition of $=_{\beta d}$. The definitions of correct book, correct line, correct context, correct expression and $=_{\beta d}$ should be given within one definition, using induction on the length of the book. This would lead to a correct but very long definition, and that is the reason why the definitions are split into smaller parts (in this paper as well as in [12]).

[8] Note that the d-normal form $\mathrm{nf}_d(\Sigma)$ of a correct expression Σ depends on the book $\mathfrak{B}$, and to be completely correct we should write $\mathrm{nf}_{d\mathfrak{B}}(\Sigma)$ instead of $\mathrm{nf}_d(\Sigma)$. We will, however, omit the subscript $\mathfrak{B}$ as long as no confusion arises.

(δ) If $\Sigma = b(\Sigma_1, \ldots, \Sigma_n)$, and $\mathfrak{B}$ contains a line $(x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$ where $\Xi_1 \in \mathcal{E}$, then $\Sigma \to_\delta \Xi_1[x_1, \ldots, x_n{:=}\Sigma_1, \ldots, \Sigma_n]$.

We say that Σ is in δ-normal form if for no expression Ω, $\Sigma \to_\delta \Omega$, and define $\twoheadrightarrow_\delta$, $\twoheadrightarrow_\delta^+$ and $=_\delta$ as usual. $\to_\delta$ depends on $\mathfrak{B}$, but as before, we drop $\mathfrak{B}$ if no confusion occurs. The relations $=_{\mathrm{d}}$ and $=_\delta$ are the same:

Lemma 2.16 *1• (Church-Rosser) If $A_1 =_\delta A_2$ then there is B such that $A_1 \to_\delta B$ and $A_2 \to_\delta B$.* *2• $\mathrm{nf}_{\mathrm{d}}(\Sigma)$ is the unique δ-normal form of Σ.* *3• $\Sigma =_\delta \Omega$ if and only if $\Sigma =_{\mathrm{d}} \Omega$.* *4• $\to_\delta$ is strongly normalising.*

Definition 2.17 • A book $\mathfrak{B}$ is part of a book $\mathfrak{B}'$, denoted as $\mathfrak{B} \subseteq \mathfrak{B}'$, if all lines of $\mathfrak{B}$ are lines of $\mathfrak{B}'$.

• A context Γ is part of a context Γ', notation $\Gamma \subseteq \Gamma'$, if all declarations $x{:}\alpha$ of Γ are declarations in Γ'.

Lemma 2.18 (Weakening) *If $\mathfrak{B}; \Gamma \vdash \Sigma : \Omega$, $\mathfrak{B} \subseteq \mathfrak{B}'$, $\Gamma \subseteq \Gamma'$ and $\mathfrak{B}'; \Gamma' \vdash$ OK then $\mathfrak{B}'; \Gamma' \vdash \Sigma : \Omega$.*

3 From AUT-68 towards a PTS λ68

To describe AUT-68 as a PTS λ68, we translate AUT-68-expressions to λ-terms:

Definition 3.1 Recall that $\mathbb{T}$ and $\mathbb{V}$ are the set of terms and variables for PTSs. We define a mapping $\overline{[\ldots]}$ from the correct expressions in $\mathcal{E}$ (relative to a book $\mathfrak{B}$ and a context Γ) to $\mathbb{T}$. We assume that $\mathcal{C} \cup \mathcal{V} \subseteq \mathbb{V}$.
• $\overline{x} \stackrel{\text{def}}{=} x$ for $x \in \mathcal{V}$; • $\overline{b(\Sigma_1, \ldots, \Sigma_n)} \stackrel{\text{def}}{=} b\overline{\Sigma_1} \cdots \overline{\Sigma_n}$; • $\overline{\langle\Omega\rangle\Sigma} \stackrel{\text{def}}{=} \overline{\Sigma}\ \overline{\Omega}$; • $\overline{\mathtt{type}} \stackrel{\text{def}}{=} *$;
• $\overline{[x{:}\Sigma]\Omega} \stackrel{\text{def}}{=} \Pi x{:}\overline{\Sigma}.\overline{\Omega}$ if $[x{:}\Sigma]\Omega$ has type $\mathtt{type}$, otherwise $\overline{[x{:}\Sigma]\Omega} \stackrel{\text{def}}{=} \lambda x{:}\overline{\Sigma}.\overline{\Omega}$;

With this translation in mind, we want to find a type system λ68 that "suits" AUT68, i.e. if Σ is a correct expression of type Ω with respect to a book $\mathfrak{B}$ and a context Γ, then we want $\mathfrak{B}', \Gamma' \vdash \overline{\Sigma} : \overline{\Omega}$ to be derivable in λ68, and vice versa. Here, $\mathfrak{B}'$ and Γ' are some suitable translations of $\mathfrak{B}$ and Γ. The search for a suitable λ68 will focus on three points: Π-formation and parameter types; constants and variables; and definitions.

3.1 The choice of the Π-formation rules and the parameter types $\P x{:}A.B$
As $\overline{\mathtt{type}} \equiv *$, Definition 2.11 clarifies which Π-rules are implied by the abstraction mechanism of AUT-68:

The rule $\dfrac{\mathfrak{B}; \Gamma \vdash \Sigma_1{:}\mathtt{type} \qquad \mathfrak{B}; \Gamma, x{:}\Sigma_1 \vdash \Omega_1{:}\mathtt{type}}{\mathfrak{B}; \Gamma \vdash [x{:}\Sigma_1]\Omega_1 : \mathtt{type}}$

translates into the PTSs Π-rule $(*, *, *)$ $\dfrac{\overline{\mathfrak{B}}, \overline{\Gamma} \vdash \overline{\Sigma_1}{:}* \qquad \overline{\mathfrak{B}}, \overline{\Gamma}, x{:}\overline{\Sigma_1} \vdash \overline{\Omega_1}{:}*}{\overline{\mathfrak{B}}, \overline{\Gamma} \vdash (\Pi x{:}\overline{\Sigma_1}.\overline{\Omega_1}) : *}$

It is, however, not immediately clear which Π-rules are induced by the parameter mechanism of AUT-68. Let $\Sigma \equiv b(\Sigma_1, \ldots, \Sigma_n)$ be a correct ex-

pression of type Ω with respect to a book $\mathfrak{B}$ and a context Γ. By Definition 2.10 there is a line $(x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$ in $\mathfrak{B}$ such that each Σ_i is a correct expression with respect to $\mathfrak{B}$ and Γ, and has a type that is definitionally equal to $\alpha_i[x_1, \ldots, x_{i-1}{:=}\Sigma_1, \ldots, \Sigma_{i-1}]$. We also know that $\Omega =_{\beta d} \Xi_2[x_1, \ldots, x_n{:=}\Sigma_1, \ldots \Sigma_n]$. Now $\overline{\Sigma} \equiv b\overline{\Sigma_1} \cdots \overline{\Sigma_n}$, and, assuming that we can derive in $\lambda 68$ that $\overline{\Sigma_i}$ has type $\overline{\alpha_i}[x_1, \ldots, x_{i-1}{:=}\overline{\Sigma_1}, \ldots, \overline{\Sigma_{i-1}}]$, it is not unreasonable to assign the type $\Pi x_1{:}\overline{\alpha_1} \cdots \Pi x_n{:}\overline{\alpha_n} to b.\overline{\Xi_2}$. We will abbreviate this last term by $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$. Then we can derive (using n times the application rule that we will introduce for $\lambda 68$) that $\overline{\Sigma}$ has type $\overline{\Omega}$ in $\lambda 68$.

It is important to notice that the type of b, $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, does not necessarily have an equivalent in AUT-68, as in AUT-68 abstractions over **type** are not allowed (only abstractions over expressions Σ that have **type** as type are possible — cf. Definition 2.11). In other words, the type of b, $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, is not necessarily a first-class citizen of AUT-68 and should therefore have special treatment in $\lambda 68$. This is the reason to create a special sort $\triangle$, in which these types of AUT-68 constants and definitions are stored. This idea originates from van Benthem Jutting and was firstly presented in [1].

If we construct $\Pi x_n{:}\overline{\alpha_n}.\overline{\Xi_2}$ from $\overline{\Xi_2}$, we must use a rule (s_1, s_2, s_3), where s_1, s_2, s_3 are sorts. Sort s_1 must be the type of $\overline{\alpha_n}$. As $\alpha_n \equiv$ **type** or α_n has type **type**, we must allow the possibilities $s_1 \equiv *$ and $s_1 \equiv \Box$. Similarly, $\Xi_2 \equiv$ **type** or Ξ_2 has type **type**, so we also allow $s_2 \equiv *$ and $s_2 \equiv \Box$. As we intended to store the new type in sort $\triangle$, we take $s_3 \equiv \triangle$.
For similar reasons, we introduce rules $(*, \triangle, \triangle)$ and $(\Box, \triangle, \triangle)$ to construct $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$ from $\Pi x_n{:}\overline{\alpha_n}.\overline{\Xi_2}$ for $n > 1$. Hence, we have the Π-rules:
$(*, *, *)$; $(*, *, \triangle)$; $(\Box, *, \triangle)$; $(*, \Box, \triangle)$; $(\Box, \Box, \triangle)$; $(*, \triangle, \triangle)$; $(\Box, \triangle, \triangle)$.

We do not have rules of the form $(\triangle, s_2, s_3)$ or $(s_1, \triangle, s_3)$ with $s_3 \equiv *$ or $s_3 \equiv \Box$. So types of sort $\triangle$ cannot be used to construct types of other sorts. In this way, we can keep the types of the λ-calculus part of AUT-68 separated from the types of the parameter mechanism: The last ones are stored in $\triangle$.

In Example 5.2.4.8 of [1], there is no rule $(*, *, \triangle)$. In principle, this rule is superfluous, as each application of rule $(*, *, \triangle)$ can be replaced by an application of rule $(*, *, *)$. Nevertheless we maintain this rule as:

- The presence of both $(*, *, *)$ and $(*, *, \triangle)$ in the system stresses the fact that AUT-68 has two type mechanisms: One provided by the parameter mechanism and one by the λ-abstraction mechanism;
- There are technical arguments to make a distinction between types formed by the abstraction mechanism and types that appear via the parameter mechanism. In this paper, we denote product types constructed by the abstraction mechanism in the usual way (so: $\Pi x{:}A.B$), whilst we will use the notation $\P x{:}A.B$ for a type constructed by the parameter mechanism. Hence, we have for the constant b above that $b : \P_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$ [9]. As an additional advantage, the resulting system will maintain Unicity of Types.

[9] we use $\P_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$ as an abbreviation for $\P x_1{:}\overline{\alpha_1} \cdots \P x_n{:}\overline{\alpha_n}.\overline{\Xi_2}$

This would have been lost if we use rules $(*,*,*)$ and $(*,*,\triangle)$ without making this difference, as we can then by these rules derive both:

$$\frac{\alpha{:}* \vdash \alpha{:}* \qquad \alpha{:}*, x{:}\alpha \vdash \alpha{:}*}{\alpha{:}* \vdash (\Pi x{:}\alpha.\alpha) : *} \text{ and } \frac{\alpha{:}* \vdash \alpha{:}* \qquad \alpha{:}*, x{:}\alpha \vdash \alpha{:}*}{\alpha{:}* \vdash (\Pi x{:}\alpha.\alpha) : \triangle}$$

3.2 The different treatment of constants and variables

When we seek to translate the AUT-68 judgement $\mathfrak{B}; \Gamma \vdash \Sigma : \Omega$ in $\lambda 68$, we must pay attention to the translation of $\mathfrak{B}$, as there is no equivalent of books in PTSs. Our solution is to store the information on identifiers of $\mathfrak{B}$ in a PTS-context. Therefore, contexts of $\lambda 68$ will have the form $\Delta; \Gamma$. The left part Δ contains type information on primitive notions and definitions, and can be seen as the translation of the information on primitive notions and definitions in $\mathfrak{B}$. The right part Γ has the usual type information on variables.

The idea to store the constant information of $\mathfrak{B}$ in the left part of the context arises naturally. Let $\mathfrak{B}$ be a correct AUT-68 book, to which we add a line $(\Gamma; b; \text{PN}; \Xi_2)$. Then $\Gamma \equiv x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$ is a correct context with respect to $\mathfrak{B}$, and $\mathfrak{B}; \Gamma \vdash \Xi_2{:}\mathtt{type}$ or $\Xi_2 \equiv \mathtt{type}$. In $\lambda 68$ we can work as follows. Assume the information on constants in $\mathfrak{B}$ has been translated into the left part Δ of a $\lambda 68$ context. We have (assuming that $\lambda 68$ is a type system that behaves like AUT-68, and writing $\overline{\Gamma}$ for the translation $x_1{:}\overline{\alpha_1}, \ldots, x_n{:}\overline{\alpha_n}$ of Γ): $\Delta; \overline{\Gamma} \vdash \overline{\Xi_2}{:}s$ ($s \equiv *$ if $\mathfrak{B}; \Gamma \vdash \Xi_2{:}\mathtt{type}$; $s \equiv \Box$ if $\Xi_2 \equiv \mathtt{type}$). Applying the $\P$-formation rule n times, we obtain $\Delta; \varnothing \vdash \P\overline{\Gamma}.\overline{\Xi_2} : \triangle$ (If Γ is the empty context, then $\P\overline{\Gamma}.\overline{\Xi_2} \equiv \overline{\Xi_2}$, and $\overline{\Xi_2}$ has type $*$ or $\Box$ instead of $\triangle$. We write $\P\overline{\Gamma}$ for $\P_{i=1}^n x_i{:}\overline{\alpha_i}$). As $\P\overline{\Gamma}.\overline{\Xi_2}$ is exactly the type that we want to give to b (see the discussion in Subsection 3.1), we use this statement as premise for the start rule that introduces b. As the right part $\overline{\Gamma}$ of the original context has disappeared when we applied the $\P$-formation rules, $b{:}\P\overline{\Gamma}.\overline{\Xi_2}$ is automatically placed at the righthand end of Δ: The conclusion of the start rule is $\Delta, b{:}\P\overline{\Gamma}.\overline{\Xi_2} \vdash b{:}\P\overline{\Gamma}.\overline{\Xi_2}$. Adding $b{:}\P\overline{\Gamma}.\overline{\Xi_2}$ at the end of Δ can be compared with adding the line $(\Gamma; b; \text{PN}; \Xi_2)$ at the end of $\mathfrak{B}$.

This process can be captured by rule: $\dfrac{\Delta; \overline{\Gamma} \vdash \overline{\Xi_2}{:}s_1 \qquad \Delta; \vdash \P\overline{\Gamma}.\overline{\Xi_2}{:}s_2}{\Delta, b{:}\P\overline{\Gamma}.\overline{\Xi_2}; \vdash b{:}\P\overline{\Gamma}.\overline{\Xi_2}}$.
Here $s_1 \in \{*, \Box\}$ (compare: $\Xi_2{:}\mathtt{type}$ or $\Xi_2 \equiv \mathtt{type}$) and $s_2 \in \{*, \Box, \triangle\}$ (usually, $s_2 \equiv \triangle$; the cases $s_2 \equiv *, \Box$ only occur if Γ is empty).

3.3 The definition system and the translation using §

A line $(x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$, in which b is a constant and $\Xi_1 \in \mathcal{E}$, represents the definition: "For all expressions $\Omega_1, \ldots, \Omega_n$ (obeying some type conditions), $b(\Omega_1, \ldots, \Omega_n)$ abbreviates $\Xi_1[x_1, \ldots, x_n{:=}\Omega_1, \ldots, \Omega_n]$, and has type $\Xi_2[x_1, \ldots, x_n{:=}\Omega_1, \ldots, \Omega_n]$." So in $\lambda 68$, the context should have $bX_1 \cdots X_n$ "is equal to" $\Xi_1[x_1, \ldots, x_n{:=}X_1, \ldots, X_n]$, for all terms $X_1, \ldots, X_n$. This can be done by writing $b{:=}\left(\lambda_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\P_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right)$ in the context instead of only $b{:}\P_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, and adding a δ-reduction rule which unfolds the definition

of b: $\Delta \vdash b \to_\delta \lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}$ whenever $b{:=}\left(\lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\P^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right) \in \Delta$. Unfolding the definition of b in a term $b\overline{\Sigma_1}\cdots\overline{\Sigma_n}$ and applying β-reduction n times gives $\overline{\Xi_1}[x_1{:=}\overline{\Sigma_1}]\cdots[x_n{:=}\overline{\Sigma_n}]$. In AUT-68 [10], this corresponds to $\Delta \vdash b(\Sigma_1,\ldots,\Sigma_n) \to_\delta \Xi_1[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]$.

This method, however, has disadvantages:

- In the AUT-68 line $(x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$, $b(\Sigma_1,\ldots,\Sigma_n)$ has $b\overline{\Sigma_1}\cdots\overline{\Sigma_n}$ as its equivalent in λ68. If $n > 0$, the latter λ68-term has $B \equiv b\overline{\Sigma_1}\cdots\overline{\Sigma_m}$ as a subterm for any $m < n$. But B has no equivalent in AUT-68: Only after B is applied to suitable terms $\overline{\Sigma_{m+1}},\ldots,\overline{\Sigma_n}$ the result $B\overline{\Sigma_{m+1}}\cdots\overline{\Sigma_n}$ has $b(\Sigma_1,\ldots,\Sigma_n)$ as its equivalent in AUT-68. Hence B must not be seen as a term directly translatable into AUTOMATH, but only as an intermediate result necessary to construct the equivalent of $b(\Sigma_1,\ldots,\Sigma_n)$. B is recognisable as an intermediate result via its type $\P^n_{i=m+1} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, of sort Δ (not $*$ or $\Box$).

  The method above allows to unfold the definition of b in B, because $b\overline{\Sigma_1}\cdots\overline{\Sigma_m}$ can reduce to $\left(\lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right)\overline{\Sigma_1}\cdots\overline{\Sigma_m}$, and we can β-reduce this term m times to $\left(\lambda^n_{i=m+1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right)[x_j{:=}\overline{\Sigma_j}]^m_{j=1}$. In AUT-68 such unfolding is not possible before *all* n arguments $\overline{\Sigma_1},\ldots,\overline{\Sigma_n}$ are applied to b, so only when the construction of the equivalent of $b(\Sigma_1,\ldots,\Sigma_n)$ has been completed;
- $\lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}$ does not necessarily have an equivalent in AUT-68. Consider for instance the constant b in the line $(\alpha{:}\mathtt{type}; b; [x{:}\alpha]x; [x{:}\alpha]\alpha)$. In this case, $\lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1} \equiv \lambda\alpha{:}*.\lambda x{:}\alpha.x$. Its equivalent in AUT-68 is $[\alpha{:}\mathtt{type}][x{:}\alpha]x$, but an abstraction $[\alpha{:}\mathtt{type}]$ cannot be made in AUT-68. [11] This is the reason why we do not incorporate $\lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}$ as a citizen of λ68.

Hence we choose another translation. The line $(x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$, where $\Xi_1 \in \mathcal{E}$, is translated by taking $b{:=}\left(\S^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\P^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right)$ instead of $b{:=}\left(\lambda^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\P^n_{i=1} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right)$ in the left part of the context. A reduction rule $bX_1\cdots X_n \to_\delta \overline{\Xi_1}[x_1,\ldots,x_n{:=}X_1,\ldots,X_n]$ is added for all terms $X_1,\ldots,X_n$. We use § instead of λ to emphasise that, though both $\S x{:}A$ and $\lambda x{:}A$ are abstractions, they are not the same kind of abstraction.

4 λ68

Here, we give λ68, show that it has the desirable properties of PTSs and that it is the PTS version of AUT-68.

Definition 4.1 (λ68)

(i) Terms of λ68 are given by $\mathcal{T} ::= \mathcal{V} \mid \mathcal{C} \mid \boldsymbol{S} \mid \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \S\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \P\mathcal{V}{:}\mathcal{T}.\mathcal{T}$, where $\boldsymbol{S}$ is the set of sorts $\{*, \Box, \Delta\}$. Free variables

[10] We can assume that the x_i do not occur in the Σ_j, so the simultaneous substitution $\Xi_1[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]$ is equal to $\Xi_1[x_1{:=}\Sigma_1]\cdots[x_n{:=}\Sigma_n]$.

[11] Compare with the situation of Section 3.1, where we found that the type of b is not necessarily a first-class citizen of AUT-68. There, we could not avoid that the type of b became a citizen of λ68 (though we made it second-class by storing it in the sort Δ).

FV(T) and "free" constants FC(T) of term T are defined as usual;

(ii) We define the notion of context inductively:
- $\varnothing;\varnothing$ is a context; DOM $(\varnothing;\varnothing) = \varnothing$;
- If $\Delta;\Gamma$ is a context, $x \in \mathcal{V}$, x does not occur in $\Delta;\Gamma$ and $A \in \mathcal{T}$, then $\Delta;\Gamma, x{:}A$ is a context (x is a newly introduced variable); DOM $(\Delta;\Gamma) =$ DOM $(\Delta;\Gamma) \cup \{x\}$;
- If $\Delta;\Gamma$ is a context, $b \in \mathcal{C}$, b does not occur in $\Delta;\Gamma$ and $A \in \mathcal{T}$ then $\Delta, b{:}A;\Gamma$ is a context (in this case b is a *primitive* constant; DOM $(\Delta, b{:}A;\Gamma) =$ DOM $(\Delta;\Gamma) \cup \{b\}$;
- If $\Delta;\Gamma$ is a context, $b \in \mathcal{C}$, b does not occur in $\Delta;\Gamma$, $A \in \mathcal{T}$, and $T \in \mathcal{T}$, then $\Delta, b{:=}T{:}A;\Gamma$ is a context (in this case b is a *defined* constant; DOM $(\Delta, b{:=}T{:}A;\Gamma) =$ DOM $(\Delta;\Gamma) \cup \{b\}$.

PRIMCONS $(\Delta;\Gamma) = \{b \in$ DOM $(\Delta;\Gamma) \mid b$ is a primitive constant$\}$; FV$(\Delta;\Gamma) =$ DOM $(;\Gamma)$ and DEFCONS $(\Delta;\Gamma) = \{b \in$ DOM $(\Delta;\Gamma) \mid b$ is a defined constant$\}$.

(iii) We define δ-reduction on terms. Let Δ be the left part of a context. If $(b{:=}(\S_{i=1}^{n}\, x_i{:}A_i.T) : (\P_{i=1}^{n}\, x_i{:}A_i.B)) \in \Delta$, and B is not $\P y{:}B_1.B_2$, then $\Delta \vdash bX_1 \cdots X_n \to_\delta T[x_1,\ldots,x_n{:=}X_1,\ldots,X_n]$ for all $X_1,\ldots X_n \in \mathcal{T}$.

We also have the usual compatibility rules on δ-reduction. We use notations like $\twoheadrightarrow_\delta, \twoheadrightarrow_\delta^+, =_\delta$ as usual. If no confusion about which Δ occurs, we simply write $bX_1 \cdots X_n \to_\delta T[x_1,\ldots,x_n{:=}X_1,\ldots,X_n]$;

(iv) We use the usual notion of β-reduction;

(v) Judgements in $\lambda 68$ have the form $\Delta;\Gamma \vdash A : B$, where $\Delta;\Gamma$ is a context and A and B are terms. If a judgement $\Delta;\Gamma \vdash A : B$ is derivable according to the rules below, then $\Delta;\Gamma$ is a *legal* context and A and B are *legal* terms. We write $\Delta;\Gamma \vdash A : B : C$ if both $\Delta;\Gamma \vdash A : B$ and $\Delta;\Gamma \vdash B : C$ are derivable in $\lambda 68$. The rules for $\lambda 68$ are given in Figure v (v, pc, and dc are shorthand for variable, primitive constant, and defined constant, resp.). The newly introduced variables in the Start-rules and Weakening-rules are assumed to be fresh. Moreover, when introducing a variable x with a "pc"-rule or a "dc"-rule, we assume $x \in \mathcal{C}$, and when introducing x via a "v"-rule, we assume $x \in \mathcal{V}$. We write $\Delta;\Gamma \vdash_{\lambda 68} A : B$ instead of $\Delta;\Gamma \vdash A : B$ if the latter gives rise to confusion.

Notice the lack of rule (§) as we do not want that terms of the form $\S\, x{:}A.B$ be first-class citizens of $\lambda 68$: they do not have an equivalent in AUTOMATH.

Example 4.2 The translation of Example 2.9 into $\lambda 68$ is given in Figure 4. [12] We see that all variable declarations of the original book have disappeared in the translation. In the original book, they do not add any new knowledge but are only used to construct contexts. In our translation, this happens in the right part of the context, instead of the left part.

[12] Because of the habit in computer science to use more than one digit for a variable, we have to write additional brackets around subterms like `proof` to keep things unambiguous.

$$\textbf{(Axiom)} \qquad ;\vdash * : \Box$$

$$\textbf{(Start : v)} \qquad \frac{\Delta;\Gamma \vdash A : s}{\Delta;\Gamma, x{:}A \vdash x : A} \qquad s \equiv *, \Box$$

$$\textbf{(Start : pc)} \qquad \frac{\Delta;\Gamma \vdash B : s_1 \qquad \Delta;\vdash \P\,\Gamma.B : s_2}{\Delta, b{:}\P\,\Gamma.B;\vdash b : \P\,\Gamma.B} \qquad s_1 \equiv *, \Box$$

$$\textbf{(Start : dc)} \qquad \frac{\Delta;\Gamma \vdash T : B : s_1 \qquad \Delta;\vdash \P\,\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.B);\vdash b : \P\,\Gamma.B} \qquad s_1 \equiv *, \Box$$

$$\textbf{(Weak : v)} \qquad \frac{\Delta;\Gamma \vdash M : N \qquad \Delta;\Gamma \vdash A : s}{\Delta;\Gamma, x{:}A \vdash M : N} \qquad s \equiv *, \Box$$

$$\textbf{(Weak : pc)} \qquad \frac{\Delta;\vdash M : N \qquad \Delta;\Gamma \vdash B : s_1 \qquad \Delta;\vdash \P\,\Gamma.B : s_2}{\Delta, b{:}\P\,\Gamma.B;\vdash M : N} \qquad s_1 \equiv *, \Box$$

$$\textbf{(Weak : dc)} \qquad \frac{\Delta;\vdash M : N \qquad \Delta;\Gamma \vdash T : B : s_1 \qquad \Delta;\vdash \P\,\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.B);\vdash M : N} \qquad s_1 \equiv *, \Box$$

$$(\Pi - \textbf{form}) \qquad \frac{\Delta;\Gamma \vdash A : * \qquad \Delta;\Gamma, x{:}A \vdash B : *}{\Delta;\Gamma \vdash (\Pi x{:}A.B) : *}$$

$$(\P - \textbf{form}) \qquad \frac{\Delta;\Gamma \vdash A : s_1 \qquad \Delta;\Gamma, x{:}A \vdash B : s_2}{\Delta;\Gamma \vdash (\P x{:}A.B) : \Delta} \qquad s_1 \equiv *, \Box$$

$$(\lambda) \qquad \frac{\Delta;\Gamma \vdash \Pi x{:}A.B : * \qquad \Delta;\Gamma, x{:}A \vdash F : B}{\Delta;\Gamma \vdash (\lambda x{:}A.F) : (\Pi x{:}A.B)}$$

$$(\textbf{App}_1) \qquad \frac{\Delta;\Gamma \vdash M : \Pi x{:}A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x{:=}N]}$$

$$(\textbf{App}_2) \qquad \frac{\Delta;\Gamma \vdash M : \P x{:}A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x{:=}N]}$$

$$(\textbf{Conv}) \qquad \frac{\Delta;\Gamma \vdash M : A \qquad \Delta;\Gamma \vdash B : s \qquad \Delta \vdash A =_{\beta\delta} B}{\Delta;\Gamma \vdash M : B}$$

Fig. 3. Rules of $\lambda 68$

Lemma 4.3 (Free Variable Lemma)
For $\Delta;\Gamma \vdash M : N$, $\Delta \equiv b_1{:}B_1, \cdots, b_m{:}B_m$ and $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ [13] *:*

- *The $b_1, \ldots, b_m \in \mathcal{C}$ and $x_1, \ldots, x_n \in \mathcal{V}$ are all distinct;*
- $\text{FC}(M), \text{FC}(N) \subseteq \{b_1, \ldots, b_m\}$; $\text{FV}(M), \text{FV}(N) \subseteq \{x_1, \ldots, x_n\}$;
- *$b_1{:}B_1, \ldots, b_{i-1}{:}B_{i-1};\vdash B_i{:}s_i$ for $s_i \in \{*, \Box, \Delta\}$; and $\Delta; x_1{:}A_1, \ldots, x_{j-1}{:}A_{j-1} \vdash A_j{:}t_j$ for $t_j \in \{*, \Box\}$.*

Lemma 4.4 • **(Start)** *Let $\Delta;\Gamma$ be a legal context. Then $\Delta;\Gamma \vdash * : \Box$, and if $b{:}A \in \Delta;\Gamma$, or $c{:=}T{:}A \in \Delta$, then $\Delta;\Gamma \vdash c : A$.*

- **(Definition)** *Let $\Delta_1, b{:=}(\S_{i=1}^n\, x_i{:}A_i.T) : (\P_{i=1}^n\, x_i{:}A_i.B), \Delta_2;\Gamma \vdash M : N$, where $B \not\equiv \P y{:}B_1.B_2$. Then $\Delta_1; x_1{:}A_1, \ldots, x_n{:}A_n \vdash T : B : s$ for $s \in \{*, \Box\}$.*

[13] In Δ, also expressions $b_i{:=}T_i{:}B_i$ may occur, but for uniformity we leave out the $:=T_i$-part.

```
  prop   :  *,
   and   :  ¶x:prop.¶y:prop.prop,
 proof   :  ¶x:prop.*,
 and-I   :  ¶x:prop.¶y:prop.¶px:(proof)x.¶py:(proof)y.(proof)((and)xy),
and-01   :  ¶x:prop.¶y:prop.¶pxy:(proof)((and)xy).(proof)x,
and-02   :  ¶x:prop.¶y:prop.¶pxy:(proof)((and)xy).(proof)y,
 and-R   :=  §x:prop.§prx :(proof)x.(and-I)xx(prx)(prx) :
                ¶x:prop.¶prx:(proof)x.(proof)((and)xx),
 and-S   :=  §x:prop.§y:prop.§pxy:(proof)((and)xy).
                (and-I)yx((and-02)xy(pxy))((and-01)xy(pxy))
                :¶x:prop.¶y:prop.¶pxy:(proof)((and)xy).(proof)((and)yx)
```

Fig. 4. Translation of Example 2.9

Definition 4.5 We define: $\Delta_1;\Gamma_1 \vdash \Delta_2;\Gamma_2$ if and only if
- If $b{:}A \in \Delta_2;\Gamma_2$ then $\Delta_1;\Gamma_1 \vdash b{:}A$; • If $b{:=}T{:}A \in \Delta_2$ then $\Delta_1;\Gamma_1 \vdash b{:}A$;
- If $b{:=}(\S_{i=1}^n x_i : A_i.U){:}B \in \Delta_2$ and $U \not\equiv \S\, y{:}B.A'$ then $\Delta_1 \vdash bx_1 \cdots x_n =_{\beta\delta} U$.

Lemma 4.6 • **(Transitivity)** *Assume* $\Delta_1;\Gamma_1 \vdash \Delta_2;\Gamma_2$ *and* $\Delta_2;\Gamma_2 \vdash B : C$. *Then* $\Delta_1;\Gamma_1 \vdash B : C$.

- **(Substitution)** *If* $\Delta;\Gamma_1, x{:}A, \Gamma_2 \vdash B : C$ *and* $\Delta;\Gamma_1 \vdash D : A$ *then* $\Delta;\Gamma_1, \Gamma_2[x{:=}D] \vdash B[x{:=}D] : C[x{:=}D]$.
- **(Thinning)** *Let* $\Delta_1;\Gamma_1$ *be a legal context, and let* $\Delta_2;\Gamma_2$ *be a legal context such that* $\Delta_1 \subseteq \Delta_2$ *and* $\Gamma_1 \subseteq \Gamma_2$. *Then* $\Delta_1;\Gamma_1 \vdash A : B \Rightarrow \Delta_2;\Gamma_2 \vdash A : B$.

Lemma 4.7 (Generation Lemma)
- *If* $x \in \mathcal{V}$ *and* $\Delta;\Gamma \vdash x{:}C$ *then* $\exists s \in \{*, \Box\}$ *and* $B =_{\beta\delta} C$ *such that* $\Delta;\Gamma \vdash B : s$ *and* $x{:}B \in \Gamma$;
- *If* $b \in \mathcal{C}$ *and* $\Delta;\Gamma \vdash b{:}C$ *then* $\exists s \in \boldsymbol{S}$ *and* $B =_{\beta\delta} C$ *such that* $\Delta;\Gamma \vdash B : s$, *and either* $b{:}B \in \Delta$ *or* $\exists T$ *such that* $b{:=}T{:}B \in \Delta$;
- *If* $s \in \boldsymbol{S}$ *and* $\Delta;\Gamma \vdash s{:}C$ *then* $s \equiv *$ *and* $C =_{\beta\delta} \Box$;
- *If* $\Delta;\Gamma \vdash MN : C$ *then* $\exists A, B$ *such that* $\Delta;\Gamma \vdash M : (\Pi x{:}A.B)$ *or* $\Delta;\Gamma \vdash M : (\P x{:}A.B)$, *and* $\Delta;\Gamma \vdash N{:}A$ *and* $C =_{\beta\delta} B[x{:=}N]$;
- *If* $\Delta;\Gamma \vdash (\lambda x{:}A.b) : C$ *then* $\exists B$ *such that* $\Delta;\Gamma \vdash (\Pi x{:}A.B) : *$, $\Delta;\Gamma, x{:}A \vdash b : B$ *and* $C =_{\beta\delta} \Pi x{:}A.B$;
- *If* $\Delta;\Gamma \vdash (\Pi x{:}A.B) : C$ *then* $C =_{\beta\delta} *$, $\Delta;\Gamma \vdash A{:}*$ *and* $\Delta;\Gamma, x{:}A \vdash B{:}*$;
- *If* $\Delta;\Gamma \vdash (\P x{:}A.B) : C$ *then* $C =_{\beta\delta} \triangle$, $\Delta;\Gamma \vdash A{:}s_1$ *for* $s_1 \in \{*, \Box\}$, *and* $\Delta;\Gamma, x{:}A \vdash B{:}s_2$ *for* $s_2 \in \{*, \Box, \triangle\}$.

Lemma 4.8 • **(Unicity of Types)** *If* $\Delta;\Gamma \vdash A : B_1$ *and* $\Delta;\Gamma \vdash A : B_2$ *then* $B_1 =_{\beta\delta} B_2$.

- **(Correctness of Types)** *If* $\Delta;\Gamma \vdash A : B$ *then there is* $s \in \boldsymbol{S}$ *such that* $B \equiv s$ *or* $\Delta;\Gamma \vdash B : s$.

- *If* $\Delta;\Gamma \vdash A : (\Pi x{:}B_1.B_2)$ *then* $\Delta;\Gamma \vdash B_1 : *$; *and* $\Delta;\Gamma, x{:}B_1 \vdash B_2 : *$.
- *If* $\Delta;\Gamma \vdash A : (\P x{:}B_1.B_2)$ *then* $\Delta;\Gamma \vdash B_1 : s_1$ *for* $s_1 \in \{*, \Box\}$; *and* $\Delta;\Gamma, x{:}B_1 \vdash B_2{:}s_2$ *for some* s_2.

In order to show some properties of the reduction relations $\to_\beta$, $\to_\delta$ and $\to_{\beta\delta}$ and as δ-reduction also depends on books, we first have to give a translation of AUT-68 books and AUT-contexts to λ68-contexts:

Definition 4.9 • Let Γ be a AUT-68-context $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$. Then $\overline{\Gamma} \stackrel{\text{def}}{=} x_1{:}\overline{\alpha_1}, \ldots, x_n{:}\overline{\alpha_n}$.

• Let $\mathfrak{B}$ be a book. We define the left part $\overline{\mathfrak{B}}$ of a context in λ68:

- $\overline{\varnothing} \stackrel{\text{def}}{=} \varnothing$;
- $\overline{\mathfrak{B}, (\Gamma; x; —; \Omega)} \stackrel{\text{def}}{=} \overline{\mathfrak{B}}$;
- $\overline{\mathfrak{B}, (\Gamma; b; \text{PN}; \Omega)} \stackrel{\text{def}}{=} \overline{\mathfrak{B}}, b{:}\P\,\overline{\Gamma}.\overline{\Omega}$;
- $\overline{\mathfrak{B}, (\Gamma; b; \Sigma; \Omega)} \stackrel{\text{def}}{=} \overline{\mathfrak{B}}, b{:=}\S\,\overline{\Gamma}.\overline{\Sigma}{:}\P\,\overline{\Gamma}.\overline{\Omega}$.

Lemma 4.10 *Assume,* Σ *is a correct expression with respect to a book* $\mathfrak{B}$.
- *1.* $\Sigma \to_\beta \Sigma'$ *if and only if* $\overline{\Sigma} \to_\beta \overline{\Sigma'}$;
- *2.* $\mathfrak{B} \vdash_{AUT\text{-}68} \Sigma \to_\delta \Sigma'$ *if and only if* $\overline{\mathfrak{B}} \vdash_{\lambda 68} \overline{\Sigma} \to_\delta \overline{\Sigma'}$.

Theorem 4.11 (Church-Rosser for $\to_{\beta\delta}$) *Let* Δ *be the left part of a context in which* M *is typable. If* $\Delta \vdash M \twoheadrightarrow_{\beta\delta} N_1$ *and* $\Delta \vdash M \twoheadrightarrow_{\beta\delta} N_2$ *then there is* P *such that* $\Delta \vdash N_1 \twoheadrightarrow_{\beta\delta} P$ *and* $\Delta \vdash N_2 \twoheadrightarrow_{\beta\delta} P$.

Lemma 4.12 (Subject Reduction) *Let* $\Delta;\Gamma \vdash A : B$.
1. If $A \to_\beta A'$ *then* $\Delta;\Gamma \vdash A' : B$. *2.* $A \to_\delta A'$ *then* $\Delta;\Gamma \vdash A' : B$. *3. If* $A \twoheadrightarrow_{\beta\delta} A'$ *then* $\Delta;\Gamma \vdash A' : B$.

Lemma 4.13 *Assume* $s \in \boldsymbol{S}$ *and* M *legal. Then* $(\Delta \vdash M =_{\beta\delta} s) \Rightarrow M \equiv s$.

Theorem 4.14 (Strong Normalisation) λ68 *is* $\beta\delta$*-strongly normalising.*

The next two theorems formally relate AUT-68 and λ68.

Theorem 4.15 *Let* $\mathfrak{B}$ *be an* AUTOMATH *book and* Γ *an* AUTOMATH *context.*
- *If* $\mathfrak{B};\Gamma \vdash_{AUT\text{-}68}$ OK *then* $\overline{\mathfrak{B}};\overline{\Gamma}$ *is legal;*
- *If* $\mathfrak{B};\Gamma \vdash_{AUT\text{-}68} \Sigma : \Omega$ *then* $\overline{\mathfrak{B}};\overline{\Gamma} \vdash_{\lambda 68} \overline{\Sigma} : \overline{\Omega}$.

Theorem 4.16 *Let* $\Delta;\Gamma \vdash_{\lambda 68} M : N$. *There is an* AUTOMATH *book* $\mathfrak{B}$ *and an* AUTOMATH *context* Γ' *such that* $\mathfrak{B};\Gamma' \vdash_{AUT\text{-}68}$ OK, *and* $\overline{\mathfrak{B}}, \overline{\Gamma'} \equiv \Delta;\Gamma$. *Also,*

(i) *If* $N \equiv \Box$ *then* $M \equiv *$;

(ii) *If* $\Delta;\Gamma \vdash_{\lambda 68} N : \Box$ *then* $N \equiv *$ *and there is* $\Omega \in \mathcal{E}$ *such that* $\overline{\Omega} \equiv M$ *and* $\mathfrak{B};\Gamma' \vdash_{AUT\text{-}68} \Omega : \mathtt{type}$;

(iii) *If* $N \equiv \triangle$ *then there is* $\Gamma'' \equiv x_1{:}\Sigma_1, \ldots, x_n{:}\Sigma_n$, $\Omega \in \mathcal{E} \cup \{\mathtt{type}\}$ *with:* • Γ', Γ'' *is correct with respect to* $\mathfrak{B}$; • $M \equiv \P\,\overline{\Gamma''}.\overline{\Omega}$; • $\Omega \equiv \mathtt{type}$ *or* $\mathfrak{B};\Gamma' \vdash_{AUT\text{-}68} \Omega : \mathtt{type}$;

(iv) *If* $\Delta;\Gamma \vdash_{\lambda 68} N : \triangle$ *then there are* $b \in \mathcal{C}$ *and* $\Sigma_1, \ldots, \Sigma_n \in \mathcal{E}$ *such that* $M \equiv b\overline{\Sigma_1} \cdots \overline{\Sigma_n}$. *Moreover,* $\mathfrak{B}$ *contains a line* $(x_1{:}\Omega_1, \ldots, x_m{:}\Omega_m; b; \Xi_1; \Xi_2)$ *such that:* • $N \equiv \left(\P_{i=n+1}^{m}\, x_i{:}\overline{\Omega_i}.\overline{\Xi_2}\right)[x_1, \ldots, x_n{:=}\overline{\Sigma_1}, \ldots, \overline{\Sigma_n}]$; • $m > n$; • $\mathfrak{B};\Gamma' \vdash_{AUT\text{-}68} \Sigma_i{:}\Omega_i[x_1, \ldots, x_{i-1}{:=}\Sigma_1, \ldots, \Sigma_{i-1}]$ $(1 \le i \le n)$;

(v) *If* $N \equiv *$ *then there is* $\Omega \in \mathcal{E}$ *where* $\overline{\Omega} \equiv M$ *and* $\mathfrak{B}; \Gamma' \vdash_{AUT\text{-}68} \Omega : \mathtt{type}$;

(vi) *If* $\Delta; \Gamma \vdash_{\lambda 68} N : *$ *then there are* $\Sigma, \Omega \in \mathcal{E}$ *such that* $\overline{\Sigma} \equiv M$ *and* $\overline{\Omega} \equiv N$, *and* $\mathfrak{B}; \Gamma' \vdash_{AUT\text{-}68} \Sigma : \Omega$, *and* $\mathfrak{B}; \Gamma' \vdash_{AUT\text{-}68} \Omega : \mathtt{type}$.

5 Conclusion

The system AUT-68 is one of several AUTOMATH-systems. Another frequently used system is AUT-QE. We shall briefly compare AUT-68 to AUT-QE and describe how we can easily adapt $\lambda 68$ to a system λQE. The system AUT-QE has many similarities with AUT-68 but differs on the following extensions:

(i) In AUT-QE we can also form the abstraction expression $[x{:}\Sigma]\mathtt{type}$ (thus extending Definition 2.5);

(ii) Inhabitants of types $[x{:}\Sigma]\mathtt{type}$ are introduced in AUT-QE by extending abstraction rules 1 and 2 of Definition 2.11 with the AUT-QE rule: $\dfrac{\mathfrak{B}; \Gamma \vdash \Sigma_1 : \mathtt{type} \qquad \mathfrak{B}; \Gamma, x{:}\Sigma_1 \vdash \Sigma_2 : \mathtt{type}}{\mathfrak{B}; \Gamma \vdash [x{:}\Sigma_1]\Sigma_2 : [x{:}\Sigma_1]\mathtt{type}}$. Like type, $[x{:}\Sigma_1]\mathtt{type}$ is not typable. In a translation to a PTS, these expressions should get type $\Box$;

(iii) In AUT-QE, there is a new reduction $\to_{\mathrm{QE}}$ on expressions, given by the rule $[x_1{:}\Sigma_1] \cdots [x_n{:}\Sigma_n][y{:}\Omega]\mathtt{type} \to_{\mathrm{QE}} [x_1{:}\Sigma_1] \cdots [x_n{:}\Sigma_n]\mathtt{type}$ (for $n \geq 0$).

The first two rules are straightforward. They correspond to an extension of $\lambda{\to}$ to λP in PTSs. It is easy to extend $\lambda 68$ with similar rules; just add the Π-formation rule $(*, \Box, \Box)$: $\dfrac{\Delta; \Gamma \vdash A : * \qquad \Delta; \Gamma, x{:}A \vdash B : \Box}{\Delta; \Gamma \vdash (\Pi x{:}A.B) : \Box}$. The third rule is unusual. It is needed because AUT-QE does not distinguish λs and Πs. In AUT-68 this did not matter, as we could always derive whether $[x{:}\Sigma]\Omega$ should be interpreted as $\lambda x{:}\Sigma.\Omega$ or as $\Pi x{:}\Sigma.\Omega$. The latter should have type type, and the first should not have type type. Though $\lambda 68$ does not have Π-conversion, it is easy to extend it to a system $\lambda\Pi 68$ following the lines of [18] by:

- Changing rule (App_1) into $\dfrac{\Delta; \Gamma \vdash M : \Pi x{:}A.B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash MN : (\Pi x{:}A.B)N}$;
- Adding a new reduction rule $\to_\Pi$ by $(\Pi x{:}A.B)N \to_\Pi B[x{:=}N]$.

In this paper we described the most basic AUTOMATH-system, AUT-68, in a PTS style. Though an attempt at such a description has been given before in [1,15], we feel our description is more accurate and unlike [1,15], pays attention to the definition and parameter systems, which are crucial in AUTOMATH. We provided a PTS called $\lambda 68$ which we showed to be the system AUT-68 written as a PTS. Although $\lambda 68$ does not include Π-conversion (while AUTOMATH does), it is easy to adapt $\lambda 68$ to include Π-conversion following the lines of [18].

The adaptation of $\lambda 68$ to a system λQE, representing the AUTOMATH-system AUT-QE is not hard, either: It requires adapting the Π-formation rule to include not only the rule $(*, *, *)$ but also $(*, \Box, \Box)$ and the introduction of the additional reduction rule of type inclusion. We leave this as a future work.

There is no doubt that AUTOMATH has had an amazing influence in theorem proving, type theory and logical frameworks. AUTOMATH however, was

developed independently from other developments in type theory and uses a λ-calculus and type-theoretical style that is unique to AUTOMATH. Writing AUTOMATH in the modern style of type theory will enable useful comparisons between type systems to take place. There are still many lessons to learn from AUTOMATH and writing it in modern style is a useful step in this direction.

When comparing $\lambda 68$ to other type systems with definitions, we find an important difference. In $\lambda 68$, the correspondence between types of definiendum and definiens differs from that of the systems in [26,18]. AUTOMATH allows *parameters* to occur in the definiens, and there is no parameter mechanism in the PTSs of [1,26,18] althrough this mechansim exists in [22,19,20].

References

[1] H.P. Barendregt. λ-calculi with types. In *Handbook of Logic in Computer Science*, pages 117 309. OUP, 1992.

[2] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* PhD thesis, Eindhoven University of Technology, 1977. Published as Mathematical Centre Tracts nr. 83, (Amsterdam 1979).

[3] L.S. van Benthem Jutting. Description of AUT-68. Technical Report 12, Eindhoven University of Technology, 1981. Also in [24], pp. 251–273.

[4] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.

[5] N.G. de Bruijn. AUTOMATH, a language for mathematics. Technical Report 68-WSK-05, T.H.-Reports, Eindhoven University of Technology, 1968.

[6] N.G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In M. Laudet, D. Lacombe, and M. Schuetzenberger, editors, *Symposium on Automatic Demonstration*, pages 29–61, IRIA, Versailles, 1968. Springer Verlag, Berlin, 1970. Lecture Notes in Mathematics **125**; also in [24].

[7] N.G. de Bruijn. The Mathematical Vernacular, a language for mathematics with typed sets. In P. Dybjer et al., editors, *Proceedings of the Workshop on Programming Languages.* Marstrand, Sweden, 1987. Reprinted in [24].

[8] N.G. de Bruijn. Reflections on Automath. Eindhoven University of Technology, 1990. Also in [24], pages 201–228.

[9] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[10] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New Jersey, 1986.

[11] H.B. Curry and R. Feys. *Combinatory Logic I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[12] D.T. van Daalen. *The Language Theory of Automath.* PhD thesis, Eindhoven University of Technology, 1980.

[13] G. Dowek et al. The Coq Proof Assistant Version 5.6, Users Guide. Technical Report 134, INRIA, Le Chesney, 1991.

[14] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Nebert, Halle, 1879. Also in [16], pages 1–82.

[15] J.H. Geuvers. *Logics and Type Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[16] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, Massachusetts, 1967.

[17] W.A. Howard. The formulas-as-types notion of construction. In [25], pages 479–490, 1980.

[18] F. Kamareddine, R. Bloo, and R. Nederpelt. On π-conversion in the λ-cube and the combination with abbreviations. *Annals of Pure and Applied Logics*, 97:27–45, 1999.

[19] F. Kamareddine, L. Laan, and R.P. Nederpelt. Refining the Barendregt cube using parameters. *Fifth International Symposium on Functional and Logic Programming, FLOPS 2001*, LNCS 2024:375 389, 2001.

[20] F. Kamareddine, L. Laan, and R.P. Nederpelt. Revisiting the notion of function. *Algebraic and Logic Programming*, 54:65–107, 2003.

[21] F. Kamareddine and R.P. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.

[22] T. Laan. *The Evolution of Type Theory in Logic and Mathematics.* PhD thesis, Eindhoven University of Technology, 1997.

[23] E. Landau. *Grundlagen der Analysis.* Leipzig, 1930.

[24] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath.* Studies in Logic and the Foundations of Mathematics **133**. North-Holland, Amsterdam, 1994.

[25] J.P. Seldin and J.R. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980.

[26] P. Severi and E. Poll. Pure type systems with definitions. In A. Nerode and Yu.V. Matiyasevich, editors, *Proceedings of LFCS'94 (LNCS* **813***)*, pages 316–328, New York, 1994. LFCS'94, St. Petersburg, Russia, Springer.

[27] J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.

[28] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume I, II, III. Cambridge University Press, 1910, 1912, 1913[1], 1925, 1925, 1927[2].

[29] J. Zucker. Formalization of classical mathematics in Automath. In *Colloque International de Logique*, Clermont-Ferrand, pages 135–145, Paris, CNRS, 1977. Colloques Internationaux du Centre National de la Recherche Scientifique, 249.

Tactics and Parameters

Gueorgui Jojgov [1]

Faculteit Wiskunde en Informatica
Technische Universiteit Eindhoven
The Netherlands

Abstract

In this paper we discuss the problem of internalizing the meta-level transformations between (representations of) incomplete proofs and terms in a theorem prover based on Type Theory. These transformations (usually referred to as *tactics*) can be seen as meta-level functions between terms representing the state of the theorem prover. Starting with parameterized variables as representations of unknown terms, we propose an extension of the Pure Type Systems (PTSs) with parameterized abstractions. We show that such a system can adequately represent *instances* of tactics, i.e. the mapping between a state and the state resulting from it by the application of a given tactic.

We establish the important meta-theoretical properties of the extended system such as confluence, subject reduction, normalization, etc.

1 Introduction

Ever since the ground-breaking work of de Bruijn on AUTOMATH [5] there has been intensive work on mechanical tools to formalize and check mathematical theories using Type Theory. In interactive theorem provers based on Type Theory one tries to construct interactively a term inhabiting a given type. Under the formulas-as-types interpretation, such a term would be an encoding of a proof of the proposition encoded by the type. Because of the undecidability of the inhabitation problem (for interesting enough systems) one has to construct the proof-terms interactively and is hence forced to work with partially constructed objects. This raises many questions about the representation and manipulation of incomplete objects in type theory and logic. Most of the research effort in the area of open terms has been dedicated to the representation problem while the formalization of manipulations of incomplete objects has stayed on the meta-level on the background.

[1] Email: G.I.Jojgov@tue.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The formalization of incomplete terms and proofs allows us to treat unknowns and incomplete terms containing unknowns as first-class objects and therefore to model inside a calculus the incomplete objects that one works with in a theorem prover. We are able to represent the *states* of a theorem prover by open terms. The manipulations of the incomplete objects however are done on the meta-level. This means that we have no formalization of the transition between the states of the prover. It is clear that to give full formalization of the process of interactive term construction one also needs to have representation of the transitions between the states. In many systems these transitions are called *tactics* and they may be of significant complexity. Some systems use full-blown programming languages (e.g. ML) as a tactic language. This is not surprising as tactics often involve pattern matching and/or unification, recursion, backtracking, complex decision and search procedures, failure handling etc. All this however happens at the meta-level outside the object calculus that we work with. In this paper we make a first step towards internalizing some meta-level transformations by providing a calculus that allows us to represent states as terms and transitions between them as functional terms. We do that by extending a calculus of open terms with abstractions over unknowns and function types over them. On a more technical side, this means that we start with a representation of unknowns by parameterized variables (e.g. $x[\Delta]$, where Δ is a list of variable declarations) in a Pure Type System (PTS) and allow abstractions over them. This lead us to a calculus with parameterized $\lambda-$abstractions (e.g. $\lambda x[\Delta]{:}A.M$) and $\Pi-$abstractions (e.g. $\Pi x[\Delta]{:}A.M$). The resulting calculus is not powerful enough to describe arbitrary tactics because it lack essential mechanisms for doing that (e.g. unification and recursion), but it is capable of describing *tactic instances*, i.e. mappings between individual states arising from a specific applications of tactics.

The paper is organized as follows: in Section 2 we give a brief description of the representation of unknowns by (hereditarily) parameterized variables and by means of examples we describe the problems and the solutions that we propose to address them. After discussing related work in Section 2.3, we introduce our extension of PTSs in Section 3 where we define the syntax, the reduction rules and the typing system. We establish some meta-properties of the system like confluence, subject reduction and normalization. We conclude with a discussion on future work in Section 4.

2 Motivation and Related Work

2.1 Representing Unknowns in Open Terms by Parameterized Variables

Throughout this paper we model unknowns that appear in terms by parameterized variables that we also call *meta-variables*. This is of course only one of the many possibilities that have been studied (see Section 2.3), but we choose this approach because it is well-suited for representing incomplete log-

ical derivations and terms (see [7,8]). In this section we will briefly point to the main issues concerning this representation and introduce notation that we will use later.

Meta-variables stem from the use of higher-order variables to represent unknown terms in unification algorithms by Miller [13]. Indeed, a parameterized variable can be seen as a higher-order function of its arguments. We need the arguments in order to record substitutions carried out in a term as a result of β-reduction. If we would like to model a function of an argument x of type A with unknown body, we can introduce a meta-variable $h[x{:}A]$ with a parameter of type A representing the unknown body and the function is then given by the term $\lambda x{:}A.h[x]$. We can apply this function to arguments $(\lambda x{:}A.h[x])b$ and even compute the result of the beta-reduction: $h[b]$. We see that the parameters help us 'remember' that the unknown represented by h was subject to a substitution. This is very important because we would like to be able to instantiate h at any time and always get the same result. So, if we instantiate $h[x{:}A]$ by x before the beta-reduction we get $(\lambda x{:}A.x)b$ that reduces to b and if we instantiate it in $h[b]$ we get b again. In other words, *parameters make instantiation and reduction commute.*

Commutation of instantiation and reduction is obtained also in the other systems of open terms known in the literature, but parameters have an advantage when one looks at the logical side of the problem. As discussed in Jojgov [8], in incomplete logical terms and proofs there are two kinds of abstractions – one is the object-level abstraction and the other one is the meta-level dependency of unknown objects on the variables that occur free in them. These dependencies need to be kept separate in order to get a faithful extension of the formulas-as-types embedding of logic in type theory to incomplete proofs and terms. To illustrate this, consider the following incomplete derivations and their translations to type theory judgments where meta-variables are represented by higher-order function variables:

$$\frac{?}{A \to B} \qquad \frac{\dfrac{\dfrac{[A]^i}{?}}{B}}{A \to B}\,{}^i \qquad \frac{[A]^i \quad \dfrac{?}{A \to B}}{\dfrac{B}{A \to B}\,{}^i}$$

$$f_a : A \to B \vdash f_a : A \to B$$

$$f_b : A \to B \vdash \lambda x{:}A.\,(f_b\,x) : A \to B$$

$$f_c : A \to B \vdash \lambda x{:}A.\,(f_c\,x) : A \to B$$

(a) (b) (c)

The '?'-symbols here represent missing part of the proof with conclusion the formula given below and assumptions given above the symbol. We notice several things: first, looking at the representation of the unknowns in the typing judgment we cannot distinguish between the three because they are all represented by a variable of type $A \to B$. This denies us the opportunity to track the progress being made towards solving the unknown. Second, we notice that

the derivations (b) and (c), although very different from a logical viewpoint (they have different sets of possible completions to finished derivations), have identical translations. We can track the problem down to the identification in the typing judgment of the object- and meta-level abstractions present in the logical system.

Parameters help us distinguish between the representations of the two levels of abstraction. The meta-dependencies are recorded as parameters. This approach has also the advantage that it avoids the need to extend the object-level function space to accommodate the meta-level dependencies. The above examples translated to a system where unknowns are represented by parameterized meta-variables look like this:

$$\begin{aligned} m_a[\,] : A \to B \vdash & \quad m_a[\,] : A \to B \\ m_b[x : A] : B \vdash & \quad \lambda p{:}A.m_b[p] : A \to B \\ m_c[\,] : A \to B \vdash & \quad \lambda x{:}A.(m_c[\,]\, x) : A \to B \end{aligned}$$

A further discussion on the possible forms of incompleteness in logical proofs and terms yields terms containing bound variables whose object-level binders have not (yet) been constructed. Such terms occur naturally in the setting of forward proof constructions that correspond to the building of a proof term from the leaves to the root. We can view such incomplete terms as unknown terms that have known subterms. As unknowns are represented by meta-variables, the known subterms can be given as arguments to the meta-variables. To account for the binding of the variables in the subterms, we need to give meta-level binding power to the meta-variables. Then a typical meta-variable instance looks like this $m[\langle\Delta_1\rangle M_1 \ldots \langle\Delta_n\rangle M_n]$. Each M_i represents a known subterm and the variables declared in Δ_i are those that are supposed to be bound by the yet unconstructed binders. As discussed in [8], to achieve that we need to use *hereditarily parameterized meta-variables*, i.e. meta-variables whose parameters can be parameterized themselves. A logic-based argument similar to the examples above can be given (see [8]) as to why we need to use parameters instead of object-level abstractions that may even be unavailable in the system (e.g. higher-order functions in the framework of first-order logic).

2.2 Representing States and Tactic Instances as Terms

In the previous section we introduced the parameterized meta-variables as a mechanism to model incomplete terms. The process of stepwise construction of a (proof)term can be modelled by a sequence of open terms representing the incomplete proof at different stages. Let us take as an example the following problem: Assume that A is a type and a, b and c are terms of this type. Assume that R is a binary relation on A that is transitive and for each x $R(x, b)$ holds. As a part of a larger proof we would like to prove $R(a, c)$. We can reduce this goal to the goal of proving $R(b, c)$ using the assumptions we

have. The initial state of the prover can be depicted as:

```
thm  :  (x:A)(R x b)
 tr  :  (x,y,z:A)(R x y)->(R y z)->(R x z)
===========
?  :  (R a c)
```

The declarations above the line are the assumptions under which we have to prove the goal $R(a, c)$. Let us collect them in the context Δ:

$$\Delta = thm{:}\Pi x{:}A.Rxb, tr{:}\Pi x, y, z{:}A.Rxy \to Ryz \to Rxz$$

We can represent the unknown proof of the goal by a meta-variable m with parameters Δ and type Rac. Then the initial state of the prover can be encoded by the judgment

$$m[\Delta]{:}Rac \vdash \lambda\Delta.m[thm, tr] : Rac$$

where $\lambda\Delta.M$ of course means $\lambda thm{:}\ldots.\lambda tr{:}\ldots.M$. At this moment we would like to use the transitivity of R by instantiating x and z by a and c. This produces three new goals to find an instantiation for y in the transitivity, and to prove the premises corresponding to Rxy and Ryz:

```
                thm  :  ...      thm  :  ...
thm  :  ...      tr  :  ...       tr  :  ...
 tr  :  ...      y?  :  A         y?  :  A
===========     ===========      ===========
y?  :  A        p?  :  (R a y?)  q?  :  (R y?  c)
```

How do we encode this new state and how is it related to the previous one? We introduce a new meta-variable for each new goal and in the place of $m[thm, tr]$ we have an application of tr:

$$\begin{aligned}
&y?[\Delta]{:}A,\\
&p?[\Delta] : (R\,a\,y?[thm, tr]),\\
&q?[thm, tr] : (R\,y?[thm, tr]\,c),\\
&\vdash \lambda\Delta.(tr\,a\,y?[thm, tr]\,c\,p?[thm, tr]\,q?[thm, tr]) : Rac
\end{aligned}$$

Now we would like to use our other assumption, thm, to solve the second goal. At this point a theorem prover would use unification to match $R\,a\,y?$ to $R\,x\,b$ and find out that in order to apply thm, x has to be instantiated by a and $y?$ has to be b. This results in the following state:

```
thm  :  ...
 tr  :  ...
===========
r?  :  (R b c)
```

And it can be represented by the judgment

$$r?[\Delta]{:}(R\,b\,c) \vdash \lambda\Delta.(tr\,a\,b\,c\,(thm\,a)\,r?[thm, tr]) : R\,a\,c$$

This does not complete the proof, but if we have a look at the two transitions between the three states, we notice that there are several steps that we do on the meta-level that are not part of our representation. We introduce new meta-variables, we use them to give solutions to (some of) the pre-existing ones and we propagate these solutions through the representation of the state. All these are the meta-steps we make at each of the two transitions. The question arises:

Can we make these meta-transformations explicit by internalizing them in the calculus?

In this paper we will give affirmative answer to this question by extending our system with abstractions over meta-variables as means to internalize the dependency of the state on its meta-variables. The corresponding application operation would play the role of explicit representation of the instantiation of meta-variables. In this system a state can be encapsulated in a term by abstracting out all its meta-variables. The transformation steps then become functions that expect terms of appropriate types matching the types of the state terms and can also be encoded by λ-terms.

As an illustration, using the abstractions and applications that we will introduce in Section 3, the first state can be encoded by $\lambda m[\Delta]{:}Rac.\lambda\Delta.m[thm, tr]$ and its type is $\Pi m[\Delta]{:}Rac.\Pi\Delta.Rac$. The transformation step leading to the second state can be given by:

$$\lambda S : (\Pi m[\Delta]{:}Rac.\Pi\Delta.Rac).$$
$$\lambda y?[\Delta]{:}A.$$
$$\lambda p?[\Delta] : (R\, a\, y?[thm, tr]).$$
$$\lambda q?[\Delta] : (R\, y?[thm, tr]\, c).$$
$$(S \cdot \langle\Delta\rangle (tr\, a\, y?[thm, tr]\, c\, p?[thm, tr], q?[thm, tr]))$$

If we apply this transformation term to the state term and normalize, we get the term

$$\lambda y?[\Delta]{:}A.$$
$$\lambda p?[\Delta] : (R\, a\, y?[thm, tr]).$$
$$\lambda q?[\Delta] : (R\, y?[thm, tr]\, c).$$
$$\lambda\Delta.(tr\, a\, y?[thm, tr]\, c\, p?[thm, tr]\, q?[thm, tr])$$

which is exactly the encoding of the second state.

2.3 Related Work

The work in this paper builds on several ideas already present in the field of open terms. The representation of holes by higher-order functions used

in Miller's work [13] on unification is in the basis of our approach to the unknowns, but we use it in a modified form because of the need to separate object- and meta-level level abstractions (see [8] for a discussion on this). This idea has been employed previously in Luo's PAL$^+$ logical framework [10] to avoid extending the object-level function space in order to accommodate meta-level functions. The handling of the scopes in open terms can be done in different ways, ALF [11] and Munoz [14] employ explicit substitutions similarly to Strecker's Typelab [16]. The use of parameters makes explicit the idea that is implicit in Typelab's handling of meta-variables where Strecker has noticed that it suffices to use explicit substitutions attached to meta-variables only. The idea to represent states as terms is introduced in the thesis of McBride [12] where he presents the OLEG framework of open terms. He uses binders for meta-variables to represent states, but it differs from our approach in several aspects. First, the meta-variables that we consider are parameterized. Instead, in OLEG the position of the binder is used to specify the context in which the meta-variable should be solved. Second, in our system the binders for meta-variables occurring in a term have a corresponding binder in the type. This means that the type of a term with a meta-variable binder may depend on the meta-variable. This allows us to make functions that expect terms with meta-variables as arguments. The corresponding application operation allows such a function to explicitly instantiate meta-variables in its arguments.

Another system that is closely related to our presentation is the $\lambda[]$-cube of Bognar [4]. In that system we have separate binders both for object-level and meta-level binders, and both of them have a corresponding binder on the type level. Our system differs from the $\lambda[]$-cube in that it allows hereditarily parameterized variables to be constructed and abstracted over, while in the cube the parameters cannot be parameterized themselves. In that sense our system extends the systems of the $\lambda[]$-cube.

The present work is a continuation of the previous discussions of open terms and proofs in higher-order logic by Geuvers and Jojgov [7,8] where the problem of extending the formulas-as-types embedding to incomplete proofs and terms has been discussed. There we internalized the notion of unknown in the calculus, in this paper we extend this formalization to the transformations of open terms.

We make heavy use of the parameter mechanism of Bloo, Kamareddine, Laan and Nederpelt [3] who extend earlier work of Poll and Severi [15]. We extend that work by introducing the more general notion of hereditary parametrization. The author believes that an extension of the C^{p}D^{p}PTSs described in that work to PTS with hereditarily parametrization and parameterized variables that results in PTS with hereditarily parameterized variables, constants and definitions (V^{h}C^{h}D^{h}PTSs) could be useful for modelling of many practical applications. Such an extension would be forthcoming in the author's thesis.

3 Pure Type Systems with Hereditarily Parameterized Variables

In this section we will present an extension of the Pure Type Systems (PTSs) introduced by Berardi [2] and Terlouw [18] as a generalization of the systems of Barendregt's λ-cube (see [1]). We assume that the reader is acquainted with the background facts about PTSs (see for example [1,6]).

We extend the standard definition of a PTS by adding parametrization to the λ- and Π-abstractions. A parameterized λ-abstraction $\lambda m[\Delta]{:}A.M$ represents a term that has abstracted out the meta-variable $m[\Delta]$ that potentially occurs in M. Such a term would have a parameterized Π-abstraction as a type: $\Pi m[\Delta]{:}A.B$. As the use of λ suggests, we can apply parameterized λ-abstractions to arguments and that would act as an explicit notation for the instantiation operation. Meta-variables however have parameters that can be used in the term that instantiates them. Therefore, we need to introduce the parameters of a meta-variable into the argument of an application: $(\lambda m[x{:}A]{:}A.m[z])\cdot\langle x{:}A\rangle x$. This term represents *explicitly* the instantiation of the meta-variable $m[x{:}A]$ by x in the term $m[z]$ (indeed, we will see that it β-reduces to z as expected). Notice how the extended application $M\cdot\langle x{:}A\rangle N$ introduces x in scope for the term N.

3.1 Syntax

Every PTS is given by a tuple $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}\rangle$ where the elements of $\mathcal{S}$ are called *sorts*, $\mathcal{A} \subseteq \mathcal{S}\times\mathcal{S}$ is the set of axioms and $\mathcal{R}\subseteq\mathcal{S}\times\mathcal{S}\times\mathcal{S}$ is a set of triples that restrict the formation of Π-types (see e.g.[6]). The set of the *pseudo-terms* of the extended PTS is given by the following grammar:

$$\mathcal{T} ::= \mathcal{S} \mid x[\langle\Delta\rangle\mathcal{T}\ldots\langle\Delta\rangle\mathcal{T}] \mid \mathcal{T}\cdot\langle\Delta\rangle\mathcal{T} \mid \lambda x[\Delta]{:}\mathcal{T}.\mathcal{T} \mid \Pi x[\Delta]{:}\mathcal{T}.\mathcal{T}$$
$$\Delta ::= \varepsilon \mid \Delta, x[\Delta]{:}\mathcal{T}$$

This definition is motivated by the intuitive meaning of the parameterized abstractions and application introduced above. $\lambda x[\Delta]{:}A.M$ and $\Pi x[\Delta]{:}A.M$ introduce the parameterized variable $x[\Delta]$ in M where it can be used provided it is given appropriate arguments. The variables declared in Δ can be used in A, but their scope does not extend to M. The term N is in the scope of the variables in Δ in an application $M\cdot\langle\Delta\rangle N$. Similarly, in a variable instance $x[\langle\Delta_1\rangle N_1\ldots\langle\Delta_n\rangle N_n]$, each N_i is in the scope of the variables in Δ_i.

In order to ease the notation, we identify the unparameterized variables and the variables with empty parameter lists.

On the level of contexts we define the notion of structural equivalence $\approx$ as follows:

$$\frac{}{\varepsilon\approx\varepsilon} \qquad \frac{\Gamma_1\approx\Gamma_2 \quad \Delta_1\approx\Delta_2}{\Gamma_1, x_1[\Delta_1]{:}A_1 \approx \Gamma_2, x_2[\Delta_2]{:}A_2}$$

The relation $\Delta \approx \Theta$ should be read as "Δ and Θ have the same structure". Note that the relation states properties of the structure of the contexts only. In particular, in the definition above A_1 and A_2 are not subject to any restrictions. We will assume that the names of the parameterized variables determine up to $\approx$-equivalence the context describing their parameters. Hence, if we talk about a variable $x[\Delta]$ then its instances $x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]$ must have the same number of actual parameters as there are elements in Δ and if Δ is the context $x_1[\Delta_1]{:}A_1, \ldots, x_n[\Delta_n]{:}A_n$ then $\Delta_i \approx \Theta_i$.

Note also that the structural equivalence relation is a weaker notion than α-equivalence as contexts that are not α-convertible can have the same structure. The need to introduce this notion arises from the possibility to do β-reductions in contexts that are explicitly recorded in terms (see the definition of β-reduction).

Example 3.1 [Well-formed pseudo-terms]

- If $x[y[z[p{:}D]{:}E]{:}F]$ is a parameterized variable then the following term is well-formed:

$$\lambda Y{:}(\Pi z[p{:}D]{:}A.B).x[\langle z[p{:}D]{:}E\rangle(Y \cdot \langle p{:}D\rangle z[p])]$$

- If $h[p[i{:}A]{:}B, q[j{:}A]{:}B \to C]{:}A \to C$ is a parameterized variable then $h[\langle i{:}A\rangle(a \cdot i), \langle j{:}A\rangle(b \cdot j)]$ is a well-formed term.

Definition 3.2 [Free and bound variables] The set of the free variables $\mathsf{FV}(-)$ of a term or a context is defined as follows:

$$\begin{aligned}
\mathsf{FV}(\varepsilon) &= \emptyset \\
\mathsf{FV}(\Delta, x[\Delta']{:}A) &= \mathsf{FV}(\Delta) \cup \mathsf{FV}(A) \setminus \mathsf{dom}(\Delta', \Delta) \cup \mathsf{FV}(\Delta') \setminus \mathsf{dom}(\Delta) \\
\mathsf{FV}(s) &= \emptyset \\
\mathsf{FV}(x[\langle\Delta_1\rangle M_1, \ldots, \langle\Delta_n\rangle M_n]) &= \{x\} \cup \bigcup\nolimits_j (\mathsf{FV}(M_j) \setminus \mathsf{dom}(\Delta_i) \cup \mathsf{FV}(\Delta_i)) \\
\mathsf{FV}(M \cdot \langle\Delta\rangle N) &= \mathsf{FV}(M) \cup \mathsf{FV}(N) \setminus \mathsf{dom}(\Delta) \cup \mathsf{FV}(\Delta) \\
\mathsf{FV}(\lambda x[\Delta]{:}A.M) &= \mathsf{FV}(M) \setminus \{x\} \cup \mathsf{FV}(A) \setminus \mathsf{dom}(\Delta) \cup \mathsf{FV}(\Delta) \\
\mathsf{FV}(\Pi x[\Delta]{:}A.M) &= \mathsf{FV}(M) \setminus \{x\} \cup \mathsf{FV}(A) \setminus \mathsf{dom}(\Delta) \cup \mathsf{FV}(\Delta)
\end{aligned}$$

An occurrence of a variable that is not free is bound. We assume that the names of the bound variables are always taken to be different from each other and from the names of the free variables.

This definition differs from the standard one in that it defines that the scope of the parameters Δ in $\Gamma, x[\Delta]{:}A$, $\lambda x[\Delta]{:}A.M$ and $\Pi x[\Delta]{:}A.B$ to be limited to A and that actual parameters (M_i in $x[\langle\Delta_1\rangle M_1 \ldots \langle\Delta_n\rangle M_n]$) and arguments of applications (N in $M \cdot \langle\Delta\rangle N$) are in the scope of extra parameters (Δ_i and Δ resp.). This shows that the application and variable instances can behave as binders.

The process of filling in a value for a parameterized variable is called *instantiation*. As instances of variables provide actual arguments for formal

parameters, we need to propagate the arguments in the term instantiating the variable. This leads to the following definition:

Definition 3.3 [Instantiation] Let Δ be the context $x_1[\Delta_1]{:}A_1, \ldots, x_n[\Delta_n]{:}A_n$ and $m[\Delta] : A$ be a meta-variable. The instantiation of $m[\Delta]$ by an arbitrary term N in the term M (notation $M\{m[\Delta] := N\}$ is defined as follows:

$$
\begin{aligned}
s\{m[\Delta] := N\} &= s \\
(m[\langle\Theta_1\rangle M_1 \ldots \langle\Theta_n\rangle M_n])\{m[\Delta] := N\} &= N\{x_1[\Theta_1^*] := M_1^*\} \ldots \{x_n[\Theta_n^*] := M_n^*\} \\
(n[\langle\Theta_1\rangle M_1, \ldots, \langle\Theta_k\rangle M_k])\{m[\Delta] := N\} &= n[\langle\Theta_1^*\rangle M_1^*, \ldots, \langle\Theta_k^*\rangle M_k^*] \\
(M_1 \cdot \langle\Theta\rangle M_2)\{m[\Delta] := N\} &= M_1^* \cdot \langle\Theta^*\rangle M_2^* \\
(\lambda y[\Theta]{:}U.\, M)\{m[\Delta] := N\} &= \lambda y[\Theta^*]{:}U^*.\, M^* \\
(\Pi y[\Theta]{:}U.\, B)\{m[\Delta] := N\} &= \Pi y[\Theta^*]{:}U^*.\, B^*
\end{aligned}
$$

where for readability M^* abbreviates $M\{m[\Delta] := N\}$.

Note that by the $\approx$-convention on variables $\Delta_i \approx \Theta_i$ and this allows us to form the instantiations $\{x_i[\Theta_i{}^*] := u_i{}^*\}$

The well-foundness of instantiation is not completely self-evident, because in the second clause of the definition we apply recursively instantiations to a possibly 'larger' term N. Note however that in that case the contexts involved in the instantiations become strictly 'smaller' (w.r.t the depth of the context, see Definition 3.7) and that ensures the termination of the process.

Example 3.4 A few examples of instantiation:

Term	**Instantiation**	**Result**
$h[]$	$\{h[] := t\}$	$= t$
$h[a]$	$\{h[x{:}A] := x\}$	$= a$
$h[t, \langle x{:}A\rangle p(x,t)]$	$\{h[y{:}A, q[x{:}A]{:}P(x,y)] := q[y]\}$	$= p(t,t)$
$h[g, h[\lambda y{:}A.y, s]]$	$\{h[f{:}\Pi x{:}A.A, x{:}A] := fx\}$	$= g((\lambda y{:}A.y)s)$

Remark 3.5 [Substitution is instantiation with no parameters] Note that if Δ is empty in an instantiation $\{x[\Delta] := t\}$ then the instantiation of x by t in M is exactly the result of the substitution of t for the free occurrences of x in M. For example:

$$(\lambda y[z{:}Ax]{:}B.x)\{x := t\} = \lambda y[z{:}At]{:}B.t$$

Example 3.6 [Variable Capture] Due to the parameters, some variables may get 'captured'. For example in the term

$$(\lambda x{:}A.h[x])\{h[x{:}A] := x\} = \lambda x{:}A.x$$

the variable x is captured by the binder which is in contrast to

$$(\lambda x{:}A.h[\,])\{h[\,] := x\} = \lambda y{:}A.x$$

where x is still free after the instantiation (Note the renaming). In both cases we instantiate h by x but in the first example x is bound and in the second is free. We note that *only variables that have been declared as parameters can get captured.*

The notion of depth reflects the number of levels of parametrization in a context or a parameterized variable.

Definition 3.7 [Depth] The parameter depth of $x[\Delta]$ is by definition $d(\Delta)$, where the depth $d(\Delta)$ of a context Δ is defined as:

$$d(\varepsilon) = 0$$
$$d(\Gamma, x[\Delta]{:}A) = \max(d(\Gamma), d(\Delta) + 1)$$

Example: $d(A{:}*, h[x{:}A]A \to A) = 2$ and $d(A{:}*, x{:}A, h{:}\Pi y{:}A.Bx) = 1$

Proposition 3.8

(i) *If* $\Delta \approx \Theta$ *then* $d(\Delta) = d(\Theta)$.

(ii) *For all* Θ *we have* $\Theta \approx \Theta\{x[\Delta] := N\}$.

Lemma 3.9 *If* $\Delta \approx \Theta$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Theta)$ *then for all* M *and* N

$$M\{x[\Delta] := N\} = M\{x[\Theta] := N\}$$

Proof. The proof proceeds by induction on the depth of Δ and a nested induction on the structure of M. □

3.2 β-reduction and confluence

Normally β-reduction is defined in terms of the capture-avoiding meta-substitution:

$$(\lambda x{:}A.M)t \to_\beta M[t/x]$$

We extend this definition to our pseudo-terms as follows:

$$(\lambda x[\Theta]{:}A.M) \cdot \langle\Delta\rangle t \to_\beta M\{x[\Delta] := t\} \text{ if } \Theta \approx \Delta$$

Note that on unparameterized terms the two reduction relations coincide. The side condition $\Theta \approx \Delta$ is needed, because we need to know that the two contexts have the same structure in order for the instantiation $\{x[\Delta] := t\}$ to be well-defined.

To establish the confluence property we follow the modular confluence proof of Takahashi [17]. Definition 3.10 introduces the notions of parallel

reduction $M \Rightarrow N$ and complete development $\#(M)$ and Lemma 3.11 states the relevant properties:

Definition 3.10 [Parallel reduction $M \Rightarrow N$ and complete development $\#(M)$]

$$M \Rightarrow M \qquad \frac{\Theta_i \Rightarrow \Theta'_i \quad t_i \Rightarrow t'_i}{x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n] \Rightarrow x[\langle\Theta'_1\rangle t'_1 \ldots \langle\Theta'_n\rangle t'_n]}$$

$$\frac{\Delta \Rightarrow \Delta' \quad A \Rightarrow A' \quad M \Rightarrow M'}{\lambda x[\Delta]{:}A.M \Rightarrow \lambda x[\Delta']{:}A'.M'} \qquad \frac{\Delta \Rightarrow \Delta' \quad A \Rightarrow A' \quad B \Rightarrow B'}{\Pi x[\Delta]{:}A.B \Rightarrow \Pi x[\Delta']{:}A'.B'}$$

$$\frac{M \Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta'}{M \cdot\langle\Delta\rangle N \Rightarrow M' \cdot\langle\Delta'\rangle N'} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta'}{(\lambda x[\Theta]{:}A].M) \cdot\langle\Delta\rangle N \Rightarrow M'\{x[\Delta'] := N'\}}\,\Theta \approx \Delta$$

$$\begin{aligned}
\#(s) &= s \\
\#(x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]) &= x[\langle\#(\Theta_1)\rangle\#(t_1) \ldots \langle\#(\Theta_n)\rangle\#(t_n)] \\
\#(\lambda x[\Delta]{:}A.M) &= \lambda x[\#(\Delta)]{:}\#(A).\#(M) \\
\#(\Pi x[\Delta]{:}A.B) &= \Pi x[\#(\Delta)]{:}\#(A).\#(B) \\
\#((\lambda x[\Theta]{:}A.M) \cdot\langle\Delta\rangle N) &= \#(M)\{x[\#(\Delta)] := \#(N)\} \\
\#(M \cdot\langle\Delta\rangle N) &= \#(M) \cdot\langle\#(\Delta)\rangle\#(N) \ (M \text{ not an abstraction})
\end{aligned}$$

Lemma 3.11 (Properties of $\Rightarrow$ and $\#$) (i) *If $M_1 \Rightarrow M_2$ and $N_1 \Rightarrow N_2$ then $M_1\{x[\Delta] := N_1\} \Rightarrow M_2\{x[\Delta] := N_2\}$*

(ii) *If $M \Rightarrow N$ then $N \Rightarrow \#(M)$.*

(iii) *If $M \Rightarrow N$ then $M \twoheadrightarrow_\beta N$.*

(iv) *If $M \rightarrow_\beta N$ then $M \Rightarrow N$.*

From (2) it follows easily that $\Rightarrow$ has the diamond property (i.e. if $M \Rightarrow P$ and $M \Rightarrow Q$ then there is a term N such that $P \Rightarrow N$ and $Q \Rightarrow N$). Then, given M, P and Q such that $M \twoheadrightarrow_\beta P$ and $M \twoheadrightarrow_\beta Q$, we have $M \Rightarrow^* P$ and $M \Rightarrow^* Q$ using (4). But then iterating the diamond property for $\Rightarrow$ we get a term N such that $P \Rightarrow^* N$ and $Q \Rightarrow^* N$. But then from (3) we have $P \twoheadrightarrow_\beta N$ and $Q \twoheadrightarrow_\beta N$ using the transitivity of $\twoheadrightarrow_\beta$.

This concludes the proof that β-reduction is confluent.

3.3 Typing system

The derivation rules of a typing system give an inductive definition of the typing relation that assigns types to terms in a given context that specifies the types of the free variables. The standard derivation rules for PTSs (see e.g. [1,6]) are parameterized by three sets $\langle\mathcal{S}, \mathcal{A}, \mathcal{R}\rangle$ and the different PTSs can be obtained by giving particular values to the three parameters. $\mathcal{S}$ is the set of sorts, $\mathcal{A}$ is a subset of $\mathcal{S} \times \mathcal{S}$ and its elements are called axioms. The set

$\mathcal{R}$ is a subset of $\mathcal{S}\times\mathcal{S}\times\mathcal{S}$ and its elements are used to restrict the Π-formation rule.

For the purposes of typing terms with parameterized variables we introduce one extra set $\mathcal{P}$ that would be a subset of $\mathcal{S}\times\mathcal{S}$ and it will be used to denote the dependencies between the type of a parameter of a variable and the type of the variable itself. Hence a PTS with parametric variables will be given by a parametric specification that is a 4−tuple $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\rangle$.

Notation 3.12 • *The notion of convertibility between two contexts (notation $\Gamma_1 =_\beta \Gamma_2$) is defined inductively as follows:*
  · $\varepsilon =_\beta \varepsilon$
  · *if* $\Gamma_1 =_\beta \Gamma_2$, $\Delta_1 =_\beta \Delta_2$ *and* $A_1 =_\beta A_2$ *then* $\Gamma_1, x[\Delta_1]{:}A_1 =_\beta \Gamma_2, x[\Delta_2]{:}A_2$
- *We will write* $x[\vec{\Theta}\vec{t}]$ *for* $x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]$.
- *Let* $\Delta = x_1[\Delta_1]{:}A_1, \ldots, x_n[\Delta_n]{:}A_n$ *and* $\vec{t} = \langle t_1, \ldots, t_n\rangle$. *We will write* $\Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta$ *for the conjunction of the judgments* $\Gamma, \Theta_k \vdash t_k{:}\delta_{k-1}A_k$ *with* $k \in [1 \ldots n]$ *where* $\delta_0 = id$, $\delta_{k+1} = \delta_k \circ \{x_{k+1}[\Theta_{k+1}] := t_{k+1}\}$ *and* $\Theta_k =_\beta \delta_{k-1}\Delta_k$.
- *By* $\{\Delta := \vec{\Theta}\vec{t}\}$ *we will denote* δ_n *from above and* $\Delta_{|i}$ *will denote the context*

$$x_1[\Delta_1]{:}A_1, \ldots, x_{i-1}[\Delta_{i-1}]{:}A_{i-1}, \Delta_i$$

Below we give the derivation rules for a PTS with hereditarily parameterized variables. As usual s and s_i denote sorts from $\mathcal{S}$.

Definition 3.13 [Derivation Rules]

$$\frac{}{\vdash s_1{:}s_2} \quad (s_1, s_2) \in \mathcal{A} \qquad \text{(axiom)}$$

$$\frac{\Gamma, \Delta \vdash A{:}s \quad \Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta}{\Gamma \vdash x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]{:}A\{\Delta := \vec{\Theta}\vec{t}\}} \quad x[\Delta]{:}A \in \Gamma \qquad \text{(start)}$$

$$\frac{\Gamma \vdash M{:}B \quad \Gamma, \Delta \vdash A{:}s \quad \Gamma, \Delta_{|i} \vdash A_i{:}s_i}{\Gamma, x[\Delta]{:}A \vdash M{:}B} \quad (s_i, s) \in \mathcal{P} \qquad \text{(weak)}$$

$$\frac{\Gamma, \Delta \vdash A{:}s_1 \quad \Gamma, x[\Delta]{:}A \vdash B{:}s_2}{\Gamma \vdash \Pi x[\Delta]{:}A.B{:}s_3} \quad (s_1, s_2, s_3) \in \mathcal{R} \qquad (\Pi)$$

$$\frac{\Gamma, x[\Delta]{:}A \vdash M{:}B \quad \Gamma \vdash \Pi x[\Delta]{:}A.B{:}s}{\Gamma \vdash (\lambda x[\Delta]{:}A.M){:}(\Pi x[\Delta]{:}A.B)} \qquad (\lambda)$$

$$\frac{\Gamma \vdash M{:}\Pi x[\Delta]{:}A.B \quad \Gamma, \Delta \vdash N{:}A}{\Gamma \vdash M \cdot\langle\Delta\rangle N{:}B\{x[\Delta] := N\}} \qquad \text{(app)}$$

$$\frac{\Gamma \vdash M{:}A \quad \Gamma \vdash B{:}s}{\Gamma \vdash M{:}B} \quad A =_\beta B \qquad \text{(conv)}$$

We briefly comment on the modifications to the rules in order to explain the intuition behind them. In the (start) rule we type the parameterized variables introduced in the context. An instance of a variable is well-typed if it has a correct number and type of arguments. The premise $\Gamma, \Delta \vdash A{:}s$ is necessary in order to ensure that Γ is a valid context in cases when there are no parameters. Each actual parameter t_i can be given a context Θ_i that

locally introduces variables that can be used in t_i. The context Θ_i is required to be β-convertible, but not necessarily equal to $\delta_{i-1}\Delta_i$ because for the Subject Reduction property we should be able to type instances in which β-reductions have been executed in Θ_i.

Using the weakening rule (weak) we can add variables to a context. Note that the parameters of the variable can be used in its type. Very much like in the C^pD^pPTSs [3], by a suitable choice of $\mathcal{P}$ the condition $(s_i, s) \in \mathcal{P}$ is used to restrict the possible parameters that a variable of a given sort can take.

As usual, the Π-formation rule is restricted by $\mathcal{R}$. The new moment is that the bound variable may have parameters. Again, the parameters in Δ can be used in A (but not in B, see Definition 3.2).

The (λ) rule abstracts parameterized variables. If we want to apply such an abstraction to an argument, the argument should be typed in a context that is extended with the parameters. This is done by the (app) rule. Note how the application $\cdot\langle\Delta\rangle$ introduces the parameters in the context of the argument.

We now proceed by establishing the important meta-properties of the system.

Lemma 3.14 (Generation Lemma) *Let $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\rangle$ be a parametric specification. Then*

(i) *If $\Gamma \vdash s{:}D$ then there is $s' \in \mathcal{S}$ such that $D =_\beta s'$ and $(s, s') \in \mathcal{A}$;*

(ii) *If $\Gamma \vdash x[\vec{\Theta}\vec{t}]{:}D$ then $\Gamma = \Gamma_1, x[\Delta]{:}A, \Gamma_2$ and there is an s such that $\Gamma_1, \Delta \vdash A{:}s$, $\Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta$ and $D =_\beta A\{\Delta := \vec{\Theta}\vec{t}\}$*

(iii) *If $\Gamma \vdash (\Pi x[\Delta]{:}A.B){:}D$ then there are sorts $(s_1, s_2, s_3) \in \mathcal{R}$ such that $\Gamma, \Delta \vdash A{:}s_1$ and $\Gamma, x[\Delta]{:}A \vdash B{:}s_2$ and $D =_\beta s_3$.*

(iv) *If $\Gamma \vdash (\lambda x[\Delta]{:}A.M){:}D$ then there are s and B such that $\Gamma \vdash \Pi x[\Delta]{:}A.B{:}s$, $\Gamma, x[\Delta]{:}A \vdash M{:}B$ and $\Pi x[\Delta]{:}A.B =_\beta D$*

(v) *If $\Gamma \vdash M \cdot\langle\Delta\rangle N{:}D$ then there are A and B such that $\Gamma \vdash M{:}\Pi x[\Delta]{:}A.B$, $\Gamma, \Delta \vdash N{:}B$ and $D =_\beta B\{x[\Delta] := N\}$.*

(vi) *if $\Gamma, x[\Delta]{:}A, \Gamma' \vdash M{:}D$ then there are s and s_i such that $\Gamma, \Delta \vdash A{:}s$ and $\Gamma, \Delta_{|i} \vdash A_i{:}s_i$, $(s_i, s) \in \mathcal{P}$*

Proof. We proceed by induction on the generation of the typing relation $\vdash$. Consider the possible cases for the last rule of a derivation assuming the lemma holds for its subderivations. We treat here only some of the cases:

(start) This means that $D \equiv A\{x[\Delta] := \vec{\Theta}\vec{t}\}$ and

$$\frac{\Gamma, \Delta \vdash A{:}s \quad \Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta}{\Gamma \vdash x[\vec{\Theta}\vec{t}]{:}A\{\Delta := \vec{\Theta}\vec{t}\}}\ x[\Delta]{:}A \in \Gamma$$

From the condition $x[\Delta]{:}A \in \Gamma$ we have $\Gamma = \Gamma_1, x[\Delta]{:}A, \Gamma_2$ and using (6) from the induction hypothesis we have $\Gamma_1, \Delta \vdash A{:}s$.

(weak) Then the last rule looks like this:

$$\frac{\Gamma \vdash M{:}D \quad \Gamma, \Delta \vdash A{:}s \quad \Gamma, \Delta_{|i} \vdash A_i{:}s_i}{\Gamma, x[\Delta]{:}A \vdash M{:}D} \; (s_i, s) \in \mathcal{P}$$

Considering the outermost constructor of M we distinguish five cases and apply the induction hypothesis for $\Gamma \vdash M{:}D$. In this way we prove that the conditions (1) (5) hold. For (6) we need to use the induction hypothesis and the premises of the rule.

(λ) This means that $D \equiv \Pi x[\Delta]{:}A.B$ and

$$\frac{\Gamma, x[\Delta]{:}A \vdash M{:}B \quad \Gamma \vdash \Pi x[\Delta]{:}A.B{:}s}{\Gamma \vdash (\lambda x[\Delta]{:}A.M){:}(\Pi x[\Delta]{:}A.B)}$$

The statement (6) follows from (6) in the induction hypothesis.

(conv) We use the fact that $=_\beta$ is transitive.

□

Lemma 3.15 (Weakening) *If* $\Gamma_0, \Gamma_1 \vdash M{:}B$, $\Gamma_0, \Delta \vdash A{:}s$ *and* $\Gamma_0, \Delta_{|i} \vdash A_i{:}s_i$ *then* $\Gamma_0, x[\Delta]{:}A, \Gamma_1 \vdash M{:}B$ *where* x *is a fresh variable and* $(s_i, s) \in \mathcal{P}$.

Lemma 3.16 (Substitution Lemma) *If* $\Gamma, x[\Delta]{:}A, \Gamma' \vdash M{:}B$ *and* $\Gamma, \Delta \vdash N{:}A$ *then* $\Gamma, \Gamma'\{x[\Delta] := N\} \vdash M\{x[\Delta] := N\}{:}B\{x[\Delta] := N\}$

Proof. By induction on the depth of Δ and a nested induction on the derivation. □

Lemma 3.17 (Correctness of types) *If* $\Gamma \vdash M{:}A$ *then either* $A =_\beta s$ *or* $\Gamma \vdash A{:}s$ *for some* s.

Proof. By induction on the derivation of $\Gamma \vdash M{:}A$ using Substitution Lemma and Generation Lemma. We treat here only the case of the (app) rule.

By Generation from $\Gamma \vdash M{:}\Pi x[\Delta]{:}A.B$ we get $\Gamma, x[\Delta]{:}A \vdash B{:}s$ for some s. Hence by Substitution $\Gamma \vdash B\{x[\Delta] := N\}{:}s$. □

Lemma 3.18 (Subject Reduction) *Let* $\Gamma \vdash M{:}A$. *Then*

(i) *If* $M \to_\beta N$ *then* $\Gamma \vdash N{:}A$

(ii) *If* $\Gamma \to_\beta \Delta$ *then* $\Delta \vdash M{:}A$

Proof. We will prove the two statements simultaneously by induction on the derivation of $\Gamma \vdash M{:}A$.

- The last rule is (start)

(i) Then $M = x[\vec{\Theta}\vec{t}]$. If the redex is in $\vec{t}$ we apply the induction hypothesis on the respective component $\Gamma, \Theta_k \vdash t_k{:}\delta_{k-1}A_k$. If the redex is in Θ_k we use the induction hypothesis for (2) to get $\Gamma, \Theta'_k \vdash t_k{:}\delta_{k-1}A_k$. Since $\Theta_k \to_\beta \Theta'_k$ and $\Theta_k =_\beta \delta_{k-1}\Delta_k$ we can apply the (start) rule.

- The last rule is (weak)

2. Then $\Gamma = \Gamma', x[\Delta]{:}A$ and the redex can be in Γ', Δ or A. In the first case we simply use the induction hypothesis and apply the (weak) rule to the result. If $\Delta \to_\beta \Delta'$, then by induction $\Gamma, \Delta' \vdash A{:}s$ and we can apply the rule again. If $A \to_\beta A'$, then $\Gamma, \Delta \vdash A{:}s$ and from the hypothesis for (1) we have $\Gamma, \Delta \vdash A'{:}s$.

- The last rule is (app) Let

$$\frac{\Gamma \vdash P{:}\Pi x[\Delta]{:}A.B \quad \Gamma, \Delta \vdash Q{:}A}{\Gamma \vdash P \cdot \langle \Delta \rangle Q{:}B\{x[\Delta] := Q\}}$$

(i) If the redex being contracted is in P, Δ or Q, then we can simply apply the induction hypothesis. If the redex is $P \cdot \langle \Delta \rangle Q$ itself then $P = \lambda y[\Theta]{:}C.R$, reduces to $R\{y[\Delta] := Q\}$. Since $\Gamma \vdash \lambda y[\Theta]{:}C.R{:}\Pi x[\Delta]{:}A.B$ is derivable, then we go up this derivation until the node in which the λ was introduced:

$$\frac{\Gamma', y[\Theta]{:}C \vdash R{:}D \quad \Gamma' \vdash \Pi y[\Theta]{:}C.D{:}s}{\Gamma' \vdash (\lambda y[\Theta]{:}C.R){:}\Pi y[\Theta]{:}C.D}$$

where Γ' is an initial segment of Γ and B is convertible to D. Using weakening we get $\Gamma, y[\Theta]{:}C \vdash R{:}D$ and by Substitution we get $\Gamma \vdash R\{x[\Delta] := Q\}{:}D\{x[\Delta] := Q\}$ which (if necessary using the conversion rule) yields $\Gamma \vdash R\{x[\Delta] := Q\}{:}B\{x[\Delta] := Q\}$

□

Definition 3.19 [Functional specification]A specification $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$ is called *functional* if:

- for all sorts s_1, s_2, s' and s'' if $(s_1, s_2, s') \in \mathcal{R}$ and $(s_1, s_2, s'') \in \mathcal{R}$ then $s' = s''$;
- for all sorts s_1, s' and s'' if $(s_1, s') \in \mathcal{A}$ and $(s_1, s'') \in \mathcal{A}$ then $s' = s''$.

Lemma 3.20 (Uniqueness of types) *If λS is functional, $\Gamma \vdash M{:}A$ and $\Gamma \vdash M{:}B$ then $A =_\beta B$.*

Proof. By induction on M using Generation. The functionality condition is used when proving the uniqueness of the types of Π-terms and sorts. □

Definition 3.21 [Quasi-Completion] Let $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ and $\lambda S' = \langle \mathcal{S}', \mathcal{A}', \mathcal{R}', \mathcal{P}' \rangle$. λS is a quasi-completion of $\lambda S'$ if the following hold:

- $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{R}' \subseteq \mathcal{R}$;
- For each $s_1, s_2 \in \mathcal{S}'$ there is an $s_3 \in \mathcal{S}$ such that $(s_1, s_2, s_3) \in \mathcal{S}$;

Theorem 3.22 (Strong Normalization) *Let $\lambda S' = \langle \mathcal{S}', \mathcal{A}', \mathcal{R}' \rangle$ be a quasi-completion of $\lambda S^P = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. Then λS^P is strongly normalizing if $\lambda S'$ is strongly normalizing.*

Proof. We define by induction a reduction- and typing- preserving map $|-|$ from λS^P into $\lambda S'$. Then an assumption that λS^P has an infinite reduction

path induces infinite reduction path in $\lambda S'$ through the map.

$$
\begin{aligned}
|s| &= s & |\varepsilon| &= \varepsilon \\
|x[\vec{\Theta}\vec{t}]| &= x|^{\lambda}\Theta_1; t_1|\ldots|^{\lambda}\Theta_1; t_1| & |\Gamma, x[\Delta]{:}A| &= |\Gamma|, x{:}|^{\Pi}\Delta; A| \\
|\lambda x[\Delta]{:}A.M| &= \lambda x{:}|^{\Pi}\Delta; A|.|M| & & \\
|\Pi x[\Delta]{:}A.M| &= \Pi x{:}|^{\Pi}\Delta; A|.|M| & |^{\sigma}\Gamma, x[\Delta]{:}A; M| &= |^{\sigma}\Gamma; \sigma x{:}|^{\Pi}\Delta; A|.M| \\
|P \cdot \langle\Delta\rangle Q| &= |P||^{\lambda}\Delta; Q| & |^{\sigma}\varepsilon; M| &= M
\end{aligned}
$$

Then we have

(i) If $\Gamma \vdash_{\lambda S^P} M{:}A$ then $|\Gamma| \vdash_{\lambda S'} |M|{:}|A|$;

(ii) If $M \to_\beta N$ then $|M| \to_\beta^+ |N|$.

For (1), since $\lambda S'$ is a completion of λS^P, we can form the types generated by $|^{\Pi}-; -|$ and those types can be used to type the terms generated by $|^{\lambda}-; -|$. The typability of the rest of the terms is not problematic. As for (2), from the definition of $|-|$ it is clear that a redex is mapped into a redex. However, each occurrence of a paramaterized variable after propagating the instantiation generates as many redexes as the number of its paramaters. After we reduce those, we are done. □

Corollary 3.23 *The systems of the λ-cube extended with hereditary parameters are strongly normalizing.*

Proof. The Extended Calculus of Constructions (ECC) of Luo [9] is a quasi-completion of all the systems of the λ-cube with hereditarily parameterized variables. Since ECC is strongly normalizing, by Theorem 3.22 each of the systems of the cube is strongly normalizing. □

4 Future Work

After obtaining a calculus that can express both open terms and the basic operations on them explicitly we intend to investigate the possibilities of extending it with operations that could make it applicable for modelling real tactics, not only tactic instances. To do that we need to internalize other essential operations like unification and recursion. Ultimately we would like to be able to have tactic terms like the one below that represents the propsitional tactic Apply:

$$
\begin{aligned}
&\mathsf{Apply}[\varphi, \psi : \mathsf{Prop}, thm : \psi] : \varphi := \\
&\quad (\varphi \sim \psi).thm \mid \\
&\quad ?A, B : \mathsf{Prop}.(\psi \sim A \to B).?m : A.\mathsf{Apply}[\varphi, B, (thm\ m)]
\end{aligned}
$$

When given two propositions φ and ψ and a proof of ψ this tactic tries to unify φ and ψ and if this is successful it returns a proof of ψ that in this case

is also a proof of φ. If the unification fails, it checks whether ψ is an arrow type by trying to unify it with $A \to B$ where A and B are meta-variables freshly introduced by the binder $?A, B$: Prop. If this is the case, the tactic makes a recursive call by eliminating the argument A with a freshly introduced meta-variable m.

Designing a calculus capable of representing tactics like Apply is a major challenge, but we hope that the present paper is the right first step towards this goal.

References

[1] Henk Barendregt. Lambda calculi with types. In Abramsky et al., editor, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[2] S. Berardi. Towards a methematical analysis of the Coquand-Huet calculus and the other systems in Barendregt's cube. Technical report, Carnegie-Mellon University and Universita di Torino, 1989.

[3] R. Bloo, Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. Parameters in Pure Type Systems. In *Proceedings of LATIN'02*. Springer, 2002.

[4] Mirna Bognar. *Contexts in Lambda Calculus*. PhD thesis, Vrije Universiteit Amsterdam, 2002.

[5] N.G. de Bruijn. A Survey of the Project AUTOMATH. In Hindley and Seldin, editors, *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[6] Herman Geuvers. *Logics and Type systems*. PhD thesis, University of Nijmegen, 1993.

[7] Herman Geuvers and G.I. Jojgov. Open Proofs and Open Terms: a Basis for Interactive Logic. In Bradfield, editor, *Proceedings of CSL'02*. Springer, 2002.

[8] Gueorgui Jojgov. Holes with Binding Power. In *Proceedings of TYPES'02*. Springer, 2003.

[9] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, July 1990.

[10] Zhaohui Luo. PAL$^+$: A lambda-free logical framework. *Journal of Functional Programming*, to appear.

[11] Lena Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf Monomorphic Type Theory with Explicit Substitutions*. PhD thesis, Chalmers University of Technology / Göteborg University, 1995.

[12] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[13] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1992.

[14] César A. Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory.* PhD thesis, INRIA, November 1997.

[15] P. Severi and E. Poll. Pure Type Systems with definitions. In *Proc. of LFCS'94, St. Petersburg, Russia*, number 813 in LNCS, Berlin, 1994. Springer Verlag.

[16] M. Strecker. *Construction and Deduction in Type Theories.* PhD thesis, Universität Ulm, 1999.

[17] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118, 1995.

[18] J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Technical report, University of Nijmegen, 1989.

On the Structure of Mizar Types

Grzegorz Bancerek [1]

Faculty of Computer Science
Białystok Technical University
Białystok, Poland

Abstract

The aim of this paper is to develop a formal theory of Mizar types. The examples are extracted from Mizar Mathematical Library (MML), some of them are simplified or presented in a bit different way. The presented theory is an approach to the structure of Mizar types as a sup-semilattice with widening (subtyping) relation as the order. It is an abstraction from the existing implementation of the Mizar verifier by Andrzej Trybulec and Czesław Byliński. The theory describes the structure of types of the base fragment of Mizar language.

1 Introduction

The MIZAR language has been developing by Andrzej Trybulec since 1973 (see [12,13,14]). It is an attempt to approximate in a formal way the common mathematical language (CML, see [5]) used in mathematical publications. MIZAR inherits a lot from CML's expressibility as well as naturalness and freedom/smoothness of reasoning. On the other hand, it is formal [2] enough to enable mechanical proof verification and other computer processing. As a result, MIZAR is successfully used for the practical formalization of mathematics. The papers [1,2,3,6,10,11] report on the state-of-art in this area. Mizar types are discussed in [3,10,11]. Mizar constructions including types are also described in [15] where examples of translation to untyped first-order syntax in DFG format are given. Introductory information on MIZAR can also be found in [4,9,7,17,8].

The goal of this paper is to give some rough description of the structure of Mizar types used by Mizar verifier which should remain stable notwithstanding dynamic changes in the implementation. On the other hand, the theory should be rich enough to enable discussion on new features in Mizar language. We do

[1] Email: bancerek@mizar.org

[2] The MIZAR syntax: http://mizar.org/language/syntax.html

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

not intend to investigate Mizar types in the spirit of type theory. Mizar types and adjectives correspond, more or less, to nouns and adjectives, respectively, in Weak Type Theory [5].

Let us mention differences between MIZAR and other systems for the formalization of mathematics with the computer (nice comparison of MIZAR and other 14 system is given in [16]). Firstly, MIZAR should be classified as a proof checker (there is no interaction with the user when the Mizar verifier provers user's hypotheses and the only answer is correct or not). Secondly, MIZAR deals with MML large cumulated human-readable data base of mathematical knowledge. MML is based on Tarski-Grothendieck set theory and Fitch-Jaśkowski proof system with classical logic. These differences have a significant impact on used type system.

The most important role of Mizar types is the following.

(i) When a variable is introduced its type is given.

(ii) For a term τ Mizar verifier computes $\theta(\tau)$, a unique type of τ. $\theta(\tau)$ is called *the type of* τ. We say that τ *has type* θ if the type of θ widens to θ, i.e., it is a subtype of θ.

(iii) Types are used in *Qualifying formula* of the form

$$\tau \text{ is } \theta$$

Note that the formula "τ is θ" might be true even if $\theta(\tau)$ and θ are not related.

Let us list some types occurring in MML chosen in a haphazard way. First, types with empty lists of arguments:

`set`, `non empty set`, `Relation`,
`reflexive transitive antisymmetric Relation`, `one-to-one Function`,
`complete LATTICE`, `solvable Group`

and types with arguments:

`Function of A,B`, `sups-preserving map of S,T`,
`normal Subgroup of G`

where `A` and `B` are of type `set`, `S` and `T` - `complete LATTICE`, and `G` - `Group`. As we see, Mizar types consist of 2 parts: a cluster of adjectives (possibly empty) and a radix type. The cluster of adjective for the last type listed above consists of one adjective `normal` and `Subgroup of G` is the radix type of it.

We must distinguish two concepts:

radix type — the construction,

mode — the constructor.

And similarly,

adjective – the construction,

attribute – the constructor.

MIZAR allows for dependent types. Namely, radix types and adjectives depend on terms only. Therefore, we may say that we get a radix type by applying a mode to a list of terms. The list has a fixed length and, more precisely, it has fixed types. An adjective is obtained by application of an attribute to a list of terms extracted from the radix type. But, we may not treat adjectives like type modifiers because of *Adjective formula* of the form

$$\tau \text{ is } \alpha$$

where list of terms for adjective α is extracted from $\theta(\tau)$.

The basic linguistic categories in MIZAR are *Type expression*, *Adjective*, *Term expression*, and *Formula expression*. All of them are mutually disjoint. For our purposes it is sufficient to use the following simplified syntax of *Type expression* and *Term expression*.

Figure 1.1 Simplified syntax

Type-expression = *Adjective-cluster Radix-type* .

Radix-type = *Mode-symbol* ["of" *Term-expression-list*] .

Adjective-cluster = { *Adjective* } .

Term-expression-list = *Term-expression* { "," *Term-expression* } .

Term-expression = *Variable* .

The syntax of Mizar types describes only the input for Mizar verifier. The verifier translates the input to some abstract form (constructor level in [15]) which for global items (theorems and definitions) is available from machine readable data base files. The abstract form is different from the input, e.g., hidden arguments are recognized and homonyms are distinguished. Therefore, Mizar types (as well as other Mizar object) cannot be treated syntactically.

2 An example of a type structure

Let us start the consideration with the following example of a Mizar text.

Two primitives of set theory, type `set` and predicate `in`, are introduced as built-in notions. According to MIZAR rules, primitives are introduced technically by definition but without definiens. So, the primitives may be formally declared as follows.

Figure 2.1 Built-in notions

```
definition  mode set;  end;

definition
  let x, y be set;
  pred x in y;
end;
```

Mode `set` has empty list of arguments and then it constructs only one type (also denoted by `set`). Type `set` is the widest type in MIZAR any Mizar type is a subtype of type `set`. Note that `set` is not a syntactic category in Mizar language unlike *Mode* and *Type expression*.

Now, we define equality `=` and inclusion `c=` as predicates,

Figure 2.2 Equality and inclusion

```
definition
  let x, y be set;
  pred x = y means
     for z being set holds z in x iff z in y;
  antonym x <> y;
  pred x c= y means
     for z being set st z in x holds z in y;
end;
```

and attribute `empty`

Figure 2.3 Empty

```
definition
  let x be set;
  attr x is empty means  not ex y being set st y in x;
end;
```

This *Attribute definition* introduces actually two adjectives: `empty` and `non empty`. The following *Existential registrations*

Figure 2.4 Empty set

```
definition
  cluster empty set;
   existence proof ... end;
  cluster non empty set;
   existence proof ... end;
end;
```

state the existence of objects of types registered. Namely, `cluster empty set` states the existence of a set which is empty and `cluster non empty set` states the existence of a set which is non empty. After these registrations, expressions `empty set` and `non empty set` become legal types. A *Mode definition*, as below, requires also a proof of existence of an object which satisfies a given condition.

Figure 2.5 Subset

```
definition
  let x be set;
  mode subset of x -> set means  it c= x;
  existence proof ... end;
end;
```

Type `set` is the mother type of the mode `subset of ...`. It means that `subset of x`, for any set `x`, widens to `set`. In other words, if some term is a `subset of x` it is also a `set`.

Next, we have two *Existential registrations*:

Figure 2.6 Empty subset

```
definition
  let x be set;
  cluster empty subset of x;
  existence proof ... end;
end;

definition
  let x be non empty set;
  cluster non empty subset of x;
  existence proof ... end;
end;
```

The following *Conditional registration* states that every subset of an empty set is empty.

Figure 2.7 Subset of an empty set

```
definition
  let x be empty set;
  cluster -> empty subset of x;
  coherence proof ... end;
end;
```

The registration above includes three elements: the list of antecedents between '`cluster`' and '`->`', the list of consequents after '`->`', and the type. In our case,

there are not any antecedents and only one consequent - adjective **empty**.

The fact that a subset of a subset of a set is a subset of the set, may be expressed by the redefinition of the mother type of the mode.

Figure 2.8 Subset of a subset

```
definition
  let x be set;
  let y be subset of x;
 redefine
  mode subset of y -> subset of x;
  coherence proof ... end;
end;
```

The correctness condition **coherence** needs the proof of the formula

```
for z being subset of y holds z is subset of x
```

The redefinition above introduces a variant of the mode **subset of** ... which has one extra implicit argument (**x**). The implicit argument is recognized from the type of the explicit argument (**y**). In this paper, the variant will be indicated by **subset(x) of y**. There is no such notation in Mizar language, but it reflects well the internal representation of the type.

One more attribute definition:

Figure 2.9 Proper

```
definition
  let x be set;
  let y be subset of x;
  attr y is proper means  y <> x;
end;
```

The attribute **proper** in the definition above has one implicit argument - the set **x**. This argument is inherited from **subset of** .... To avoid misunderstanding, the argument will be presented explicitly in this paper and the adjectives will look like **proper(x)** and **non proper(x)**. Unfortunately, in the current version of Mizar language it is not possible to write arguments of adjectives explicitly, and, consequently, to do a conditional registration like

```
let x be set;
let y be subset of x;
cluster proper(y) -> proper(x) subset of y;
```

But we may do the following registrations:

Figure 2.10 Proper subset

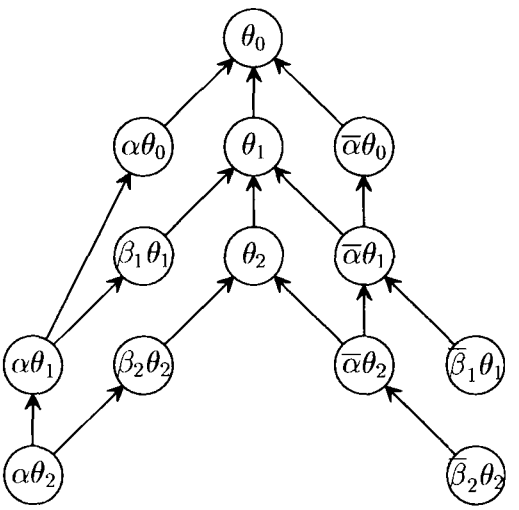
```
definition
  let x be set;
  cluster non proper subset of x;
  existence proof ... end;
end;

definition
  let x be empty set;
  cluster -> non proper subset of x;
  existence proof ... end;
end;

definition
  let x be non empty set;
  cluster empty -> proper subset of x;
   coherence proof ... end;
  cluster non proper -> non empty subset of x;
   coherence proof ... end;
  cluster proper subset of x;
   existence proof ... end;
end;
```

Let `x` denote a variable of the type `non empty set` and let `y` denote a variable of the type `non empty subset of x`. The structure of widening of types with arguments `x` and `y` may be expected to look as follows

Figure 2.11 First attempt to the structure of widening



where θ_0, θ_1, and θ_2 stand for `set`, `subset of x`, and `subset(x) of y` and α,

$\overline{\alpha}$, β_1, $\overline{\beta}_1$, β_2, and $\overline{\beta}_2$ stand for `empty`, `non empty`, `proper(x)`, `non proper(x)`, `proper(y)`, and `non proper(y)`, respectively.

The type $\alpha\theta_1$, i.e., `empty subset of x`, is equal to `empty proper(x) subset of x` according to the registration from Figure 2.10. Similarly, the type `non proper(x) subset of x` is equal to `non proper(x) non empty subset of x`, etc.

Variable `y` has two types: `subset of x` and `set`. The expression

```
y qua set
```

introduces a term which is equal to `y` but has only type `set`. Type

```
subset of (y qua set)
```

has the argument of type `set` and, in consequence, does not widen to `subset of x` like `subset(x) of y` does. However, type `subset(x) of y` which widens to `subset of x` widens also to `subset of (y qua set)`. The structure in Figure 2.11 does not include such types. Eventually, the structure of widening is more complicated. It is presented in Figure 2.12 for types with adjective `empty` and in Figure 2.13 for types with adjective `non empty`. Type `subset of (y qua set)` is represented by θ'_2.

Figure 2.12 Structure of widening: `empty`

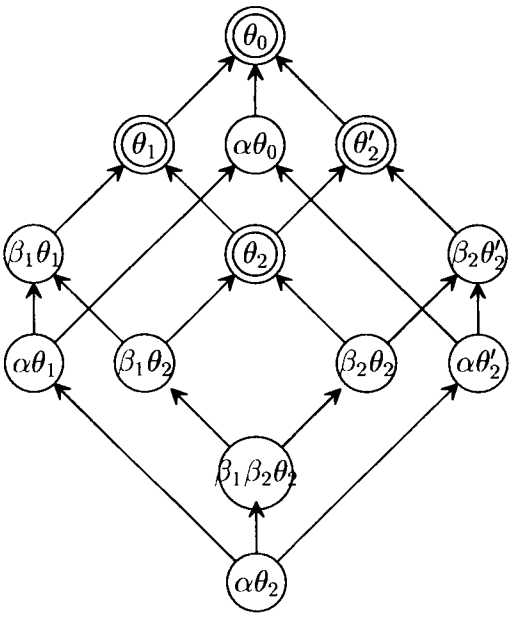
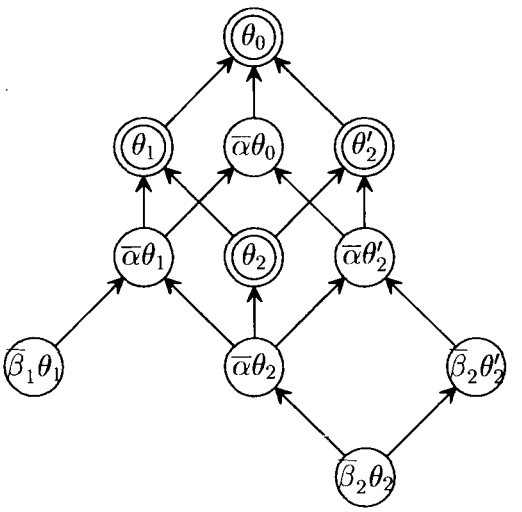


Figure 2.13 Structure of widening: `non empty`



3 Basic concepts

In this section we give an approach to the widening (subtyping) relation of Mizar types and connection of types and adjectives.

Definition 3.1 An upper-bounded poset $\mathcal{S} = \langle \mathit{Types}, \preceq, \top \rangle$ is called *a widening structure* if

(⊔) $\mathcal{S}$ is sup-semilattice and

(WF) $\mathcal{S}$ is Noetherian (the relation $\preceq$ is well-founded).

The ordering relation $\preceq$ on *Types* is called *widening relation* of the widening structure.

The condition (⊔) means that each pair $\{\theta_1, \theta_2\}$ of types has the least upper bound $\theta_1 \sqcup \theta_2$ w.r.t. $\preceq$. I.e.,

(1) $\theta_1 \preceq \theta_1 \sqcup \theta_2$ and $\theta_2 \preceq \theta_1 \sqcup \theta_2$,

(2) if $\theta_1 \preceq \theta$ and $\theta_2 \preceq \theta$, then $\theta_1 \sqcup \theta_2 \preceq \theta$

for any type θ.

The condition (WF) means that each non empty set T of types, $T \subseteq \mathit{Types}$, has maximal element w.r.t. $\preceq$. It means, also, that there is no infinite sequence of types

$$\theta_1 \prec \theta_2 \prec \theta_3 \prec \ldots$$

where $\prec$ is the irreflexive part of $\preceq$, i.e. $\prec = \preceq \setminus \mathrm{id}_{\mathit{Types}}$.

The greatest (widest) type $\top$ in the example is the type `set`. The widening relation is indicated by arrows, e.g.,

`proper(y) subset(x) of y` $\preceq$ `subset(x) of y` $\preceq$ `subset of x` $\preceq$ `set`

Fact 3.2 *Any ideal I of a widening structure has the maximum,* $\sup I \in I$.

Proof. I has maximal element θ by (WF). The type θ is the supremum of I because for any $\theta' \in I$ the type $\theta \sqcup \theta'$ belongs to I as I is an ideal. Then, $\theta \sqcup \theta' = \theta'$ by maximality. So, $\theta' \preceq \theta$. □

Definition 3.3 A tuple $\mathcal{A} = \langle \mathit{Types}, \mathit{Adjs}, \preceq, \top, \mathbf{non}, \mathit{adjs} \rangle$, where

$$\mathbf{non} : \mathit{Adjs} \to \mathit{Adjs},$$

$$\mathit{adjs} : \mathit{Types} \to \mathrm{Fin}(\mathit{Adjs})$$

is called *a structure of types and adjectives* or, simply, *a TA structure* if

(⊔,WF) $\mathcal{S} = \langle \mathit{Types}, \preceq, \top \rangle$ is a widening structure,

(**non**) **non** is an involution without fix points,

(X) if $\alpha \in \mathit{adjs}(\theta)$, then $\mathbf{non}\,\alpha \notin \mathit{adjs}(\theta)$,

(H) *adjs* is ⊔-homomorphism from $\mathcal{S}$ into $\mathrm{Fin}(\mathit{Adjs})^{\mathrm{op}}$.

$\mathrm{Fin}(\mathit{Adjs})$ is the set of all finite subsets of *Adjs* and $\mathrm{Fin}(\mathit{Adjs})^{\mathrm{op}} = \langle \mathrm{Fin}(\mathit{Adjs}), \supseteq \rangle$ is the sup-semilattice opposite to the semilattice of finite subsets with inclusion as ordering relation. *Adjs* is the set of *adjectives* and $\mathit{adjs}(\theta)$ is the set of all adjectives possessed by type θ.

The condition (**non**) means that for any adjective α

$$\mathbf{non}\,\mathbf{non}\,\alpha = \alpha \tag{3}$$

$$\mathbf{non}\,\alpha \neq \alpha \tag{4}$$

The condition (H) means that for all types θ_1 and θ_2,

$$\mathit{adjs}(\theta_1 \sqcup \theta_2) = \mathit{adjs}(\theta_1) \cap \mathit{adjs}(\theta_2) \tag{5}$$

and, particularly, *adjs* is antitone

$$\text{if } \theta_1 \preceq \theta_2, \text{ then } \mathit{adjs}(\theta_2) \subseteq \mathit{adjs}(\theta_1). \tag{6}$$

Note that the condition (H) does not imply $\mathit{adjs}(\top) = \emptyset$.

In the example the set of adjectives contains the following adjectives: `empty`, `non empty`, `proper(x)`, `non proper(x)`, `proper(y)`, and `non proper(y)`. In this case the operations from definition 3.3 are as follows:

$$\mathbf{non}(\texttt{empty}) = \texttt{non empty}$$

$$\mathbf{non}(\texttt{non empty}) = \texttt{empty}$$

$$\mathit{adjs}(\texttt{empty subset(x) of y}) = \{\texttt{empty}, \texttt{proper(x)}, \texttt{proper(y)}\}$$

$$\mathit{adjs}(\texttt{non proper(y) subset(x) of y}) = \{\texttt{non empty}, \texttt{non proper(y)}\}$$

etc.

Definition 3.4 We say that adjective α is an adjective of type θ or that θ is a type with adjective α if $\alpha \in \mathit{adjs}(\theta)$. The set of all types with adjective α is denoted by $\mathit{types}(\alpha)$,

$$\mathit{types}(\alpha) = \{\theta \in \mathit{Types} : \alpha \in \mathit{adjs}(\theta)\}. \tag{7}$$

In the example *types*(`empty`) includes

{`empty set`, `empty subset of x`, `empty subset of y`, `empty subset(x) of y`}

As simple conclusions of definitions 3.3 and 3.4 we get the following facts.

Fact 3.5 *Functions adjs and types are conjugate*

$$\alpha \in adjs(\theta) \text{ iff } \theta \in types(\alpha) \tag{8}$$

and

$$adjs(\theta) = \{\alpha \in Adjs : \theta \in types(\alpha)\}. \tag{9}$$

Fact 3.6 *Adjectives* α *and* **non** α *cannot appear in the same type,*

$$types(\alpha) \cap types(\mathbf{non}\,\alpha) = \emptyset. \tag{10}$$

Lemma 3.7 *The set* $types(\alpha)$ *is empty or an ideal in* $\mathcal{S}$.

Proof. The set $types(\alpha)$ is lower because if $\theta' \preceq \theta \in types(\alpha)$, then $\alpha \in adjs(\theta) \subseteq adjs(\theta')$ by (8) and (6). Thus $\theta' \in types(\alpha)$ again by (8). The set $types(\alpha)$ is directed because if θ, $\theta' \in types(\alpha)$, then $\alpha \in adjs(\theta) \cap adjs(\theta') = adjs(\theta \sqcup \theta')$ by (8) and (5). Eventually, $\theta \sqcup \theta' \in types(\alpha)$. □

4 Applying adjectives

The applicability of adjectives to types (modification of a type by an adjective) are disscused in this section.

Definition 4.1 An adjective α *is applicable to* the type θ if there is a type $\theta' \in types(\alpha)$ such that $\theta' \preceq \theta$.

The adjective α is applicable to the type θ if $\alpha \in adjs(\theta)$.

In the example, adjective `empty` is applicable to all types from Figure 2.12. Adjective `empty` is not applicable to type `non proper(x) subset of x`.

Let us note that if α is applicable to θ, then the set $\{\theta' \in types(\alpha) : \theta' \preceq \theta\}$ is an ideal, since it is a non empty intersection of two ideals:

$$types(\alpha) \qquad \text{and} \qquad \downarrow\theta = \{\theta' \in Types : \theta' \preceq \theta\}.$$

So, the set has the maximum by fact 3.2 and then the following definition is correct.

Definition 4.2 If an adjective α is applicable to a type θ, then *the application* of α to θ is defined by

$$\alpha * \theta = \sup\{\theta' \in types(\alpha) : \theta' \preceq \theta\}. \tag{11}$$

Corollary 4.3 *If the adjective* α *is applicable to the type* θ*, then the type* $\alpha * \theta$ *satisfies*

$$\alpha * \theta \preceq \theta, \tag{11.1}$$

$$\alpha \in adjs(\alpha * \theta), \tag{11.2}$$

(11.3) $$\alpha * \theta \in types(\alpha),$$

(11.4) *for each type* $\theta' \preceq \theta$ *if* $\alpha \in adjs(\theta')$, *then* $\theta' \preceq \alpha * \theta$.

Corollary 4.4 *If* $\alpha \in adjs(\theta)$, *then*

(11.5) $$\alpha * \theta = \theta.$$

Lemma 4.5 *If an adjective* α *is applicable to a type* θ *and an adjective* β *is applicable to the type* $\alpha * \theta$, *then* β *is applicable to* θ, α *is applicable to* $\beta * \theta$, *and*

(12) $$\alpha * (\beta * \theta) = \beta * (\alpha * \theta)$$

Proof. Sets $types(\alpha) \cap \downarrow\theta$ and $types(\beta) \cap \downarrow(\alpha * \theta)$ are non empty. Therefore there exists a type $\vartheta \in types(\alpha) \cap types(\beta)$ which widens to θ. It means that β is applicable to θ and α is applicable to $\beta * \theta$.

$$\begin{aligned}\alpha * (\beta * \theta) &= \sup\{\theta' \in types(\alpha) : \theta' \preceq \beta * \theta)\}\\ &= \sup\{\theta' \in types(\alpha) \cap types(\beta) : \theta' \preceq \theta)\}\\ &= \sup\{\theta' \in types(\beta) : \theta' \preceq \alpha * \theta)\}\\ &= \beta * (\alpha * \theta)\end{aligned}$$

□

In the example every adjective is applicable to `set`. Particularly, `proper(x)` is applicable to `set` and the result of application is `proper(x) subset of x`. So, we may apply an adjective with an argument which is not inherited from the type to which we apply it. The adjective `proper(x)` is also applicable to `subset of (y qua set)`,

`proper(x)` $*$ `subset of (y qua set)` = `proper(x) subset(x) of y`.

The last type is not expressible in Mizar language but it exists in internal process of Mizar verifier. These cases suggest that the application should be restricted to types widening to a type which can export needed arguments. This type will be called a *subject type* of an adjective. The subject type is the type from the definition of the attribute (after appropriate substitution):

`empty` and `non empty`	have subject type	`set`,
`proper(x)` and `non proper(x)`	have subject type	`subset of x`,
`proper(y)` and `non proper(y)`	have subject type	`subset of (y qua set)`.

The subject type of an adjective α is an upper bound of the sets $types(\alpha)$ and $types(\mathbf{non}\,\alpha)$. When both sets are non empty the subject type is equal to

$$\sup(types(\alpha) \cup types(\mathbf{non}\,\alpha)).$$

Such situation holds for `empty` and `proper(x)` as below. In the cases of `proper(z)`, where `z` denotes `set`, the set $types$(`proper(z)`) is empty (there is no existential registration with such an adjective) and

$\sup(types$(`proper(z)`$) \cup types$(`non proper(z)`$))$ = `non proper(z) subset of z`

which is not intended to be the subject type of `proper(z)`. On the other hand, when `e` denotes `empty set`, then `non proper(e)` $\in$ *adjs*(`subset of e`), then
types(`proper(e)`) $= \emptyset$ and

$$\sup(\mathit{types}(\texttt{proper(e)}) \cup \mathit{types}(\texttt{non proper(e)})) = \texttt{subset of e}.$$

Eventually, we introduce the following

Definition 4.6 A function $sub : Adjs \to Types$ is *a subject function* of a TA structure $\mathcal{A}$ if for any adjective α

(S0) *sub* absorbs operation **non**, i.e., $sub(\alpha) = sub(\mathbf{non}\,\alpha)$ for any $\alpha \in Adjs$,

(S1) $sub(\alpha)$ is an upper bound of the set $types(\alpha) \cup types(\mathbf{non}\,\alpha)$ and, moreover,

(S2) $sub(\alpha) = sup(types(\alpha) \cup types(\mathbf{non}\,\alpha))$ when both sets, $types(\alpha)$ and $types(\mathbf{non}\,\alpha)$, are non empty.

In our example we can use the subject function *sub* such that

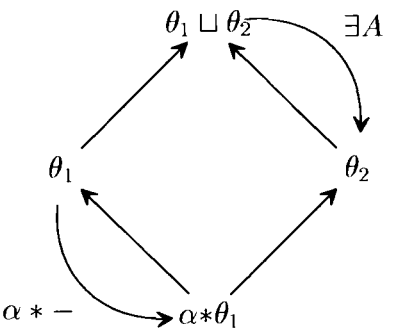
$$sub(\texttt{empty}) = \texttt{set},$$
$$sub(\texttt{proper(a)}) = \texttt{subset of (a qua set)}$$

for any `a`. If `a` is a variable of type `set`, then `subset of (a qua set)` is equal to `subset of a`.

Definition 4.7 Let $\mathcal{A}$ be a TA structure with subject function *sub*. An adjective α is *properly applicable* to a type θ if $\theta \preceq sub(\alpha)$ and α is applicable to θ. The set A of adjectives is *properly applicable* to θ if there exists a permutation $\alpha_1, \ldots, \alpha_n$ of A such that α_1 is properly applicable to θ and α_{i+1} is properly applicable to $\alpha_i * (\ldots * (\alpha_1 * \theta) \ldots)$ for $1 \leq i < n$. The type $\alpha_n * (\ldots * (\alpha_1 * \theta) \ldots)$ is *the application* of A to θ and is denoted by $A * \theta$.

In the example, `empty` is properly applicable to every type from Figure 2.12 but `proper(y)` is properly applicable to types widening to `subset of (y qua set)` only.

Definition 4.8 A TA structure $\mathcal{A}$ with subject function *sub* satisfies *commutativity law* if for any types θ_1, θ_2 and any adjective α such that α is properly applicable to θ_1 and $\alpha * \theta_1 \preceq \theta_2$, there exists a set of adjectives A properly applicable to $\theta_1 \sqcup \theta_2$ satisfying $A * (\theta_1 \sqcup \theta_2) = \theta_2$.



5 Radix types

Definability of radex types in introduced AT structure are presented in this section.

Definition 5.1 For any types θ and θ' we will write $\theta \circ\!\!\rightarrow \theta'$ if θ widens to θ' and there exists an adjective $\alpha \in adjs(\theta)$ such that $\alpha \notin adjs(\theta')$, α is properly applicable to θ', and $\alpha * \theta' = \theta$.

In the example we have:

```
                  empty subset(x) of y ○→ proper(x) proper(y) subset(x) of y
                  empty subset(x) of y ○→ proper(y) subset(x) of y
                  empty subset(x) of y ○→ subset(x) of y
proper(x) proper(y) subset(x) of y ○→ proper(y) subset(x) of y
          proper(y) subset(x) of y ○→ subset(x) of y
      non proper(x) subset(x) of y ○→ non empty subset(x) of y
      non proper(x) subset(x) of y ○→ subset(x) of y
          non empty subset(x) of y ○→ subset(x) of y
```

Fact 5.2 *Relation $\circ\!\!\rightarrow \subseteq \prec$. In consequence it is terminating.*

Lemma 5.3 *Let $\mathcal{A}$ be a TA structure with subject function sub. Assume that $\mathcal{A}$ satisfies commutativity law. Then the reduction $\circ\!\!\rightarrow$ has unique normal form property.*

Proof. It is enough to show that $\circ\!\!\rightarrow$ has weak Church Rosser property. Then let us assume that $\theta_1 \leftarrow\!\!\circ\ \theta \circ\!\!\rightarrow \theta_2$ for some types θ, θ_1, and θ_2. So, there exist adjectives α and β such that α is properly applicable to θ_1, β is properly applicable to θ_2, and $\alpha * \theta_1 = \theta = \beta * \theta_2$. Hence by commutativity law we may find sets A, $B \subseteq adjs(\theta)$ properly applicable to $\theta_1 \sqcup \theta_2$ such that $\theta_2 = B * (\theta_1 \sqcup \theta_2)$ and $\theta_1 = A * (\theta_1 \sqcup \theta_2)$. We choose A and B to be minimal w.r.t. inclusion. Let the permutations $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_m$ of A and B satisfy conditions of definition 4.7:

$$\theta_1 = \alpha_n * (\ldots * (\alpha_1 * (\theta_1 \sqcup \theta_2))), \qquad \theta_2 = \beta_m * (\ldots * (\beta_1 * (\theta_1 \sqcup \theta_2))).$$

Eventually, by minimality of A and B

$$\theta_1 \circ\!\!\rightarrow \alpha_{n-1} * (\ldots * (\alpha_1 * (\theta_1 \sqcup \theta_2))) \circ\!\!\rightarrow \ldots \circ\!\!\rightarrow \alpha_1 * (\theta_1 \sqcup \theta_2) \circ\!\!\rightarrow \theta_1 \sqcup \theta_2,$$

$$\theta_2 \circ\!\!\rightarrow \beta_{m-1} * (\ldots * (\beta_1 * (\theta_1 \sqcup \theta_2))) \circ\!\!\rightarrow \ldots \circ\!\!\rightarrow \beta_1 * (\theta_1 \sqcup \theta_2) \circ\!\!\rightarrow \theta_1 \sqcup \theta_2,$$

what ends the proof. □

Definition 5.4 The *radix type* of a type θ, denoted by $radix(\theta)$, is the unique normal form of θ with respect to the reduction $\circ\!\!\rightarrow$.

Fact 5.5 *For any type θ,*

$$radix(\theta) \preceq \theta.$$

Proof. It is an immediate consequence of fact 5.2 and transitivity of $\preceq$ ($\prec$).□

Theorem 5.6 *The radix type may be defined by*

$$radix(\theta) = \sup\{\theta' \in Types : \exists_{A \text{ properly applicable to } \theta}\ A * \theta' = \theta\} \tag{13}$$

Proof. For a derivation of the normal form of θ we have

$$\theta = \alpha_1 * \theta_1 \circ\!\!\rightarrow \theta_1 = \alpha_2 * \theta_2 \circ\!\!\rightarrow \ldots \circ\!\!\rightarrow \theta_{n-1} = \alpha_n * \theta_n \circ\!\!\rightarrow \theta_n = radix(\theta) \tag{14}$$

Then, the set $\{\alpha_1, \ldots, \alpha_n\}$ is properly applicable to $radix(\theta)$ and $\theta = \{\alpha_1, \ldots, \alpha_n\} * radix(\theta)$. Hence,

$$radix(\theta) \preceq \sup\{\theta' \in Types : \exists_{A \text{ properly applicable to } \theta}\ A * \theta' = \theta\}$$

Opposite widening is the result of commutativity law. Namely, let us observe that if α is properly applicable to ϑ and $\alpha * \vartheta = \theta \preceq radix(\theta)$, then there exists a set $A \subseteq adjs(\theta)$ such that A is properly applicable to $\vartheta \sqcup radix(\theta)$ and $A * (\vartheta \sqcup radix(\theta)) = radix(\theta)$. This means that $adjs(\vartheta \sqcup radix(\theta)) = adjs(radix(\theta))$ as the type $radix(\theta)$ is a normal form w.r.t. $\circ\!\!\rightarrow$. So, it also means that $\vartheta \sqcup radix(\theta) = radix(\theta)$ and, consequently, $\vartheta \preceq radix(\theta)$. Hence, using induction we may show that every type θ' from the set in (13) widens to $radix(\theta)$. □

Lemma 5.7 *Let α be an adjective properly applicable to a type θ. If $\alpha * \theta \preceq \theta' = radix(\theta')$, then $\theta \preceq \theta'$.*

Proof. As in the observation made in the proof of theorem 5.6, $\theta \sqcup radix(\theta') = \theta'$ and $\theta \preceq \theta'$. □

Lemma 5.8 *Function radix is monotone,*

$$\text{if } \theta \preceq \theta', \text{ then } radix(\theta) \preceq radix(\theta') \tag{15}$$

for any types θ and θ'.

Proof. Let us assume that $\theta \preceq \theta'$. Then $\theta \preceq radix(\theta')$ by fact 5.5. For type θ we have a derivation like (14). Applying lemma 5.7 inductively to types θ_i from the derivation we obtain $radix(\theta) \preceq radix(\theta')$. □

Lemma 5.9 *Application of an adjective does not change radix type,*

$$radix(\alpha * \theta) = radix(\theta) \tag{16}$$

for any adjective α properly applicable to a type θ.

Proof. $\alpha * \theta \circ\!\!\rightarrow \theta$ or $\alpha * \theta = \theta$. Then $radix(\alpha * \theta) = radix(\theta)$ by uniqueness of a normal form. □

6 Further work

As the further work we want to extend this theory to include other features of Mizar language. Simultaneously, the theory is formalized in MIZAR itself.

References

[1] Bancerek, G., *Development of the theory of continuous lattices in* MIZAR, in "Symbolic Computation and Automated Reasoning", M. Kerber and M. Kohlhase, Eds., A K Peters, 2001.

[2] Bancerek, G., N. Endou, and Y. Shidama, *Lim-inf convergence and its compactness*, Mechanized Mathematics and Its Applications, **2** (2002), 29–35.

[3] Bancerek, G., and P. Rudnicki, *A compendium of continuous lattices in* MIZAR, Journal of Automated Reasoning, **29** (2002), 189–224.

[4] Bonarska, E., "An introduction to PC Mizar," *Fondation Ph. le Hodey*, Brussels, 1990. `http://mizar.org/project/bonarska.ps.gz`

[5] Kamareddine, F., and R. Nederpelt, *A refinement of de Bruijn's formal language of mathematics*, to appear, 2003.

[6] Milewski, R., and Ch. Schwarzweller, *Algebraic requirements for the construction of polynomial rings*, Mechanized Mathematics and Its Applications, **2** (2002), 1–8.

[7] Muzalewski, M., "An outline to PC Mizar," *Fondation Ph. le Hodey*, Brussels, 1993.

[8] Nakamura, Y., "Mizar lectures notes," 4th edition. Shinshu University, Nagano, 2001. `http://markun.cs.shinshu-u.ac.jp/kiso/projects2/proofchecker/mizar/Mizar4/index-e.html`

[9] Rudnicki, P., *An overview of the Mizar project*, Preceedings of the 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology, Bastad, 1992. `http://mizar.org/project/MizarOverview.ps.gz`

[10] Rudnicki, P., Ch. Schwarzweller, and A. Trybulec, *Commutative algebra in the Mizar system.* Journal of Symbolic Computation, **32** (2001), 143–169.

[11] Rudnicki, P., and A. Trybulec, *On equivalents of well-foundedness*, Journal of Automated Reasoning, **23** (1999), 197–234.

[12] Trybulec, A., *Informationslogische sprache Mizar*, Dokumentation-Information, Heft 33, Ilmenau, 1977.

[13] Trybulec, A., *The Mizar-QC/6000 Logic Information Language*, ALLC Bulletin, **6** (1978), No 2.

[14] Trybulec, A., *The Mizar Logic Information Language*, Studies in Logic, Grammar and Rhetoric, **1** (1980), Białystok.

[15] Urban, J., *Translating Mizar for first order theorem provers*, in A. Asperti, B. Buchberger, J. H. Davebport (Eds.): "MKM 2003", LNCS 2594: 203-215, 2003.

[16] Wiedijk, F., *Comparing mathematical provers.*, in A. Asperti, B. Buchberger, J. H. Davebport (Eds.): "MKM 2003", LNCS 2594: 188-202, 2003.

[17] Wiedijk, F., "Mizar: An Impression," `http://www.cs.kun.nl/~freek/notes`

Explicit Substitutions à la de Bruijn: the local and global way

Fairouz Kamareddine [1]

School of Mathematical and Computational Sciences
Heriot-Watt Univ., Riccarton
Edinburgh EH14 4AS, Scotland

Alejandro Ríos [2]

Department of Computer Science
University of Buenos Aires
Buenos Aires, Argentina

Abstract

Kamareddine and Nederpelt [9], resp. Kamareddine and Ríos [11] gave two calculi of explicit of substitutions highly influenced by de Bruijn's notation of the λ-calculus. These calculi added to the explosive pool of work on explicit substitution in the past 15 years. As far as we know, calculi of explicit substitutions: a) are unable to handle *local substitutions*, and b) have answered (positively or negatively) the question of the *termination* of the underlying calculus of substitutions. The exception to a) is the calculus of [9] where substitution is handled both locally and globally. However, the calculus of [9] does not satisfy properties like confluence and termination. The exception to b) is the λs_e-calculus [11] for which termination of the s_e-calculus, the underlying calculus of substitutions, remains unsolved. This paper has two aims:

(i) To provide a calculus à la de Bruijn which deals with local substitution and whose underlying calculus of substitutions is terminating and confluent.

(ii) To pose the problem of the termination of the substitution calculus of [11] in the hope that it can generate interest as a termination problem which at least for curiosity, needs to be settled. The answer here can go either way. On the one hand, although the $\lambda\sigma$-calculus [1] does not preserve termination, the σ-calculus itself terminates. On the other hand, could the non-preservation of termination in the λs_e-calculus imply the non-termination of the s_e-calculus?

[1] Email: fairouz@macs.hw.ac.uk
[2] Email: rios@dc.uba.ar

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Given $(\lambda x.xx)y$, one may not be interested in having yy as the result but rather only $(\lambda x.yx)y$. In other words, only one occurrence of x is substituted by y and the substitution can be continued later. Such *local* substitution is a major issue in functional language implementation [15]. Yet, most calculi of explicit substitutions are not able to handle this process. This paper presents an explicit substitution calculus which is able to handle local substitution.

There are two main styles of explicit substitution: the $\lambda\sigma$- and the λs_e-styles. The $\lambda\sigma$-calculus [1] reflects in its choice of operators and rules the calculus of categorical combinators [3]. The main innovation of the $\lambda\sigma$-calculus is the division of terms in two sorts: sort `term` and sort `substitution`. λs_e [11] departs from this style of explicit substitutions in two ways. First, it keeps the classical and unique sort `term` of the λ-calculus. Second, it does not use some of the categorical operators, especially those which are not present in the classical λ-calculus. The λs_e has two new operators which reflect the substitution and updating that are present in the meta-language of the λ-calculus, and so it can be said to be closer to the λ-calculus from an intuitive point of view, rather than a categorical one. The λs_e is based on the λs-calculus [10] which is a refinement of the calculus of [9] that was influenced by the Automath style and, as a result was able to handle local as well as global substitutions. The calculus of [9] however does not enjoy confluence and termination and refining it into λs (and λs_e) led to loss of local substitutions. As far as we know any explicit substitution calculus other than that of [9] is unable to handle local substitutions. For a survey of calculi of explicit substitutions, and a comparison between both $\lambda\sigma$- and λs_e-styles, see [13].

The $\lambda\sigma$- and λs_e-calculi, although in different styles, enjoy some common properties: they are both confluent, they both fail to preserve the termination of the λ-calculus, and they both simulate β-reduction. However, although the underlying substitution calculus of $\lambda\sigma$ is known to be terminating, this question remains unsettled for λs_e. This is frustrating. This question has been settled for any other calculus of explicit substitutions, so why has it proved very hard for λs_e? This paper reports on the status of this question so far.

Since the calculus of [9] and the calculus of local substitutions we will give in this paper are better described in a notation [8] highly influenced by de Bruijn's λ-calculus, we will separate the section dealing with local substitutions from that dealing with the termination of s_e.

2 The local substitution calculus

Since we are going to discuss and continue the work of [9] we shall present our calculus in *item notation* (cf. [8]). In this notation we write $a\,b = (b\,\delta)a$, $\lambda a = (\lambda)a$, $a\,\sigma^i b = (b\,\sigma^i)a$ and $\varphi^i_k a = (\varphi^i_k)a$. The σ^i-operator is the operator for explicit substitution at level i and the φ^i_k-operator stands for the explicit

updating. The following nomenclature is used: $(b\,\delta)$, (λ), $(c\,\sigma^i)$, (φ^i_k) are called *items* (δ-, λ-, σ- and φ-items, respectively) and b and c the *bodies* of the respective items. A sequence of items is called a *segment*. Every term can be written as $\overline{s}\,\mathbf{n}$, where $\mathbf{n}$ is a de Bruijn index, with a convenient segment $\overline{s}$.

In order to treat local substitution [9] proposed the following rules:

$\sigma_{0\delta}$-*transition*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$((c\,\sigma^i)b\,\delta)a$	
$\sigma_{1\delta}$-*transition*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(b\,\delta)(c\,\sigma^i)a$	
σ-*destruction 1*	$(c\,\sigma^i)\mathtt{i}$	$\longrightarrow$	c	
σ-*destruction 2*	$(c\,\sigma^i)\mathtt{j}$	$\longrightarrow$	$\mathtt{j}$	if $\mathtt{j} \neq \mathtt{i}$

These rules are enough to prevent confluence. For example:
$(2\sigma^1)(1\,\delta)1 \to_{\sigma_{0\delta}\text{-}tr} ((2\sigma^1)1\,\delta)1 \to_{\sigma\text{-}dest\,1} (2\,\delta)1$
$(2\sigma^1)(1\,\delta)1 \to_{\sigma_{1\delta}\text{-}tr} (1\,\delta)(2\sigma^1)1 \to_{\sigma\text{-}dest\,1} (1\,\delta)2$

[9] gave a σ-*generation* rule to start β-reduction by generating a σ^1-operator:

σ-*generation*	$(b\,\delta)(\lambda)a$	$\longrightarrow$	$(b\,\delta)(\lambda)((\varphi^1_0)b\,\sigma^1)a$

Note that the starting δ-λ pair is kept after reduction. This enables the reuse of the rule to substitute another occurrence of the intended variable.

Considering only the rules introduced so far, the calculus presents another problem: terms which are strongly normalising in the classical λ-calculus, lose this property in the new calculus, and this occurs even if the application of the σ-*generation* rule is restricted to the case when the abstractor binds at least one occurrence of a de Bruijn number in a. Here is an example:
$(1\,\delta)(\lambda)(2\,\delta)1 \to_{\sigma-gen} (1\,\delta)(\lambda)((\varphi^1_0)1\,\sigma^1)(2\,\delta)1 \to_{\sigma_{0\delta}-tr}$
$(1\,\delta)(\lambda)(((\varphi^1_0)1\,\sigma^1)2\,\delta)1 \to_{\sigma-dest\,2} (1\,\delta)(\lambda)(2\,\delta)1 \to_{\sigma-gen} \cdots$

In order to solve the problem of confluence we will introduce a calculus where the rules $\sigma_{0\delta}$-*transition* and $\sigma_{1\delta}$-*transition* are modified as follows:

σ-δ-*transition 1*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(c\,\sigma^i)((c\,\sigma^i)b\,\delta)a$
σ-δ-*transition 2*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(c\,\sigma^i)(b\,\delta)(c\,\sigma^i)a$

Therefore, we shall be keeping the starting σ^i-item in order to reuse it. But we shall need a rule to dispose of this σ^i-item once all possible substitutions have been performed. We could try, for instance, the following:

σ-*disposal*	$(c\,\sigma^i)a$	$\longrightarrow$	a	if $\mathtt{i} \notin FV(a)$

But this rule is not enough to get rid of the σ^i-item. For example:
$(1\,\sigma^1)(1\,\delta)2 \to_{\sigma\text{-}\delta\text{-}tr\,1} (1\,\sigma^1)((1\,\sigma^1)1\,\delta)2 \to_{\sigma\text{-}dest\,1} (1\,\sigma^1)(1\,\delta)2 \to_{\sigma\text{-}\delta\text{-}tr\,1} \cdots$

The problem is that after the substitution is performed on the index 1 we have again 1 and hence 1 will always be free in the scope of $(1\sigma^1)$.

We can try to add the classical σ-δ-*transition* rule to ensure that the σ^i-item will be disposed of at some time:

σ-δ-*transition*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$((c\,\sigma^i)b\,\delta)(c\,\sigma^i)a$

But the inclusion of this rule forces us to *justify* the new calculus, since it

stands for *global* substitution and is *always* needed to dispose of the σ^i-items.

In principle, we have two choices for the *σ-generation* rule: either we keep it as in [9] (see above) or we decide not to preserve the δ-λ-pairs and we state it as it is usually given in calculi of explicit substitutions (cf. [10,12]):

new σ-generation $\qquad (b\,\delta)(\lambda)a \quad \longrightarrow \quad (b\,\sigma^1)a$

If we admit this *new σ-generation* rule and keep our choice of operators, we are going to end up with either the λs-calculus (cf. [10]) if we decide for global updatings, or with the λt-calculus (cf. [12]) if we decide for partial updatings. But, none of these calculi permit local substitutions. Also, since these calculi do not preserve the δ-λ-pairs, their σ-destruction rules must update the free variables and hence in both calculi we have:

σ-destruction 3 $\qquad (c\,\sigma^i)\mathtt{j} \quad \longrightarrow \quad \mathtt{j}-\mathtt{1} \qquad \text{if } \mathtt{j} > \mathtt{i}$

But with this rule and the new σ-δ-transition rules we lose confluence:
$(1\,\sigma^1)(3\,\delta)1 \to_{\sigma\text{-}\delta\text{-}tr\,1} (1\,\sigma^1)((1\,\sigma^1)3\,\delta)1 \to_{\sigma-dest\,3} (1\,\sigma^1)(2\,\delta)1 \to_{\sigma\text{-}\delta\text{-}tr}$
$((1\,\sigma^1)2\,\delta)(1\,\sigma^1)1 \to_{\sigma-dest\,3} (1\,\delta)(1\,\sigma^1)1 \to_{\sigma-dest\,1} (1\,\delta)1$
And the following derivation is also available:
$(1\,\sigma^1)(3\,\delta)1 \to_{\sigma\text{-}\delta\text{-}tr} ((1\,\sigma^1)3\,\delta)(1\,\sigma^1)1 \to_{\sigma-dest\,3} (2\,\delta)(1\,\sigma^1)1 \to_{\sigma-dest\,1} (2\,\delta)1$
Therefore, we discard the *new σ-generation* rule in order to avoid *σ-destruction 3* and choose to keep the first version of these rules.

Finally, since the *σ-generation* rule preserves the δ-λ pair we need a rule to dispose of the pair once all the possible substitutions have been carried on, i.e. when the abstractor in the δ-λ pair does not bind any de Bruijn index. When discarding the δ-λ pair we must update the de Bruijn numbers that stand for free variables. We shall perform this updating by introducing a new family of operators: μ^i and rewriting rules for their propagation.

2.1 A first attempt

With this intuition behind our calculus, we give a formal presentation.

Definition 2.1 The terms of the calculus are given by the following grammar:

$$\Lambda\sigma\mu ::= \mathbb{N} \mid (\Lambda\sigma\mu\,\delta)\Lambda\sigma\mu \mid (\lambda)\Lambda\sigma\mu \mid (\Lambda\sigma\mu\,\sigma^i)\Lambda\sigma\mu \mid (\varphi_k)\Lambda\sigma\mu \mid (\mu^i)\Lambda\sigma\mu$$

where $i \geq 1$, $k \geq 0$. We let a, b, c, ... range over $\Lambda\sigma\mu$.

Note that the updating operators contain only one index. This is because our calculus will work with partial updatings and therefore, as for the λt-calculus [12], the lower index is enough to deal with the updating mechanism.

The notion of free variable in our calculus needs the following definition:

Definition 2.2 Let $N \subset \mathbb{N}$ and $k \geq 0$. We define

(i) $N \setminus k = \{n-k : n \in N, n > k\}$, $N + k = \{n+k : n \in N\}$

(ii) $N_{\rho k} = \{n \in N : n \rho k\}$, where $\rho \in \{<, \leq, >, \geq\}$.

We can define now the free variables of a term in $\Lambda\sigma\mu$.

σ-*generation*	$(b\,\delta)(\lambda)a$	$\longrightarrow$	$(b\,\delta)(\lambda)((\varphi_0)b\,\sigma^1)a$	if $1 \in FV(a)$
μ-*generation*	$(b\,\delta)(\lambda)a$	$\longrightarrow$	$(\mu^1)a$	if $1 \notin FV(a)$
σ-λ-*transition*	$(b\,\sigma^i)(\lambda)a$	$\longrightarrow$	$(\lambda)((\varphi_0)b\,\sigma^{i+1})a$	
σ-δ-*transition*	$(b\,\sigma^i)(a_1\delta)a_2$	$\longrightarrow$	$((b\,\sigma^i)a_1\delta)\,(b\,\sigma^i)a_2$	
σ-δ-*transition 1*	$(b\,\sigma^i)(a_1\delta)a_2$	$\longrightarrow$	$(b\,\sigma^i)((b\,\sigma^i)a_1\delta)a_2$	
σ-δ-*transition 2*	$(b\,\sigma^i)(a_1\delta)a_2$	$\longrightarrow$	$(b\,\sigma^i)(a_1\delta)\,(b\,\sigma^i)a_2$	
σ-*destruction*	$(b\,\sigma^i)\mathbf{n}$	$\longrightarrow$	$\begin{cases} b & \text{if } n = i \\ \mathbf{n} & \text{if } n \neq i \end{cases}$	
φ-λ-*transition*	$(\varphi_k)(\lambda)a$	$\longrightarrow$	$(\lambda)(\varphi_{k+1})a$	
φ-δ-*transition*	$(\varphi_k)(a_1\delta)a_2$	$\longrightarrow$	$((\varphi_k)a_1\delta)(\varphi_k)a_2$	
φ-*destruction*	$(\varphi_k)\mathbf{n}$	$\longrightarrow$	$\begin{cases} \mathbf{n+1} & \text{if } n > k \\ \mathbf{n} & \text{if } n \leq k \end{cases}$	
μ-λ-*transition*	$(\mu^i)(\lambda)a$	$\longrightarrow$	$(\lambda)(\mu^{i+1})a$	
μ-δ-*transition*	$(\mu^i)(a_1\delta)a_2$	$\longrightarrow$	$((\mu^i)a_1\delta)(\mu^i)a_2$	
μ-*destruction*	$(\mu^i)\mathbf{n}$	$\longrightarrow$	$\begin{cases} \mathbf{n-1} & \text{if } n > i \\ \mathbf{n} & \text{if } n \leq i \end{cases}$	

Fig. 1. The $\lambda\sigma\mu$-calculus

Definition 2.3 The *set of free variables* of a term in $\Lambda\sigma\mu$ is defined by:

$FV(\mathbf{n}) = \{n\}$ $\qquad$ $FV((\varphi_k)a) = FV(a)_{\leq k} \cup (FV(a)_{>k} + 1)$

$FV((b\,\delta)a) = FV(b) \cup FV(a)$ $\qquad$ $FV((\mu^i)a) = FV(a)_{\leq i} \cup (FV(a)_{>i} \setminus 1)$

$FV((\lambda)a) = FV(a) \setminus 1$ $\qquad$ $FV((b\,\sigma^i)a) = FV(a)_{<i} \cup (FV(a)_{>i} \setminus 1) \cup FV(b)$

Definition 2.4 The *$\lambda\sigma\mu$-calculus* is the reduction system $(\Lambda\sigma\mu, \to)$ where $\to$ is the least compatible relation (with the operators of $\Lambda\sigma\mu$) generated by the set $\lambda\sigma\mu$ of rules in Figure 2.4. The calculus of substitutions associated with the $\lambda\sigma\mu$-calculus is the reduction system generated by the set $\lambda\sigma\mu - \{\sigma\text{-}generation, \mu\text{-}generation\}$ and we call it the $\sigma\mu$-calculus.

Note that the problem of loss of strong normalisation for terms which are strongly normalising in the classical λ-calculus still persists. For example:

$(1\,\sigma^1)(2\,\delta)1 \to_{\sigma\text{-}\delta\text{-}tr\,1} (1\,\sigma^1)((1\,\sigma^1)2\,\delta)1 \to_{\sigma-dest} (1\,\sigma^1)(2\,\delta)1 \to \cdots$

Note also that this calculus is not confluent. E.g., let $a = ((1\,\delta)1\,\sigma^1)(2\,\delta)1$:

$a \to_{\sigma\text{-}\delta\text{-}tr\,2} ((1\,\delta)1\,\sigma^1)(2\,\delta)((1\,\delta)1\,\sigma^1)1 \to_{\sigma-dest} ((1\,\delta)1\,\sigma^1)(2\,\delta)(1\,\delta)1 \to_{\sigma\text{-}\delta\text{-}tr\,2}$
$((1\,\delta)1\,\sigma^1)(2\,\delta)((1\,\delta)1\,\sigma^1)(1\,\delta)1 \twoheadrightarrow_{\sigma\text{-}\delta\text{-}tr,\sigma-dest} (2\,\delta)((1\,\delta)1\,\delta)(1\,\delta)1$

But also $a \to_{\sigma\text{-}\delta\text{-}tr} (((1\,\delta)1\,\sigma^1)2\,\delta)((1\,\delta 1)\sigma^1)1 \twoheadrightarrow_{\sigma-dest} (2\,\delta)(1\,\delta)1$

Finally, $\lambda\sigma\mu$ not only does not solve the problem of confluence and does not preserve strong normalisation, but it also is not a first order rewriting system in the classical sense since both generation rules are conditional and extra and maybe costly calculations must be performed to evaluate the conditions.

The only properties that we have proved, concern a subsystem of $\lambda\sigma\mu$, we call it $\sigma\mu^-$, and is obtained by deleting σ-δ-*tr 1* and σ-δ-*tr 2* from $\sigma\mu$.

Lemma 2.5 *$\sigma\mu^-$ is SN and CR and the set of $\sigma\mu^-$-normal forms is exactly the set of pure terms.*

Proof. Analogous to the proof of this property for λt [12]. □

2.2 A better attempt

In the previous section we have given several counterexamples, the majority of which are based on the fact that rules like σ-δ-*tr 1* and σ-δ-*tr 2* can be used several times to perform the *same* substitution. Therefore, these rules are not adequate to formalise the notion of local substitution.

In order to prevent a rule like σ-δ-*tr 1* to evaluate the same substitution several times we are going to introduce a unary operator L to mark the term where the substitution has been locally performed and we will not allow the substitution to be evaluated again on marked terms. Let us try the following:

preliminary σ-δ-local 1 $\quad (c\,\sigma^i)(b\,\delta)a \longrightarrow (c\,\sigma^i)((L)(c\,\sigma^i)b\,\delta)a$

Now this rule poses still the problem of normalisation:
$(c\,\sigma^i)(b\,\delta)a \to (c\,\sigma^i)((L)(c\,\sigma^i)b\,\delta)a \to (c\,\sigma^i)((L)(c\,\sigma^i)(L)(c\,\sigma^i)b\,\delta)a \to \cdots$

To prevent this we propose to introduce another family of σ-operators that we denote σ^i_{Loc} and we modify the rule as follows:

σ-δ-local 1 $\quad (c\,\sigma^i)(b\,\delta)a \longrightarrow (c\,\sigma^i_{Loc})((L)(c\,\sigma^i)b\,\delta)a$

And to dispose of these new operators we add:

σ_{Loc}-disposal 1 $\quad (c\,\sigma^i_{Loc})((L)b\,\delta)a \longrightarrow (b\,\delta)(c\,\sigma^i)a$

We must also add the following, in order to be able to perform local substitution in the other branch of the application:

σ-δ-local 2 $\quad (c\,\sigma^i)(b\,\delta)a \longrightarrow (c\,\sigma^i_{Loc})(b\,\delta)(L)(c\,\sigma^i)a$

σ_{Loc}-disposal 2 $\quad (c\,\sigma^i_{Loc})(b\,\delta)(L)a \longrightarrow ((c\,\sigma^i)b\,\delta)a$

We are approaching the right solution but the confluence problem persist:
$(c\,\sigma^i_{Loc})((L)b\,\delta)(L)a \to_{\sigma_{Loc}-disp\,1} (b\,\delta)(c\,\sigma^i)(L)a$
And on the other hand $(c\,\sigma^i_{Loc})((L)b\,\delta)(L)a \to_{\sigma_{Loc}-disp\,2} ((c\,\sigma^i)(L)b\,\delta)a$

But this problem has an easy solution: split the family of operators σ^i_{Loc} into one family that stands for the local substitution performed in the left branch of the application and another family for the right branch. We denote these families σ^i_L and σ^i_R, respectively. Hence, we propose:

σ_R-*generation*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(c\,\sigma^i_R)((L)(c\,\sigma^i)b\,\delta)a$
σ_R-*destruction*	$(c\,\sigma^i_R)((L)b\,\delta)a$	$\longrightarrow$	$(b\,\delta)(c\,\sigma^i)a$
σ_L-*generation*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(c\,\sigma^i_L)(b\,\delta)(L)(c\,\sigma^i)a$
σ_L-*destruction*	$(c\,\sigma^i_L)(b\,\delta)(L)a$	$\longrightarrow$	$((c\,\sigma^i)b\,\delta)a$

With this formulation we solve several problems at the same time: the distinction between σ^i_R and σ^i_L allow us to obtain confluence for the calculus of substitution and the distinction between σ^i-operators on one hand and σ^i_R and σ^i_L on the other is a good sign for the preservation of strong normalisation. Moreover, with this formulation we are not forced to preserve the δ-λ pairs and hence we do not need to introduce conditions on free or bound variables and furthermore we do not need the introduction of the μ-operator and all the rules that it generates. Now, we present formally the calculus:

Definition 2.6 The terms of the calculus are given by the following grammar:

$$\Lambda s_L ::= \mathbb{N} \mid (\Lambda s_L\,\delta)\Lambda s_L \mid (L)\Lambda s_L \mid (\lambda)\Lambda s_L \mid (\Lambda s_L\,\sigma^i)\Lambda s_L \mid (\varphi^i_k)\Lambda s_L$$
$$\mid (\Lambda s_L\,\sigma^i_L)\Lambda s_L \mid (\Lambda s_L\,\sigma^i_R)\Lambda s_L \quad \text{where } i \geq 1,\ k \geq 0$$

We let a, b, c, ... range over Λs_L. Note that we come back to the updating operators of λs. In fact, the calculus we will define is λs where σ-δ-*transition* is replaced by the four rules above. Note also that $\Lambda \subset \Lambda s \subset \Lambda s_L$.

Definition 2.7 The λs_L-*calculus* is the reduction system $(\Lambda s_L, \rightarrow_{\lambda s_L})$, where $\rightarrow_{\lambda s_L}$ is the least compatible reduction on Λs_L generated by the rules in Figure 2. We use λs_L to denote this set of rules. The calculus of substitutions associated with the λs_L-calculus is the reduction system generated by the set $\lambda s_L - \{\sigma\text{-}generation\}$ and we call it the σ_L-calculus.

Lemma 2.8 shows that the λs-calculus can be simulated in the λs_L-calculus:

Lemma 2.8 *Let a, $b \in \Lambda s$, if $a \rightarrow_{\lambda s} b$ then $a \rightarrow_{\lambda s_L} b$.*

Proof. It is enough to show that the σ-δ-*transition* rule can be simulated in the λs_L-calculus. This may be done by consecutive application either of σ_R-*generation* and σ_R-*destruction* or σ_L-*generation* and σ_L-*destruction*. □

We conclude now that the λs_L-calculus simulates classical β-reduction:

Corollary 2.9 *Let a, $b \in \Lambda$, if $a \rightarrow_\beta b$ then $a \rightarrow_{\lambda s_L} b$.*

Proof. Using the previous lemma and the simulation of β in λs (cf. [10]). □

We are going to prove now confluence and strong normalisation of the σ_L-calculus, in order to have existence and uniqueness of σ_L-normal forms.

Lemma 2.10 *The σ_L-calculus is locally confluent.*

σ-*generation*	$(b\,\delta)(\lambda)a$	$\longrightarrow$	$(b\,\sigma^1)a$
σ-λ-*transition*	$(b\,\sigma^j)(\lambda)a$	$\longrightarrow$	$(\lambda)(b\,\sigma^{j+1})a$
σ_R-*generation*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(c\,\sigma^i_R)((L)(c\,\sigma^i)b\,\delta)a$
σ_R-*destruction*	$(c\,\sigma^i_R)((L)b\,\delta)a$	$\longrightarrow$	$(b\,\delta)(c\,\sigma^i)a$
σ_L-*generation*	$(c\,\sigma^i)(b\,\delta)a$	$\longrightarrow$	$(c\,\sigma^i_L)(b\,\delta)(L)(c\,\sigma^i)a$
σ_L-*destruction*	$(c\,\sigma^i_L)(b\,\delta)(L)a$	$\longrightarrow$	$((c\,\sigma^i)b\,\delta)a$
σ-*destruction*	$(b\,\sigma^j)\mathbf{n}$	$\longrightarrow$	$\begin{cases} \mathbf{n-1} \text{ if } n > j \\ (\varphi^j_0)b \text{ if } n = j \\ \mathbf{n} \quad \text{if } n < j \end{cases}$
φ-λ-*transition*	$(\varphi^i_k)(\lambda)a$	$\longrightarrow$	$(\lambda)(\varphi^i_{k+1})a$
φ-δ-*transition*	$(\varphi^i_k)(a_1\delta)a_2$	$\longrightarrow$	$((\varphi^i_k)a_1\delta)(\varphi^i_k)a_2$
φ-*destruction*	$(\varphi^i_k)\mathbf{n}$	$\longrightarrow$	$\begin{cases} \mathbf{n+i-1} \text{ if } n > k \\ \mathbf{n} \quad \text{if } n \leq k \end{cases}$

Fig. 2. The λs_L-calculus

Proof. By Knuth-Bendix Theorem it is enough to study the critical pairs. There is only one, namely the one generated by the σ_R-*generation* and σ_L-*generation* rules. It can be closed using σ_L-*destruction* and σ_R-*destruction.* □

The proof of SN is not immediate. We envisage a proof by structural induction split in the following lemmas. We note SN the set of terms in Λs_L which are σ_L-strongly normalising. Our aim is to prove that $SN = \Lambda s_L$.

Lemma 2.11 *Let a, $b \in \Lambda s_L$, the following hold:*

(i) $(b\,\delta)a \in SN$ *iff* $a \in SN$ *and* $b \in SN$.

(ii) $(\lambda)a \in SN$ *iff* $a \in SN$.

(iii) $(L)a \in SN$ *iff* $a \in SN$.

Proof. No σ_L-rule has an application, an abstraction or a mark at the root.□

In the following lemmas we use the notation: $\lg(a)$ stands for the length of term a and is defined as usual, $\mathrm{dp}(a)$ stands for the depth of a, i.e. the length of the longest derivation to its σ_L-normal form. We use $\mathrm{dp}(a)$ for $a \in SN$.

Lemma 2.12 *For $i \geq 1$ and $k \geq 0$, if $a \in SN$ then $(\varphi^i_k)a \in SN$.*

Proof. By induction on the ordinal $(\mathrm{dp}(a), \lg(a))$.

If $(\mathrm{dp}(a), \lg(a)) = (0, 1)$ then $a = \mathbf{n}$; obvious. If $\varphi^i_k a$ is a normal form, then obvious. Therefore we study all possible reducts of $(\varphi^i_k)a$ and prove them SN.

If $(\varphi_k^i)a \to (\varphi_k^i)b$, with $a \to b$, we conclude by IH since $\mathrm{dp}(a) > \mathrm{dp}(b)$.

If the reduction is at the root we must analyse the three possible rules. We just study *φ-δ-transition*: We have $(\varphi_k^i)((b\,\delta)c) \to ((\varphi_k^i)b\,\delta)(\varphi_k^i)c$. Now $\mathrm{dp}((b\,\delta)c) \geq \mathrm{dp}(b)$, $\mathrm{dp}((b\,\delta)c) \geq \mathrm{dp}(c)$, $\mathrm{lg}((b\,\delta)c) > \mathrm{lg}(b)$ and $\mathrm{lg}((b\,\delta)c) > \mathrm{lg}(c)$. Hence by IH, $(\varphi_k^i)b \in SN$ and $(\varphi_k^i)c \in SN$, and we use Lemma 2.11.1. □

Lemma 2.13 *For $i \geq 1$, if a, $b \in SN$ then $(b\,\sigma^i)a \in SN$.*

Proof. By induction on the ordinal $(\mathrm{dp}(a), \mathrm{lg}(a), \mathrm{dp}(b))$.

If $(\mathrm{dp}(a), \mathrm{lg}(a), \mathrm{dp}(b)) = (0, 1, 0)$, then $a = \mathbf{n}$ and b is in normal form. The result is obvious if $n \neq i$, whereas if $n = i$ we use the previous lemma.

The proof follows now the lines of the previous lemma, but an interesting case arises when considering the reduction at the root by the *σ_R-generation* or the *σ_L-generation* rule. Let us study for instance the latter.

Therefore we have $a = (d\,\delta)c$ and $(b\,\sigma^i)(d\,\delta)c \to (b\,\sigma_L^i)(d\,\delta)(L)(b\,\sigma^i)c$. Let us assume that $(b\,\sigma_L^i)(d\,\delta)(L)(b\,\sigma^i)c \notin SN$.

Since $\mathrm{dp}((d\,\delta)c) \geq \mathrm{dp}(c)$ and $\mathrm{lg}((d\,\delta)c) > \mathrm{lg}(c)$, by IH we have $(b\,\sigma^i)c \in SN$ and by Lemma 2.11.3, $L((b\,\sigma^i)c) \in SN$. Now, since $a \in SN$, we have $d \in SN$, and by Lemma 2.11.1, we conclude $(d\,\delta)(L)(b\,\sigma^i)c \in SN$.

Therefore, since $b \in SN$, there must be an infinite derivation beginning at $(b\,\sigma_L^i)(d\,\delta)(L)(b\,\sigma^i)c$ which reduces at the root. Furthermore, since there are no rules which reduce applications or marks, there exist d', c', b' such that $d \twoheadrightarrow d'$, $(b\,\sigma^i)c \twoheadrightarrow c'$, $b \twoheadrightarrow b'$ and $(b\,\sigma_L^i)(d\,\delta)(L)(b\,\sigma^i)c \twoheadrightarrow (b'\,\sigma_L^i)(d'\,\delta)(L)c' \to ((b'\,\sigma^i)d'\,\delta)c' \to \cdots$ But the fact that this derivation is infinite is a contradiction because by IH, we have $(b\,\sigma^i)c \in SN$, and hence $c' \in SN$, and also by IH we have $(b\,\sigma^i)d \in SN$, and hence $(b'\,\sigma^i)d' \in SN$. Therefore, by Lemma 2.11.1, $((b'\,\sigma^i)d'\,\delta)c' \in SN$. We conclude that $(b\,\sigma_L^i)(d\,\delta)(L)(b\,\sigma^i)c$ must be SN. □

Lemma 2.14 *For $i \geq 1$, if a, $b \in SN$ then $(b\,\sigma_L^i)a \in SN$ and $(b\,\sigma_R^i)a \in SN$.*

Proof. By induction on the ordinal $(\mathrm{dp}(a), \mathrm{dp}(b))$. Use the previous lemma when considering the reduction at the root. □

Theorem 2.15 *The σ_L-calculus is strongly normalising.*

Proof. By induction on a we prove that every $a \in \Lambda s_L$ is SN.

If $a = \mathbf{n}$, it is obviously SN.

If $a = (c\,\delta)b$ or $a = (\lambda)b$ or $a = (L)b$, use Lemma 2.11.

If $a = (\varphi_k^i)b$ use Lemma 2.12.

If $a = (c\,\sigma^i)b$ use Lemma 2.13.

If $a = (c\,\sigma_L^i)b$ or $a = (c\,\sigma_R^i)b$ use Lemma 2.14. □

Theorem 2.16 *The σ_L-calculus is confluent.*

Proof. By Newman's Lemma, the previous lemma and Lemma 2.10. □

3 The status of the open question of termination of s_e

The λs_e-calculus, like the $\lambda\sigma$-calculus, simulates β-reduction, is confluent (on open terms[3]) [11] and does not preserve strong normalisation (we say does not have PSN) [6]. However, although strong normalisation (SN) of the σ-calculus (the substitution calculus associated with the $\lambda\sigma$-calculus) has been established, it is still unknown whether strong normalisation of the s_e-calculus (the substitution calculus associated with the λs_e-calculus) holds. Only weak normalisation of the s_e-calculus is known so far [11].

The s_e-calculus (see Definition 3.8) has the *σ-σ-transition* rule which seems to be responsible for the difficulties in establishing SN of s_e. However, Zantema showed that the *σ-σ-transition* scheme on its own is SN [11].

This section is a discussion of the status of strong normalisation of the s_e-calculus. We show that the set of rules s_e is the union of two disjoint sets of rules *σ-σ-tr.*+*φ-σ-tr.* and the rest of the rules where each of these two sets gives a calculus which is SN. However, commutation does not hold and hence modularity cannot be used to obtain SN of s_e. In addition, the distribution elimination [17] and recursive path ordering methods are not applicable and we remain unsure whether s_e is actually SN or not.

3.1 The classical λ-calculus in de Bruijn notation

We assume the reader familiar with de Bruijn notation [5]. We define Λ, the *set of terms with de Bruijn indices*, by: $\Lambda ::= \mathbb{N} \mid (\Lambda\Lambda) \mid (\lambda\Lambda)$.
We use $a, b, \ldots$ to range over Λ and $m, n, \ldots$ to range over $\mathbb{N}$ (positive natural numbers). Furthermore, we assume the usual conventions about parentheses and avoid them when no confusion occurs. Throughout the whole article, $a = b$ is used to mean that a and b are syntactically identical. We write $\rightarrow^+$ and $\twoheadrightarrow$ to denote the transitive and the reflexive transitive closures of a reduction notion $\rightarrow$. We say that a reduction $\rightarrow$ is *compatible on* Λ when for all a, b, $c \in \Lambda$, we have $a \rightarrow b$ implies $a\,c \rightarrow b\,c$, $c\,a \rightarrow c\,b$ and $\lambda a \rightarrow \lambda b$.

As β-reduction à la de Bruijn involves the substitution of a variable $\mathtt{n}$ for a term b in a term a, we need to update the terms:

Definition 3.1 Let the *updating functions* $U_k^i : \Lambda \rightarrow \Lambda$ for $k \geq 0$ be $i \geq 1$ be:

$$U_k^i(ab) = U_k^i(a)\,U_k^i(b) \qquad U_k^i(\mathtt{n}) = \begin{cases} \mathtt{n} + \mathtt{i} - 1 & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k\,. \end{cases}$$

$$U_k^i(\lambda a) = \lambda(U_{k+1}^i(a))$$

Definition 3.2 The *meta-substitutions at level j*, for $j \geq 1$, of a term $b \in \Lambda$ in a term $a \in \Lambda$, denoted $a\{\!\{\mathtt{j} \leftarrow b\}\!\}$, is defined inductively on a as follows:

[3] The λs_e-calculus is confluent on the whole set of open terms whereas $\lambda\sigma$ is confluent on the open terms without metavariables of sort `substitution` as is shown in [16].

$$(a_1 a_2)\{\!\{\mathtt{j} \leftarrow b\}\!\} = (a_1\{\!\{\mathtt{j} \leftarrow b\}\!\})\,(a_2\{\!\{\mathtt{j} \leftarrow b\}\!\})$$

$$(\lambda a)\{\!\{\mathtt{j} \leftarrow b\}\!\} = \lambda(a\{\!\{\mathtt{j}+1 \leftarrow b\}\!\})$$

$$\mathtt{n}\{\!\{\mathtt{j} \leftarrow b\}\!\} = \begin{cases} \mathtt{n}-1 & \text{if } n > j \\ U_0^j(b) & \text{if } n = j \\ \mathtt{n} & \text{if } n < j\,. \end{cases}$$

Definition 3.3 *β-reduction* is the least compatible reduction on Λ generated by: (β-rule) $(\lambda a)\,b \to_\beta a\{\!\{\mathtt{1} \leftarrow b\}\!\}$
The *λ-calculus à la de Bruijn*, is the reduction system with rewriting rule β.

3.2 The λs- and λs_e-calculi

λs [10] handles explicitly the meta-operators of definitions 3.1 and 3.2. Hence, the syntax of the λs-calculus is obtained by adding two families of operators :

- $\{\sigma^j\}_{j\geq 1}$, which denotes the explicit substitution operators. The term $a\,\sigma^j b$ stands for term a where all free occurrences of the variable corresponding to de Bruijn index j are to be substituted by term b.
- $\{\varphi_k^i\}_{k\geq 0\ i\geq 1}$, which denotes the updating functions necessary when working with de Bruijn numbers to fix the variables of the term to be substituted.

Definition 3.4 The *set Λs of terms of the λs-calculus* is given as follows:

$$\Lambda s ::= I\!N \mid \Lambda s \Lambda s \mid \lambda \Lambda s \mid \Lambda s\,\sigma^j \Lambda s \mid \varphi_k^i \Lambda s \quad \textit{where} \quad j,\, i \geq 1\,,\ \ k \geq 0\,.$$

We take a, b, c to range over Λs. A term of the form $a\,\sigma^j b$ is called a *closure*. Furthermore, a term containing neither σ's nor φ's is called a *pure term*.

A reduction $\to$ on Λs is *compatible* if for all a, b, $c \in \Lambda s$, if $a \to b$ then $a\,c \to b\,c$, $c\,a \to c\,b$, $\lambda a \to \lambda b$, $a\,\sigma^j c \to b\,\sigma^j c$, $c\,\sigma^j a \to c\,\sigma^j b$ and $\varphi_k^i a \to \varphi_k^i b$.

To *σ-generation* which mimicks the β-rule, we add a set of rules which are the equations in definitions 3.1 and 3.2 oriented from left to right.

Definition 3.5 The *λs-calculus* is the reduction system $(\Lambda s, \to_{\lambda s})$, where $\to_{\lambda s}$ is the least compatible reduction on Λs generated by the set λs of the rules of Figure 3. The *s-calculus*, the *calculus of substitutions associated with the λs-calculus*, is the reduction system generated by the set $s = \lambda s - \{\sigma\text{-generation}\}$.

Lemma 3.6 (cf. [10]) *The following holds:*

(i) *(SN and CR of s) The s-calculus is strongly normalising and confluent on Λs. Hence, every term a has a unique s-normal form denoted $s(a)$.*

(ii) *The set of s-normal forms is exactly Λ.*

(iii) *For all a, $b \in \Lambda s$ we have:* $s(a\,b) = s(a)s(b)$, $s(\lambda a) = \lambda(s(a))$, $s(\varphi_k^i a) = U_k^i(s(a))$, $s(a\,\sigma^j b) = s(a)\{\!\{\mathtt{j} \leftarrow s(b)\}\!\}$.

(iv) *Let a, $b \in \Lambda s$, if $a \to_{\sigma-gen} b$ or $a \twoheadrightarrow_{\lambda s} b$ then $s(a) \twoheadrightarrow_\beta s(b)$.*

(v) *(Soundness) Let a, $b \in \Lambda$, if $a \twoheadrightarrow_{\lambda s} b$ then $a \twoheadrightarrow_\beta b$.*

(vi) *(Simulation of β-reduction) Let a, $b \in \Lambda$, if $a \to_\beta b$ then $a \twoheadrightarrow_{\lambda s} b$.*

(vii) *(CR of λs) The λs-calculus is confluent on Λs.*

σ-*generation*	$(\lambda a)\, b$	$\longrightarrow$	$a\,\sigma^1\, b$
σ-λ-*transition*	$(\lambda a)\, \sigma^j b$	$\longrightarrow$	$\lambda(a\sigma^{j+1} b)$
σ-*app-transition*	$(a_1\, a_2)\, \sigma^j b$	$\longrightarrow$	$(a_1\, \sigma^j b)\, (a_2\, \sigma^j b)$
σ-*destruction*	$\mathtt{n}\, \sigma^j b$	$\longrightarrow$	$\begin{cases} \mathtt{n}-1 & \text{if } n > j \\ \varphi_0^j\, b & \text{if } n = j \\ \mathtt{n} & \text{if } n < j \end{cases}$
φ-λ-*transition*	$\varphi_k^i(\lambda a)$	$\longrightarrow$	$\lambda(\varphi_{k+1}^i\, a)$
φ-*app-transition*	$\varphi_k^i(a_1\, a_2)$	$\longrightarrow$	$(\varphi_k^i\, a_1)\, (\varphi_k^i\, a_2)$
φ-*destruction*	$\varphi_k^i\, \mathtt{n}$	$\longrightarrow$	$\begin{cases} \mathtt{n}+\mathtt{i}-1 & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k \end{cases}$

Fig. 3. The λs-calculus

(viii) *(Preservation of SN) Pure terms which are strongly normalising in the λ-calculus are also strongly normalising in the λs-calculus.*

Open terms were introduced in the λs-calculus as follows (see [11]):

Definition 3.7 The set of *open terms*, noted Λs_{op} is given as follows:

$$\Lambda s_{op} ::= \mathbf{V} \mid I\!N \mid \Lambda s_{op} \Lambda s_{op} \mid \lambda \Lambda s_{op} \mid \Lambda s_{op}\, \sigma^j \Lambda s_{op} \mid \varphi_k^i \Lambda s_{op}$$

where $j,\, i \geq 1\,,\ k \geq 0$ and where $\mathbf{V}$ stands for a set of variables, over which $X,\, Y,\, \ldots$ range. We take $a,\, b,\, c$ to range over Λs_{op}. Furthermore, *closures*, *pure terms* and *compatibility* are defined as for Λs.

Working with open terms one loses confluence as shown by the example:

$$((\lambda X)Y)\sigma^1 1 \to (X\sigma^1 Y)\sigma^1 1 \qquad ((\lambda X)Y)\sigma^1 1 \to ((\lambda X)\sigma^1 1)(Y\sigma^1 1)$$

and $(X\sigma^1 Y)\sigma^1 1$ and $((\lambda X)\sigma^1 1)(Y\sigma^1 1)$ have no common reduct. This example also shows that even local confluence is lost. In order to solve this problem, [11] added to the λs-calculus a set of rules that guarantees confluence.

Definition 3.8 The set of rules λs_e is λs together with the rules of Figure 4. The λs_e-*calculus* is the reduction system $(\Lambda s_{op}, \to_{\lambda s_e})$ where $\to_{\lambda s_e}$ is the least compatible reduction on Λs_{op} generated by the set of rules λs_e. The s_e-*calculus*, the *calculus of substitutions associated with the λs_e-calculus*, is the rewriting system generated by the set of rules $s_e = \lambda s_e - \{\sigma\text{-}generation\}$.

Lemma 3.9 (cf. [11]) *The following holds:*

(i) *(WN and CR of s_e) The s_e-calculus is weakly normalising and confluent.*

(ii) *(Simulation of β-reduction) Let $a,\, b \in \Lambda$, if $a \to_\beta b$ then $a \twoheadrightarrow_{\lambda s_e} b$.*

(iii) *(CR of λs_e) The λs_e-calculus is confluent on open terms.*

σ-σ-*transition*	$(a\,\sigma^i b)\,\sigma^j\,c$	$\longrightarrow$	$(a\,\sigma^{j+1}\,c)\;\sigma^i\,(b\,\sigma^{j-i+1}\,c)$	if	$i \leq j$
σ-φ-*transition 1*	$(\varphi^i_k\,a)\,\sigma^j\,b$	$\longrightarrow$	$\varphi^{i-1}_k\,a$	if	$k < j < k+i$
σ-φ-*transition 2*	$(\varphi^i_k\,a)\,\sigma^j\,b$	$\longrightarrow$	$\varphi^i_k(a\,\sigma^{j-i+1}\,b)$	if	$k+i \leq j$
φ-σ-*transition*	$\varphi^i_k(a\,\sigma^j\,b)$	$\longrightarrow$	$(\varphi^i_{k+1}\,a)\,\sigma^j\,(\varphi^i_{k+1-j}\,b)$	if	$j \leq k+1$
φ-φ-*transition 1*	$\varphi^i_k\,(\varphi^j_l\,a)$	$\longrightarrow$	$\varphi^j_l\,(\varphi^i_{k+1-j}\,a)$	if	$l+j \leq k$
φ-φ-*transition 2*	$\varphi^i_k\,(\varphi^j_l\,a)$	$\longrightarrow$	$\varphi^{j+i-1}_l\,a$	if	$l \leq k < l+j$

Fig. 4. The extra rules of the λs_e-calculus

(Beta)	$(\lambda a)\,b$	$\longrightarrow$	$a\,[b \cdot id]$
(VarId)	$\mathbf{1}\,[id]$	$\longrightarrow$	$\mathbf{1}$
(VarCons)	$\mathbf{1}\,[a \cdot s]$	$\longrightarrow$	a
(App)	$(a\,b)[s]$	$\longrightarrow$	$(a\,[s])\,(b\,[s])$
(Abs)	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a\,[\mathbf{1} \cdot (s \circ \uparrow)])$
(Clos)	$(a\,[s])[t]$	$\longrightarrow$	$a\,[s \circ t]$
(IdL)	$id \circ s$	$\longrightarrow$	s
(ShiftId)	$\uparrow \circ id$	$\longrightarrow$	$\uparrow$
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s
(Map)	$(a \cdot s) \circ t$	$\longrightarrow$	$a\,[t] \cdot (s \circ t)$
(Ass)	$(s_1 \circ s_2) \circ s_3$	$\longrightarrow$	$s_1 \circ (s_2 \circ s_3)$

Fig. 5. The $\lambda\sigma$-calculus

(iv) *(Soundness) Let* $a, b \in \Lambda$, *if* $a \twoheadrightarrow_{\lambda s_e} b$ *then* $a \twoheadrightarrow_\beta b$.

3.3 The $\lambda\sigma$-calculus and the termination of the σ-calculus

Definition 3.10 The syntax of the *$\lambda\sigma$-calculus* [1] is given by:

$$\textbf{Terms} \qquad \Lambda\sigma^t ::= 1 \mid \Lambda\sigma^t\Lambda\sigma^t \mid \lambda\Lambda\sigma^t \mid \Lambda\sigma^t[\Lambda\sigma^s]$$

$$\textbf{Substitutions} \quad \Lambda\sigma^s ::= id \mid \uparrow \mid \Lambda\sigma^t \cdot \Lambda\sigma^s \mid \Lambda\sigma^s \circ \Lambda\sigma^s$$

The set, denoted $\lambda\sigma$, of rules of the *$\lambda\sigma$-calculus* is given in Figure 5.

The set of rules of the *σ-calculus* is $\lambda\sigma - \{(Beta)\}$. We use $a, b, c, \ldots$ to range over $\Lambda\sigma^t$ and $s, t, \ldots$ to range over $\Lambda\sigma^s$. For every substitution s we define the *iteration of the composition of* s inductively as $s^1 = s$ and $s^{n+1} = s \circ s^n$. We use the convention $s^0 = id$. Note that the only de Bruijn index used is $\mathbf{1}$, but we can code $\mathbf{n}$ as $\mathbf{1}[\uparrow^{n-1}]$. So, $\Lambda \subset \Lambda\sigma^t$.

Theorem 3.11 *The σ-calculus is strongly normalising (SN).*

There are various proofs of this theorem in the literature:

(i) The first strong normalisation proof of σ is based on the strong normalisation of $SUBST$ [7], which is, within CCL, the set of rewriting rules that compute the substitutions. See [7].

(ii) The proof in [4] shows the termination of σ via a strict translation from σ to another calculus σ_0 (an economic variant of σ) and the termination of σ_0. The calculus σ_0 is one sorted and treats both $\circ$ and [] as $\circ$, observing that $\circ$ and [] behave in the same way.

(iii) Zantema gives two proofs in [17,18]. The first is based on a suitable generalisation of polynomial orders to show the termination of the calculus σ_0 given below (and hence the termination of σ). The second uses semantic labelling to show the termination of σ.

We will explain why these techniques for showing SN of σ do not apply to s_e.

Definition 3.12 [The σ_0-calculus] The set of terms $\Lambda\sigma_0$ of the σ_0-calculus has the abstract syntax $s, t ::= 1 \mid id \mid \uparrow \mid \lambda s \mid s \circ t \mid s \cdot t$.
The set, denoted σ_0, of rules of the calculus is the following:

(VrId)	$1 \circ id \to 1$	(ShId)	$\uparrow \circ id \to \uparrow$
(VrCons)	$1 \circ (s \cdot t) \to s$	(Abs)	$(\lambda s) \circ t \to \lambda(s \circ (1 \cdot (t \circ \uparrow)))$
(ShCons)	$\uparrow \circ (s \cdot t) \to t$	(Map)	$(s \cdot t) \circ u \to (s \circ u) \cdot (t \circ u)$
(IdL)	$id \circ s \to s$	(Ass)	$(s \circ t) \circ u \to s \circ (t \circ u)$

Remark 3.13 σ_0 is a particular case of the system $Subst$ of CCL. Rules $(VrId)$ and $(ShId)$ are particular cases of the right identity rule. Hence, the techniques of (i) and (ii) above for showing SN for $SUBST$ and σ_0 will have similar status with respect to s_e.

The methods of techniques (i) .. (iii) above do not apply to s_e:

- **Problem 1: Unable to use recursive path ordering** By taking a look at the s_e-rules (Definition 3.8), it becomes obvious that the unfriendly rules, with respect to SN, are *σ-σ-transition* and to a lesser extent *φ-σ-transition.* These rules prevent us from finding an order on the set of operators in order to solve the normalisation problem with a recursive path ordering (rpo).
- **Problem 2: Unable to use Zantema's distribution elimination lemma.** The s_e-rules "look like" associative rules but unfortunately they are not; e.g. in *σ-σ-transition* one could think the σ^j-operator distributes over the σ^i-operator, but it is not a "true" distribution: σ^j changes to σ^{j+1} when acting on the first term and to σ^{j-i+1} when acting on the second. This prevents use of Zantema's distribution elimination method [17] to show SN.

Another technique to show SN is modularity where SN is proved for certain

subcalculi os s_e which are shown to satisfy a commutation property. We show in the next section that indeed s_e can be divided into two subcalculi which are SN, but that unfortunately, the needed commutation results do not hold.

3.4 Dividing s_e in two disjoint sets $s + *\varphi$ and $*\sigma$

Definition 3.14 We define the following sets of rules:
$*\varphi = \{\sigma\text{-}\varphi\text{-tr.1}, \sigma\text{-}\varphi\text{-tr.2}, \varphi\text{-}\varphi\text{-tr.1}, \varphi\text{-}\varphi\text{-tr.2}\}$,
$*\sigma = \{\sigma\text{-}\sigma\text{-tr.}, \varphi\text{-}\sigma\text{-tr.}\}$,
$*\varphi^{-} = \{\sigma\text{-}\varphi\text{-tr.1}, \varphi\text{-}\varphi\text{-tr.2}\}$, $*\varphi^{--} = \{\sigma\text{-}\varphi\text{-tr.2}, \varphi\text{-}\varphi\text{-tr.1}\}$.

Note that $s_e = (s+*\varphi)+*\sigma$. We shall prove in this section that both calculi generated by the set of rules $s+*\varphi$ (Theorem 3.17) and $*\sigma$ (Theorem 3.28) are SN. Unfortunately, these calculi do not possess the property of commutation needed to ensure that their union s_e is SN (see Example 3.31).

3.5 SN of $s + *\varphi$

We prove that $s + *\varphi$ is SN by giving a weight that decreases by reduction. We begin by defining two weight functions needed for the final weight:

Definition 3.15 Let $P : \Lambda s_{op} \to I\!N$ and $W : \Lambda s_{op} \to I\!N$ be defined by:

$P(X) = P(\mathbf{n}) = 2$	$W(X) = W(\mathbf{n}) = 1$
$P(a\,b) = P(a) + P(b)$	$W(a\,b) = W(a) + W(b) + 1$
$P(\lambda a) = P(a)$	$W(\lambda a) = W(a) + 1$
$P(a\,\sigma^j b) = j * P(a) * P(b)$	$W(a\,\sigma^j b) = 2 * W(a) * (W(b) + 1)$
$P(\varphi^i_k a) = (k+1) * (P(a) + 1)$	$W(\varphi^i_k a) = 2 * W(a)$

Lemma 3.16 *For a, $b \in \Lambda s_{op}$ the following hold:*

(i) *If $a \to_{s+*\varphi} b$ then $W(a) \geq W(b)$.*
(ii) *If $a \to_{s+*\varphi^-} b$ then $W(a) > W(b)$.*
(iii) *If $a \to_{*\varphi--} b$ then $P(a) > P(b)$.*

Proof. By induction on a: if the reduction is internal, the induction hypothesis applies; otherwise, the theorem must be checked for each rule. □

Theorem 3.17 *The $s + *\varphi$-calculus is SN.*

Proof. The previous lemma ensures that the ordinal $(W(a), P(a))$ decreases with the lexicographical order for each $s + *\varphi$-reduction. □

3.6 The $\lambda\omega$- and $\lambda\omega_e$-calculi

Recall that the $*\sigma$-calculus consists of the two painful rules σ-σ-tr. and φ-σ-tr. which are at the heart of our inability to use the rpo method or the methods

of Zantema. In order to establish SN of $*\sigma$, we will use an isomorphism established in [13] between λs_e and $\lambda\omega_e$, a calculus written in the $\lambda\sigma$-style.

In order to express λs-terms in the $\lambda\sigma$-style, [13] split the closure operator of $\lambda\sigma$ (denoted in a semi-infix notation as $-[-]$) in a family of closures operators that were denoted also with a semi-infix notation as $-[-]_i$, where i ranges on the set of natural numbers. [13] also admitted as basic operators the iterations of $\uparrow$ and therefore had a countable set of basic substitutions $\uparrow^n$, where n ranges on the set of natural numbers. By doing so, the updating operators of λs become available as $-[\uparrow^n]_i$. Finally, [13] introduced a *slash* operator of sort $\mathtt{term} \rightarrow \mathtt{substitution}$ which transforms a term a into a substitution $a/$. This operator may be considered as *consing with id* (in the $\lambda\sigma$-jargon) and was first introduced and exploited in the $\lambda\upsilon$-calculus (cf. [2]). Here is the formalisation of this syntax and the rewriting rules of $\lambda\omega$:

Definition 3.18 The set $\Lambda\omega$ of terms of the $\lambda\omega$-calculus, is defined as $\Lambda\omega^t \cup \Lambda\omega^s$, where $\Lambda\omega^t$ and $\Lambda\omega^s$ are mutually defined as follows ($j \geq 1$ and $i \geq 0$):

$$\textbf{Terms} \qquad \Lambda\omega^t ::= I\!N \mid \Lambda\omega^t\Lambda\omega^t \mid \lambda\Lambda\omega^t \mid \Lambda\omega^t[\Lambda\omega^s]_j$$

$$\textbf{Substitutions} \quad \Lambda\omega^s ::= \uparrow^i \mid \Lambda\omega^t/$$

The set, denoted $\lambda\omega$, of rules of the *$\lambda\omega$-calculus* is given as follows:

σ-generation	$(\lambda a)\, b$	$\longrightarrow$	$a\,[b/]_1$
σ-app-transition	$(a\, b)[s]_j$	$\longrightarrow$	$(a\,[s]_j)\,(b\,[s]_j)$
σ-λ-transition	$(\lambda a)[s]_j$	$\longrightarrow$	$\lambda(a[s]_{j+1})$
σ-/-destruction	$\mathtt{n}[a/]_j$	$\longrightarrow$	$\begin{cases} \mathtt{n-1} & \text{if } n > j \\ a[\uparrow^{j-1}]_1 & \text{if } n = j \\ \mathtt{n} & \text{if } n < j \end{cases}$
σ-$\uparrow$-destruction	$\mathtt{n}[\uparrow^i]_j$	$\longrightarrow$	$\begin{cases} \mathtt{n+i} & \text{if } n \geq j \\ \mathtt{n} & \text{if } n < j \end{cases}$

The set of rules of the *ω-calculus* is $\lambda\omega - \{\sigma - generation\}$. We use $a, b, c, \ldots$ to range over $\Lambda\omega^t$ and $s, t, \ldots$ to range over $\Lambda\omega^s$.

Definition 3.19 Let $\mathbf{V}$ stand for a set of variables, over which X, Y, ... range. The set $\Lambda\omega_{op}$ of *open terms*, is defined as $\Lambda\omega^t_{op} \cup \Lambda\omega^s_{op}$, where $\Lambda\omega^t_{op}$ and $\Lambda\omega^s_{op}$ are mutually defined as follows ($j \geq 1$ and $i \geq 0$):

$$\textbf{Open Terms} \quad \Lambda\omega^t_{op} ::= \mathbf{V} \mid I\!N \mid \Lambda\omega^t_{op}\Lambda\omega^t_{op} \mid \lambda\Lambda\omega^t_{op} \mid \Lambda\omega^t_{op}[\Lambda\omega^s_{op}]_j$$

$$\textbf{Substitutions} \quad \Lambda\omega^s_{op} ::= \uparrow^i \mid \Lambda\omega^t_{op}/$$

We take a, b, c to range over $\Lambda\omega^t_{op}$ and s, t, ... over $\Lambda\omega^s_{op}$. *Closures*, *pure terms* and *compatibility* are defined as expected. The set $\lambda\omega_e$ of rules of the

$\lambda\omega_e$-*calculus* is obtained by adding to $\lambda\omega$ the new following rules:

σ-/-*transition*	$a\,[b/]_k[s]_j$	$\longrightarrow$	$a\,[s]_{j+1}[b[s]_{j-k+1}/]_k$ if $k \leq j$
/-↑-*transition*	$a\,[\uparrow^i]_k[b/]_j$	$\longrightarrow$	$\begin{cases} a[b/]_{j-i}[\uparrow^i]_k & \text{if } k+i \leq j \\ a[\uparrow^{i-1}]_k & \text{if } k \leq j < k+i \end{cases}$
↑-↑-*transition*	$a\,[\uparrow^i]_k[\uparrow^l]_j$	$\longrightarrow$	$\begin{cases} a[\uparrow^l]_{j-i}[\uparrow^i]_k & \text{if } k+i < j \\ a[\uparrow^{i+l}]_k & \text{if } k \leq j \leq k+i \end{cases}$

The set of rules of the ω_e-*calculus* is $\lambda\omega_e - \{\sigma\text{-}generation\}$.

Remark 3.20 Note that the rule schemes /-↑ and ↑-↑ can be merged into the single scheme $a\,[\uparrow^i]_k[s]_j \to a[s]_{j-i}[\uparrow^i]_k$ for $k+i<j$ but they must be kept distinct when $k+i=j$ if SN is to hold. The ↑-↑-scheme, if admitted when $k+i=j$, may generate an infinite loop (e.g., if $i=k=l=1$ and $j=2$).

[13] established an isomorphism between λs_e and $\lambda\omega_e$ and also between λs and $\lambda\omega$. These isomorphisms translate properties of λs and λs_e to $\lambda\omega$ and $\lambda\omega_e$, respectively. Hence, all the results mentioned above concerning λs and λs_e translate into corresponding results for the sort `term` to $\lambda\omega$ and $\lambda\omega_e$.

Theorem 3.21 (cf. [13]) *The following hold:*

(i) *The ω-calculus is SN and confluent on $\Lambda\omega^t$.*

(ii) *Let $a, b \in \Lambda$. If $a \twoheadrightarrow_{\lambda\omega} b$ then $a \twoheadrightarrow_\beta b$. If $a \to_\beta b$ then $a \twoheadrightarrow_{\lambda\omega} b$.*

(iii) *The $\lambda\omega$-calculus is confluent on $\Lambda\omega^t$.*

(iv) *Pure terms which are SN in the λ-calculus are also SN in the $\lambda\omega$-calculus.*

(v) *The ω_e-calculus is weakly normalising and confluent.*

(vi) *The $\lambda\omega_e$-calculus is confluent on open terms.*

(vii) *Let $a, b \in \Lambda$. If $a \twoheadrightarrow_{\lambda\omega_e} b$ then $a \twoheadrightarrow_\beta b$. If $a \to_\beta b$ then $a \twoheadrightarrow_{\lambda\omega_e} b$.*

3.7 SN of $*\sigma$

To prove SN for $*\sigma$ we will use the isomorphism presented in Section 3.6 and the technique that Zantema used to prove SN for the calculus whose only rule is σ-σ-*transition* (cf. [11]). Following this isomorphism, the schemes σ-σ-*tr.* and φ-σ-*tr.* of λs_e both translate into the same scheme of $\lambda\omega_e$, namely σ-/-*transition* of Definition 3.19. Hence, to show that $*\sigma$ is SN, it is enough to show that the calculus whose only rule is σ-/-*transition*, let us call it σ-/-calculus, is SN. To do so, we use the following Lemma of Zantema (cf. [14]):

Lemma 3.22 *Any reduction relation $\to$ on a set T satisfying the three properties below is strongly normalising:*

(i) *$\to$ is weakly normalising.*

(ii) $\rightarrow$ *is locally confluent.*

(iii) $\rightarrow$ *is increasing, i.e.,* $\exists$ *function* $f : T \mapsto I\!N$ *where* $a \rightarrow b \Rightarrow f(a) < f(b)$.

For weak normalisation of the σ-/ $-$ *calculus* we use the technique of [11]:

Definition 3.23 We say that $c \in \Lambda\omega^t$ is an *external normal form* if $c = a[s_1]_{i_1} \cdots [s_n]_{i_n}$ where $a \neq e[d/]_k$ and if $s_k = b_k/$ then $i_k > i_{k+1}$. We denote the set of external normal forms ENF.

Lemma 3.24 *Let* $c = a[s_1]_{i_1} \cdots [s_n]_{i_n} \in ENF$ *and let* $i_n \leq i_{n+1}$ *and* $s_n = b_n/$ *then there exists a* σ-/*-derivation* $c \rightarrow^+ a[t_1]_{j_1} \cdots [t_{n+1}]_{j_{n+1}} \in ENF$ *such that* $j_{n+1} = i_n$ *and for every* r *with* $1 \leq r \leq n+1$ *we have either* $t_r = s_k$ *for some* $k \leq n+1$ *or* $t_r = (a_p[s_{n+1}])/$ *for some* $s_p = a_p/$ *with* $1 \leq p \leq n$.

Proof. By induction on n. □

Lemma 3.25 *Let* $c = a[s_1]_{i_1} \cdots [s_n]_{i_n}$ *such that* $a \neq e[d/]_k$. *There exists a* σ-/*-derivation* $c \twoheadrightarrow a[t_1]_{j_1} \cdots [t_n]_{j_n} \in ENF$ *such that for every* r *with* $1 \leq r \leq n+1$ *we have either* $t_r = s_k$ *for some* $k \leq n$ *or* $t_r = (a_{p_{r\,1}}[s_{p_{r\,2}}]_{k_2} \cdots [s_{p_{r\,n}}]_{k_n})/$ *with* $1 \leq p_{r\,1} \leq \cdots \leq p_{r\,n} \leq n$ *and with some* $s_p = a_{p_{r\,1}}/$ *(*$1 \leq p \leq n$*).*

Proof. By induction on n, using the previous lemma. □

Lemma 3.26 *The* σ-/*-calculus is weakly normalising.*

Proof. Assume a term c not having a normal form for which every term smaller (in size) than c admits a normal form. Let $c = a[s_1]_{i_1} \cdots [s_n]_{i_n}$ such that $a \neq e[d/]_k$. By Lemma 3.25, $c \twoheadrightarrow a[t_1]_{j_1} \cdots [t_n]_{j_n} \in ENF$. As a, t_1, $\cdots$ t_n are all smaller than c, they admit a normal form. Replacing each of them by its normal form in $a[t_1]_{j_1} \cdots [t_n]_{j_n}$ gives a normal form for c. Absurd. □

Theorem 3.27 *The* σ-/*-calculus is strongly normalising on* $\Lambda\omega^t$.

Proof. Use Lemma 3.22. (i) was shown in Lemma 3.26. (ii) follows from a critical pair analysis and (iii) is shown by choosing $f(a)$ to be the size of a. □

Since both rule schemes in $*\sigma$ translate into the single σ-/ rule scheme, the isomorphism gives:

Theorem 3.28 *The* $*\sigma$*-calculus is strongly normalising.*

3.8 Modularity fails

Now that $s + *\varphi$ and $*\sigma$ are SN the question arises whether the whole system can be shown SN using a modularity result. The answer is negative for the classical modularity theorem of Bachmair-Dershowitz, which we recall here.

Definition 3.29 A rewrite relation R commutes over S if whenever $a \rightarrow_S b \rightarrow_R c$, there is an alternative derivation $a \rightarrow_R d \rightarrow_{R \cup S} c$.

Theorem 3.30 (Bachmair-Dershowitz-85) *Let* R *commute over* S. *The combined system* $R \cup S$ *is SN iff* R *and* S *both are SN.*

Example 3.31 shows that no commutation exists between $s + *\varphi$ and $*\sigma$ and so the Bachmair-Dershowitz's Theorem cannot be used to get SN of s_e.

Example 3.31 To show that $*\sigma$ does not commute over $s+*\varphi$, let $k+i \leq j$, $h \leq j-i+1$ and $h > k+1$. Now take the following derivation:
$(\varphi_k^i(a\,\sigma^h b))\,\sigma^j c \to_{*\varphi} \varphi_k^i((a\,\sigma^h b)\,\sigma^{j-i+1}c) \to_{\sigma-\sigma-tr} \varphi_k^i((a\,\sigma^{j-i+2}c)\,\sigma^h(b\,\sigma^{j-i-h+2}c))$
It is easy to see that $(\varphi_k^i(a\,\sigma^h b))\,\sigma^j c$ does not contain any $*\sigma$-redex.

On the other hand, $s + *\varphi$ does not commute over $*\sigma$ either:
Let $i \leq j$ and let us consider the following derivation:
$((\lambda a)\,\sigma^i b)\,\sigma^j c) \to_{\sigma-\sigma-tr} ((\lambda a)\,\sigma^{j+1}c)\,\sigma^i(b\,\sigma^{j-i+1}c) \to_s (\lambda(a\,\sigma^{j+2}c))\,\sigma^i(b\,\sigma^{j-i+1}c)$
But reducing the only s-redex in $((\lambda a)\,\sigma^i b)\,\sigma^j c)$ we get $(\lambda(a\,\sigma^{i+1}b))\,\sigma^j c$ which also has a unique s-redex. Reducing it we get $\lambda((a\,\sigma^{i+1}b)\,\sigma^{j+1}c)$ and now there is only the *σ-σ-transition* redex which gives us $\lambda((a\,\sigma^{j+2}c)\sigma^{i+1}(b\,\sigma^{j-i+1}c))$ which has no further redexes. Therefore, $(\lambda(a\,\sigma^{j+2}c))\,\sigma^i(b\,\sigma^{j-i+1}c)$ cannot be reached from $((\lambda a)\,\sigma^i b)\,\sigma^i c)$ with an s_e-derivation beginning with an s-step.

4 Conclusion

This paper attempted two goals:

(i) It gave a calculus of explicit substitutions which allows local as well as global substitutions. We showed that this calculus simulates beta reduction and that the underlying calculus of substitutions is strongly normalising and confluent. A calculus of local explicit substitutions was given in [9], however that calculus did not enjoy good theoretical properties.

(ii) It explained the problems faced in showing that the s_e-calculus is strongly normalising. We are not sure whether the answer is positive or negative at this stage. We leave this problem as a challenge to the community.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375 416, 1991.

[2] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Functional Programming*, 6(5), 1996.

[3] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, 1986. Revised edition : Birkhäuser (1993).

[4] P-L Curien, T. Hardin, and A. Ríos. Strong normalisation of substitutions. *Logic and Computation*, 6:799 817, 1996.

[5] N. G. de Bruijn. Lambda-Calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381 392, 1972.

[6] B. Guillaume. *Un calcul des substitutions avec etiquettes.* PhD thesis, Université de Savoie, Chambéry, France, 1999.

[7] T. Hardin and A. Laville. Proof of Termination of the Rewriting System SUBST on CCL. *Theoretical Computer Science*, 46:305–312, 1986.

[8] F. Kamareddine and R. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.

[9] F. Kamareddine and R. P. Nederpelt. On stepwise explicit substitution. *International Journal of Foundations of Computer Science*, 4(3):197–240, 1993.

[10] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. Proceedings of PLILP'95. *LNCS*, 982:45–62, 1995.

[11] F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.

[12] F. Kamareddine and A. Ríos. Bridging de Bruijn indices and variable names in explicit substitutions calculi. *Logic Journal of the IGPL*, 6(6):843–874, 1998.

[13] F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-styles of explicit substitutions. *Logic and Computation*, 10(3):349–380, 2000.

[14] J.-W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, II, 1992.

[15] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[16] A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites.* PhD thesis, Université de Paris 7, 1993.

[17] H. Zantema. Termination of term rewriting: interpretation and type elimination. *J. Symbolic Computation*, 17(1):23–50, 1994.

[18] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

Remarks on Isomorphisms of Simple Inductive Types

David Chemouil, Sergei Soloviev [1,2]

IRIT, Université Paul Sabatier
118, route de Narbonne
31062 Toulouse, France

Abstract

We study isomorphisms of types in the system of simply-typed λ-calculus with inductive types and recursion operators. It is shown that in some cases (multiproducts, copies of types), it is possible to add new reductions in such a way that strong normalisation and confluence of the calculus are preserved, and the isomorphisms may be regarded as intensional w.r.t. a stronger equality relation.

1 Introduction

1.1 Presentation

This work is part of a larger project where we are exploring the possibilities of extensions preserving strong normalisation and confluence of standard reduction systems by new reductions of the form $f'\ (f\ t) \longrightarrow t$ where f' is in some sense an inverse of f.

The way this notion of invertibility may be understood is one of the questions we are investigating. A possibility would be to take the invertibility w.r.t extensional equality of functions between inductive types.

Here, we shall consider the simply-typed λ-calculus, equipped with inductive types (*i.e* recursive types satisfying a condition of strict positivity) and structural recursion schemes on these types.

In this short paper, we will focus on two particular cases where the usefulness of this extension seems obvious. Namely, we shall study some isomorphisms of products (defined as inductive types) and the notion of *copy* of a type

[1] E-mails: chemouil@irit.fr, soloviev@irit.fr

[2] Work partly funded by Project ISOT (STIC 21) from Department STIC of CNRS, and Liapunov Institute.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1.2 Isomorphisms of Types

Let us first recall a few facts and definitions about isomorphisms of types.

Definition 1.1 Consider a typed λ-calculus, equipped with an equivalence relation $\sim$ on terms, a term $\mathrm{id}_A : A \to A$ for any type A and a composition operator $\circ$ (with suitable typing) verifying the following conditions, for any function $f : A \to B$:

$$f \circ \mathrm{id}_A \sim f \qquad\qquad \mathrm{id}_B \circ f \sim f$$

Then, two types A and B are said to be *isomorphic* (written $A \cong B$) if there exist two λ-terms $f : A \to B$ and $g : B \to A$ such that

$$f \circ g \sim \mathrm{id}_B \qquad\qquad g \circ f \sim \mathrm{id}_A$$

In this case, g is often written f^{-1} and called the *inverse* of f.

Until now, isomorphisms of types have mostly been studied in various first- or second-order λ-calculi, where $\sim$ is usually generated by $\beta\eta$-conversion [3], $\mathrm{id}_A \mathrel{\widehat{=}} \lambda x : A \cdot x$ and $\circ \mathrel{\widehat{=}} \lambda g : B \to C \cdot \lambda f : A \to B \cdot \lambda x : A \cdot g\ (f\ x)$ (for any types A, B, and C). As an example, we have the following result:

Proposition 1.2 ([20]; [9,11]) *All isomorphisms holding in $\lambda^1\beta\eta_{\to,\times,1}$, the first-order simply-typed λ-calculus with binary products and unit type (or, equivalently, in cartesian closed categories), are obtainable by finite compositions of the following "base" of seven isomorphisms:*

$$A \times B \cong B \times A \qquad A \times (B \times C) \cong (A \times B) \times C$$

$$(A \times B) \to C \cong A \to (B \to C) \qquad A \to (B \times C) \cong (A \to B) \times (A \to C)$$

$$A \times 1 \cong A \qquad A \to 1 \cong 1 \qquad 1 \to A \cong A$$

1.3 Isomorphisms of Inductive Types

Now, it is our view that, as long as inductive types are concerned, *intensional isomorphisms*, in ordinary sense, lack expressivity. To view this problem in a larger context, one needs a notion of *extensionality*.

Definition 1.3 Two types A and B are *extensionally isomorphic* (written $A \cong\!\!\!\!\!\approx\ B$) if there exists two λ-terms $f : A \to B$ and $g : B \to A$ such that

$$\forall x : A \cdot g\ (f\ x) \sim x \quad \text{and} \quad \forall y : B \cdot f\ (g\ y) \sim y\ .$$

(Note that $\cong$ and $\cong\!\!\!\!\!\approx$ are both equivalence relations.)

[3] It was shown in [10] that with β-conversion solely, the only invertible term is the identity.

Obviously, we have $A \cong B \Rightarrow A \approx B$, but the converse is usually not true. One way to achieve this kind of isomorphisms would be to add extensional reduction rules to the calculi, such as η rules, surjective pairing, etc. However, many calculi don't come equipped with extensional reduction rules, for various reasons (decidability, confluence, etc); though some positive results do exist, e.g [16,13,15]. Hence, in this paper, we will mainly be interested with $\beta\iota$-reduction only (where ι-reduction is the rule associated to structural recursion over inductive types).

Of course, extensional isomorphisms are *provable* by induction, but they are not *computable*, *i.e*, one doesn't have (for example)

$$\lambda x : A \cdot f^{-1}\ (f\ x) \longrightarrow_{\beta\iota} \lambda x : A \cdot x.$$

Without appealing to full extensionality, we think that, if f and f^{-1} are mutually invertible extensional isomorphisms, it is worth considering the addition of new reduction rules (call them σ-reductions, following [6]) as follows:

$$f\ (f^{-1}\ x) \longrightarrow_{\sigma} x \quad \text{and} \quad f^{-1}\ (f\ x) \longrightarrow_{\sigma} x.$$

1.4 Outline of the paper

In Sect. 2, we quickly give essential definitions of a simply-typed λ-calculus with inductive types.

Then, in Sect. 3, we quickly present a small lemma ("Deferment Lemma") that is of interest in the next section.

In Sect. 4, we illustrate the addition of rewrite rules on n-ary products. We show that, for products, strong normalisation and confluence are preserved for a rewrite rule corresponding to commutativity, while it is not the case for associativity, unless we also add surjective pairing.

Finally, in Sect. 5, we study the notion of *isomorphic copy* of a type, and how a rewrite rule corresponding to it may or not be added to the calculus.

2 Simply-Typed λ-Calculus with Inductive Types

We define the simply-typed λ-calculus with inductive types, which may be seen as an extension of Gödel's system $\mathcal{T}$. Some references on λ-calculus and inductive types may be found in [4,19,5,22,18,8]. Furthermore, most of our notations and results concerning rewrite systems are taken from [1]. For a given reduction $\longrightarrow_R$, we write $\longrightarrow_R^+$ for its transitive closure, and $\longrightarrow_R^*$ for its reflexive-transitive closure.

2.1 Types

Throughout this paper, we consider an infinite set $\mathcal{S} = \{\alpha, \beta, \ldots\}$ of *type variables*. We also consider an infinite set of variables $\mathcal{V}$ (with $\mathcal{V} \cap \mathcal{S} = \varnothing$),

and an infinite set $\mathcal{C}$ of *inductive-type constructors* (or *introduction operators*), with $\mathcal{C} \cap \mathcal{S} = \mathcal{C} \cap \mathcal{V} = \varnothing$.

Moreover, as usual in this sort of presentation, we consider all terms and types up to α-conversion, *i.e* the names of bound variables are irrelevant.

Note 1 *In the following, the sign* $\equiv$ *will denote syntactic equality, and definitions will be introduced in the calculus with the sign* $\widehat{=}$. *Furthermore, we will use the common notation* let $x = e_1$ in e_2 *for* $e_2[e_1/x]$.

Definition 2.1 The set of *pre-types* is generated by the following grammar rules:

$$\begin{array}{lll} \mathrm{Ty} ::= & \alpha \mid (\mathrm{Ty} \to \mathrm{Ty}) \mid \mathrm{Ind}(\alpha)[\mathrm{CS}] \\ \mathrm{CS} ::= & \mathrm{CL} \mid \varepsilon \\ \mathrm{CL} ::= & c : \mathrm{Ty} \mid c : \mathrm{Ty}\,;\, \mathrm{CL} \end{array}$$

with $c \in \mathcal{C}$ (as usual, ε denotes the empty word). Of course, we require that any constructor belong to only one inductive type.

Note 2 *We consider that* $\to$ *is right associative, hence* $\tau_1 \to (\tau_2 \to \tau_3)$ *will be subsequently written* $\tau_1 \to \tau_2 \to \tau_3$.

An inductive type with n constructors $c_1, \ldots, c_n$ in $\mathcal{C}$, each of arity k_i (with $1 \leqslant i \leqslant n$), is then of the form

$$\mathrm{Ind}(\alpha)[\,c_1 : \sigma_1^1 \to \ldots \to \sigma_1^{k_1} \to \alpha\,;\, \ldots\,;\, c_n : \sigma_n^1 \to \ldots \to \sigma_n^{k_n} \to \alpha\,],$$

where the part between brackets is bound by $\mathrm{Ind}(\alpha)$. Moreover, every $\sigma_i \equiv \sigma_i^1 \to \ldots \to \sigma_i^{k_i} \to \alpha$ must verify certain conditions, as explained below.

Definition 2.2 A *strictly positive operator* τ over a type variable α (written τ spos α) is inductively defined by the following rules:

$$\frac{}{\alpha \text{ spos } \alpha} \qquad \frac{\alpha \notin \mathrm{FV}(\tau_1) \qquad \tau_2 \text{ spos } \alpha}{\tau_1 \to \tau_2 \text{ spos } \alpha}$$

Definition 2.3 An *(inductive) schema* τ over a type variable α (written τ sch α) is inductively defined by the following rules:

$$\frac{}{\alpha \text{ sch } \alpha} \qquad \frac{\alpha \notin \mathrm{FV}(\tau_1) \qquad \tau_2 \text{ sch } \alpha}{\tau_1 \to \tau_2 \text{ sch } \alpha} \qquad \frac{\tau_1 \text{ spos } \alpha \qquad \tau_2 \text{ sch } \alpha}{\tau_1 \to \tau_2 \text{ sch } \alpha}$$

Intuitively, a schema σ is of the form $\sigma^1 \to \ldots \to \sigma^k \to \alpha$, where every σ^j is itself:

- either a type not containing α (we call this σ^j a *non-recursive operator*);
- or a type of the form $\sigma^j \equiv \nu_1 \to \ldots \to \nu_m \to \alpha$ (we call this σ^j a *strictly positive operator*), where α does not appear in any ν_ℓ.

Note 3 *Given a schema* $\sigma \equiv \sigma^1 \to \ldots \to \sigma^k \to \alpha$, *we will denote by* $\mathrm{SP}_\alpha(\sigma)$ *the set of indices* j *(with* $1 \leqslant j \leqslant k$*) such that* σ^j *is a strictly positive operator over* α, i.e $\mathrm{SP}_\alpha(\sigma) = \{j \mid 1 \leqslant j \leqslant k \wedge \sigma^j \text{ spos } \alpha\}$. *This set will be useful because it corresponds to arguments (of a given constructor) on which a recursive call may be carried out.*

Definition 2.4 A *type* τ (written $\tau : \star$) is inductively defined by the following rules:

$$\frac{\alpha \in \mathcal{S}}{\alpha : \star} \qquad \frac{\tau_1 : \star \qquad \tau_2 : \star}{\tau_1 \to \tau_2 : \star} \qquad \frac{c_i \in \mathcal{C} \qquad \sigma_i : \star \qquad \sigma_i \text{ sch } \alpha \qquad (1 \leqslant i \leqslant n)}{\mathrm{Ind}(\alpha)[c_1 : \sigma_1 ; \ldots ; c_n : \sigma_n] : \star}$$

Example 2.5 With these rules, it is possible to define the types of natural numbers, of Brouwer's ordinals and of lists of natural numbers (normally, these inductive types should have different constructor names, we used some common names for the sake of readibility):

$$\begin{aligned} \mathrm{Nat} &\mathrel{\widehat{=}} \mathrm{Ind}(\alpha)[0 : \alpha \,|\, \mathrm{S} : \alpha \to \alpha] \\ \mathrm{Ord} &\mathrel{\widehat{=}} \mathrm{Ind}(\alpha)[0 : \alpha \,|\, \mathrm{S} : \alpha \to \alpha \,|\, \mathrm{L} : (\mathrm{Nat} \to \alpha) \to \alpha] \\ \mathrm{ListNat} &\mathrel{\widehat{=}} \mathrm{Ind}(\alpha)[\mathrm{nil} : \alpha \,|\, \mathrm{cons} : \mathrm{Nat} \to \alpha \to \alpha]. \end{aligned}$$

Note that any inductive type τ generates a *recursor* (or *structural-recursion operator*) $\mathcal{R}_{\tau,\kappa}$ to any type κ. This will be further explained in the next section concerned with terms of the language.

2.2 Terms

We will now define the terms of our calculus.

Definition 2.6 The set of *terms* is generated by the following grammar rule:

$$M \;::=\; c \;\mid\; x \;\mid\; (\lambda x : \tau \cdot M) \;\mid\; (M\ M) \;\mid\; \mathcal{R}_{\tau,\kappa}\,,$$

where $x \in \mathcal{V}$, $c \in \mathcal{C}$ and τ and κ are types.

Note 4 *Application is left-associative, hence* $(\ldots(M_1\ M_2)\ \ldots)\ M_n)$ *can be written* $M_1 \ldots M_n$. *In the same way, abstraction is right-associative, hence* $(\lambda x_1 : \tau_1 \cdot (\lambda x_2 : \tau_2 \cdot M))$ *can be written* $\lambda x_1 : \tau_1 \cdot \lambda x_2 : \tau_2 \cdot M$

We now define a syntactic operation that will be useful to assert typing rules for terms.

Definition 2.7 Let τ be an inductive type, $\sigma \equiv \sigma^1 \to \ldots \to \sigma^k \to \alpha$ a schema over α in τ, and κ a type. Let $\{j_p\}_{p=1,\ell} = \mathrm{SP}_\alpha(\sigma)$. Then, we define

$$\Upsilon_\tau(\sigma, \kappa) \equiv \sigma^1[\tau/\alpha] \to \ldots \to \sigma^k[\tau/\alpha] \to \sigma^{j_1}[\kappa/\alpha] \to \ldots \sigma^{j_\ell}[\kappa/\alpha] \to \kappa.$$

Definition 2.8 We now present the typing rules for the calculus:

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha}\ (\text{AX}) \qquad \frac{\tau \equiv \text{Ind}(\alpha)[\ldots; c : \sigma; \ldots] \qquad \tau : \star}{\Gamma \vdash c : \sigma[\tau/\alpha]}\ (\text{CONSTR})$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{(\lambda x : \tau_1 \cdot M) : \tau_1 \to \tau_2}\ (\lambda) \qquad \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash (M\ N) : \tau_2}\ (\text{APP})$$

$$\frac{\begin{array}{c}\tau \equiv \text{Ind}(\alpha)[c_1 : \sigma_1; \ldots; c_n : \sigma_n] \\ \Gamma \vdash M_i : \Upsilon_\tau(\sigma_i, \kappa) \qquad (1 \leqslant i \leqslant n)\end{array}}{\Gamma \vdash (\mathcal{R}_{\tau,\kappa}\ M_1\ \ldots\ M_n) : \tau \to \kappa}\ (\text{ELIM})$$

2.3 Reduction

Definition 2.9 We define the usual *β-reduction* rule as follows:

$$(\lambda x : \tau \cdot M)\ N\ \longrightarrow_\beta\ M[N/x]\ .$$

Now, we define the ι-reduction. However, to do so, we first need to make a technical definition which will be helpful.

Definition 2.10 Let $\nu \equiv \nu_1 \to \ldots \to \nu_m \to \alpha$ be a strictly positive operator over α. Then, we define

$$\Xi(R, N, \nu) \equiv \lambda z_1 : \nu_1 \cdot \ldots \cdot \lambda z_m : \nu_m \cdot R\ (N\ z_1\ \ldots\ z_m)\ .$$

Of course, in the special case where $m = 0$, we have $\Xi(R, N, \nu) \equiv R\ N$.

Definition 2.11 Now, let $\sigma \equiv \sigma^1 \to \ldots \to \sigma^k \to \alpha$ be a schema over α, and let $\{j_p\}_{p=1,\ell} = \text{SP}_\alpha(\sigma)$. Then, we define *ι-reduction* by

$$\mathcal{R}_{\tau,\kappa}\ M_1\ \ldots\ M_n\ (c_i\ N_1\ \ldots\ N_{k_i}) \longrightarrow_\iota M_i\ N_1, \ldots\ N_{k_i}\ N'_{j_1}\ \ldots\ N'_{j_\ell}\,,$$

where $N'_{j_p} \equiv \Xi(\mathcal{R}_{\tau,\kappa}\ M_1\ \ldots\ M_n, N_{j_p}, \sigma_{j_p})$, for all $1 \leqslant p \leqslant \ell$.

Examples of rules for some basic inductive types are given in Figure 1 on the following page.

Proposition 2.12 *For the simply-typed λ-calculus with inductive types, βι-reduction is strongly normalising and confluent.*

See for example [8].

3 A Deferment Lemma

There are many lemmas concerning with strong normalisability of a relation $\longrightarrow_{RS}$ when $\longrightarrow_R$ and $\longrightarrow_S$ are strongly normalising. Though the lemma we

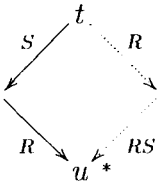
$$
\begin{array}{lll}
\mathcal{R}_{\mathrm{Nat},\kappa}\ a\ f\ 0 & \longrightarrow_\iota & a \\
\mathcal{R}_{\mathrm{Nat},\kappa}\ a\ f\ (\mathrm{S}\ p) & \longrightarrow_\iota & f\ p\ (\mathcal{R}_{\mathrm{Nat},\kappa}\ a\ f\ p) \\
 & & \\
\mathcal{R}_{\mathrm{Ord},\kappa}\ a\ f\ g\ 0 & \longrightarrow_\iota & a \\
\mathcal{R}_{\mathrm{Ord},\kappa}\ a\ f\ g\ (\mathrm{S}\ p) & \longrightarrow_\iota & f\ p\ (\mathcal{R}_{\mathrm{Ord},\kappa}\ a\ f\ g\ p) \\
\mathcal{R}_{\mathrm{Ord},\kappa}\ a\ f\ g\ (\mathrm{L}\ k) & \longrightarrow_\iota & g\ k\ (\lambda z : \mathrm{Nat} \cdot (\mathcal{R}_{\mathrm{Ord},\kappa}\ a\ f\ g\ (k\ z))) \\
 & & \\
\mathcal{R}_{\mathrm{ListNat},\kappa}\ a\ f\ \mathrm{nil} & \longrightarrow_\iota & a \\
\mathcal{R}_{\mathrm{ListNat},\kappa}\ a\ f\ (\mathrm{cons}\ h\ t) & \longrightarrow_\iota & f\ h\ t\ (\mathcal{R}_{\mathrm{ListNat},\kappa}\ a\ f\ t)
\end{array}
$$

Fig. 1. Recursion rules for some basic inductive types

consider below is close to many results in the folklore, we could not find its exact formulation in the literature.

Note also that this lemma is not equivalent to the so-called Postponement Lemma for η-contractions in pure λ-calculus, see e.g [3] p. 386.

Definition 3.1 Let $\longrightarrow_R$ and $\longrightarrow_S$ be two reductions. Then, $\longrightarrow_S$ is *deferable w.r.t* $\longrightarrow_R$ if, for all terms t and u such that $t \longrightarrow_S \longrightarrow_R u$, there is a derivation $t \longrightarrow_R \longrightarrow^*_{RS} u$.



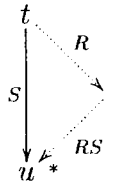
Lemma 3.2 (Deferment Lemma) *Let $\longrightarrow_R$ and $\longrightarrow_S$ be two strongly normalising relations. Then, if $\longrightarrow_S$ is deferable w.r.t $\longrightarrow_R$, $\longrightarrow_{RS}$ is strongly normalising.*

Proof. Let $\longrightarrow_R$ and $\longrightarrow_S$ be two strongly normalising relations, such that $\longrightarrow_S$ is deferable w.r.t $\longrightarrow_R$. Let us suppose that $\longrightarrow_{RS}$ is not strongly normalising, and show that it leads to a contradiction.

If $\longrightarrow_{RS}$ is not strongly normalising, then $\longrightarrow^*_{RS}$ consists of an infinite alternation of $\longrightarrow^*_R$ and $\longrightarrow^*_S$. Then, one can inductively "lift" $\longrightarrow_R$-reductions by deferring every $\longrightarrow_S$-reduction followed by an $\longrightarrow_R$-reduction, thus building an infinite derivation of $\longrightarrow_R$ steps. This contradicts the fact that $\longrightarrow_R$ is strongly normalising. □

In fact, we can prove a slightly more powerful lemma whose premises occur however less in practice.

Definition 3.3 Let $\longrightarrow_R$ and $\longrightarrow_S$ be two reductions. Then, $\longrightarrow_S$ is *0-deferable w.r.t* $\longrightarrow_R$ if, for all terms t and u such that $t \longrightarrow_S u$, there is a derivation $t \longrightarrow_R \longrightarrow^*_{RS} u$.



Lemma 3.4 (0-Deferment Lemma) *Let* $\longrightarrow_R$ *and* $\longrightarrow_S$ *be two strongly normalising relations. Then, if* $\longrightarrow_S$ *is 0-deferable w.r.t* $\longrightarrow_R$, $\longrightarrow_{RS}$ *is strongly normalising.*

Proof. Immediate, because 0-deferment implies deferment. □

Remark 3.5 Since the submission of this paper, we found some references about what we call Deferment Lemma (cf. [2,14] and most notably [12]). While we shall keep calling this property "deferment" in the current paper, we intend to use the preferable term "adjournement" afterwards, following Delia Kesner (private communication).

4 Multiproducts

Let us define a schema of inductive types representing n-ary products:

$$\Pi_n\ A_1 \ldots A_n \mathrel{\widehat{=}} \mathrm{Ind}(\alpha)[\,\langle\cdot\rangle_n : A_1 \to \ldots \to An \to \alpha\,]\ ,$$

with recursion operator $(\!|\cdot|\!)_n$ defined by

$$(\!|\cdot|\!)_n : (A_1 \to \ldots \to A_n \to B) \to (\Pi_n\ A_1 \ldots A_n \to B)$$
$$(\!|f|\!)_n\ \langle a_1 \ldots a_n\rangle_n \longrightarrow_\iota f\ a_1 \ldots a_n\ .$$

The projections p^n_k are defined as $(\!|\lambda x_1 : A_1 \cdot \ldots \cdot \lambda x_n : A_n \cdot x_k|\!)_n$.

Remark 4.1 One may note that the product of morphisms $f_i : C \to A_i$ (with $1 \leqslant i \leqslant n$) is definable, without the elimination operator, by

$$\mathrm{prod}_n\ f_1 \ldots f_n \mathrel{\widehat{=}} \lambda z : C \cdot \langle f_1\ z, \ldots, f_n\ z\rangle_n\ .$$

However, many familiar properties of product and projections do not hold intensionally. For example, we have $\langle p^2_1\ x, p^2_2\ x\rangle_2 \neq_{\beta\iota} x$ for $x : \Pi_2\ A\ B$. In fact, this property, usually known as *surjective pairing*, stipulates that the product is unique.

4.1 Commutativity of Products

Now, let ϱ be a permutation of $\{1, \ldots, n\}$. The permutation of $\Pi_n\ A_1 \ldots A_n$ induced by ϱ is denoted $\overline{\varrho}$, and defined as $(\!|\lambda x_1 : A_1 \cdot \ldots \cdot \lambda x_n : A_n \cdot \langle x_{\varrho(1)}, \ldots, x_{\varrho(n)}\rangle_n|\!)_n$.

Proposition 4.2 *For any term* $t : \Pi_n\ A_1 \dots A_n$ *and permutations* ϱ *and* ω *defined on* $\{1, \dots, n\}$, *the equality* $\overline{\varrho \circ \omega}\ t =_{\beta\iota} \overline{\varrho}\ (\overline{\omega}\ t)$ *is provable.*

Still, while we can prove this proposition by induction, it is important to note that the equality is not computable for an arbitrary t, but just when $t \equiv \langle t_1, \dots, t_n \rangle_n$ for some n (cf. Sect. 1.3 on page 3). Note also that for mutually inverse permutations ϱ and ϱ^{-1}, $\overline{\varrho}$ and $\overline{\varrho^{-1}}$ are mutually inverse extensional isomorphisms.

Now, for given mutually inverse permutations ϱ and ϱ^{-1}, let us add the following rewrite rules to the system of $\beta\iota$-reductions:

$$\overline{\varrho}\ (\overline{\varrho^{-1}}\ x) \longrightarrow_\sigma x \qquad\qquad \overline{\varrho^{-1}}\ (\overline{\varrho}\ x) \longrightarrow_\sigma x\ .$$

(Note that $\overline{\varrho}$ and $\overline{\varrho^{-1}}$ are *concrete*, *i.e* constant, terms of the calculus.)

Remark 4.3 To lighten the notation, let us write π and π' for $\overline{\varrho}$ and $\overline{\varrho^{-1}}$. We will also make use of diagrams, as is usually done for this kind of proof.

Lemma 4.4 *σ-reduction is strongly normalising.*

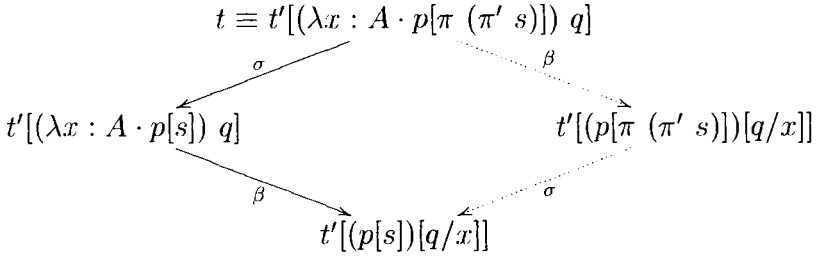
Proof. Take the length of terms as an ordering. □

Theorem 4.5 *$\beta\iota\sigma$-reduction is strongly normalising.*

Proof. We show that σ-reduction is deferable w.r.t β-reduction (case i) and w.r.t ι-reduction (case ii).

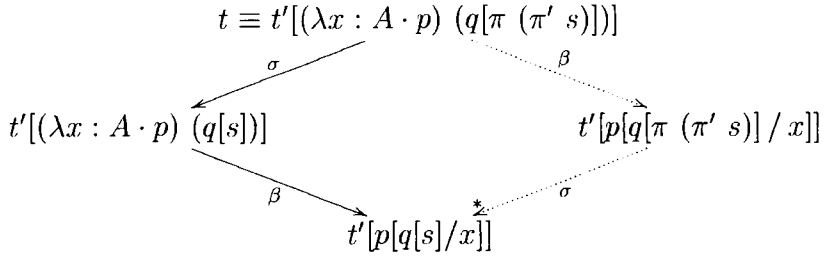
(i) For β-reduction. The crucial case is when the σ-redex occurs inside a β-redex.

i.1. As a first possibility, we may have $t \equiv t'[(\lambda x : A \cdot p[\pi\ (\pi'\ s)])\ q]$. Note that π and π' do not contain variables.



i.2. We may also have $t \equiv t'[(\lambda x : A \cdot p)\ (q[\pi\ (\pi'\ s)])]$, in which case the term p may contain many (or zero) occurrences of x, which requires

to carry as many σ-reductions.



(ii) For ι-reduction.

ii.1. The crucial case occurs when a ι-redex may interact with π and π', hence we must have $t \equiv t'[\pi\ (\pi'\ \langle s_1, \ldots, s_n \rangle_n)]$. But then, it is immediate to see that $t \longrightarrow_\sigma t'[\langle s_1, \ldots, s_n \rangle_n]$ can also be performed by the derivation: $t'[\pi\ (\pi'\ \langle s_1, \ldots, s_n \rangle_n)] \longrightarrow_\iota \longrightarrow^+_{\beta\iota} t'[\langle s_1, \ldots, s_n \rangle_n]$. This is a trivial case of 0-deferment.

ii.2. In other cases, the ι-redex doesn't interfere with σ-reduction, therefore deferment is obviously possible.

□

Theorem 4.6 *$\beta\iota\sigma$-reduction is confluent.*

Proof. First, as $\beta\iota\sigma$-reduction is strongly normalising, it is enough to show local confluence (by Newman's Lemma), *i.e* for all terms t, w, w' such that $t \longrightarrow_{\beta\iota\sigma} w$ and $t \longrightarrow_{\beta\iota\sigma} w'$, there exists a term u such that $w \longrightarrow^*_{\beta\iota\sigma} u$ and $w' \longrightarrow^*_{\beta\iota\sigma} u$.

By Lemma 2.12, $\beta\iota$-reduction is confluent. For σ-reductions alone, by Newman's Lemma it is enough to show local confluence. The critical pairs induced by σ-reduction are joinable; hence by the Critical Pair Theorem, σ-reduction is locally confluent. Therefore, for $\beta\iota\sigma$-reductions there are only two extra cases to be considered depending on whether one carries a β- or ι-reduction (combined with σ-) as a first step.

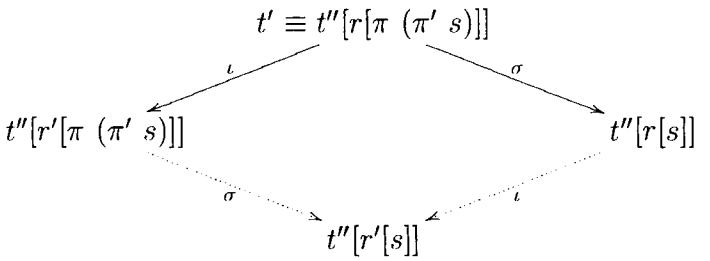
(i) If it is a ι-reduction, then $t \equiv t'[\pi\ (\pi'\ s)]$, and there are 4 possible cases: the ι-redex is in s, the ι-redex has no intersection with $\pi\ (\pi'\ s)$, the ι-redex contains $\pi\ (\pi'\ s)$, or the ι-redex is in $\pi\ (\pi'\ s)$ and intersects with π or π'.

i.1. We have $t \equiv t'[\pi\ (\pi'\ (s'[r]))]$, r being a ι-redex. Then, if t ι-reduces to $t'[\pi\ (\pi'\ s'[r'])]$ and σ-reduces to $t'[s'[r]]$, it is possible to "close" the fork by $t'[\pi\ (\pi'\ s'[r'])] \longrightarrow_\sigma t'[s'[r']]$ and $t'[\pi\ (\pi'\ s'[r'])] \longrightarrow_\iota t'[s'[r']]$.

i.2. Once more, the order is indifferent.

i.3. One has $t' \equiv t''[r[\pi\ (\pi'\ s)]]$. The upper-left ι-reduction cannot affect $\pi\ (\pi'\ s)$ since this part doesn't begin with an introduction operator. (In general, the lower-left reduction would possibly be $\longrightarrow^*_\sigma$ since the number of σ-redexes may change when ι-reduction is applied, but it

is not the case for products.)



i.4. In fact, the ι-redex should coincide with $(\pi'\ s)$, since $(\pi'\ s)$ doesn't begin with an introduction operator, so it cannot be $\pi\ (\pi'\ s)$ (here, we use the concrete definition of π and π'). Thus, s must be of the form $\langle s_1, \ldots, s_n \rangle_n$. But, for all elements of this form, we have $\pi\ (\pi'\ \langle s_1, \ldots, s_n \rangle_n) \longrightarrow_\iota \longrightarrow^+_{\beta\iota} \langle s_1, \ldots, s_n \rangle_n$, hence local confluence holds trivially in this case.

(ii) For β-reduction, cases ii.1 and ii.2 are similar to cases i.1 and i.2, thus treated as above.

ii.3 If $t \equiv t'[(\lambda x : A \cdot p[\pi\ (\pi'\ s)])\ q]$, and $t \longrightarrow_\beta t'[(p[\pi\ (\pi'\ s)])[q/x]]$ and $t \longrightarrow_\sigma t'[(\lambda x : A \cdot p[s])\ q]$, closing the "fork" is straightforward by observing that both terms σ- and β-reduce respectively in one step to $t'[(p[s])[q/x]]$. (Note that this situation appears because π and π' are closed terms.)

ii.4 In the last case, where $t \equiv t'[(\lambda x : A \cdot p)\ (q[\pi\ (\pi'\ s)])]$, the number of occurrences of x in p may influence the number of σ-reductions to perform to close the diagram. Thus, if $t \longrightarrow_\beta t'[p[q[\pi\ (\pi'\ s)]\,/x]]$ and $t \longrightarrow_\sigma t'[(\lambda x : A \cdot p)\ (q[s])]$, we may need a sequence of reductions $t'[p[q[\pi\ (\pi'\ s)]\,/x]] \longrightarrow^*_\sigma t'[p[q[s]/x]]$ while a one-step β-reduction only would be necessary on the other term: $t'[(\lambda x : A \cdot p)\ (q[s])] \longrightarrow_\beta t'[p[q[s]/x]]$.

□

4.2 Associativity of Products

As just seen, products enjoy the commutativity property. However, the associativity does not hold in general, *i.e*, it is not the case that, for example, $\Pi_2\ (\Pi_2\ A\ B)\ C \cong \Pi_2\ A\ (\Pi_2\ B\ C)$. This is so because there is an occurence of $\Pi_2\ A\ B$ (or $\Pi_2\ B\ C$) *inside* another Π_2. Thus, the "isomorphisms" g and g' would be defined in the following way:

$$\begin{aligned} g &: \Pi_2\ (\Pi_2\ A\ B)\ C \to \Pi_2\ A\ (\Pi_2\ B\ C) \\ &\widehat{=} (\!|\lambda p : \Pi_2\ A\ B \cdot \lambda c : C \cdot \langle p_1^2\ p, \langle p_2^2\ p, c \rangle_2 \rangle_2|\!) \end{aligned}$$

and

$$g' : \Pi_2\ A\ (\Pi_2\ B\ C) \to \Pi_2\ (\Pi_2\ A\ B)\ C$$
$$\widehat{=} (\!|\lambda a : A \cdot \lambda q : \Pi_2\ B\ C \cdot \langle \langle a, p_1^2\ q \rangle_2, p_2^2\ q \rangle_2 |\!) \ .$$

Then, for a term $\langle p, c \rangle_2$, with $p : \Pi_2\ A\ B$ and $c : C$, one has:

$$g'\ (g\ \langle p, c \rangle_2) \longrightarrow_\iota \longrightarrow_\beta g'\ \langle p_1^2\ p, \langle p_2^2\ p, c \rangle_2 \rangle_2$$
$$\longrightarrow_\iota \longrightarrow_\beta \langle \langle p_1^2\ p, p_2^2\ p \rangle_2, c \rangle_2 \neq_{\beta\iota} \langle p, c \rangle_2$$

because of the lack of surjective pairing. It is interesting to note that, even with extensionality on canonical elements, the isomorphism establishing associativity of binary product does not hold.

4.3 Retractions

Now, let us consider some correspondances between n-products for different n, for example $\Pi_3\ A\ B\ C$ and $\Pi_2\ (\Pi_2\ A\ B)\ C$. Define

$$f : \Pi_2\ (\Pi_2\ A\ B)\ C \to \Pi_3\ A\ B\ C$$
$$\widehat{=} (\!|\lambda y : \Pi_2\ A\ B \cdot \lambda z : C \cdot \langle p_1^2\ y, p_2^2\ y, z \rangle_3 |\!)_2$$

and

$$f' : \Pi_3\ A\ B\ C \to \Pi_2\ (\Pi_2\ A\ B)\ C$$
$$\widehat{=} (\!|\lambda x : A \cdot \lambda y : B \cdot \lambda z : C \cdot \langle \langle x, y \rangle_2, z \rangle_2 |\!)_3 \ .$$

For $\langle t, u, v \rangle_3 : \Pi_3\ A\ B\ C$, we have:

$$f\ (f'\ \langle t, u, v \rangle_3) \longrightarrow_\iota \longrightarrow_\beta f\ \langle \langle t, u \rangle_2, v \rangle_2 \longrightarrow_\iota \longrightarrow_\beta \langle t, u, v \rangle_3 \ .$$

However, for $\langle y, z \rangle_2 : \Pi_2\ (\Pi_2\ A\ B)\ C$, we have:

$$f'\ (f\ \langle y, z \rangle_2) \longrightarrow_\iota \longrightarrow_\beta f'\ \langle p_1^2\ y, p_2^2\ y, z \rangle_3$$
$$\longrightarrow_\iota \longrightarrow_\beta \langle \langle p_1^2\ y, p_2^2\ y \rangle_2, z \rangle_2 \neq_{\beta\iota} \langle y, z \rangle_2 \ ,$$

once again because the type $\Pi_2\ A\ B$ doesn't enjoy surjective pairing. This means that even in an extensional sense (on canonical elements), f is only a *retraction*, and not an isomorphism. Of course, the same situation will appear if we consider the product of n elements expressed with Π_n, and using a superposition of Π_k for $k < n$. While we will not consider deeply the case of retractions in this paper, we think they deserve attention for further studies: this example suggests that Π_3 might be considered as the "canonical" representation of triples, for being the retract of all representations of triples. One may note that this observation demonstrates the usefulness of adding new reductions gradually. The correspondence between products of different

arity described above would remain hidden if surjective pairing was already present.

4.4 Surjective Pairing

Let us add the rule $\langle p_1^2\ x, p_2^2\ x\rangle_2 \longrightarrow_{SP} x$ (if x is of product type) to the system with $\beta\iota$-reductions. We will now show that the Deferment Lemma may also be applied to prove strong normalisation of a system of $\beta\iota$SP-reductions.

Consider a SP-reduction followed by some β- or ι-reduction.

$$t[\langle p_1^2\ s, p_2^2\ s\rangle_2] \longrightarrow_{SP} t[s] \longrightarrow_\iota t^*[s^*]\ .$$

If s does not have the form $\langle s_1, s_2\rangle_2$ or it does but the reduction does not use this occurrence of $\langle\cdot,\cdot\rangle_2$ then deferment is obviously possible.

Suppose the reduction that follows SP is ι, then t should be a term of the form $t[\langle p_1^2\ s, p_2^2\ s\rangle_2] \equiv t'[(\!|f|\!)_2\ \langle p_1^2\ s, p_2^2\ s\rangle_2]$ where $s : \Pi_2\ A\ B$, $s_1 : A$, $s_2 : B$, $f : A \to B \to C$ and we have

$$t'[(\!|f|\!)_2\ \langle p_1^2\ s, p_2^2\ s\rangle_2] \longrightarrow_{SP} t'[(\!|f|\!)_2\ \langle s_1, s_2\rangle_2] \longrightarrow_\iota t'[f\ s_1\ s_2]\ .$$

This can be replaced by

$$\begin{aligned} t'[(\!|f|\!)_2\ \langle p_1^2\ s, p_2^2\ s\rangle_2] &\longrightarrow_\iota t'[f\ (p_1^2\ s)\ (p_2^2\ s)] \\ &\longrightarrow_\iota\longrightarrow_\beta t'[f\ s_1\ (p_2^2\ s)] \longrightarrow_\iota\longrightarrow_\beta t'[f\ s_1\ s_2] \end{aligned}$$

(a trivial case of deferment). It is easy to see that local confluence will hold as well.

5 Isomorphic Copies of (Non-)Algebraic Types

The notion of the copy of a type is a very important one, and occurs quite often in many developments. For example, such operations are frequently used in tree-processing programs such as compilers. In this section, we study how isomorphisms may be used to devise an extended notion of copy, namely the *isomorphic copy* (for want of a better name).

Let us consider two extensionally isomorphic types A and B with isomorphisms $f : A \to B$ and $f^{-1} : B \to A$, and a type

$$C \equiv \mathrm{Ind}(\alpha)[\,c_1 : \sigma_1^1 \to \ldots \to \sigma_1^{k_1} \to \alpha\,;\ \ldots\ ;\ c_n : \sigma_n^1 \to \ldots \to \sigma_n^{k_n} \to \alpha\,]\ ,$$

possibly containing occurrences of A. An *isomorphic copy* C' of C differs by names of introduction operators, e.g $c'_1, \ldots, c'_n$, and by the fact that each "atomic" occurrence of A in C is replaced by an occurrence of B in C' (that is to say: A will be replaced by B only if it occurs either as a non-recursive operator, or as the premise —*i.e*, the type of an argument— of a strictly positive operator).

The reader who prefers a less abstract setting may suppose the isomorphisms between A and B belong to the class studied in section 4. It can be also intensional isomorphism, e.g., permutation of premisses of a functional type.

The definitions below also may be modified in such a way that only some selected occurrences of A are considered.

Now, let us define a function icopy : $C \to C'$ which converts canonical objects from one type to the other. Formally, icopy is of the form $\mathcal{R}_{C,C'}\ M_1 \dots M_n$. For every constructor $c_i : \sigma_i^1 \to \dots \to \sigma_i^{k_i} \to C$, let $\{j_p\}_{p=1,\ell} = \mathrm{SP}_\alpha(\sigma)$ and let us denote every strictly positive operator $\sigma_i^{j_p}$ by $\nu_{i,j,1} \to \dots \nu_{i,j,p_{i,j}} \to \alpha$. Then, we have

$$M_i \equiv \lambda x_1 : \sigma_i^1[C/\alpha] \cdot \dots \cdot \lambda x_{k_i} : \sigma_i^{k_i}[C/\alpha] \cdot \\ \lambda w_{j_1} : \sigma_i^{j_1}[C'/\alpha] \cdot \dots \cdot \lambda w_{j_\ell} : \sigma_i^{j_\ell}[C'/\alpha] \cdot c'_i\ \delta_1 \dots \delta_{k_i}$$

where

$$\delta_m \equiv \begin{cases} \text{(a)} & \lambda z_1 : \nu'_{i,m,1} \cdot \dots \cdot \lambda z_p : \nu'_{i,m,p_{i,m}} \cdot w_m\ z'_1 \dots z'_p & \text{if } m \in j_1, \dots, j_\ell; \\ \text{(b)} & f\ x_m & \text{if } \sigma_i^m \equiv A; \\ \text{(c)} & x_m & \text{otherwise;} \end{cases}$$

and, for $1 \leqslant r \leqslant p_{i,m}$:

- $\nu'_{i,m,r} \equiv B$ and $z'_r \equiv f^{-1}\ z_r$ if $\nu_r \equiv A$;
- $\nu'_{i,m,r} \equiv \nu_{i,m,r}$ and $z'_r \equiv z_r$ otherwise.

The function $\mathrm{icopy}^{-1} : C' \to C$ is defined similarly.

We may now consider the behaviour of icopy and icopy^{-1} w.r.t introduction operators, assuming that the new σ-reductions $\mathrm{icopy}^{-1}\ (\mathrm{icopy}\ x) \longrightarrow_\sigma x$ *and* $f^{-1}\ (f\ x) \longrightarrow_\sigma x$ are added. The main observation is that

$$\mathrm{icopy}^{-1}\ (\mathrm{icopy}\ (c_i\ t_1 \dots t_{k_i})) \longrightarrow^+_{\beta\iota} c_i\ t'_1 \dots t'_{k_i}$$

where t'_j:

- is t_j in case (c);
- is $f^{-1}\ (f\ t_j)$ in case (b);
- and is of the form $\lambda z_1 : \nu_{i,j,1} \cdot \dots \cdot \lambda z_p : \nu_{i,i,p_{i,j}} \cdot \mathrm{icopy}^{-1}\ (\mathrm{icopy}\ (t_j\ z'_1 \dots z'_p))$ where $z'_r \equiv f^{-1}\ (f\ z_r)$ if $\nu_r \equiv A$, $z'_r \equiv z_r$ otherwise, in case (a).

Now, suppose we have a term of the form $q[\mathrm{icopy}^{-1}\ (\mathrm{icopy}\ (c_i\ t_1 \dots t_{k_i}))]$.

Then, by one single σ-reduction, we have

$$q[\text{icopy}^{-1}\ (\text{icopy}\ (c_i\ t_1 \ldots t_{k_i}))] \longrightarrow_\sigma q[c_i\ t_1 \ldots t_{k_i}]\ .$$

But we may try to defer this σ-reduction. First, we have

$$q[\text{icopy}^{-1}\ (\text{icopy}\ (c_i\ t_1 \ldots t_{k_i}))] \longrightarrow^+_{\beta\iota} q[c_i\ t'_1 \ldots t'_{k_i}]\ .$$

Now, the deferment will depend on which cases the t'_j are in. In case (c), we have $t'_j \equiv t_j$, so no more reduction is to be done to close the diagram. If case (b) happens, some σ-reductions will be needed:

$$q[c_i\ t'_1 \ldots t'_{k_i}] \longrightarrow^+_\sigma q[c_i\ t_1 \ldots t_{k_i}]\ .$$

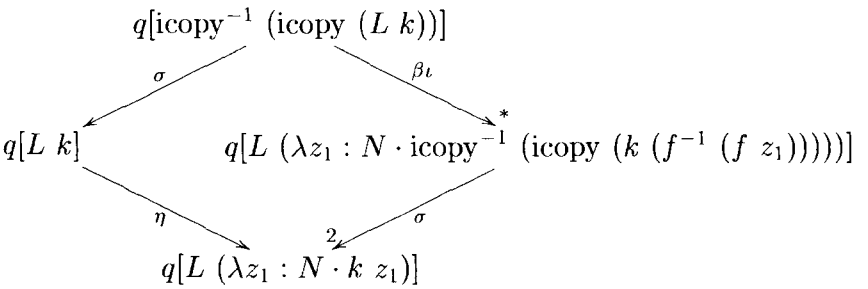
Finally, if case (a) happens, carrying some σ-reductions may lead to an unclosed diagram:

$$q[c_i\ t'_1 \ldots t'_{k_i}] \longrightarrow^+_\sigma q[c_i\ t''_1 \ldots t''_{k_i}]\ ,$$

where t''_j may begin by some abstractions. This situation will not happen only in the specific case, similar in result to case (b), where σ^j_i is a strictly positive operator over α of null arity, *i.e* $\sigma^j_i \equiv \alpha$. For example, this is the case for the 'S' constructor of ordinals. In the general case however (*i.e* with σ^j_i being a strictly positive operator over α of non-null arity), the only way to close the diagram seems to add further η-expansions in the following way:

$$q[c_i\ t_1 \ldots t_{k_i}] \longrightarrow^+_\eta q[c_i\ t''_1 \ldots t''_{k_i}]\ .$$

As an example, we have, for the 'L' constructor of ordinals the following reduction graph:



As a conclusion, if we only meet cases (c) and (b), and case (a) with only null-arity strictly positive operators, it is always possible to (0-)defer σ-reductions in the calculus. Thus $\beta\iota\sigma$-reduction is strongly normalising for "algebraic" types. Confluence follows easily, with a similar proof as for Theorem 4.6 on page 10.

As we briefly discussed above, our "strategy" is to add new reductions one by one. Thus, even the result for algebraic types only opens a large field of

applications for icopy, generated by isomorphisms of parameters introduced previously.

The difficult case is when "non-algebraic" types occur. Recently we obtained a proof for this case and the system with η-expansion.

Definition 5.1 We define η-expansion as follows:

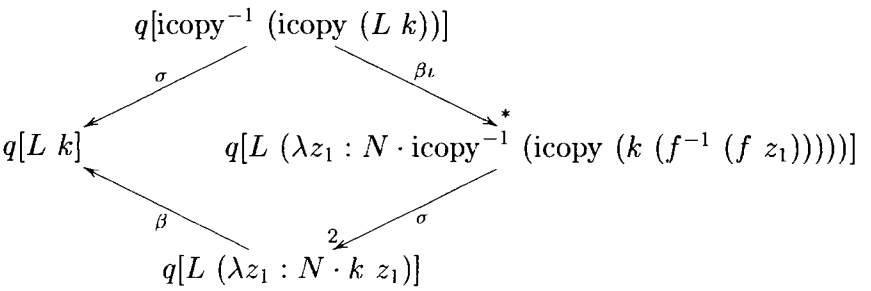
$$M \longrightarrow_\eta \lambda x : A \cdot M\ x \quad \text{if} \quad \begin{cases} M \text{ is of function type } A \to B \\ M \text{ is neither an abstraction nor applied.} \end{cases}$$

In detailed form the proof is too long to be presented here and we shall only give an outline.

The main observation used in this proof is that if the terms $t_1, \ldots, t_{k_i}$ above are in η-expanded form then

$$q[c_i\ t_1 \ldots t_{k_i}] \ \overset{+}{\underset{\beta}{\longleftarrow}}\ q[c_i\ t''_1 \ldots t''_{k_i}] \ .$$

E.g., the diagram for 'L' constructor may be closed differently:



Since we consider the system with η-expansions, we need a proof that the system with $\beta\iota$ and η-expansions is strongly normalising and confluent (we currently have a sketch of this proof).

To prove strong normalisation of the system extended not only by η but by σ-reductions related to icopy we assume that there is an infinite reduction sequence including σ reductions.

To use the observation above we need a lemma that shows that this reduction sequence will remain infinite if we insert appropriate η-expansions (to make the terms t in case (a) η-expanded).

After that, using a modification of deferment (to take into account the condition that the terms t are η-expanded) we show that it would be possible to obtain an infinite sequence consisting of $\beta\eta\iota$ only and this contradiction shows that the system with σ is SN.

The proof is completed by verification of confluence.

Acknowledgement

We would like to thank Roberto Di Cosmo for helpful discussions, and Freiric Barral for his help and proofreading.

6 Conclusion

The systems based on intensional equality (e.g., many proof assistants) often puzzle mathematically-oriented users because some familiar functional equalities (such as equalities related to commutativity and associativity of product) are no more viewed as computational and their use may require additional and heavy proof development. The arguments in favor of the equality based only on $\beta\iota$-reduction (or even $\beta\eta\iota$) may look nice from the foundational point of view but, pragmatically speaking, there is no harm if an extension of a reduction system doesn't destroy properties such as strong normalisation and confluence.

In this short paper, we studied two cases that seem of interest: extensions of reduction systems related to products and also to *"isomorphic" copies* of a type.

As for products, using the Deferment Lemma, we were able to prove that adding a rewriting rule corresponding to commutativity of products keeps the calculus strongly normalising and confluent. The same lemma also enabled us to show that adding surjective pairing to the system of $\beta\iota$-reductions does not break normalisation and confluence properties.

Secondly the notion of *isomorphic copy*, is useful for a clean distinction between the multiple uses of the type itself and of its copies. E.g., in proof assistants, the type of Even numbers is often defined as a copy of type Nat together with an appropriate coercion Even $\to$ Nat. Combining this coercion with the isomorphism copy defined above, we may obtain representations of classes of numbers modulo 2^n. Furthermore, isomorphic copies of non-algebraic types may require a notion of η-expansion, and hence to show that $\beta\eta\iota\sigma$-reduction is strongly normalising and confluent.

There are several recent works where normalisation in extended reduction systems is considered (e.g., [21] or [7,8]). This makes the perspective seem quite optimistic.

The calculus we considered here (the simply-typed λ-calculus with inductive types) is a compromise between the richness provided by inductive constructions and the relative simplicity of simply-typed systems. In the case of dependent types, one will meet more difficulties because new reductions will influence type-equality as well.

The subject needs more investigation but appropriate methods (e.g., a modification of H. Goguen's Typed Operational Semantics, see [17]) will prob-

ably lead to useful results of the same type as presented here.

References

[1] Baader, F. and T. Nipkow, "Term Rewriting and All That," Cambridge University Press, New York, 1998.

[2] Bachmair, L. and N. Dershowitz, *Commutation, transformation, and termination*, in: J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, Lecture Notes in Computer Science **230** (1986), pp. 5–20.

[3] Barendregt, H. P., "The Lambda Calculus - Its Syntax and Semantics," North-Holland, Amsterdam, 1984.

[4] Barendregt, H. P., *Lambda calculi with types*, in: D. M. Gabbai, S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, Oxford University Press, Oxford, 1992 .

[5] Barras, B., S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi and B. Werner, *The Coq proof assistant reference manual : Version 6.1*, Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France (1997).

[6] Barthe, G. and O. Pons, *Type isomorphisms and proof reuse in dependent type theory*, in: F. Honsell and M. Miculan, editors, *Proceedings 4th Int. Conf. on Found. of Software Science and Computation Structures, FoSSaCS'01, Genova, Italy, 2–6 Apr. 2001*, Lecture Notes in Computer Science **2030**, Springer-Verlag, Berlin, 2001 pp. 57–71.

[7] Blanqui, F., J.-P. Jouannaud and M. Okada, *The calculus of algebraic constructions*, in: P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)* (1999), pp. 301–316.

[8] Blanqui, F., J.-P. Jouannaud and M. Okada, *Inductive-data-type systems*, Theoretical Computer Science **272** (2002), pp. 41–68.

[9] Bruce, K., R. Di Cosmo and G. Longo, *Provable isomorphisms of types*, Technical Report 90-14, LIENS, École Normale Supérieure, Paris (1990).

[10] Dezani-Ciancaglini, M., *Characterization of normal forms possessing inverse in the $\lambda - \beta - \eta$-calculus*, Theoretical Computer Science **2** (1976), pp. 323–337.

[11] Di Cosmo, R., "Isomorphisms of Types: From λ-Calculus to Information Retrieval and Language Design," Progress in Theoretical Computer Science, Birkhäuser, Boston, MA, 1995.

[12] Doornbos, H. and B. von Karger, *On the union of well-founded relations*, Logic Journal of the IGPL **6** (1998), pp. 195–201.

[13] Dowek, G., G. Huet and B. Werner, *On the definition of the eta-long normal form in type systems of the cube*, in: H. Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, Nijmegen, The Netherlands, 1993.

[14] Geser, A., "Relative Termination," Ph.D. thesis, Universität Passau, Passau, Germany (1990).

[15] Geuvers, H., *The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi*, in: *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Santa Cruz, California, 1992, pp. 453–460.

[16] Geuvers, H., "Logics and Type Systems," Ph.D. thesis, Computer Science Institute, Katholieke Universiteit Nijmegen (1993).

[17] Goguen, H., *A typed operational semantics for type theory*, LFCS report ECS-LFCS-94-304, University of Edinburgh, Department of Computer Science (1994).

[18] Luo, Z., "Computation and Reasoning: A Type Theory for Computer Science," Number 11 in International Series of Monographs on Computer Science, Oxford University Press, 1994.

[19] Paulin-Mohring, C., *Inductive definitions in the system Coq. Rules and properties*, in: M. Bezem and J. F. Groote, editors, *Proceedings of the 1^{st} International Conference on Typed Lambda Calculi and Applications, TCLA '93*, Utrecht, The Netherlands, Lecture Notes in Computer Science **664** (1993), pp. 328–345.

[20] Soloviev, S. V., *The category of finite sets and Cartesian closed categories*, in: *Theoretical Applications of Methods of Mathematical Logic III*, Zapiski Nauchnykh Seminarov LOMI **105**, Nauka, Leningrad, 1981 pp. 174–194, english translation in *Journal of Soviet Mathematics*, 22(3) (1983), 1387–1400.

[21] Walukiewicz, D., *Termination of rewriting in the calculus of constructions*, Lecture Notes in Computer Science **1443** (1998).

[22] Werner, B., *Méta-théorie du Calcul des Constructions Inductives*, Thèse Univ. Paris VII, France (1994).

Polymorphic type-checking for the ramified theory of types of *Principia Mathematica*

M. Randall Holmes [1]

Department of Mathematics, Boise State University, 1910 University Drive, Boise, Idaho 83725-1555, USA

Abstract

A formal presentation of the ramified theory of types of the *Principia Mathematica* of Russell and Whitehead is given. The treatment is inspired by but differs sharply from that in a recent paper of Kamareddine, Nederpelt and Laan. A complete algorithm for determining typability and most general polymorphic types of propositional functions of the ramified theory of types is presented, unusual in requiring reasoning about numerical inequalities in the course of deduction of type judgments (to support unification of orders). Software implementing these algorithms has been developed by the author, and examples of the use of the software are presented. This is an abridged version of a longer paper which may appear later elsewhere.

This paper was inspired by reading [3], where Kamareddine, Nederpelt and Laan present a formalization of the ramified theory of types (usually to be abbreviated *RTT*) of [5], the *Principia Mathematica* of Russell and Whitehead (hereinafter *PM*). It is surprising that the theory of types of *PM* (the oldest one) is nowhere given a rigorous formal description; in fact, *PM* has no notation for types! There are various formal systems of ramified type theory in the literature (the author has even presented one in [1]), but the one in [3] is the only one known to us that is close to *PM* in details of its notation.

While reading [3], we developed a type checker ([2]) for its version of *RTT*. We used the same notation for propositional functions that is used in [3] (except that we were able to omit type labels on quantified variables, which makes our notation closer to that of *PM*), but we took a quite different approach to reasoning about types. From the checker it is possible to "reverse engineer" a formal treatment of the type system of *RTT* different from that of [3], which we give here.

This is an abridged version of a longer paper which we hope to publish elsewhere. Here we omit a section which discusses differences between the

[1] Email: holmes@math.boisestate.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

notation of [3] and the original notation of *PM*.

The logical world of *PM* is inhabited by *individuals* and *propositional functions*. We will usually abbreviate "propositional function" as "pf" (following [3]). In this section we introduce notation for these.

An individual is denoted by one of the symbols $a_1, a_2, a_3, \ldots$ (in the computer implementation, `a1, a2, a3...`). We call these symbols "individual constants".

Before we present the notation for propositions, we need to introduce variables and primitive relation symbols. A variable is one of the symbols $x_1, x_2, x_3, \ldots$ (`x1, x2, x3...` in the computer implementation). A primitive relation symbol is a string of upper-case letters with a numerical subscript indicating its arity (in the paper, R_1 and S_2 are primitive relation symbols: these would be `R1` and `S2` in the computer implementation).

We note that we will freely use the word "term" in the sequel for any piece of notation, whether propositional notation, the name of an individual, or a variable.

Now we present the definition of notation for propositions. The notion of free occurrence of a variable in a proposition is defined at the same time. It is worth noting here that notation for a proposition is usually but not always also notation for a propositional function (pf).

atomic proposition: A symbol $R_n(v_1, \ldots, v_n)$ consisting of a primitive relation symbol with arity n followed by a list of n arguments v_i, each of which is either a variable x_{j_i} or an individual constant a_{j_i}, is an atomic proposition. ($R_0()$ is also an atomic proposition in the system of [3], and for us as well for now. The software that motivates this paper supports the ability to turn on or off a requirement that primitive relation symbols and propositional functions have positive arity). The free occurrences of variables in an atomic proposition are exactly the typographical occurrences of variables in it.

negation: If P is a proposition, then $\neg P$ (`~P` in the computer implementation) is a proposition, the negation of the proposition P. The free occurrences of variables in $\neg P$ are precisely the free occurrences of variables in P.

binary propositional connectives: If P and Q are propositions, then $(P \vee Q)$ is a proposition. Other connectives can be defined. In the computer implementation, propositional connectives are strings of lower case letters: `(P v Q)`, `(P implies Q)`, `(P and Q)`, `(P iff Q)`. The free occurrences of variables in $(P \vee Q)$ are the free occurrences of variables in P and Q; defined binary connectives would have the same rule.

quantifiers: If P is a proposition in which the variable x_i occurs free (this condition is what requires us to define "free variable" at the same time as "propositional notation"), $(\forall x_i.P)$ is a proposition (this is written `[xi]P` in the computer implementation). The existential quantifier $(\exists x_i.P)$ (written

[Exi]P in the computer implementation) can be introduced by definition. The free occurrences of variables in $(\forall x_i.P)$ are the free occurrences of variables other than x_i in P (and similarly for any other quantifier).

In [3], the structure of the typing algorithm required the attachment of explicit type labels to variables bound by quantifiers. In our system, this is not necessary. This is closer to the situation in *PM*, where no type indices appear (There is no notation for types in *PM*, so there can't be type indices; there are occasional appearances of numerical superscripts representing "orders").

propositional function application ("matrix" and general): If x_i is a variable and $A_1, \ldots, A_n$ is an argument list in which each A_i is of one of the forms a_{j_i} (an individual constant), x_{j_i} (a variable) or P_i (notation for a proposition, representing a pf), then $x_i(A_1, \ldots, A_n)$ and $x_i!(A_1, \ldots, A_n)$ are propositions. In the latter notation, the exclamation point indicates that the "order" of the type of the variable x_i is as low as possible: this will be clarified when types and orders are discussed. The notation $x_i!(A_1, \ldots, A_n)$ does not appear in [3]; its use in this paper is a generalization of the notation for "matrices" (predicative functions) in *PM*. $x_i()$ is also a proposition in the system of [3] (the variable x_i represents a proposition in this case); $x_i()$ and $x_i!()$ are propositions for us as well for now: if we require that primitive relation symbols and pfs have positive arity, then we exclude such propositions). The free occurrences of variables in $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, \ldots, A_n)$ are the head occurrences of x_i and those A_i's which are variables: note that the free occurrences of variables in those A_i's which are pf notations are *not* free occurrences of variables in $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, \ldots, A_n)$.

completeness of definition: All propositional notations are constructed in this way.

As usual, an occurrence of a variable in a proposition which is not free is said to be bound. Note that a variable x_i is not a propositional notation.

The notation for a propositional function is the same as the notation for a proposition: a pf is construed as a function of the variables which appear in it (or rather of the variables which appear free in it). When 0-ary predicates are forbidden (this is arguably the case in *PM* (see remark on p. 38) and is supported as an option by our checker), a propositional notation must contain a free variable to represent a pf; otherwise a propositional notation without free variables will represent a 0-ary propositional function. The full system of the checker also allows propositional variables (which are used in *PM*) but does not allow occurrences of propositional variables in pfs.

Since we do not have head binders in the notation for pfs to determine the order of multiple arguments, we allow the order of the indices of the variables (which we may refer to occasionally as "alphabetical order") to determine the order in which arguments are to be supplied to the function. This follows *PM*.

We refer to the atomic propositions and the propositional function applica-

tion terms as "logically atomic", and to other terms as "logically composite".

We now give the recursive definition of simultaneous substitution of a list of individuals, variables and/or propositional functions A_k for variables x_{i_k} in a proposition P, for which we use the notation $P[A_k/x_{i_k}]$. The clauses of the definition follow the syntax. It is required that the subscripts i_k be distinct for different values of k.

atomic propositions: Let $R_n(v_1, \ldots, v_n)$ be an atomic proposition. For each v_i and index k, define v_i' as A_k if v_i is x_{i_k}; define v_i' as v_i if v_i is not any x_{i_k}. If any v_i' is a pf notation, $R_n(v_1, \ldots, v_n)[A_k/x_{i_k}]$ is undefined; otherwise $R_n(v_1, \ldots, v_n)[A_k/x_{i_k}]$ is defined as $R_n(v_1', \ldots, v_n')$.

negation: $(\neg P)[A_k/x_{i_k}] = \neg(P[A_k/x_{i_k}])$

binary propositional connectives: $(P \vee Q)[A_k/x_{i_k}] = (P[A_k/x_{i_k}] \vee Q[A_k/x_{i_k}])$. The rule is the same for any binary propositional connective.

quantification: Let $(\forall x_j.P)$ be a quantified sentence (the rule is the same for any quantifier). Define A_k' as x_j in case $i_k = j$ and as A_k otherwise. Then $(\forall x_j.P)[A_k/x_{i_k}]$ is defined as $(\forall x_j.P[A_k'/x_{i_k}])$.

pf variable application: Let $x_j(V_1, \ldots, V_n)$ or $x_j!(V_1, \ldots, V_n)$ be a proposition built by application. Define B' for any notation B as A_k if B is x_{i_k} and as B otherwise. We define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x_j'(V_1', \ldots, V_n')$ and $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x_j'!(V_1', \ldots, V_n')$ except in the case where x_j' is a pf notation Q: in this case something rather more complicated happens. It will be undefined unless there are precisely n variables which occur free in Q. If there are n variables which occur free in Q, define t_k so that x_{t_k} is the kth free variable in Q in alphabetical order. Then define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ or $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $Q[V_k'/x_{t_k}]$.

There is a serious difficulty with this "definition". Consider the pf $\neg x_1(x_1)$ (this certainly is a pf by our definition above). Now substitute $\neg x_1(x_1)$ for the variable x_1 in the proposition $\neg x_1(x_1)$ itself. We will obtain the negation of the result of replacing x_1 with $\neg x_1(x_1)$ in $x_1(x_1)$. Giving $\neg x_1(x_1)$ the name R for the moment, we see that the result of the latter substitution will be $R[R/x_1]$; but this is exactly the substitution we started out trying to make, so we have landed in an infinite regress. This illustrates the fact that the circularity of the proposed "definition" of substitution is essential – in the last clause, there is no guarantee that the instance of substitution $Q[V_k'/x_{t_k}]$ to be carried out is "simpler" in any way than the original substitution $x_j'(V_1, \ldots, V_n)[A_k/x_{i_k}]$ being defined, and our example shows that it need not be.

It is hoped that the reader will notice that this is essentially Russell's paradox of naive set theory. Our solution will be the official solution of *PM*: we will impose a type system, under which the term $\neg x_1(x_1)$ will fail to denote a pf, and the problem will disappear. For the moment, we withdraw the definition of substitution, and will return to it after we have presented the type system.

The self-contained approach to the definition of substitution taken here may be contrasted with the rather elaborate invocation of λ-calculus in [3]. Though our definition appears to have failed at this point, the type system will allow us to give the definition above as a legitimate inductive definition. The reason we can do this and the authors of [3] cannot is that their definition of the typing algorithm depends on the notion of substitution, and ours does not. (Our algorithm does depend on the notion of substitution into notations for *types*, as we will see below, but the definition of substitution into types does not present logical difficulties presented by the definition of substitution into propositions or pfs).

We follow [3] in presenting the simple theory of types without orders first, though historically it was presented by Ramsey as a simplification of the ramified theory of types of *PM*.

The base type of the system of *PM* is the type 0 inhabited by individuals. (Nothing prevents the adoption of additional base types, or indeed the avoidance of commitment to any base type at all).

All other types are inhabited by propositional functions. In the simple theory of types, the type of a pf is determined precisely by the list of types of its arguments.

We introduce notation for simple types:

Individuals: 0 is a type notation.

Propositions: () is a type notation (for the type of propositions).

Propositional Functions: If $t_1, \ldots, t_n$ are type notations, $(t_1, \ldots, t_n)$ is a type notation. (If pfs were required to have positive arity, we would require $t_i \neq ()$ here).

Variable Types: For each variable x_i, we provide a type notation $[x_i]$. (This notation is an innovation for this paper: it represents an unknown (polymorphic) type to be assigned to x_i; these types may also be called "polymorphic types").

Completeness of Definition: All simple type notations are derived in this way.

No Nontrivial Identifications: Types not containing variable types are equal precisely if they are typographically identical.

As is noted in [3], there is no notation for types in *PM*: this notation is apparently due to Ramsey (except for our innovation of variable types, whose purpose will become clear below).

Our aim in this essay is to avoid the necessity of assigning types overtly to variables, which is truer to the approach taken in *PM* itself. It is useful to consider what a system with explicit type assignment would look like, though.

The type assignment is represented as a partial function from terms to types: $\tau(x_i)$ is the type to be assigned to x_i, and more generally $\tau(t)$ is the type to be assigned to the individual constant, variable, or propositional function

t. Types in the range of τ are constant types (they contain no type variables $[x_i]$). We require that bound variables be typed as well as free variables, and identity of variables implies identity of type regardless of free or bound status. We stipulate that every variable is in the range of τ and that the inverse image of each type under τ contains infinitely many variables: this has the same effect as providing infinitely many variables labelled with each type. The following rules simultaneously tell us which terms are typable (have values under τ) and how to compute the value of τ if there is one. Functions τ satisfying these rules are called "type functions on P", where P is a fixed proposition or propositional function.

individuals: If x_i appears as an argument in an atomic subproposition of P, $\tau(x_i) = 0$. $\tau(a_i) = 0$ for any individual constant a_i.

propositional functions: If Q is a propositional function appearing as a subterm of P, every subterm of Q has a value under τ, and the n free variables of Q, indexed in increasing order, are x_{i_k}, $\tau(Q) = (\tau(x_{i_1}), \ldots, \tau(x_{i_n}))$. If Q contains no free variables, then $\tau(P) = ()$.

variable application: If $x_j(A_1, \ldots, A_n)$ or $x_j!(A_1, \ldots, A_n)$ is a subterm of P, then $\tau(x_j) = (\tau(A_1), \ldots, \tau(A_n))$.

These rules are to be understood as additional restrictions on well-formedness of terms: a term P is to be considered well-formed iff there is a type function τ on P. Notice that the value of τ at every term (or its lack of value) is completely determined by the values of τ at variables. The process described terminates by induction on the structure of propositional notations: to compute the type (or assess the typability) of any notation other than a variable or individual constant, we appeal only to the types of proper subterms of that notation, and we are given types of variables and individual constants at the outset.

We now proceed to develop a system for expressing and reasoning about type assignments to subterms of propositional functions, adopting rules on the basis of their validity for an intended interpretation in terms of type functions.

There are four kinds of type judgments. In the following, P stands for a propositional function or proposition, t, u stand for types (variable types $[x_i]$ are permitted to appear as types and as components of complex types) and x_i stands for a variable. The meanings of these judgments will be modified by a redefinition of the notion of "type function on P" which will be given below.

ill-typedness: "P is ill-typed" is defined as "there is no type function τ on P".

propositional function type assignment: "P has type t" means "for all type functions τ on P, $\tau(P) = t$", where any type $[x_i]$ appearing in t is interpreted as $\tau(x_i)$.

variable type assignment: "x_i has type t in P" means "for all type functions τ on P, $\tau(x_i) = t$", where any type $[x_j]$ appearing in t is interpreted

as $\tau(x_j)$.

type equality: "$t = u$ in P" is defined as "for all type functions τ on P, $t = u$", where any type $[x_j]$ appearing in t or u is interpreted as $\tau(x_j)$.

We now develop rules for deduction about type judgments, showing that the rules are valid in the intended interpretation.

We begin with the observation that the conditions defining a type function on P depend only on the appearances of variables in logically atomic subterms of P: these conditions assign types to arguments appearing in atomic propositions, to propositional functions, which can only appear as arguments of propositional function application terms, and to the head variables of propositional function application terms. It follows immediately from this that τ is a type function on P under precisely the same conditions under which it is a type function on $\neg P$ or on $(\forall x_i.P)$ (if the latter is well-formed), since these terms contain precisely the same logically atomic subterms. Further, it follows that any type function on $(P \vee Q)$ is also a type function on P and on Q, since it will satisfy the conditions on logically atomic subterms of P and Q, since the set of logically atomic subterms of $(P \vee Q)$ is the union of the set of logically atomic subterms of P and the set of logically atomic subterms of Q.

These facts can be expressed as rules for reasoning about type judgments:

negations: $\neg P$ is ill-typed iff P is ill-typed. x_i has type t in $\neg P$ iff x_i has type t in P.

quantification: $(\forall x_i.P)$ (if well-formed) is ill-typed iff P is ill-typed. x_j has type t in $(\forall x_i.P)$ iff x_j has type t in P.

binary propositional connectives: If P or Q is ill-typed, $(P \vee Q)$ is ill-typed. If x_i has type t in P or x_i has type t in Q, then x_i has type t in $(P \vee Q)$.

There are three kinds of occurrences of variables in logically atomic subterms: a variable can appear as an argument of an atomic proposition, as the head variable of a pf application term, or as an argument of a pf application term. The following rules express the type judgments we can make about occurrences of variables in each context:

individual variables: If $x_i = A_k$ in $R_n(A_1, \ldots, A_n)$, then x_i has type 0 in $R_n(A_1, \ldots, A_n)$.

applied variables: If A_i has type t_i for each i, then x_j has type $(t_1, \ldots, t_n)$ in $x_j(A_1, \ldots, A_k)$ or $x_j!(A_1, \ldots, A_k)$.

argument variables: x_i has type $[x_i]$ in P for any propositional function P (this kind of occurrence gives us no type information).

In this way a possibly variable type may be assigned to each occurrence of a variable on the basis of its logically atomic context. This is called the "local" type of the occurrence. However, more than one typographically different type

may be assigned to the same variable. For example, x_1 is assigned type 0 and type $[x_1]$ in $R_1(x_1) \vee x_2(x_1)$. Different types assigned to the same variable will of course be equal. We can express this in terms of type judgments.

multiple types: If x_i has type t in P and x_i has type u in P then $t = u$ in P.

variable type equations: If $[x_i] = t$ in P then x_i has type t in P.

Definition: We assign an integer *arity* to each type which is not a type variable. 0 has arity -1. () has arity 0. $(t_1, \ldots, t_n)$ has arity n. Note that a type may have variable type components, but it will still have arity if it is not itself a type variable. Note also that types which are equal will have equal arity if their arity is defined.

type distinction: If t and u each have arity and have distinct arities and $t = u$ in P, then P is ill-typed.

absurdity: If P is ill-typed, then P has type t, $t = u$ in P and x_i has type t in P for any t, u, and x_i (this is obviously true under the intended interpretation - we need it for a completeness result).

componentwise equality: If $(t_1, \ldots, t_n) = (u_1, \ldots, u_n)$ in P, then $t_i = u_i$ in P for each i.

type substitution: If x_i has type t in P and x_j has type u in P, then x_j has the type $u[t/[x_i]]$ obtained by substituting t for all occurrences of $[x_i]$ in u.

A consideration related to type substitution is that no type can be ill-founded: the type of a variable x_i cannot have $[x_i]$ as a proper component.

ill-foundedness: If x_i has type t in P and $t[t/[x_i]] \neq t$, then P is ill-typed.

Finally, we need the rule for typing propositional functions.

propositional function type: If the variables free in P, listed in order of increasing index, are $(x_{i_1}, \ldots, x_{i_n})$ and x_{i_k} has type t_k for each k, then P has type $(t_1, \ldots, t_n)$.

An additional rule is stated which we do not use in the computer implementation for simple type theory (though we do use it in ramified type theory), but which is needed for a completeness result for type functions as we have defined them.

types from arguments: If x_i has type t in A_k, then x_i has type t in $x_j(A_1, \ldots, A_n)$ and $x_j!(A_1, \ldots, A_n)$.

It should be clear that each of these rules is sound for the intended interpretation. We will prove that this set of rules is complete for the intended interpretation as well.

Theorem: For each propositional function P, there is a type t such that "P has type t" is deducible from the rules above and the types possible as values $\tau(P)$ for a type function τ on P are precisely the types obtainable by substituting arbitrary types for each type variable appearing in t.

Proof of Theorem: We describe the computation of the type t. The idea is to construct a set of judgments "x_i has type t_i" deducible using the type judgment rules which satisfies all the rules for a type function except that t_i's may type variables: arbitrary instantiation of the type variables (and extension of the function to variables not appearing in P) then yields a true type function.

Begin the construction of the set of judgments by computing the "local" type of each occurrence of each variable x_i. We prove the theorem by structural induction: we assume that each pf argument of pf application terms can be assigned a type satisfying the conditions of the theorem (so that we can assign types to the head variables of these terms).

This fails to induce a type function on P (mod instantiation of type variables with concrete types) only if more than one type is assigned to the same variable. We describe a procedure for resolving such situations.

If any variable is assigned types of different arities, or if any variable x_i is assigned a type which contains $[x_i]$ as a proper component, the process terminates with the judgment that P is ill-typed.

If x_i is assigned any type t which is not a variable type (t may be a composite type with variable components) replace all occurrences of $[x_i]$ in types assigned to other variables with the type t. If x_i is assigned type $[x_j]$ ($j \neq i$), replace all occurrences of the type $x_{\mathtt{min}\{i,j\}}$ in types assigned to all variables with the type $x_{\mathtt{max}\{i,j\}}$. This is justified by the type substitution rule. In the process described below, carry out these substitutions whenever a new type assignment is made. Notice that such a substitution will occur at most once for any given variable x_i, since it eliminates the target type everywhere. Of course, if $[x_i]$ is introduced as a proper component of the type of x_i, terminate with a judgment of ill-typedness.

If x_i is assigned types $[x_j]$ and t in P, add the judgment "x_j has type t in P" and eliminate the type assignment "x_i has type $[x_j]$ in P" (note that all occurrences of $[x_j]$ will then be eliminated if t is not a type variable). In one special case we proceed differently: if x_i is assigned types $[x_j]$ and $[x_k]$, we assign x_i, x_j, and x_k the type $x_{\mathtt{max}\{i,j,k\}}$.

If x_i is assigned types $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ in P, the judgments $t_i = u_i$ follow for each relevant i. From these equality judgments continue to deduce further equality judgments in the same way. This process will terminate with either a judgment that P is ill-typed or a finite nonempty set of nontrivial judgments of the form $[x_k] = v_k$, each of which has "x_k has type v_k" as a consequence, which we add to our list of type assignments. Assign to x_i the type which results if all these types x_k are replaced with the corresponding v_k's in either of the two types being reconciled (the same type results in either case). Note that no new assignment to x_i can result, because $[x_i]$ cannot be a component of the type assigned to x_i unless P is ill-typed.

This process must terminate, because each step of the process described

eliminates at least one variable type $[x_i]$ from consideration or terminates with a judgment of ill-typedness.

When the process terminates, we will either have concluded that P is ill-typed (and this judgment will be honest because the rules are sound for the intended interpretation) or we will have obtained a set of type assignments to the variables appearing in P satisfying the conditions for a type function: any instantiation of type variables appearing in these types with constant types will give a type function on P.

It is important to note that this is a type algorithm based on the quite standard approach of type unification implemented, for example, in the type checking of the computer language *ML* (a standard reference is [4]).

We can now salvage the definition of substitution given above.

Convention: We stipulate henceforth that propositional notations are well-formed iff they are well-formed under the original definition and the judgment "P is ill-typed" cannot be deduced using the algorithm given above.

Theorem: $P[A_k/x_{i_k}]$, defined as above, will be well-defined as long as there is a fixed set of substitutions σ of types for polymorphic type variables such that the type of each A_k is the result of applying σ to the type of x_{i_k} in P.

Proof of Theorem: We only need to consider the case in which a propositional function Q is substituted for the variable x_j in a term $x_j(A_1, \ldots, A_n)$ or $x_j!(A_1, \ldots, A_n)$.

We reproduce the problematic clause from the definition of substitution.

"Let $x_j(V_1, \ldots, V_n)$ or $x_j!(V_1, \ldots, V_n)$ be a proposition built by application. We carry out the substitution of a finite list of terms A_k for corresponding variables x_{i_k}. Define B' for any notation B as A_k if B is typographically x_{i_k} and as B otherwise. We define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x'_j(V'_1, \ldots, V'_n)$ and $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x'_j!(V'_1, \ldots, V'_n)$ except in the case where x'_j is a pf notation Q: in this case something rather more complicated happens. It will be undefined unless there are precisely n variables which occur free in Q. If there are n variables which occur free in Q, define t_k so that x_{t_k} is the kth free variable in Q in alphabetical order. Then define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ or $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $Q[V'_k/x_{t_k}]$."

The type of the pf Q being substituted for x_j in P is the image under the fixed substitution σ of the type of x_j in P, and so is the image under σ of a proper component of the type of P. Thus, by a structural induction on types, the substitution $Q[V'_k/x_{t_k}])$ into Q used to define the substitution into P succeeds, because the image under σ of the type of Q is simpler than the image under σ of the type of P. Note that because P is well-typed, that substitution $Q[V'_k/x_{t_k}])$ will meet the typing conditions we require for substitutions: the fact that Q has the same type that x_j has in P, each V'_k has the same type as V_k in P, and $x_j(V_1, \ldots, V_n)$ is a subterm of P is sufficient to see this.

So the problem of substitution is solved by the adoption of simple type

theory.

The motivation behind the ramified theory is as follows. The type of a pf in *STT* is determined by the types of its arguments, and all types of its arguments must be proper components of its type and thus simpler than its type. It can said further (though such qualms are no longer fashionable) that understanding the meaning of a pf involves understanding the entire type over which any quantified variable appearing in the function ranges, so the type of a pf must be more complex than that of any variable over which quantification occurs in the pf. More concretely, Russell suggests in *PM* that a quantified sentence is to be understood as expressing an infinitary conjunction or disjunction in which sentences referring to every object of the type quantified over must occur. If quantified sentences are to be interpreted in this way, then the appearance of a quantified variable in a propositional function of the same type as the propositional function or of a more complex type would lead to formal circularity on expansion to infinitary form.

The restriction is enforced in *RTT* by adding to each type a new feature, a non-negative integer called its "order". The order of type 0 (the type of individuals) is 0 (zero). The type () of propositions in simple type theory is partitioned into types $()^n$ for each natural number n, where the order n will be the least natural number greater than the order of the type of any variable which occurs in the proposition (including quantified variables). A propositional function P containing n free variables x_{i_k} (listed in increasing order) with types t_k will have type $(t_1, \ldots, t_n)^m$, where m is the smallest natural number greater than the order of any of the types t_k and the order of the type of any variable quantified in P. A similar rule applies to the typing of head variables x_i in expressions $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, \ldots, A_n)$: the type of x_i will be $(t_1, \ldots, t_n)^r$ where each t_k is the type of A_k, and the order r is larger than the orders of the t_k's; in the term $x_i!(A_1, \ldots, A_n)$, the order r must be the smallest order larger than all orders of t_k's.

Polymorphic type-checking for this system is made difficult by the fact that a polymorphic type $[x_i]$ has unknown order (denoted by $|x_i|$) and a term $x_i(A_1, \ldots, A_n)$ has only a lower bound on its order, and so it is necessary to do a certain amount of arithmetical reasoning on unknown orders. A typical order is the maximum of a natural number n and several expressions of the form $|x_i| + m$. Unification of orders is a not entirely trivial problem.

This is all made concrete as follows. We begin with the definition of formal polymorphic orders.

Natural numbers are polymorphic orders. $|x_i|$ is a polymorphic order for each x_i. Formal maxima of polymorphic orders are polymorphic orders and so is the formal sum of a polymorphic order and a natural number.

Elementary properties of maximum and addition allow us to reduce any polymorphic order to a canonical form, which will be the maximum of a single natural number (if the natural number is 0 it is omitted) and a list of expressions $|x_i| + m$ (if m is 0 it is omitted) presented in ascending order of the

parameter i. Adding a natural number to such a standard form and taking the maximum of two such standard forms are computable operations.

If m and n are polymorphic types, we say $m > n$ when $\max(m, n+1) = m$. This is not a total order, of course.

The result $u[m/|x_i|]$ of substituting a polymorphic order m for the polymorphic order $|x_i|$ in a polymorphic order u is the result of replacing the occurrence of $|x_i|$ in u (if there is one: otherwise the result of the substitution is u) with m, then simplifying to canonical form.

Substitution into orders is needed to handle changes in order which take place when a more detailed type is substituted for a polymorphic type variable.

Now we are in a position to define ramified types (and their orders, simultaneously).

individuals: 0 is a ramified type of order 0.

propositions: If n is a polymorphic order, $()^n$ is a ramified type of order n.

propositional functions: If $t_1, \ldots, t_n$ are ramified types and m is a polymorphic order greater than the order of any of the types t_k, then $(t_1, \ldots, t_n)^m$ is a ramified type of order m.

polymorphic types: For each variable x_i, there is a ramified type $[x_i]$ of order $|x_i|$.

We present the rules for a term-typing function τ as above. Notice that here the orders will be fixed non-negative integers.

individuals: If x_i appears as an argument in an atomic proposition, $\tau(x_i) = 0$. $\tau(a_i) = 0$ if a_i appears.

propositional functions: If P is a propositional function and the n free variables of P, indexed in increasing order, are x_{i_k}, $\tau(P) = (\tau(x_{i_1}), \ldots, \tau(x_{i_n}))^m$, where m is one greater than the maximum of the orders of the types of the variables appearing in P (free or bound). If P contains no free variables, then $\tau(P) = ()^m$, where m is one greater than the maximum of the orders of the types of the variables appearing in P.

variable application: If $x_j!(A_1, \ldots, A_n)$ is a term, then $\tau(x_j) = (\tau(A_1), \ldots, \tau(A_n))^m$, where m is one plus the maximum of the orders of the types of the A_i's. If $x_j(A_1, \ldots, A_n)$ is a term, then $\tau(x_j) = (\tau(A_1), \ldots, \tau(A_n))^m$, for some m greater than the order of the type of any A_i.

Notice that in the ramified theory there is an additional case where the type of a variable cannot be rigidly deduced from its context: as before, the type of a variable argument to a pf variable is polymorphic (though it may be determined from other features of the context) and in addition the order of the type of x_j in a term $x_j(A_1, \ldots, A_n)$ only has a lower bound, not a fixed value (though further information in the context might fix the order or further restrict it). This will be reflected in additional appearances of polymorphic variables in our algorithm.

We will regard a pf as well-formed when there is a type function τ which assigns a type to that function. Some pfs will have many possible types, as above, which will be indicated by the appearance of type variables $[x_i]$ and order variables $|x_i|$ in the type resulting from the algorithm.

We now develop rules for deduction about type judgments, showing that the rules are valid in the intended interpretation. Our development will be parallel to the development for simple type theory above. We present only those clauses which differ from the clauses in the development for STT.

applied variables: If A_i has type t_i for each i, and the order of t_k is o_k for each k, then x_j has type $(t_1, \ldots, t_n)^r$ in $x_j!(A_1, \ldots, A_k)$, where $r = 1 + \max(o_1, ..., o_k)$, and x_j has type $(t_1, \ldots, t_n)^s$ in $x_j(A_1, \ldots, A_k)$, where $s = \max(|x_j|, o_1 + 1, \ldots, o_n + 1)$.

Definition: We assign an integer *arity* to each type which is not a type variable. 0 has arity -1. () has arity 0. $(t_1, \ldots, t_n)^m$ has arity n. Note that a type may have variable type components, but it will still have arity if it is not itself a type variable. Note also that types which are equal will have equal arity if their arity is defined. (this clause appears simply because order appears in composite types – note that the order has no effect on the arity).

componentwise equality: If $(t_1, \ldots, t_n)^{m_1} = (u_1, \ldots, u_n)^{m_2}$ in P, then $t_i = u_i$ in P for each i. (this clause appears, again, simply because order is mentioned; it is also the case that $m_1 = m_2$ will hold, but we have no judgment of this form available to us).

type substitution: If x_i has type t in P and x_j has type u in P, then x_j has the type $u[t/[x_i]]$ obtained by substituting t for all occurrences of $[x_i]$ in u. If x_i has type t in P and $u = v$ in P, then $u[t/[x_i]] = v[t/x_i]$ in P. (this clause appears because we need substitution into equality judgments; such a rule would be valid in STT but is not needed there).

In the rules above and below, it is important to note that substitution of a type t for a type variable $[x_i]$ also has the effect of substituting the order of t for all occurrences of the order variable $|x_i|$.

ill-foundedness: If x_i has type t in P and $t[t/[x_i]] \neq t$, then P is ill-typed. (Note that the computation of $t[t/[x_i]]$ includes the reduction of its order to standard form).

There is a form of circularity which does not lead to ill-typedness: a variable x_i may have a type whose order t is a maximum of types including $|x_i|$; the calculation of $t[t/[x_i]]$ includes the simplification of the order of t, which will reduce $t[t/[x_i]]$ to t.

propositional function type: If the variables free in P, listed in order of increasing index, are $(x_{i_1}, \ldots, x_{i_n})$, and the variables quantified in P are $(x_{i_{n+1}}, \ldots, x_{i_m})$, x_{i_k} has type t_k for each k and type t_k has order o_k for each k, then P has type $(t_1, \ldots, t_n)^r$, where $r = 1 + \max(o_1, \ldots, o_m)$.

It should be clear from our discussion that each of these rules is sound for the intended interpretation. However, this set of rules is not complete.

We now introduce the notion of "bounding variable" of an order.

Definition: If an order n is presented in the standard form $\max(n_0, n_1 + |x_{i_1}|, \ldots, n_k + |x_{i_k}|)$, and some n_j with $(j \neq 0)$ is equal to 0, then x_{i_j} is said to be a "bounding variable" of n.

It is important to observe that the only orders deduced by any of our rules which can have bounding variables are the polymorphic orders $|x_i|$ themselves and the orders assigned to x_j in terms $x_j(A_1, \ldots, A_n)$, which have bounding variable $|x_j|$. Any other polymorphic order that we assign is the successor $1+n$ of some order n, and it is clear that no successor order can have a bounding variable.

Further, the following rule clearly holds for types assigned by our algorithm:

bounding variables: If x_i has type t in P and the order of t has bounding variable x_j, then x_j has type t in P.

The reason for this is that any rule which assigns a type with bounding variable x_j in the first instance actually assigns this type to the variable x_j. Further, this implies that we can assume that any type with a bounding variable has only one, since the types of the bounding variables can be shown to be equal by this rule, and type substitution can then be used to eliminate one of them.

We present an incomplete but often successful algorithm for computation of the type of a proposition or propositional function P in RTT. This algorithm follows the STT algorithm very closely.

Provisional algorithm: We describe the computation of the type t. The idea, as before, is to construct a set of judgments "x_i has type t_i" deducible using the type judgment rules which satisfies all the rules for a type function except for possibly containing type variables: arbitrary instantiation of the type variables then yields a true type function.

Begin the construction of the set of judgments by computing the "local" type of each occurrence of each variable x_i. The algorithm is recursive in the same way as the STT algorithm: we assume that each pf argument of pf application terms has been successfully assigned a type.

If any variable is assigned types of different arities, or if any variable x_i is assigned a type which contains $[x_i]$ as a proper component, the process terminates with the judgment that P is ill-typed (note that if x_i is assigned a type with bounding variable $|x_i|$, this does not lead to forbidden circularity).

If x_i is assigned any type t which is not a variable type (including composite types with variable components) replace all occurrences of $[x_i]$ in types assigned to other variables with the type t. Note that this does not necessarily eliminate all occurrences of x_i: if the type of x_i has bounding

variable x_i, occurrences of $|x_i|$ will remain.

The assignment of type $[x_j]$ to a variable x_i is handled as in the *STT* algorithm.

Such substitutions will usually occur at most once for any given variable x_i, since the target type is usually eliminated everywhere. Of course, if $[x_i]$ is introduced as a proper component of the type of x_i, terminate with a judgment of ill-typedness. The exception in which the variable x_i is assigned a type with bounding variable x_i remains to be considered. Notice that as soon as a variable is assigned a type which does not have a bounding variable, any type which that variable may have been assigned which had a bounding variable will be converted to a form which does not have a bounding variable by the global substitution process.

If x_i is assigned types $[x_j]$ and t in P, add the judgment "x_j has type t in P" and eliminate the type assignment "x_i has type $[x_j]$ in P", except in two special situations which follow: if x_i is assigned types $[x_j]$ and $[x_k]$, we assign x_i, x_j, and x_k the type $x_{\max\{i,j,k\}}$, as in the *STT* algorithm. If the type t has bounding variable x_j, it must be the case that the judgment "x_j has type t in P" has already been made. In this case we define t' as $t[[x_{\max\{i,j\}}]/x_j]$ and assign this type to both x_i and x_j, replacing all occurrences of $[x_i]$ and $[x_j]$ in all type judgments with $[x_{\max\{i,j\}}]$. Observe that in each case at least one polymorphic type has been eliminated from all type judgments.

If x_i is assigned types $(t_1, \ldots, t_n)^{m_1}$ and $(u_1, \ldots, u_n)^{m_2}$ in P, the judgments $t_i = u_i$ follow for each relevant i. From these equality judgments continue to deduce further equality judgments in the same way, ending up with a finite set of nontrivial judgments "x_k has type v_k" which can be used to unify the two composite types just as in the *STT* algorithm.

If x_i is assigned types $(t_1, \ldots, t_n)^{m_1}$ and $(u_1, \ldots, u_n)^{m_2}$ in P, or if x_i is assigned types $()^{m_1}$ and $()^{m_2}$, the orders m_1 and m_2 should be the same. If m_1 has bounding variable x_j and m_2 has no bounding variable, we make the additional judgment "x_j has type $(u_1, \ldots, u_n)^{m_2}$ in P" and replace all occurrences of $|x_j|$ with m_2 (other occurrences of $[x_j]$ should already have been eliminated). We proceed symmetrically if m_2 has a bounding variable and m_1 has no bounding variable. If m_1 and m_2 have bounding variables x_j and x_k respectively, we make the additional judgments "x_j has type $(u_1, \ldots, u_n)^{m_2}$ in P" and "x_k has type $(t_1, \ldots, t_n)^{m_1}$ in P", then replace all occurrences of $|x_j|$ and $|x_k|$ (there should be no frank occurences of $[x_j]$ or $[x_k]$) in type judgments with $|x_{\max\{j,k\}}|$. Both of these maneuvers are justified by the bounding variable rule.

This process must terminate. Each step of the process described eliminates at least one variable type $[x_i]$ from consideration (along with any occurrences of $|x_i|$) or terminates with a judgment of ill-typedness.

When the process terminates, we will either have concluded that P is ill-typed (and this judgment will be honest because the rules are sound for the intended interpretation) or we will have type assignments to the

variables appearing in P almost satisfying the conditions for a type function: "almost" because the same variable may be assigned distinct ramified types corresponding to the same simple type but having typographically different orders. If each variable has been assigned a unique type by the end of the process, then the algorithm succeeds in defining a type function τ up to assignments of concrete type values to type variables, as above.

This algorithm is still based on the quite standard approach of type unification implemented, for example, in the type checking of the computer language *ML* (see [4]).

The algorithm above is sound but incomplete. If it yields a type, it will always be a correct type, but there are propositions and pfs which cannot be typed by this algorithm but which are typable in *RTT*. In practice, the algorithm is quite good; it is not easy to write a typable term of *RTT* which it will not type (though we shall present an example).

A complete algorithm requires true order unification. This will depart from the usual methods of type checking, because it will require reasoning about numerical inequalities.

It might seem that we would need new judgments "$m = n$ in P", where m, n are orders, but in fact the type judgment "$()^m = ()^n$ in P" is equivalent. We do allow ourselves the abbreviation "$m = n$ in P" for "$()^m = ()^n$ in P" where it is clear that orders are being discussed (we call type equality judgments of this form "order equality judgments"), but not in the statement of the following (obviously sound) additional rules:

componentwise equality of composite types (order): If $(t_1, \ldots, t_n)^{m_1} = (u_1, \ldots, u_n)^{m_2}$ in P, then $()^{m_1} = ()^{m_2}$ in P.

order substitution: If x_i has type t in P and m is the order of t, and $()^p = ()^q$ in P, then $()^{p[m/|x_i|]} = ()^{q[m/x_i]}$ in P.

We outline our basic approach to reasoning about order unification. An order equality judgment in standard form will take the form $\max\{n_0, n_1 + |x_{i_1}|, \ldots, n_k + |x_{i_k}|\} = \max\{m_0, m_1 + |x_{j_1}|, \ldots, m_l + |x_{j_l}|\}$. This is equivalent to a disjunction of conditions, each of which asserts the equality of one of the terms of the first maximum with one of the terms of the second maximum along with the inequalities asserting that the two chosen terms are greater than or equal to the other terms of the respective maxima from which they are taken. If one or both of the orders has a bounding variable, the bounding variable is the only possible maximum chosen (which simplifies the calculation in these cases by reducing the number of cases).

All of the resulting statements can be expressed using assertions of the form $|x_i| \geq n$, $|x_i| \leq n$, or $|x_i| - |x_j| \leq n$, where n is an integer. Any equation or inequality between terms of the forms n_0 or $n_k + |x_{i_k}|$ can be converted to a conjunction of inequalities of the forms above by substracting an appropriate quantity from each side of the equality or inequality and converting an equation to the conjunction of two inequalities in the obvious way. Any conjunct

of the form $|x_i| \leq r$ where $r < 0$ (which will also be obtained (e.g.) from an equation $|x_i| + m = |x_i| + n$ where $m \neq n$) can be used to conclude that an entire conjunction is false.

We now describe the computation of complete conditions for well-typedness of a term from a number of order equality judgments. Convert each order equality judgment to a disjunction of conjunctions of inequalities of the forms described above. A conjunction of disjunctions of conjunctions is converted to a disjunction of conjunctions in the obvious way.

Now each conjunction of inequalities is processed separately. Present all inequalities in a uniform way by rewriting $|x_i| \leq n$, $|x_i| \geq n$ as $|x_i| - 0 \leq n$, $0 - |x_i| \leq -n$, respectively. Every inequality is then written in the form $A - B \leq n$. For each x_i which appears, include $0 - |x_i| \leq 0$, $0 - 0 \leq 0$ and $|x_i| - |x_i| \leq 0$ in the conjunction. Wherever $A - B \leq n_1$ and $A - B \leq n_2$ both appear, retain just $A - B \leq \min\{n_1, n_2\}$. Wherever $A - B \leq m$ and $B - C \leq n$ both appear, add $A - C \leq m + n$ to the conjunction. Apply these operations repeatedly if necessary. If any conjunct of the form $|x_i| - 0 \leq r$ with $r < 0$ or $|x_i| - |x_i| \leq r$ with $r < 0$ appears, conclude that the conjunct is false. We claim that this procedure will produce a canonical complete conjunction equivalent to the conjunction we started with.

Lemma: Any conjunction of a set of inequalities of the form $A - B \leq n$, where A and B are either 0 or variables with natural number values, is converted to a canonical equivalent form by the procedure described above.

Proof of Lemma: The proof of the Lemma is omitted from this abridged version of the paper.

Conjunctions can then be simplified by eliminating redundant conjuncts (a conjunct is redundant if eliminating the conjunct then computing the canonical form gives the same result as computing the canonical form of the original conjunction): in practice this gives quite manageable displayed forms for conditions.

Once each disjunct is computed, identical disjuncts or conjunctions weaker than other disjuncts can be recognized and eliminated (by comparing canonical forms) and a simplified form of the disjunction of conditions under which the term is well-typed can be computed (or ill-typedness can be reported if all conjuncts reduce to falsehood).

This can be applied to produce a complete algorithm: use the provisional algorithm described above to generate a list of type assignments whose failures of uniqueness are induced only by failures to unify order, then apply the procedure described above to reduce the order equality judgments that are required to arithmetic assertions about polymorphic orders. Note that under the resulting conditions it is possible to select any of the types given for each variable or propositional function as correct, since all types given for any one object will be equal under the conditions derived from the unification of the orders.

The simplification of the arithmetic conditions on polymorphic orders made possible by the use of canonical forms for conjunctions combined with the elimination of redundant conjuncts and disjuncts is essential for manageable-sized output (earlier versions showed this) and gives good results.

The reasoning above was informal arithmetical reasoning. It is theoretically interesting to observe that it can be handled by an extension of our system of type judgments. This is not how the software does it, and we do not discuss the details in this abridged version of the paper.

Here we omit a section in which comparisons between the system of this paper and the system of [3] is found, except for the comment in the following paragraph. The other points listed in the section found here in the unabridged paper are made (perhaps briefly) elsewhere in the paper.

The range of terms recognized as well-typed by our system is far larger than that recognized by the system of [3], and apparently larger than that recognized by *PM*! The system of [3] only supports types all of whose component types are "predicative". Probably the modifications of the system required to lift this restriction would not be extensive. On reading [3] originally, we thought this was a weakness of their development, but in fact it seems to reflect the intentions of the authors of *PM*: see p. 165. However, we think that more complex impredicative pfs would be needed for work in *PM* without the axiom of reducibility (and if one assumes this axiom one might as well work in *STT*).

We are working in *RTT* in all examples, but the software does not display order superscripts on types when the order is the smallest possible. Some features of the output of our software are suppressed.

```
Term input:
     S2(a1,a2)
final type list:
unconditional type:
     ()
```

Just as in example 49, clause 1, of [3], the propositional notation $S(a_1, a_2)$ is recognized as a proposition because it contains no free variables.

```
Term input:
     (R1(x1) v S1(x1))
final type list:
     x1:  0
unconditional type:
     (0)
```

This is parallel to the second example in clause 2 in example 49 of [3].

```
Term input:
     (R1(x1) v S1(x2))
final type list:
```

```
     x1:  0
     x2:  0
unconditional type:
     (0,0)
```

This term $R_1(x_1) \vee R_2(x_2)$ would be treated quite differently from the term above in the system of [3], whereas the treatment of both propositional functions in the system of this paper is very similar. In both terms, our checker first generates the list of free variables, then each free variables is typed using local rules, and the types of the free variables are listed to form the type of the pf.

The system of [3] uses a different (and more usual) kind of context than our system. The form of a type judgment of the system of [3] is $\Gamma \models f : t$, where f is a term, t is the type assigned to that term, and Γ, the "context", is a list of assignments of types to variables. In our system, a type judgment about an entire term (propositional notation) has no context, while type judgments about variables have as context the term in which they appear.

In the system of [3], the term $R_1(x_1) \vee R_2(x_1)$ is typed by first considering the typing of $R_1(a_1) \vee R_2(a_1)$, which is immediately seen to have type (), and in which the term a_1 has type 0, then using the rule for typing substitutions to insert new component with type 0 into the type () of $R_1(a_1) \vee R_2(a_1)$ to obtain the type (0). The term $R_1(x_1) \vee R_2(x_2)$ is typed by observing that the two disjuncts have the property that all variables of the first are alphabetically prior to the variables of the second, typing the first and the second as (0) in the same way we typed the previous term, then concluding that the type of the whole is the "product" $(0, 0)$ of two copies of (0) (speaking somewhat loosely). This might give some idea of the very different flavor of the two approaches.

```
Term input:
     [x1](x1!() v ~x1!())
final type list:
     x1:  ()
unconditional type:
     ()^1
```

This is example 51 from [3]. Order is important in this example. Note that the variable x_1 represents a proposition (a 0-ary propositional function); the order of its type is 0. The entire term is also a proposition (it contains no free variables, because x_1 is bound by the quantifier) but its order is at least 1, because it must be greater than the order of the quantified variable. It is precisely 1 because we used "predicative" pf application. The order 0 of the type of x_1 is not displayed because it is as small as possible.

We can see an explicit polymorphic type by implementing the term in Remark 58 of [3], stipulating that the application is predicative.

```
Term input:
     x2!(x1)
```

```
final type list:
     x1:  [x1]
     x2:  ([x1])
unconditional type:
     ([x1],([x1]))
```

In this term, x_1 is of a completely unknown type $[x_1]$, while x_2 is seen to be of type $([x_1])$ (it is a predicate of objects of type $[x_1]$), so the whole term is of type $([x_1],([x_1]))$: the order of the components is determined by the fact that x_1 is alphabetically prior to x_2.

In [3], two different derivations are given, showing how two different types can be assigned to this pf, whereas here we get a single computation yielding *all* types. If we get more information from the context, the type will become more specific:

```
Term input:
     (x2!(x1) v S1(x1))
final type list:
     x1:  0
     x2:  (0)
unconditional type:
     (0,(0))
```

Here we know from local information elsewhere in the term that the type of x_1 is 0, so we get a more specific type for the whole pf.

We now give a large example. There are two different conditions under which the given pf is well-typed.

```
Term input:
     (x1!(x2,x2) v x1!([x3][x5]x3!(x5,x8),[x6][x9]x6!(x4,x9)))
unconditional type:
     ?!?
conditional type:
     ((([x8])^max(|x5|+2,
     |x8|+2,2),([x8])^max(|x8|+2,|x9|+2,2)),
     ([x8])^max(|x5|+2,|x8|+2,2))
  WITH
     |x5| <= |x9| and
     |x8| <= |x9| and
     |x9| <= |x5|
  OR
     |x5| <= |x8| and
     |x9| <= |x8|
```

In more standard notation, the propositional function is

$$x_1!(x_2,x_2) \vee x_1!((\forall x_3.(\forall x_5.x_3(x_5,x_8))),(\forall x_6.(\forall x_9.(x_6!(x_4,x_9)))))$$

The entire term is a propositional function of the arguments x_1 and x_2; it is necessary to figure out what the types of x_1 and x_2 are. Because of the presence of the subterm $x_1!(x_2, x_2)$, we know that the two arguments of any occurrence of x_1 must be of the same type. So the propositional functions $(\forall x_3.(\forall x_5.x_3(x_5, x_8)))$ and $(\forall x_6.(\forall x_9.(x_6!(x_4, x_9)))$ are of the same type. Each of these is a function of one variable, x_8 in one case and x_4 in the other, so x_4 and x_8 are of the same type. This base type is polymorphic: we know nothing about it.

Now we need to analyze orders. The order of the type of $(\forall x_3.(\forall x_5.x_3(x_5, x_8)))$ is two greater than the maximum of the orders of $[x_5]$ and $[x_8]$. The increment of two is because x_3 has type one greater than this maximum, and the order is raised one more because of the quantifier over the type of x_3. Similarly, the order of the type of $(\forall x_6.(\forall x_9.(x_6!(x_4, x_9)))$ is two greater than the maximum of the order of $[x_4] = [x_8]$ and the order of $[x_9]$. These two orders have to be the same. There are two ways for this to happen: either the order of $[x_5]$ is greater than the order of $[x_8]$, in which case the order of $[x_9]$ also has to be greater than the order of $[x_8]$ and actually must be the same as the order of $[x_5]$, or the order of $[x_8]$ is greater than or equal to the orders of $[x_5]$ and $[x_9]$ (which in this case need not be the same). And these two cases are what the output above describes.

The type of x_1 will be $([x_2], [x_2])$; the type of x_2 will be (x_8). So the underlying simple type of this expression is $(((([x_8]), ([x_8])), ([x_8]))$, and this is what we see above, adorned with appropriate orders.

We omit a section on applications to proof-checking for *PM* which will appear in the unabridged paper.

References

[1] Holmes, M. Randall, "Subsystems of Quine's "New Foundations" with Predicativity Restrictions", *Notre Dame Journal of Formal Logic*, vol. 40, no. 2 (spring 1999), pp. 183-196.

[2] Holmes, M. Randall, software files (in standard ML) `rtt.sml` (source for the type checker) and `rttdemo.sml` (demonstration file), accessible at `http://math.boisestate.edu/~holmes/holmes/rttcover.html`.

[3] Kamareddine, F., Nederpelt, T., and Laan, R., "Types in mathematics and logic before 1940", *Bulletin of Symbolic Logic*, vol. 8, no. 2, June 2002.

[4] Milner, R., "A theory of type polymorphism in programming", *J. Comp. Sys. Sci.*, 17 (1978), pp. 348-375.

[5] Whitehead, Alfred N. and Russell, Bertrand, *Principia Mathematica (to *56)*, Cambridge University Press, 1967.