THE PROBLEM OF BUREAUCRACY AND IDENTITY OF PROOFS FROM THE PERSPECTIVE OF DEEP INFERENCE

Alessio Guglielmi (TU Dresden and University of Bath)
17.6.2005

Abstract

Deep inference offers possibilities for getting rid of much bureaucracy in deductive systems, and, correspondingly, to come up with interesting notions of proof identity. We face now the problem of *designing* formalisms which are intrinsically bureaucracy-free. Since we have a design problem, it is important to elaborate definitions that will remain useful for many years to come. I propose a discussion of several proposals. The discussion will hopefully be also a good way of introducing deep inference to those who don't know it.

In my talk I will explain in detail and with examples all the notions quickly sketched below. It is apparently extremely simple stuff, but there are subtle issues that only experienced proof theorists might appreciate; I will try to address them. The proposed solutions are currently discussed on the mailing list Frogs. By the time of the workshop, in addition to my proposed definitions, I will have also the opinions of the participants to the discussions.

Bureaucracy and Identity

Bureaucracy and identity of proofs are intimately related.

There is no formal notion of bureaucracy, but I guess the consensus is that, when two proofs are *morally the same*, but they differ in *inessential details*, then this is due to *bureaucracy*.

If this is so, we should conclude that eliminating bureaucracy should lead us to eliminate the inessential details that blur the `sameness´, i.e., identity, of proofs.

We should agree that, for any given logic, there are several possible notions of identity of proofs, and people can invent more and more of them.

Given a notion of identity and a formalism, either the formalism is able to express the identical proofs or it isn't: in the latter case, we have bureaucracy, and we have an enemy.

Our goal is to attack some specific, important kinds of bureaucracy, in order to improve the ability of proof theory to deal with bureaucracy. It is hopeless to try and define bureaucracy once and for all. However, it is now possible to define formalisms which get rid of the most brutal and medieval forms of bureaucracy.

Bureaucracy in the Formalism and in the Deductive System

I will use in the following the syntax of the calculus of structures (CoS) [WS].

There are several sources of bureaucracy, and I think it is convenient to address them separately. It should be possible to make a broad distinction between bureaucracy induced by the formalism and bureaucracy induced by the specific deductive system used (in the given formalism).

What I call formalism A [1] takes care of bureaucracy of the kind

```
    [R' T']           [R' T']
  r'-------         r -------
    [R' T ]           [R  T']
  r -------   vs.   r'------- ,
    [R  T ]           [R  T ]
```

where the order of the application of two inference rules doesn't morally matter. In formalism A, one can write

```
     R'    T'
  [r--- r'---]
     R     T
```

and the problem is solved. This is an example of formalism-related bureaucracy: CoS only sees the two derivations above, and doesn't express the one below.

However, consider a deductive system where associativity is explicit (I mean, we are not working modulo associativity). Consider the following two derivations:

```
      [[a  a] a]
  ass -----------
      [a [a  a]]             [[a a] a]
   ac_ ----------         ac_ ---------
      [a  a]                    [a  a]
   ac_ ------       vs.      ac_ ------ .
       a                          a
```

They might be considered `morally the same´, but it is difficult to fix the problem in the formalism definition. Perhaps, a better idea is to fix the deductive system. For example, one can propose a deductive system with sort of a `general atomic contraction´, quotient by associativity, and go for the derivation

```
      [a a a]
  gac_ ------- .
         a
```

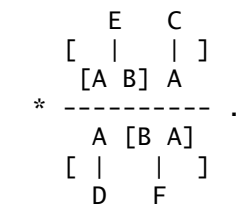
So, this could be an example of fixing the bureaucracy problems by fixing the deductive system (inside a given formalism).

A Problem with Commutativity and Associativity

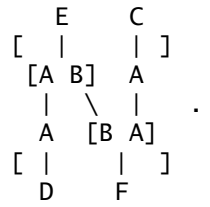
In my opinion, the *very first* source of bureaucracy in *all deductive systems* in *all formalisms* is associativity and commutativity (when present) in formulae. I mean, in most cases, we do not want to distinguish formulae, and so proofs, just because of the order of associations, right?

The only practical way of dealing with commutativity is working under an equivalence relation that takes care of it. Associativity offers some more options. Anyway, working under associativity and commutativity, in a deductive system, is difficult. Actually, it is also dangerous.

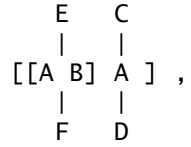
Consider



This is a derivation (in formalism B [2]) in which two derivations are vertically composed by *, and we work under commutativity and associativity. The problem is that this is the only way we have in formalism B for representing (what I could graphically and imprecisely represent as)



However, the same derivation above could also stand for



and this of course is *morally different*!

What can we do? Well, we could stop working under commutativity and associativity: this way we could easily distinguish between the two

cases. However, if we drop commutativity and associativity, we get back all the bureaucracy in formulae, with a vengeance, because now this bureaucracy scales up to proof composition.

Possible Solutions

Apart from developing the ideas in [2], there is now the possibility of designing a geometric formalism that solves the problems mentioned above, which I called *wired deduction*. I posted its possible definition(s) to the mailing list Frogs, and this generated a discussion [3]. For convenience, I reproduce in the appendix the email with the definition, but there is no room for reporting all the issues discussed on Frogs.

It is too early to tell whether this is the long-term solution we are looking for, however, the new formalism certainly works well for classical logic, and this is what I'd like to show at the workshop, since the ideas of wired deduction are all clearly exposed also in the case of classical logic.

References

[1] Alessio Guglielmi. Formalism A. URL:
http://iccl.tu-dresden.de/~guglielm/p/AG11.pdf.

[2] Alessio Guglielmi. Formalism B. URL:
http://iccl.tu-dresden.de/~guglielm/p/AG13.pdf.

[3] Alessio Guglielmi, Stéphane Lengrand and Lutz Straßburger. Emails at URLs:
http://thread.gmane.org/gmane.science.mathematics.frogs/219,
http://thread.gmane.org/gmane.science.mathematics.frogs/220.

Web Site

[WS] http://alessio.guglielmi.name/res/cos.

Appendix

```
Delivered-To:Frogs
Date: Tue, 15 Mar 2005 17:32:09 +0100
To: Frogs,                          Michel Parigot
From: Alessio Guglielmi
Subject: [Frogs] Wires and pipes
Cc: Dominic Hughes
List-Post: Frogs
List-Page: <http://frogs.prooftheory.org>

Hello,

in this message I propose a formalism and a deductive system for classical
propositional logic.

The formalism, which I'd like to call `wired deduction' (weird deduction!) should
be the first example of deductive derivation net: it is an intrinsically
```

bureaucracy-free, deductive and geometric formalism. It naturally subsumes CoS and formalisms A and B.

As usual, I will define the formalism by way of a deductive system, and the natural choice is classical propositional logic. It is possible to define the formalism in isolation, in two ways: 1) geometrically, as a set of graph-forming rules; 2) deductively, by the definition for formalism B I showed at the workshop, *enriched by wires* (see below for what wires are).

These definitions are almost trivial after you see a deductive system. Before posting their details, I would like to receive some reactions about the deductive system, which you find below.

This email has three parts: Motivations, Intuition and Technicalities. Reading up to Intuition should be enough to get a good idea, if you know already about CoS and KS.

I would be very grateful if somebody checks the technicalities, though. They are nontrivial and unfortunately very combinatorial. Clearly, they might be wrong, but it should be possible to fix any mistake without changing the general picture.

Please, if you check, let me know, even if you find no mistakes (positive information is still very useful!).

Ciao,

-Alessio

MOTIVATION
==========

The general motivation is devising a formalism which is bureaucracy-free and *intrinsically* so. Moreover, we want the formalism to be geometric.

Bureaucracy-free means that it should be possible to express, inside the formalism, canonical representatives of derivations which are `morally the same' according to some notion. See the message

<http://article.gmane.org/gmane.science.mathematics.frogs/219>

for an exposition of these ideas. See also the notes

<http://iccl.tu-dresden.de/~guglielm/p/AG11.pdf> ,
<http://iccl.tu-dresden.de/~guglielm/p/AG13.pdf> ,
<http://www.iam.unibe.ch/~kai/Current/prty.pdf> .

`Intrinsically' bureaucracy-free means that the formalism disallows the very formation of some redundant derivations. This notion is closely related, somehow, to the idea that the formalism should be geometric.

For a formalism, being `geometric' means that derivations are some sort of graphs over which one operates locally and modulo some basic symmetries like those due to commutativity and associativity.

Much of the inspiration for wired deduction comes from subatomic logic, especially the idea of wires and the ww_ rule. See

<http://iccl.tu-dresden.de/~guglielm/p/AG8.pdf> .

In the design of the deductive system for classical logic, I wanted to get rid once and for all of the unit equations. They have no strong justification in terms of `war to bureaucracy' and they cause some technical problems. I think that the system below is particularly convincing in this respect. It might be possible to do better than I did, in the sense that some technicalities can perhaps be simplified.

INTUITION

I will attempt here a completely informal exposition of the system for classical logic, called KSw, which should be sufficient for people who know CoS and KS. The technical definitions are in the Technicalities part of this message (they are still subject to changes, of course).

We need to operate under associativity and commutativity, in order to get rid of bureaucracy in formulae. However, in some cases we need to keep track of `where the atoms come from'. The obvious solution would be to disambiguate these situations by resorting to occurrences. Wires go one step further: they allow following atoms and their transformations throughout a whole derivation.

The main idea goes as follows: there is a denumerable set of wires. Wires are neither created nor destroyed. To wires we associate atoms, and the association may vary in the course of the derivation.

Moreover, at any given time bunches of wires are organised into a tree of logical relations, which also can change over time. For example, the following is a derivation of a from (a V a) (so, it's a contraction):

```
[a a]
 | |
[a f] .

 1 2
```

There are two wires, 1 and 2, vertically disposed, and we assume that time flows vertically going upwards. In the beginning, a is associated to 1 and f is associated to 2. In traditional logic, a would be a propositional variable and f would be the `false' unit; both are atoms for us. Wires 1 and 2 are in a disjunction relation (indicated as usual in CoS). After some time wire 2 gets the value a, but the logical relation between wires does not change. We indicate this situation by the rule

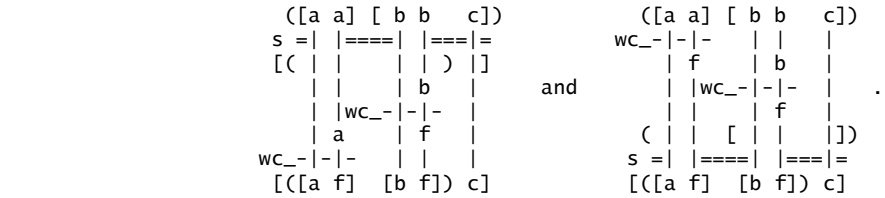
```
     [a a]
wc_ -|-|- ,
     [a f]
```

which is of course supposed to apply in the middle of other wires. This is an example of atomic rule. A special feature of wired deduction is that atomic rules only work on wires, their values and their relations, by `going through' a predetermined amount of them (in the case above, two). Atomic rules `see' which atoms wires carry.

There is another kind of rule, the local rule. These are different than atomic rules, they only see *bunches* of wires, called *pipes*, and they reshuffle their logical relations. For example, take the switch rule

```
    (A [B  C])
s  =|==|==|=  .
  [(A  B) C]
```

Through pipes, logical inferences can go up and down, provided they don't stumble one on another. For example, consider the following derivations:



As you can see, the two wc_ rules can freely go up and down and pass through the holes in the s rule. This example shows how wired deduction deals with both type A and type B bureaucracy.

This also shows that employing this geometric criterion (sort of an elastic deformation of graphs) avoids the problem of representing non-canonical derivations: all derivations are canonical, and convergence of the `rewriting system' is trivial.

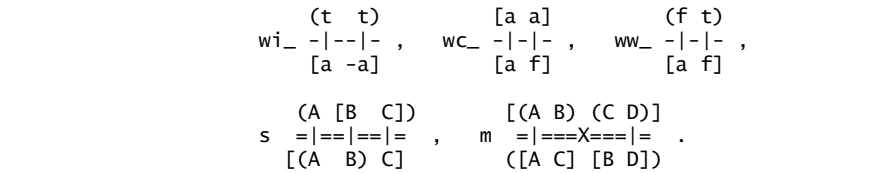
So, what are derivations? Derivations are nets of the kind seen above, whose general shape is

In other words, the premiss R is in the middle of any number of t wires in conjunction and the conclusion T is in the middle of any number of f wires in a disjunction. *No wires are created or destroyed*: in this sense, this formalism is always *linear*. A proof, of course, is a derivation with all t's in the premiss.

This setup clearly works for classical logic, and all our results with CoS, and my preliminary work with subatomic proof theory, tell us that this should work for *any* logic. Of course, in general f and t are simply the units of whatever disjunction and conjunction one has, for example they would be bottom and one for linear logic.

The general geometry is given by the wires: we *always* assume that they live under a commutative and associative equivalence. In the case of non-commutative logic, the non-commutativity will be represented by the logical relation between wires, not by their geometry. In other words: if you take the horizontal section of any derivation like the ones above, you always have a relation web (back to the origins).

So, this is propositional classical logic's system KSw; notice that there are *no equations*:



Of course, I have to show that this system is complete for classical logic (soundness is trivial). I could do it semantically, for example by showing how to

get disjunctive normal forms and then realising resolution and appeal to its completeness.

However, I also want to check that the complexity of proofs does not grow wrt KS, which is the place where we mostly study it. So, in the technicalities, you will find a complete proof of the admissibility of KS equations for KSw.

What is the secret of success? Part of the reason is in the fact that we can assume to have an unlimited supply of t's in conjunction and f's in disjunction. These atoms can be brought wherever they are needed by the switch rule. Doing this way does generate a small amount of bureaucracy of the deductive-system kind, for contraction and weakening rules. However, it is very easy to get rid of this bureaucracy by simple permutations. This is not very geometrical, but it is more geometrical than basically allowing [R f] = R and (R t) = t everywhere, so I went that way. In any case, there always is bureaucracy associated to the piling up of contraction and weakenings, as I showed in the previous message to Frogs, and this can be dealt with by using an appropriate deductive system with non-local rules (or by using a straightforward equivalence on proofs).

If you have more or less clear what I tried to explain above, you can jump directly to section 3 of the Technicalities and see KSw in action while getting rid of KS equations.

TECHNICALITIES
==============

1 LANGUAGE

Definition We define the following:

- WW is a denumerable set of _wires_; we denote wires by natural numbers.

- PP is a set of _pipes_; we denote pipes by A, B, C, D and various decorations.

- SSF is the language of _scheme skeleton formulae_, produced by

      SSF ::= WW | PP | [SSF SSF] | (SSF SSF)

  and such that no wire and no pipe appears twice in any element of SSF; we denote scheme skeleton formulae by K; an _instance_ of a scheme skeleton formula K is a scheme skeleton formula obtained by replacing in K any pipes by scheme skeleton formulae.

Example K = [(1 2) A] is a scheme skeleton formula, while [(1 1) A] is not. K' = [(1 2) (A B)] is an instance of K, while [(1 2) (A A)] is not. [(1 2) ([3 4] (5 6))] is an instance of K'.

Definition AA is a denumerable set of _atoms_; we denote atoms by a, b and c; on atoms we have an involution -: AA -> AA (i.e., --a = a); two special atoms f and t, called _units_, belong to AA, and -f = t. A _scheme formula_ is a couple (K, wr K -> AA), where wr K is the set of wires appearing in K; if no pipes appear in K, then the scheme formula is a _formula_; formulae are denoted by F.

Example If K = [(1 2) A] then (K, {1 -> a, 2 -> t}) is a scheme formula; if

K' = [(1 2) 3] then (K', {1 -> a, 2 -> t, 3 -> a}) is a formula, corresponding to the classical propositional logic formula ((a ^ t) V a).

Definition The equivalence == on SSF is defined as the minimal equivalence relation such that

```
        [K K'] == [K' K] ,
        (K K') == (K' K) ,
   [K [K' K"]] == [[K K'] K"] ,
   (K (K' K")) == ((K K') K") ,
          if K == K' then [K K"] == [K' K"] and  (K K") == (K' K") .
```

The equivalence == is applied naturally wherever scheme skeleton formulae appear. _Structures_, denoted by P, Q, R, T, U and V, are formulae modulo ==.

Examples and Notation We usually omit indicating wires, and we write, for example, [(a t) a] in the place of ([(1 2) 3], {1 -> a, 2 -> t, 3 -> a}). We have that [(a b) [f a]] == [a [f (b a)]]. We drop unnecessary parentheses, so [a [f (b a)]] can be written as [a f (b a)].

Definition Two structures R and T are _isomorphic_ if in their respective ==-equivalence classes there are two formulae which are equal modulo some permutation of wires.

Example Let

```
   R = ([([1 2] 3)]_==, {1 -> a, 2 -> b, 3 -> c}) ,
   T = ([(4 [5 6])]_==, {4 -> c, 5 -> a, 6 -> b}) .
```

Clearly,

```
   T = ([([5 6] 4)]_==, {5 -> a, 6 -> b, 4 -> c}) ;
```

we can consider the permutation {1 <-> 5, 2 <-> 6, 3 <-> 4}, and this shows that R and T are isomorphic.

Notation We usually do not indicate pipes, rather we use structure notation. So, for example, ([A B] C) is indicated as ([R T] U). This allows for an important shortcut: when we repeat letters, like in (R R), we mean any structure

```
   ([(K K')]_==, m), where m: wr (K + K') -> AA,
```

such that ([K]_==, m') and ([K']_==, m") are isomorphic, where m' and m" are the restrictions of m to wr K and wr K', respectively.

Example ([R R] a) can be instantiated as ([(T T U) (T T U)] a) and ([(b f) (b f)] a), for example, but not as ([b c] a). ([R T] a) instead does not impose any restriction on R and T.

Definition An _atomic inference rule_ is any expression of the kind

```
      F
   r ---,
      F'
```

where F and F' are formulae such that the same wires appear in F and F'; r is the _name_ of the rule. We adopt a notation such that wires are not explicitly indicated, but they can be `followed', for example

```
       ((1 2), {1 -> f, 2 -> t})
   ww_ ------------------------
       ([1 2], {1 -> a, 2 -> f})
```

is denoted by

```
       (f t)            (t f)
   ww_ -|-|-   or   ww_  -X-  .
       [a f]            [a f]
```

Definition A _local inference rule_ is any expression of the kind

```
      K
   r --- ,
      K'
```

where K and K' are scheme skeleton formulae where no wires appear and such that the same pipes appear in both; r is the _name_ of the inference. We adopt a notation where we join vertically the pipes; for example

```
     (A [B C])
   s ---------
     [(A B) C]
```

is denoted by

```
      (A [B  C])            (A [C B])
   s  =|==|==|=    or    s  =|===X=   .
     [(A  B) C]            [(A  B) C]
```

Example System KSw for classical propositional logic is defined by the following rules

```
       (t  t)         [a a]         (f t)
   wi_ -|--|- ,   wc_ -|-|- ,   ww_ -|-|- ,   for all a in AA,
       [a -a]         [a f]         [a f]

      (A [B  C])       [(A B) (C D)]
   s  =|==|==|=  ,   m  =|===X===|=  .
     [(A  B) C]        ([A C] [B D])
```

The first three rule (schemes) are atomic, the last two are local. They are called, respectively _wired interaction_, _wired contraction_, _wired weakening_, _switch_ and _medial_.

2 COMPOSITION OF RULES

This part needs to be completed. For now, suffice to say that we compose rules like in the calculus of structures. Of course, it is possible to define more geometric notions of compositions, like for formalism B.

3 CLASSICAL PROPOSITIONAL LOGIC

Proposition The _contraction_ rule

```
      [P P]
   c_ ~|~^~ ,
       P
```

is derivable for KSw.

Proof By structural induction on P. If P = a then consider

```
        S[a  a]
   wc_  |-|--|-
        |[a  f]
    s* =|=|==|=  .
       [S{a} f ]
```

If P = [R T] then consider

```
      [R R T T]
   c_  | |~|~^~
      [R R |  ]
   c_ ~|~^~|    .
      [R   T  ]
```

If P = (R T) then consider

```
      [(R T) (R T)]
    m  =|===X===|=
      ([| R] [T T])
   c_   | |  ~|~^~
      ([R R]  |   )
   c_  ~|~^~  |
      ( R     T   )
```

<>

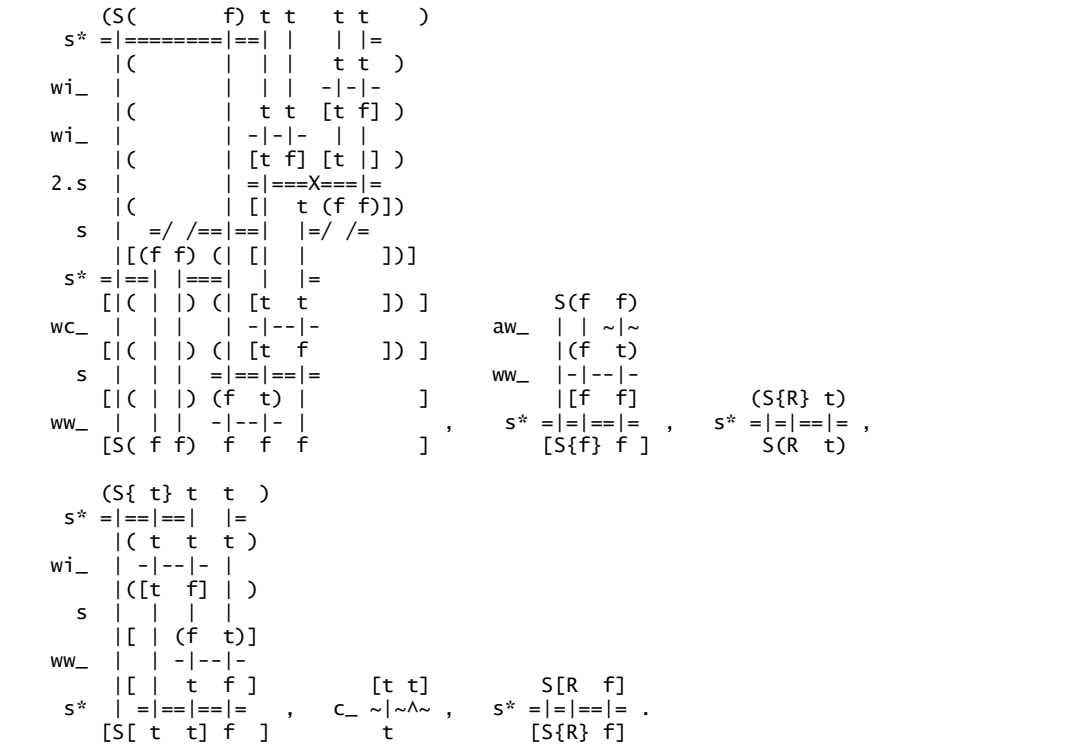
Proposition The following rules

```
       f
   aw_ ~|~   (_atomic weakening_),
       a

      [a a]
   ac_ ~|~^~   (_atomic contraction_),
       a

       f          (f f)          R
   r1 ~!^!~ ,   r2 ~|~^~ ,   r3 ~|~!~ ,
      (f f)         f           (R t)

      t           [t t]        [R f]
   r5 ~|~!~ ,   r6 ~|~^~ ,   r8 ~|~^~
      [t t]         t            R
```

are derivable for KSw.

Proof Consider, respectively:

```
      (S{f} t)
    s* =|=|==|=
       |(f  t)
   ww_  |-|--|-
       |[a  f]
    s* =|=|==|= ,
       [S{a} f]

      [a a]
   c_ ~|~^~ ,
       a
```

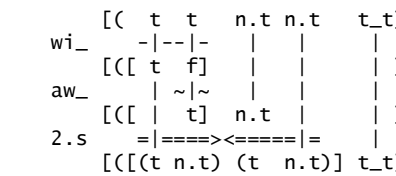


<>

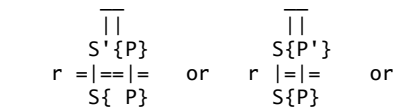
Theorem <PP> In KSw, if S{P} is provable then S[P P] is provable.

Proof Induction on the length of the proof D of S{P}.

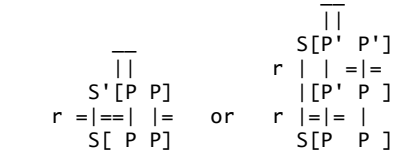
Base Case: If D = [(t t_t) f_f], we have to show that
[([(t n.t) (t n.t)] t_t) f_f] is provable, for n>= 0. Take



Inductive Cases: If the bottommost rule instance in the proof of S{P} is like
in



then use the induction hypothesis on



Otherwise, the following cases are possible:

```
                                                  __
                                                  ||
                                              S'[Q (t  t)]
1  S{ } = S'[a { }], P = [-a Q] and D = wi_ |  | -|--|- : Consider
                                              S'[Q  a -a ]
```

```
                  __
                  ||
       S'[Q Q (t t) (t  t)]
   wi_  |  | |  | |  -|--|-
        | [| | (t t)  a -a ]
   wi_  |  | | -|-|-  |  |
        | [| | -a a   a  | ]
   wc_  |  | |  |-|---|- |
        | [| |  | a   f -a ]
    s* =|==| |  | |    =X=    ;
       [S'[Q Q -a a  -a] f  ]
```

when Q is empty the argument is the same.

```
                                                  __
                                                  ||
                                              S'[Q a a]
2  S{ } = S'[{ } f], P = [Q a] and D = wc_ |  |-|-|-: Consider
                                              S'[Q a f]
```

```
              __
              ||
       S'[Q Q a a a a]
   wc_ |  | | |-|-|-|
       | [| | | f a a]
    c_ |  | | | |~|~^~ ;
       S'[Q Q a f a ]
```

when Q is empty the argument is the same.

```
                                                  __
                                                  ||
                                              S'[Q a a]
3  S{ } = S'[a { }], P = [f Q] and D = wc_ |  |-|-|-: Consider
                                              S'[Q a f]
```

```
              __
              ||
       S'[Q Q a a a a]
   wc_ |  | | | |-|-|-
       | [| | a a a f]
   wc_ |  | |-|-|-| |
       | [| | f a a |]
    c_ |  | | |~|~^~|  ;
       S'[Q Q f a   f]
```

when Q is empty the argument is the same.

```
                                                  __
                                                  ||
                                              S'[Q (f t)]
4  S{ } = S'[{ } f], P = [Q a] and D = ww_ |  | -|-|- : Consider
                                              S'[Q  a f ]
```

```
                __
                ||
       S'[Q Q (f t) (t f)]
   ww_ |  | |  | |  -|-|-
       | [| | (f t)  f a ]
   ww_ |  | | -|-|-  | |
       | [| |  a f   | | ]
    c_ |  | |  |~|~~~^~|   ;
       S'[Q Q  a f     a ]
```

when Q is empty the argument is the same.

```
                                                  __
                                                  ||
                                              S'[Q (f t)]
5  S{ } = S'[a { }], P = [f Q] and D = ww_ |   | -|-|- : Consider
                                              S'[Q  a f ]
```

```
              __
              ||
      S'[Q Q (t f) (f t)]
ww_ |   |  |  |  -|-|-
    | [| | (t f)  a f ]
ww_ |   |  | -|-|-  | |
    | [| |  f a   a | ]
 c_ |   |  |  |~|~~~^~|   ;
    S'[Q Q  f a     f ]
```

when Q is empty the argument is the same.

```
                                                      __
                                                      ||
                                                  S'[Q (R [T  U U'])]
6  S{ } = S'[{ } U'], P = [(R T) U Q] and D = s |   | =|==|==| |=    :
   Consider                                       S'[Q (R  T) U U'  ]
```

```
                      __
                      ||
     S'[Q Q (R [T  U U']) ([U' U  T] R)]
   s |   |  |   |  | |      =|   |==|==|=
     | [| | (| [|   | | ])    |   | (|   |)]
   s |   | | =|==|==| |=      |   |  |   |
     | [| | (|   |) | U'      U' | (|   |)]
   c_ |  | |  |   |  |~^~~~~~~|~ |  |   |   ,
     S'[Q Q (R  T) U          U' U (T  R)]
```

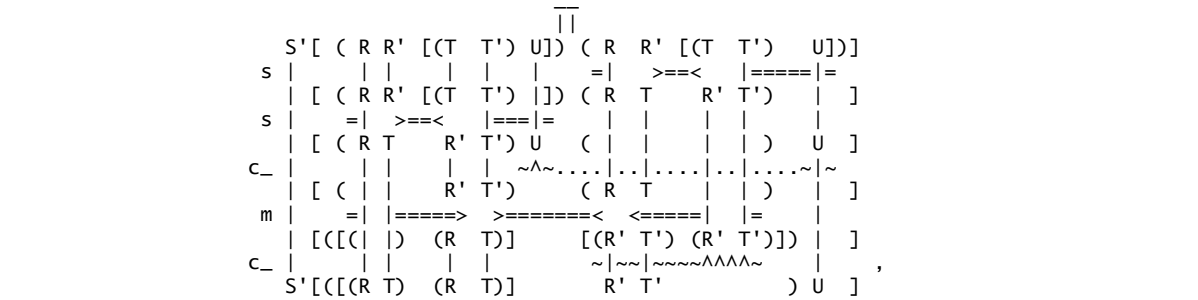
when Q or U are empty the argument is the same.

```
                                                      __
                                                      ||
                                                  S'[Q (R [T  U U'])]
7  S{ } = S'[(R T) { } U'], P = [U Q] and D = s |   | =|==|==| |=    :
   Consider                                       S'[Q (R  T) U U'  ]
```

```
                      __
                      ||
     S'[Q Q (R [T  U U']) ([U' U  T] R)]
   s |   |  |   |  | |      =|   |==|==|=
     | [| | (| [|   | | ])    |   | (|   |)]
   s |   | | =|==|==| |=      |   |  |   |
     | [| | (|   |) | U'      U' | (|   |)]
   c_ |  | |  |   |  |~^~~~~~~|~ |  |   |
     S'[Q Q (R  T) U          U' U (T  R)]
   c_ |  | | ~|~~|~.|........|..|.~^^^^~ ,
     S'[Q Q (R  T) U          U' U       ]
```

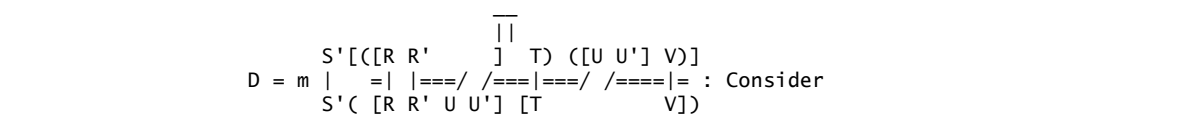
when Q or U are empty the argument is the same.

```
                                                      __
                                                      ||
                                                  S'( R R' [(T T') U])
8  S{ } = S'[({ } R' T') U], P = (R T) and D = s |   =| |====| |===|= :
   Consider                                       S'[(R R'   T T') U]
```



when R or R' or T or T' are empty, but (R R'), (T T'), (R T) and (R' T') are not empty, the argument is the same. The only case remaining to consider is when (R' T') is empty, i.e., S{ } = S'[{ } U], P = (R T), but this reduces to case 6.

9 S{ } = S'([{ } R' U'] [T V]), P = [R U] and



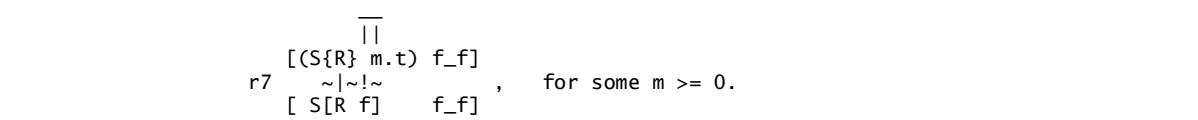
when R or R' or U or U' are empty, but [R R'] and [U U'] are not empty, the argument is the same.

<>

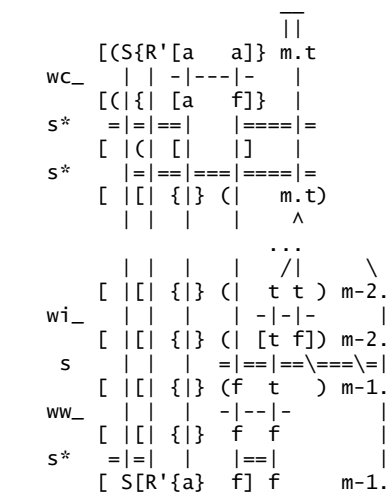
Proposition <R7> The rule

is admissible for system KSw.

Proof Consider the topmost instance of r7 in a proof in KSw + {r7}:



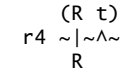
Let R = R'{a}, for some a; by Theorem <PP> there is a proof



Proceed eliminating r7 instances one by one.

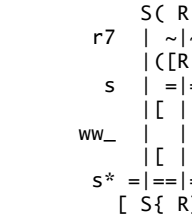
<>

Proposition The rule



is admissible for KSw.

Proof Consider



If S(R t) is provable in KSw, by Proposition <R7> then S{R} is also provable.

<>

Theorem System KSw is equivalent to system KS.

Proof All rules and equations of KS have admissible counterparts in KSw.

<>

TECHNISCHE UNIVERSITÄT DRESDEN

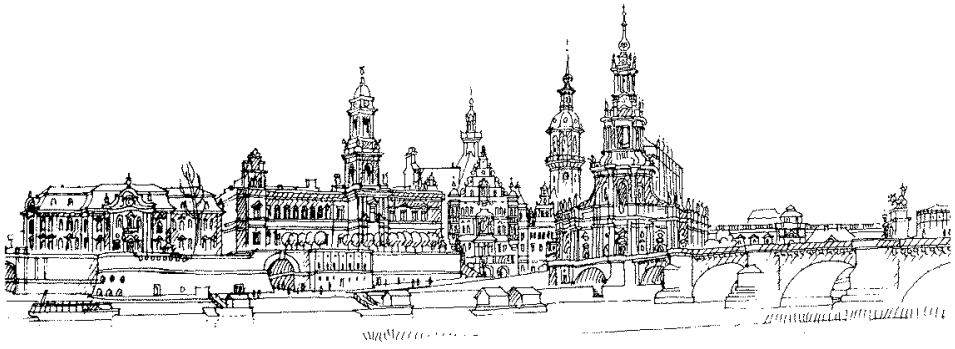
Fakultät Informatik

Technische Berichte
Technical Reports
ISSN 1430-211X

FI05-08-Juli 2005

Paola Bruscoli, François Lamarche
and Charles Stewart (Eds.)

**Structures and Deduction –
the Quest for the Essence of Proofs
(satellite workshop of ICALP 2005)**



Technische Universität Dresden
Fakultät Informatik
D−01062 Dresden
Germany
URL: http://www.inf.tu−dresden.de/