Deduction and computation

Gérard Huet

▶ **To cite this version:**

Gérard Huet. Deduction and computation. RR-0513, INRIA. 1986. inria-00076041

HAL Id: inria-00076041
https://inria.hal.science/inria-00076041

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA
CENTRE DE ROCQUENCOURT

Rapports de Recherche

N° 513

DEDUCTION AND COMPUTATION

Gérard HUET

Avril 1986

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France

Tél. : (1) 39 63 55 11

Deduction and Computation

Gérard Huet

Résumé

Nous présentons dans un cadre commun les concepts syntaxiques essentiels à une théorie unifiée des preuves et des calculs.

Abstract

We present in a unified framework the basic syntactic notions of deduction and computation.

PAPIER RÉCUPÉRÉ ET RECYCLÉ

Deduction and Computation

Gérard Huet

We present in a unified framework the basic syntactic notions of deduction and computation.

1 Terms and types

1.1 General notations

We assume known elementary set theory and algebra. $\mathcal{N}$ is the set $\{0, 1, ...\}$ of natural numbers, $\mathcal{N}_+$ the set of positive natural numbers. We shall identify the natural n with the set $\{0, ..., n-1\}$, and thus 0 is also the empty set $\emptyset$. Every finite set S is isomorphic to n, with n the cardinal of S, denoted $n = |S|$. If A and B are sets, we write $A \rightarrow B$, or sometimes B^A, for the set of functions with domain A and codomain B.

1.2 Languages, concrete syntax

Let Σ be a finite alphabet. A *string* u of *length* n is a function in $n \rightarrow \Sigma$. The set of all strings over Σ is

$$\Sigma^* = \bigcup_{n \in \mathcal{N}} \Sigma^n$$

We write $|u|$ for the length n of u. We write u_i for $u(i-1)$, when $i \leq n$. The null string, unique element of Σ^0, is denoted Λ. The unit string mapping 1 to $a \in \Sigma$ is denoted 'a'. The concatenation of strings u and v, defined in the usual fashion, is denoted $u \hat{} v$, and when there is no ambiguity we write e.g. 'abc' for 'a' $\hat{}$ 'b' $\hat{}$ 'c'. When $u \in \Sigma^*$ and $a \in \Sigma$, we write $u \cdot a$ for $u \hat{}$ 'a'. We define an ordering $\leq$ on Σ^*, called the *prefix* ordering, by

$$u \leq v \Leftrightarrow \exists w \quad v = u \hat{} w$$

If $u \leq v$, the residual w is unique, and we write $w = v/u$. We say that occurrences u and v are *disjoint*, and we write $u|v$, iff u and v are unrelated by the partial ordering $\leq$. Finally we let $u < v$ iff $u \leq v$ with $u \neq v$.

The set Σ^* has the structure of a monoïd, that is:

$$Ass : (u \hat{} v) \hat{} w = u \hat{} (v \hat{} w)$$

$$IdL : \Lambda \hat{} u = u$$

$$IdR : u \hat{} \Lambda = u$$

Actually, Σ^* is the *free* monoïd generated by Σ.

Examples.

1. $\Sigma = 0$. We get $\Sigma^* = 1$.

2. $\Sigma = 1$. We get $\Sigma^* = \mathcal{N}$. Strings are here natural numbers in unary notation, and concatenation corresponds to addition.

3. $\Sigma = 2 = \{0, 1\}$ (the Booleans). The set Σ^* is the set of all binary words.

4. $\Sigma = \mathcal{N}_+$. We call the elements of Σ^* *occurrences*. When $u = w \cdot m$ and $v = w \cdot n$, with $m < n$, we say that u is *left* of v, and write $u <_L v$.

1.3 Terms: abstract syntax

We first define a *tree domain* as a subset D of $\mathcal{N}_1^*$ closed under $<$ and $<_L$:

$$u \in D \ \wedge \ v < u \ \Rightarrow \ v \in D$$

$$u \in D \ \wedge \ v <_L u \ \Rightarrow \ v \in D.$$

We say that M is a Σ-tree iff $M \in D \rightarrow \Sigma$, for some tree domain D. We write $D = D(M)$, and we say that D is the set of occurrences in M. M is said to be finite whenever D is.

We shall now use occurrences to designate nodes of a tree, and the subtree starting at that node. If $u \in D(M)$, we define the Σ-tree M/u as mapping occurrence v to $M(u \hat{} v)$. We say that M/u is the sub-tree of M at occurrence u. If N is also a Σ-tree, we define the *graft* $M[u \leftarrow N]$ as the Σ-tree mapping v to $N(w)$ whenever $v = u \hat{} w$ with $w \in D(N)$, and to $M(v)$ if $v \in D(M)$ and not $u \leq v$.

We need one auxiliary notion, that of *width* of a tree. If $M \in \Sigma^*$, we define the (top) width of M as

$$||M|| \ = \ \max\{n \mid \text{`}n\text{'} \in D(M)\}$$

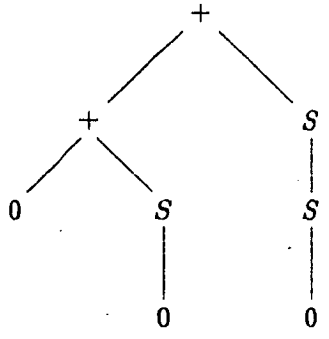
We shall now consider Σ a *graded* alphabet, that is given with an *arity* function α in $\Sigma \rightarrow \mathcal{N}$. We then say that M is a Σ-term iff M is a Σ-tree verifying the supplementary consistency condition:

$$\forall u \in D(M) \ \ ||M/u|| \ = \ \alpha(M(u))$$

That is, every subtree of M is of the form $F(M_1, M_2, ..., M_n)$, with $n = \alpha(F)$. We write $T(\Sigma)$ for the set of Σ-terms. If $M_1, M_2, ...M_n \in T(\Sigma)$ and $F \in \Sigma$, with $\alpha(F) = n$, then $M = F(M_1, M_2, ...M_n)$ is easily defined as a Σ-term. This gives $T(\Sigma)$ the structure of a Σ-algebra. Since conversely the decomposition of M is uniquely determined, we call $T(\Sigma)$ the *completely free* Σ-algebra.

Example

With $\Sigma = \{+, S, 0\}$, $\alpha(+) = 2$, $\alpha(S) = 1$, $\alpha(0) = 0$, the following structure represents a Σ-term:



The following proposition is easy to prove by induction. All occurrences are supposed to be universally quantified in the relevant tree domain.

Proposition 1.

$$\textit{Embedding}: \quad M[u \leftarrow N]/(u \hat{} v) \ = \ N/v$$

$$\textit{Associativity}: M[u \leftarrow N][u \hat{} v \leftarrow P] \ = \ M[u \leftarrow N[v \leftarrow P]]$$

$$\textit{Persistence} : M[u \leftarrow N]/v \;=\; M/v \quad (u|v)$$
$$\textit{Commutativity} : M[u \leftarrow N][v \leftarrow P] \;=\; M[v \leftarrow P][u \leftarrow N] \quad (u|v)$$
$$\textit{Distributivity} : M[u \leftarrow N]/v \;=\; (M/v)[u/v \leftarrow N] \quad (v \leq u)$$
$$\textit{Dominance} : M[u \leftarrow N][v \leftarrow P] \;=\; M[v \leftarrow P] \quad (v \leq u)$$

We define the length $|M|$ of a (finite) term M recursively by:

$$|F(M_1, ..., M_2)| \;=\; 1 + \Sigma_{i=1}^{n} |M_i|$$

1.4 Parsing

It is well-known that the term in the example above can be represented unambiguously as a Σ-string, for instance in *prefix polish notation*, that is here: $++0S0SS0$. This result is not very interesting: such strings are neither good notations for humans, nor good representations for computers, since the graft operation necessitates unnecessary copying. We shall discuss later good machine representations, using binary graphs. As far as human readibility is concerned, we assume known parsing techniques. This permits to represent terms, on an extended alphabet with parentheses and commas, which is closer to standard mathematical practice. Also, infix notation and indentation permit to keep in the string some of the tree structure more apparent. We shall not make explicit the exact representation grammar, and allow ourselves to write freely for instance $(0 + S(0)) + S(S(0))$. Note that we avoid explicit quotes as well, which permits us to mix freely meta-variables with object structures, like in $S(M)$, where M is a meta-variable denoting a Σ-term.

1.5 Terms with variables, substitution

The idea is to internalize the notation $S(M)$ above as a term $S(x)$ over an extended alphabet containing special symbols of arity 0 called *variables*.

Let V be a denumerable set disjoint from Σ. We define the set of terms with variables, $T(\Sigma, V)$, in exactly the same way as $T(\Sigma \cup V)$, extending the arity function so that $\alpha(x) = 0$ for every x in V. The only difference between the variables and the constants (symbol of arity 0) is that a constant has an existential import: it denotes a value in the domain we are modelling with our term language, whereas a variable denotes a term. The difference is important only when there are no constants in Σ, since then $T(\Sigma)$ is empty.

All of the notions defined for terms extend to terms with variables. We define the set $V(M)$ of variables occurring in M as:

$$V(M) \;=\; \{x \in V \mid \exists u \in D(M) \quad M(u) = x\}$$

and we define the number of distinct variables in M as $\nu(M) \;=\; |V(M)|$.

We shall now formalize the notion of substitution of terms for variables in a term containing variables. From now on, the sets Σ and V are fixed, and we use T to denote $T(\Sigma, V)$. A *substitution* σ is a function in $V \rightarrow T$, identity almost everywhere. That is, the set $D(\sigma) \;=\; \{x \in V \mid \sigma(x) \neq x\}$ is finite. We call it the *domain* of σ. Substitutions are extended to morphisms over T by

$$\sigma(F(M_1, ..., M_n)) \;=\; F(\sigma(M_1), ..., \sigma(M_n))$$

Bijective substitutions are called *permutations*. When $U \subseteq V$, we write σ_U for the restriction of substitution σ to U. It is easy to show that, for all σ, M and U:

$$V(M) \subseteq U \;\Rightarrow\; \sigma(M) = \sigma_U(M).$$

Alternatively, we can define the replacement $M[x \leftarrow N]$ as

$$M[u_1 \leftarrow N]...[u_n \leftarrow N]$$

where $\{u_1, ..., u_n\} = \{u \mid M(u) = x\}$ and then

$$\sigma(M) = M[x \leftarrow \sigma(x) \mid x \in V(M)]$$

with an obvious notation.

We now define the quasi-ordering $\leq$ of *matching* in T by:

$$M \leq N \Leftrightarrow \exists\sigma \quad N = \sigma(M)$$

It is easy to show that if such a σ exists, $\sigma_{V(M)}$ is unique. We shall call it the *match of N by M*, and denote it by N/M.

We define $M \equiv N \Leftrightarrow M \leq N \wedge N \leq M$. When $M \equiv N$, we say that M and N are *isomorphic*. This is equivalent to say that $M = \sigma(N)$ for some permutation σ. Note that $M \equiv N$ implies $|M| = |N|$. Finally, we define

$$M > N \Leftrightarrow N \leq M \wedge \neg M \leq N.$$

Proposition. $>$ is a well-ordering on T.

Proof. We show that $M > N$ implies $\mu(M) > \mu(N)$, with $\mu(M) = |M| - \nu(M)$.

Let φ be any bijection between $T \times T$ and V. We define a binary operation $\cap$ in T by:

$$F(M_1, ..., M_n) \cap F(N_1, ..., N_n) = F(M_1 \cap N_1, ..., M_n \cap N_n)$$

$$M \cap N = \varphi(M, N) \quad \text{in all other cases.}$$

$M \cap N$ is uniquely determined from φ and, for distinct φ's, is unique up to $\equiv$.

Proposition. $M \cap N$ is a g.l.b. of M and N under the match quasi-ordering.

Let $\mathbf{T}$ be the quotient set $T/\equiv$, completed with a maximum element $\top$. From the propositions above we conclude:

Theorem. $\mathbf{T}$ is a complete lattice.

Corollary. If two terms M and N have an upper bound, i.e. a common instance $\sigma(M) = \sigma'(N)$, they have a l.u.b. $M \cup N$, which is a most general such instance: $\sigma = \sigma_0; \tau$, $\sigma' = \sigma'_0; \tau$. The term $M \cup N$ is unique modulo $\equiv$ and may be found by the *unification* algorithm.

Proposition.

$$D(\sigma(M)) = D(M) \cup \bigcup_{\{u \mid M(u) \in V\}} \{u \hat{} v \mid v \in D(\sigma(M(u)))\}$$

$$\forall u \in D(M) \quad M(u) \in V \Rightarrow \sigma(M)/u \hat{} v = \sigma(M(u))/v \quad (v \in D(\sigma(M(u))))$$

$$M(u) \in \Sigma \Rightarrow \sigma(M)/u = \sigma(M/u)$$

$$\sigma(M)[u \leftarrow \sigma(N)] = \sigma(M[u \leftarrow N])$$

1.6 Graph representations, dags

We represent trees by binary graphs of adr pairs. An adr consists in one tag bit, and one byte field interpreted either as an address in the graph memory, or as a natural number. In this last case, the natural 0 is reserved for *nil*, the empty list of trees. Symbols from Σ are coded as positive naturals. If tree M is represented at the graph address $adr1$ and the list L is represented at address $adr2$, then the list $M \cdot L$ is represented by the graph node $(M \cdot L)$. Finally, the tree $F(L)$ is represented by $(F \cdot L)$.

This is the standard way of representing trees and lists in the language LISP. A precise description of the memory allocation implementation of such schemes is beyond the scope of these notes.

Terms are of course represented as trees. A global table holds the arity function. There are several possibilities for the representation of variables. They may be represented as symbols. But then the scope structure must be computed by an algorithm, rather than being implicit in the structure. Also a global scanning of the term is necessary to determine its set of variables, and substitution involves copying of the substituted term. For these reasons, variables are often represented rather as integer offsets in stacks of bindings. Such "structure sharing" representations are now standard for PROLOG implementations.

A precise account of the various representations schemes for term structures, and of the accompanying algorithms, is out of the scope of these notes. It should be born in mind that the crucial problem is memory utilization: the trade-off between copying and sharing is often the deciding factor for an implementation. Languages with garbage-collected structures, such as LISP, are ideal for programming "quick and dirty" prototypes. But serious implementation efforts should aim at good algorithmic performance on realistic size applications.

The crucial algorithms in formula and proof manipulation are matching, unification, substitution and grafting. First-order unification has been specially well studied. A linear algorithm is known [122], but in practice quasi-linear algorithms based on congruence classes operations are preferred [99,100]. Furthermore, these algorithms extend without modification to unification of infinite rational terms represented by finite graphs [64].

Implementation methods may be partitioned into two families. Some depend on logical properties (e.g. sharing subterms in dags arising from substitution to a term containing several occurrences of the same variable). Some are purely statistical (e.g. sharing structures globally through hash-coding techniques). Particular applications require a careful analysis of the optimal trade-off between logical and statistical techniques.

There is no comprehensive survey on implementation issues. Some partial aspects are described in [8,141,101,99,164,159,115,40,1,32,42,19,45,145,160].

2 Inference rules

We shall now study *inference systems*, defined by inference rules. The general form of an inference rule is:

$$IR: \frac{P_1 \quad P_2 \quad \ldots \quad P_n}{Q}$$

where the P_i's and Q are *propositions* belonging to some formal language. We shall here regard these propositions as *types*, and the inference rule as the description of the signature of IR considered as a typed operator. More precisely, IR has arity n, P_i is the type of its i-th argument, and Q is the

type of its result. Well-typed terms composed of inference operators are called the *proofs* defined by the inference system. Let us now examine a few familiar inference systems.

2.1 The trivial homogeneous case: Arities

A graded alphabet Σ may be considered as the simplest inference systems, where types are reduced to arities. I.e., the set of propositions is 1, and an operator F of arity n is an inference rule

$$F: \frac{0\ 0\ \ldots\ 0}{0}$$

(with n zero's in the numerator). A Σ-proof corresponds to our Σ-terms above.

2.2 Finite systems of types: Sorts

The next level of inference systems consist in choosing a finite set S of elementary propositions, usually called *sorts*. For instance, with $S = \{int, bool\}$, and Σ defined by:

$$0: int \quad S: int \to int \quad true: bool \quad false: bool \quad if: bool, int \to int$$

where we use the alternative syntax $P_1, ..., P_n \to Q$ for an inference rule, the term $if(true, 0, S(0))$ is of sort *int*, i.e. it is a proof of proposition *int*.

As another example, consider the puzzle "Missionaries and Cannibals". We call *configuration* any triple $\langle b, m, c\rangle \in 2 \times 4 \times 4$. The boolean b indicates the position of the boat, m (resp. c) is the number of missionaries (resp. cannibals) on the left bank. The set of states S is the set of *legal* configurations, that obey the condition

$$P(m, c) \equiv m = c \text{ or } m = 0 \text{ or } m = 3$$

There are thus 10 distinct states or sorts. The rules of inference comprise first a constant denoting the starting configuration:

$$s_0: \langle 0, 3, 3\rangle$$

then the transitions carrying p missionaries and q cannibals from left to right:

$$L_{m,c,p,q}: \langle 0, m, c\rangle \to \langle 1, m-p, c-q\rangle \quad (m \geq p, c \geq q, P(m, c), P(m-p, c-q), 1 \leq p+q \leq 2)$$

and finally the transitions $R_{m,c,p,q}$, which are inverses of $L_{m,c,p,q}$. The game consists in finding a proof of $\langle 1, 0, 0\rangle$.

This simple example of a finite group of transformations applies to more complex tasks, such as Rubik's cube. All state transition systems can be described in a similar fashion. Examples of such proofs are parse-trees of regular grammars, where the inference rules signatures correspond to a finite automaton transition graph. Slightly more complicated formalisms allow subsorts, i.e. containment relationships between the sorts, i.e. implications between the elementary propositions. These systems reduce to simple sorts by considering dummy transitions corresponding to the implicit coercions.

2.3 Types as terms: standard proof trees

We shall here describe our types as terms formed over an alphabet Φ of type operators, which we shall call *functors*. For the moment, we shall assume that we have just one category of such propositions, i.e. the functors have just an arity. The alphabet Σ of inference rules determines the legal proof trees.

Example. Combinatory logic.

We take as functors a set Φ of constants Φ_0, plus a binary operator $\rightarrow$, which we shall write in infix notation. We call *functionality* a term in $T(\Phi)$. We have three families of rules in Σ. In the following, the meta-variables A, B, C denote arbitrary functionalities. The operators of the K and S families are of arity 0, the operators of the *App* family are binary.

$$K_{A,B}: \ A \rightarrow (B \rightarrow A)$$

$$S_{A,B,C}: \ (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

$$App_{A,B}: \ \frac{A \rightarrow B \quad A}{B}$$

Here is an example of a proof. Let A and B be any functionalities, $C = B \rightarrow A$, $D = A \rightarrow C$, $E = A \rightarrow A$, $F = A \rightarrow (C \rightarrow A)$, $G = D \rightarrow E$. The term

$$App_{D,E}(App_{F,G}(S_{A,C,A}, K_{A,C}), K_{A,B})$$

has type E, i.e. it gives a proof of the proposition $A \rightarrow A$.

We express formally that proof M proves proposition P in the inference system Σ as: $\Sigma \models M : P$. That is, we think of a theorem as the type of its proof tree. Proof-checking is identified with type-checking. Here this is a simple consistency check; that is, if operator F is declared in Σ as: $F: \ P_1, ..., P_n \rightarrow Q$ and if $\Sigma \models M_i : P_i$ for $1 \leq i \leq n$, then $\Sigma \models F(M_1, ..., M_n) : Q$.

2.4 Polymorphism: Rule schemas

This next level of generality consists in authorizing variables in the propositional terms. This is very natural, since it internalizes the meta-variables used to index families of inference rules as propositional variables. The rules of inference become thus polymorphic operators, whose types are expressions containing free variables. This is the traditional notion of schematic inference rule from mathematical logic.

Example. The example from the previous section is more naturally expressed in this polymorphic formalism. We replace the set Φ_0 by a set of variables V, and now we have just 3 rules of inference: K, S and *App*. The types can be completely dispensed with, since a well typed term possesses a most general type, called its principal type. For instance, in the example above, the proof $App(App(S, K), K)$ has a principal type $A \rightarrow A$, with $A \in V$. This term is usually written $I = SKK$ in combinatory logic, where the concrete syntax convention is to write combinator strings to represent sequences of applications associated to the left.

The notion of principal type, first discovered by Hindley in the combinatory logic context, and independently by Milner for ML type-checking [111], is actually completely general:

Theorem. Let Σ be any signature of polymorphic operators over a functor signature Φ. Let M be a legal proof term. Then M possesses a principal type $\tau \in T(\Phi, V)$. That is, $\Sigma \models M : \tau$, and for all $\tau' \in T(\Phi, V)$, $\Sigma \models M : \tau'$ implies $\tau \leq \tau'$.

Proof. This is an easy application of the unification theorem.

By now we have developed enough formalism to make sense out of our "propositions as types" paradigm. Actually, the example we have discussed above is the fragment of propositional logic known as "minimal logic". When regarding the functor $\rightarrow$ as (intuitionistic) implication, and *App* as the usual inference rule of Modus ponens, K and S are the two axioms of minimal logic presented as a Hilbert calculus. Combinatory logic is thus the calculus of proofs in minimal logic [37].

Actually combinators don't just have a type, they have a value. They can be *defined* with definition equations in terms of application. Using the concrete syntax mentioned above, we get for instance K and S defined by the following equations:

$$Def_K : \; K\ x\ y \;=\; x$$

$$Def_S : \; S\ x\ y\ z \;=\; x\ z\ (y\ z).$$

Exercice. Verify that the two equations above, when seen as unification constraints, define the expected principal types for K and S.

This point of view of considering equality axiomatizations of the proof structures corresponds to what the proof-theorists call *cut elimination*. That is, the two equations above can be used as rewrite rules in order to eliminate redundancies corresponding to useless detours in the proofs. We shall develop more completely this point of view of *computation as proof normalization* in section 4.4 below.

The current formalism of inference rules typed by terms with variables corresponds to proof theory's intuitionistic sequents, and to automated reasoning's Horn clauses. For instance, a PROLOG [24] interpreter may be seen in this framework as a proof synthesis method. Given an alphabet Σ of polymorphic inference rules (usually called definite clauses), and a proposition τ over functor alphabet Φ, it returns a proof term M such that M is a legal Σ-proof term of principal type τ' instance of τ:

$$\Sigma \models M : \tau' \geq \tau.$$

With $\sigma = \tau'/\tau$, we say that σ is a PROLOG *answer* to the *query* τ. Of course this explanation is incomplete; we have to explain that PROLOG finds all such instances by a backtrack procedure constructing proofs in a bottom-up left-to-right fashion, using operators from Σ in a specific order (the order in which clauses are declared); this last requirement leads to incompleteness, since PROLOG may loop with recursively composable operators, whereas a different order might lead to termination of the procedure. Also, PROLOG may be presented several goals together, and they may share certain variables, but this may be explained by a simple extension of the above proof-synthesis explanation.

We claim that this explanation of PROLOG is more faithful to reality than the usual one with Horn clauses. In particular, our explanation is completely constructive, and we do not have to explain the processes of conjunctive normalization and Skolemization. Furthermore, there is no distinction in Φ between predicate and function symbols, consistently with most PROLOG implementations.

2.5 Proof terms with variables, natural deduction.

The example above demonstrated the difficulty of proofs presented in a Hilbert style. The completely trivial theorem $\forall A \cdot A \to A$ had a complicated proof using three axioms and two applications of modus ponens. Of course one could consider adding combinator I as an axiom, but this is only begging the question since other trivial natural theorems would present similar difficulties. And of course there is no easy way to decide which combinators are well-typed. For instance, Peirce's law:

$$Peirce: \ ((A \to B) \to A) \to A$$

although a propositional tautology easily checkable by the truth-table method, is *not* intuitionistically valid.

The natural proof of $A \to A$ consists in, given a proof x of A, returning merely x as a proof of A. that is, the *natural* proof of $A \to A$ is the (polymorphic) identity algorithm. This method of proof usually proceeds through the deduction theorem below.

Deduction theorem. Let Γ be any set of propositions, A and B be two propositions. We have $\Gamma, A \vdash B$ iff $\Gamma \vdash A \to B$.

The deduction theorem holds in any reasonable system of logic. It can be proved easily in minimal logic, by induction on the size of proofs. Unfortunately, the deduction theorem is a *meta* theorem, i.e. a mathematical theorem of the meta-theory analyzing the proof system, as opposed to the theorems, or well-typed proof terms inside the proof system.

We shall see in section 4 that it is easy to internalize *deductions* as proof terms with variables, called *sequents*. This point of view will lead to logic presented in *natural deduction* style, that is to λ-calculus formalisms. Before investigating this next level of expressive power, we consider in the next section a particularly important inference system Σ, that of *equational logic*.

3 Rewriting inference and equational logic

3.1 The classical presentation

Equational logic is classically presented as a restriction of first-order logic, where the only predicate symbol is $=$, and the only non-logical axioms are universal equalities between terms containing free variables. For instance, the theory of groups is classically presented over the functor alphabet

$$\Phi \ = \ \{*, ^{-1}, 1\}$$

by the equations:

$$Idl: \ \ 1 * x \ = \ x$$

$$Invl: \ \ x^{-1} * x \ = \ 1$$

$$Ass: \ \ (x * y) * z \ = \ x * (y * z)$$

and the class of all first order models of these equations is called the *variety* of groups. The well known completeness theorem of Birkhoff states that a universally quantified equation between terms over Φ is valid in the variety iff it can be deduced from the axioms using the rules of substitution and of replacement of equal for equal.

3.2 The proof-theoretic formalization

Here we ignore the abstract notion of model and concentrate on the rules of inference. We assume given a functor alphabet Φ given with arity function α, in which we distinguish an atom $\rightarrow$ given with arity 2. The substitution inference rule disappears, since it is implicit from the polymorphism of other rules. The replacement of equals for equals is decomposed into elementary steps of term replacement rules:

$$Id_A : \ A \rightarrow A \qquad \textit{Reflexivity}$$

$$; : \ \frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C} \qquad \textit{Transitivity}$$

which specify that the rewriting arrow $\rightarrow$ is a quasi-ordering. Now we must state that $\rightarrow$ is compatible with the rest of the Φ-structure. That is, for every functor F in $\Phi - \{\rightarrow\}$ of arity n and for every $i \leq n$ we take a congruence rule:

$$Funct_{F,i} \ : \ \frac{A \rightarrow B}{F(A_1, ..., A_{i-1}, A, A_{i+1}, ..., A_n) \rightarrow F(A_1, ..., A_{i-1}, B, A_{i+1}, ..., A_n)} \qquad \textit{Congruence}$$

If we add the rule of symmetry we get the theory of equality, where we usually use symbol $=$ instead of $\rightarrow$:

$$Op \ : \ \frac{A = B}{B = A} \qquad \textit{Symmetry}$$

The non-logical axioms of the variety are then added as so many constants. For instance, over groups, we obtain a proof of proposition $y = x^{-1} * (x * y)$ by the term

$$Op(Funct_{*,1}(Invl); Idl); Ass \ : \ y = x^{-1} * (x * y)$$

Exercise: Show a proof of $x * 1 = x$ using the inference system Σ above.

The conclusion we may draw from the example above is that, beyond its apparent simplicity, equational reasoning may indeed be quite complicated. The rule of symmetry is specially hard to use since it expresses a commutativity of $\rightarrow$, harder to visualize than the easier monoïd structure implicit from the rules *Id* and ";". It is then natural to ask:

1) Can we eliminate Op
2) More generally, can we normalize equational proofs?

3.3 The categorical viewpoint

This viewpoint gives a prominent role to the monoïd structure of the quasi-ordering $\rightarrow$. Simplifying the presentation, we may present a *category* as presented by a set of *objects Obj*, which we shall here confuse with the set of (closed) terms over some functor alphabet Φ, and by a set of *arrows* (or morphisms) which we shall here confuse with the set of (closed) proofs generated from some inference system Σ containing initially the two rules:

$$Id_A \ : \ A \rightarrow A \qquad \textit{Identity}$$

$$; : \ \frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C} \qquad \textit{Composition}$$

Whenever $f : A \rightarrow B$, we say that arrow f has *domain* A and *codomain* B. Furthermore, it is specified that the proofs are quotiented by a congruence $=$ verifying:

$$Idl : \ Id; f \ = \ f$$

$$Idr: \ f;Id \ = \ f$$

$$Ass: \ (f;g);h \ = \ f;(g;h)$$

So we see that a category is a structure obtained as a hybrid of quasi-ordering and of monoïd, to which it reduces in the two degenerate cases (i.e. $|f : A \rightarrow B| \leq 1$ and $|Obj| = 1$). Note that we have given the same name to axiom Idl as for the axiomatization of groups above, although here the operator ";" is a Σ-operator, and not just a Φ-operator like "*". However the unification theorem allows us to make implicit the type of variable f above, and the overloading of "*Idl*" may be seen as a reflection principle.

If **A** and **B** are two categories, a functor F from **A** to **B** associates to every object A of **A** an object $F(A)$ of **B**, and to every arrow $f : A \rightarrow B$ an arrow $F(f) : F(A) \rightarrow F(B)$ such that the following functorial conditions hold:

$$F(Id) \ = \ Id$$

$$F(f;g) \ = \ F(f);F(g)$$

We see a great analogy between the notion of rewriting inference system and the main categorical notions. Actually, the categorical viewpoint is richer in that the functors have sorts themselves (i.e., the categories), and poorer in that they do not yet have arities (i.e. we just have monadic functors so far). In order to build-in arities we shall need products, and a full categorical account of minimal logic is obtained by a further adjunction, namely exponentiation. But we shall defer this explanation until we develop natural deduction in section 4. We have given this elementary development of category theory essentially to justify our terminology. The congruence rule of term formation explains a functoriality condition on the object part, and the functoriality condition on the arrow part of the functor expresses the congruence property for rewriting. Substitutivity in rewrite rules is expressed by defining them as natural transformations between the functors denoted by the two sides of the rule. That is, a *natural transformation* τ between functors F and G (both from category **A** to category **B**) is a mapping associating to every object A of **A** an arrow $\tau_A : F(A) \rightarrow G(A)$ such that

$$\tau_A;G(f) \ = \ F(f);\tau_B$$

And if we consider equations rather than simply rewrite rules, the symmetry inference rule is interpreted as the existence of inverses to arrows. Equations are thus defined as natural isomorphisms.

Category theory is explained in Mac Lane [94]. The categorical viewpoint for algebra has been developed by Lawvere and others [97]. Its application to proof theory is explained (in a somewhat complicated form) in Szabo [152].

3.4 Confluence and Termination

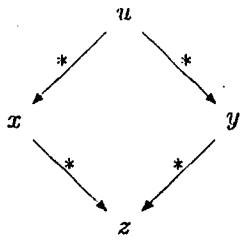
We come back to the problem of eliminating the symmetry rule. Let now $\rightarrow$ be any binary relation over some set S, $\rightarrow^*$ be its reflexive-transitive closure, $\leftrightarrow^*$ be its equivalence closure. We say that $\rightarrow$ verifies the *Church-Rosser condition* iff

$$x \leftrightarrow^* y \ \Leftrightarrow \ \exists z \quad x \rightarrow^* z \ \wedge \ y \rightarrow^* z$$

It is easy to show that this condition is equivalent to *confluence*, i.e.

$$u \rightarrow^* x \ \wedge \ u \rightarrow^* y \ \Leftrightarrow \ \exists z \quad x \rightarrow^* z \ \wedge \ y \rightarrow^* z$$

that is, diagrammatically:



When $\rightarrow$ is a confluent relation, normal forms (i.e. terminal elements) are unique whenever they exist, and equality (i.e. $\leftrightarrow^*$) may be decided by rewriting. That is, deduction may be replaced by computation, and symmetry is eliminated in all but one instance.

For instance, the following set of 10 rewrite rules defines a confluent term rewriting system for group theory:

$$1 * x \rightarrow x$$

$$x^{-1} * x \rightarrow 1$$

$$(x * y) * z \rightarrow x * (y * z)$$

$$x * 1 \rightarrow x$$

$$x * x^{-1} \rightarrow 1$$

$$(x^{-1})^{-1} \rightarrow x$$

$$1^{-1} \rightarrow 1$$

$$x * (x^{-1} * y) \rightarrow y$$

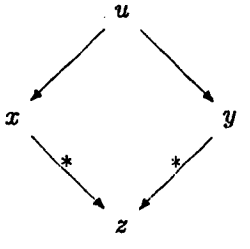
$$x^{-1} * (x * y) \rightarrow y$$

$$(x * y)^{-1} \rightarrow y^{-1} * x^{-1}$$

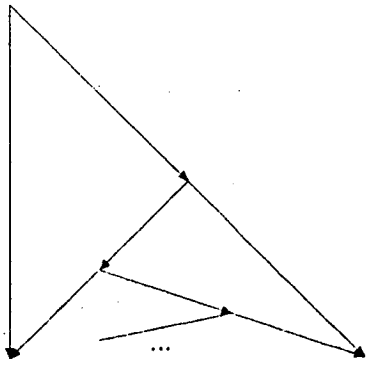
The rewrite relation associated with such a term rewriting system R is defined by $M \rightarrow_R N$ iff there exists a rule $\alpha \rightarrow \beta$ in R and an occurrence u in $D(M)$ such that $M/u = \sigma(\alpha)$ for some substitution σ, and $N = M[u \leftarrow \sigma(\beta)]$. It is clear that the group axioms are decided by the system R above. Conversely all rewrite rules in R may be shown to be valid equations in group theory (see the exercise above). What is less obvious is to decide the confluence of R. We shall see in the next section that is is easy to show that it is *locally confluent*, in the sense that:

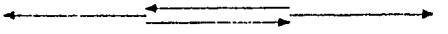
$$u \rightarrow x \ \wedge \ u \rightarrow y \ \Leftrightarrow \ \exists z \quad x \rightarrow^* z \ \wedge \ y \rightarrow^* z$$

that is, diagrammatically:



However, local confluence is not enough to prove confluence, as shown by the following counter-examples:





3.5 The Nœtherian case: Knuth and Bendix

The problem encountered with the above counter-examples is that the rewriting relation possessed infinite chains. Let us say that relation $\rightarrow$ is *Nœtherian* iff there is no infinite chain $x_1 \rightarrow x_2 \rightarrow \ldots$ (Then, its transitive closure $\rightarrow^+$ is a well-founded ordering). We remark that $\rightarrow$ is Nœtherian over S iff every non-empty subset of S admits a minimal element with respect to $\rightarrow^+$.

Now let us say that a predicate P over S is *$\rightarrow$-hereditary* iff

$$\forall x \in S \quad [\forall y \quad x \rightarrow^+ y \;\Rightarrow\; P(y)] \;\Rightarrow\; P(x).$$

Now we may state an important induction principle.

Principle of Nœtherian Induction: Let $\rightarrow$ be a Nœtherian relation over S. Then for every $\rightarrow$-hereditary predicate P we have $\forall x \in S \;\; P(x)$.

It is easy to validate this induction principle using the above remark, by considering the set of all x's such that not $P(x)$. And now we may show that local confluence implies confluence for Nœtherian relations.

Newman's lemma. A Nœtherian relation is confluent iff it is locally confluent.

Proof: Nœtherian induction on predidate P defined as:

$$P(u) \;=\; \forall x, y \quad u \rightarrow^* x \;\wedge\; u \rightarrow^* y \;\Rightarrow\; \exists z \quad x \rightarrow^* z \;\wedge\; y \rightarrow^* z$$

We now explain the Knuth-Bendix decision procedure for the confluence of Nœtherian term rewriting systems. First let us give an algorithm.

Superposition algorithm. Let $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ be two rewrite rules in R, let $u \in D(\alpha_1)$ and $M = \alpha_1/u$ be such that M is a non-variable term unifiable with α_2. Let $N = \sigma_1(M) = \sigma_2(\alpha_2)$ be a principal instance, with $V(N) \cap V(\alpha_1) = \emptyset$. We say that the *superposition* of $\alpha_2 \rightarrow \beta_2$ on $\alpha_1 \rightarrow \beta_1$ *at* u determines the *critical pair* $\langle P, Q \rangle$, with $P = \sigma_1(\alpha_1)[u \leftarrow \sigma_2(\beta_2)]$ and $Q = \sigma_1(\beta_1)$.

Examples.

- $G(B,x) \rightarrow K(x)$ superposes on $F(x,G(x,A)) \rightarrow H(x)$ at '2' to give $P = F(B,K(A))$ and $Q = H(B)$.
- $H(H(x)) \rightarrow K(x)$ superposes on itself at '1' to give $P = H(K(y))$ and $Q = K(H(y))$.

The Knuth-Bendix theorem. The relation $\rightarrow_R$ is locally confluent iff for every critical pair $\langle P,Q\rangle$ there exists N such that $P \rightarrow_R^* N$ and $Q \rightarrow_R^* N$.

Corollary. If R is a Nœtherian term rewriting system, its confluence is decidable.

The above theorem may be used to check the local confluence of the 10 group rewrite rules above. Assuming termination, this shows that any group equality may be decided by mere rewriting.

Actually, the method may be extended to systems of rules that fail the test. If for some critical pair $\langle P,Q\rangle$ we reduce P and Q to two distinct irreducible terms P' and Q', we have generated an interesting lemma $P' = Q'$, which is an equational consequence of the rules considered as equations. It may be possible to give an orientation to this new equality for forming an extended term rewriting system, while preserving the finite termination property. This is the basis of the Knuth-Bendix completion method, which attempts to complete a term rewriting system to a confluent one. This method may be considered a way of compiling a canonical form algorithm from an equational specification.

We cannot describe the method fully here. The main ideas are that unresolved critical pairs are kept as new rewrite rules, and that all rules are kept inter-reduced. The procedure may stop with a canonical system, it may fail because termination is impossible to establish, or it may loop. Whenever it does not fail, it gives a semi-decision procedure for the original equational theory, as explained in Huet [66]. More detailed expositions of the method may be found in [84,65,71].

Failure may result from some permutative consequence such as commutativity. The method has been extended in various ways in order to consider rewritings modulo such permutative axioms. For instance, Peterson and Stickel [127] have shown that it was possible to extend the method to complete equational presentations, where one or several functors were assumed to be associative and commutative, using Stickel's associative-commutative unification algorithm [151,43]. This method has been extended by Jouannaud and Kirchner [73].

Various other extensions of the Knuth-Bendix procedure have been proposed, for handling constructors (free functors) [69] and for solving word problems in finitely presented algebras [90]. The Knuth-Bendix completion procedure and its extensions give a general framework to simplification techniques.

As example of canonical term rewriting system we give distributive lattices. Here $\cap$ and $\cup$ are assumed to be associative and commutative. The canonical set consists in the following four rules:

$$x \cap (x \cup y) \rightarrow x$$

$$x \cup (y \cap z) \rightarrow (x \cup y) \cap (x \cup z)$$

$$x \cup x \rightarrow x$$

$$x \cap x \rightarrow x$$

Exercise. Show that the other distributivity law is a consequence of the above rules.

Finally, we show the canonical system for Boolean algebras. Now the connectives $\wedge$ and $\oplus$ (exclusive or) are assumed to be associative and commutative.

$$x \wedge 1 \rightarrow x$$

$$x \wedge 0 \rightarrow 0$$

$$x \wedge x \rightarrow x$$

$$x \oplus 0 \rightarrow x$$

$$x \oplus x \rightarrow 0$$

$$(x \oplus y) \wedge z \rightarrow (x \wedge z) \oplus (y \wedge z)$$

This canonical set can be used to decide propositional calculus, using the following translations:

$$\neg x \rightarrow x \oplus 1$$

$$x \vee y \rightarrow x \oplus y \oplus (x \wedge y)$$

$$x \Rightarrow y \rightarrow x \oplus (x \wedge y) \oplus 1$$

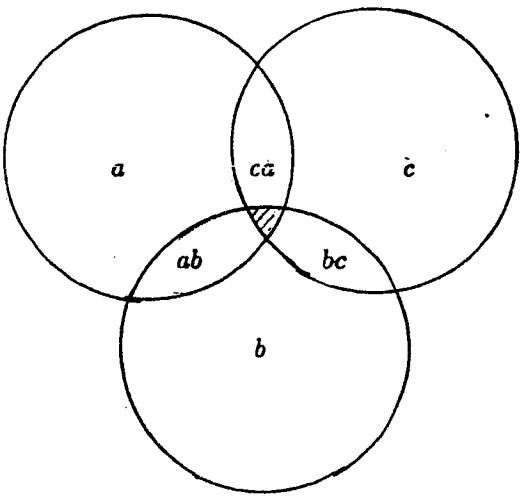
The resulting decision method is basically the method of Venn's diagrams, as the following example demonstrates. With three propositional letters a, b and c, the proposition

$$(a \wedge \neg b) \vee (b \wedge \neg c) \vee (c \wedge \neg a)$$

reduces to its canonical form:

$$a \oplus b \oplus c \oplus a \wedge b \oplus b \wedge c \oplus c \wedge a$$

which can easily be "seen" as a disjoint union of regions in the following Venn diagram:



This example also shows that disjunctive normal form is *not* a canonical form, since the above proposition possesses another d.n.f.

$$(b \wedge \neg a) \vee (c \wedge \neg b) \vee (a \wedge \neg c)$$

or, as Quine puts it, a formula may have distinct minimal sets of prime implicants [133].

3.6 Sequential computations

We now consider term rewriting systems with two constraints:

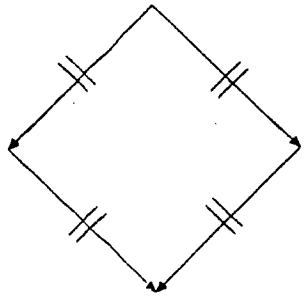
(a) left linearity: for every $\alpha \to \beta$ in R, every variable of α occurs exactly once

(b) non ambiguity: there are no critical pairs

As we shall see, these systems are always confluent, their termination is unnecessary. Functional programming languages, and more generally operational semantics rules can usually be expressed as such systems of rewrite rules [58]. As a very simple example, consider the system of two rules Def_K and Def_S defining the combinators S and K.

We shall here define the main notions of computation using rewrite rules. The full theory is given in Huet-Lévy [70]. We call redex in term M an occurrence $u \in D(M)$ such that $\alpha \leq M/u$ for some left-hand side α of a rule in R. We define the reduction relation $\to_R$ associated with R in the same way as in the preceding section. We shall assume R fixed from now on, and write simply $\to$ for reduction. Let $M \to N$ at redex occurrence $u \in D(M)$, using rule $\alpha \to \beta \in R$. Let now v be any redex in M. We define the set $v\backslash u$ of *residuals* of v as a set of redexes in N defined as follows. If $v = u$, $v\backslash u = \emptyset$. If $v < u$ or $v|u$, then $v\backslash u = \{v\}$. Finally, if $v > u$, this means, by non-overlapping, that v is below some variable x of α. By linearity, x has a unique occurrence in α, which we shall denote by x as well. That is, $v = u \hat{} x \hat{} w$ for some w. Now let X be the set of occurrences of variable x in β. We define $v\backslash u = \{u \hat{} y \hat{} w \mid y \in X\}$.

Thus redex v may have zero, one or more residuals in N. Intuitively, these residuals are the places where one must reduce in N in order to effect the computation step consisting in reducing at redex v in M. Actually, on the natural dag implementation all the occurrences of $v\backslash u$ denote the same shared node of the dag representing N. Symmetrically the same holds of $u\backslash v$. And as expected we have a local confluence diagram, where the single steps u and v confluate using all the steps in $v\backslash u$ (resp. $u\backslash v$). However, this is not sufficient, since we do not want to require $\to$ to be Nœtherian. However, it is easy to notice that all the redexes in $v\backslash u$ are mutually disjoint, and that any residual of some redex is always disjoint from any residual of some other disjoint redex. Thus it is natural to extend the reduction relation $\to$ to parallel reduction of a set of mutually disjoint redexes, a relation we shall write $\twoheadrightarrow$. If $M \twoheadrightarrow N$ using set of redexes U, then for every set V of mutually disjoint redexes in M, we define the residuals of V by U as: $V\backslash U = \{w \in v\backslash u \mid u \in U \wedge v \in V\}$. And now we have a strong confluence property:



which extends easily to multi-steps derivations A and B, yielding:

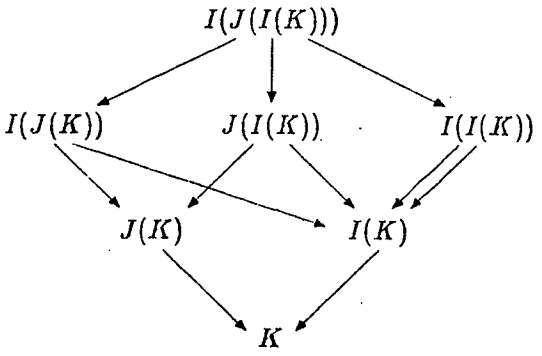
The parallel moves theorem. Let A and B be two co-initial derivations. Define $A \cup B$ as $A; B\backslash A$. Then $A \cup B \equiv B \cup A$, in the sense that these two derivations are co-final, and preserve

residuals.

The categorical viewpoint. The category whose objects are terms, and whose arrows from M to N are parallel derivations, quotiented by the equivalence $\equiv$, admits pushouts.

Corollary. The reduction relation $\rightarrow$ has the Church-Rosser property.

Caution! The lattice structure given by the parallel moves theorem is on *derivations*, and *not* on terms. For instance, if we consider the system R consisting solely of the rules $I(x) \rightarrow x$ and $J(x) \rightarrow x$, the following derivations diagram shows that the terms $I(J(K))$ and $J(I(K))$ do not possess a l.u.b.



Note that this phenomenon may be traced to the existence of two non-equivalent derivations between $I(I(K))$ and $I(K)$. This shows that the categorical viewpoint is the right one here: we need to talk in terms of arrows, not just relations between terms.

The standardization theorem. It is always possible to compute in an outside-in manner.

We do not have the space here to explain in a rigorous manner what *outside-in* exactly means. We just remark that this may be more complicated than merely reducing the leftmost-outermost redex, i.e. the redex minimum in the total ordering on occurrences defined by $u <_{LO} v$ iff either $u < v$, or else there exist $u' \leq u$ and $v' \leq v$ with $u' <_L v'$. For instance, with R consisting of $F(x, D) \rightarrow C$, $B \rightarrow D$ and $\bot \rightarrow \bot$, the standard derivation going from $F(\bot, B)$ to its canonical form C is:

$$F(\bot, B) \rightarrow F(\bot, D) \rightarrow C$$

whereas the leftmost-outermost rule leads to a non-terminating loop.

This theorem suggests it can be used to define a *computation rule*, usable to drive an interpreter computing "lazily". However, this is not true, since the standard derivation in a derivation class is not simply a function of the starting term. For instance, consider Gustave's function, an example due to G.G. Berry; with R consisting of the 3 rules $F(0, 1, x) \rightarrow A$, $F(x, 0, 1) \rightarrow A$ and $F(1, x, 0) \rightarrow A$, nothing tells us where to compute next in a term $F(M_1, M_2, M_3)$. This example is just as bad as the classical "parallel-or" definition, an obviously non-sequential example ruled out here because we do not admit critical pairs.

Thus it is clear that we do not have strong enough syntactic restrictions on the left-hand sides of our rewrite rules to be able to define sequential computation rules. The key to such restrictions is to adapt the Kahn-Plotkin sequentiality theory [76] to define a notion of "needed" redex. This

leads to the notion of sequential term rewriting system. A further refinement, strong sequentiality, gives a decidable criterion which may be used to drive efficient interpreters which look for a needed redex in linear time, using a generalization to trees of the Knuth-Morris-Pratt string-matching algorithm [85]. This theory is completely explained in Huet-Lévy [70].

In practice, we obtain easy criterions for strong sequentiality in the particular cases of systems with constructors, and "left" systems such as systems of combinators definitions.

4 Natural deduction and λ-calculus

4.1 Proofs with variables: sequents

We now come back to the general theory of proof structures. We saw earlier that the Hilbert presentation of minimal logic was not very natural, in that the trivial theorem $A \to A$ necessitated a complex proof $S\ K\ K$. The problem is that in practice one does not use just proof terms, but *deductions* of the form

$$\Gamma \vdash A$$

where Γ is a set of (hypothetic) propositions.

Deductions are exactly proof terms *with variables*. Naming these hypothesis variables and the proof term, we write:

$$\{...[x_i : A_i]... \mid i \leq n\} \vdash M : A$$

with $V(M) \subseteq \{x_1, ..., x_n\}$. Such formulas are called *sequents*. Since this point of view is not very well-known, let us emphasize this constatation:

Sequents represent proof terms with variables.

Note that so far our notion of proof construction has not changed:

$\Gamma \vdash_\Sigma M : A$ iff $\vdash_{\Sigma \cup \Gamma} M : A$, i.e. the hypotheses from Γ are used as supplementary axioms, in the same way that in the very beginning we have defined $T(\Sigma, V)$ as $T(\Sigma \cup V)$.

4.2 The deduction theorem

This theorem, fundamental for doing proofs in practice, gives an equivalence between proof terms with variables and functional proof terms:

$$\Gamma \cup \{A\} \vdash B \quad \Leftrightarrow \quad \Gamma \vdash A \to B$$

That is, in our notations:

a) $\Gamma \vdash M : A \to B \quad \Rightarrow \quad \Gamma \cup \{x : A\} \vdash (M\ x) : B$

This direction is immediate, using App, i.e. Modus Ponens.

b) $\Gamma \cup \{x : A\} \vdash M : B \quad \Rightarrow \quad \Gamma \vdash [x]M : A \to B$

where the term $[x]M$ is given by the following algorithm.

Schönfinkel's abstraction algorithm:

$$\begin{array}{lll} [x]\,x & = & I \qquad (= S\ K\ K) \\ [x]\,y & = & K\ y \qquad (y \neq x) \\ [x]\,(M\ N) & = & S\ [x]M\ [x]N \end{array}$$

Note that this algorithm motivates the choice of combinators S and K (and optionally I). Again we stress a basic observation:

Schönfinkel's algorithm is the essence of the proof of the deduction theorem.

Now let us consider the rewriting system R defined by the rules Def_K and Def_S, optionally supplemented by:

$$Def_I : \ I\ x\ =\ x$$

and let us write $\triangleright$ for the corresponding reduction relation.

Fact. $\quad ([x]M\ \ N) \quad \triangleright^* \quad M[x \leftarrow N].$

We leave the proof of this very important property to the reader. The important point is that the abstraction operation, together with the application operator and the reduction $\triangleright$, define a *substitution* machinery. We shall now use this idea more generally, in order to internalize the deduction theorem in a basic calculus of functionality. That is, we forget the specific combinators S and K, in favor of abstraction seen now as a new term constructor.

4.3 λ calculus.

Here we give up Σ-terms in general, in favor of λ-terms constructed by 3 elementary operations:

x	*variable*
$(M\ N)$	*application*
$[x]M$	*abstraction*

This last case is usually written $\lambda x \cdot M$, whence the name λ-notation. The λ-notation is first a non-ambiguous notation for expressions denoting functions. For instance, the function of two arguments which computes sin of its first argument and adds it to cos of its second is written

$$[x]\,[y]\,sin(x) + cos(y)$$

The variables x and y are *bound* variables, that is they are dummies and their name does not matter, as long as there are no clashes. This defines a congruence of renaming of bound variables usually called α-conversion. Another method is to adopt de Bruijn's indexes, where variable names disappear in favor of positive natural numbers [15]. We define recursively the sets λ_n of λ-expressions valid in a context of length $n \geq 0$ as follows:

$$\begin{array}{rcll} \lambda_n & = & k & (1 \leq k \leq n) \\ & | & (M\ N) & (M, N \in \lambda_n) \\ & | & []M & M \in \lambda_{n+1}. \end{array}$$

Thus integer n refers to the variable bound by the n-th abstraction above it. For instance, the expression $[](1\ [](1\ 2))$ corresponds to $[x](x\ [y](y\ x))$. This example shows that, although more rigorous from a formal point of view, the de Bruijn naming scheme is not fit for human understanding, and we shall now come back to the more usual concrete notation with variable names.

The fact observed above is now edicted as a computation rule, usually called *β-reduction*. Let $\triangleright$ be the smallest relation on λ-expressions compatible with application and abstraction and such that:

$$([x]M\ \ N) \quad \triangleright \quad M[x \leftarrow N].$$

We call λ-calculus the λ-notation equipped with the β-reduction computation rule $\triangleright$. λ-calculus is the basic calculus of substitution, and β-reduction is the basic computation mechanism of functional programming languages. Here is an example of computation:

$$([x]\,[y]\,(x\ (y\ x))\ \ [u]\,(u\ u)\ \ [v]\,[w]\,v)$$

$$\triangleright\ ([y]\,([u]\,(u\ u)\ \ (y\ \ [u]\,(u\ u)))\ \ [v]\,[w]\,v)$$

$$\triangleright^2\ ([u]\,(u\ u)\ \ [w]\,[u]\,(u\ u))$$

$$\triangleright\ ([w]\,[u]\,(u\ u)\ \ [w]\,[u]\,(u\ u)) \quad \triangleright \quad [u]\,(u\ u)$$

We briefly sketch here the syntactic properties of λ-calculus. Similarly to the theory developed above, the notion of residual can be defined. However, the residuals of a redex may not always be disjoint, and thus the theory of derivations is more complex. However the parallel moves lemma still holds, and thus the Church-Rosser property is also true. Finally, the standardization theorem holds, and here it means that it is possible to compute in a leftmost-outermost fashion. These results, and more details, in particular the precise conditions under which β-reduction simulates combinatory logic calculus, are precisely stated in Barendregt [4].

We finally remark that λ-calculus computations may not always terminate. For instance, with $\Delta = [u]\,(u\ u)$ and $\bot = (\Delta\ \Delta)$, we get $\bot \triangleright \bot \triangleright \ldots$ A more interesting example is given by

$$Y = [f]\,([u]\,(f\ (u\ u))\ \ [u]\,(f\ (u\ u)))$$

since $(Y\ f) \triangleright^* (f\ (Y\ f))$ shows that Y defines a general fixpoint operator. This shows that (full) λ-calculus is inconsistent with logic. What could $(fix\ \neg)$ mean? As usual with such paradoxical situations, it is necessary to introduce *types* in order to stratify the definable notions in a logically meaningful way. Thus, the basic inconsistency of Church's λ-calculus, shown by Rosser, led to Church's theory of types [22]. On the other hand, λ-calculus as a pure computation mechanism is perfectly meaningful, and Strachey prompted Scott to develop the theory of reflexive domains as a model theory for full λ-calculus. But let us first investigate the typed universe.

4.4 Gentzen's system N of natural deduction

The idea of λ-notation proofs underlies Gentzen's natural deduction inference rules [48], where *App* is called $\rightarrow$-elim and *Abs* is called $\rightarrow$-intro. The role of variables is taken by the base sequents:

$$Axiom_A \ : \ A \vdash A$$

together with the structural *thinning* rule:

$$Thinning \ : \ \frac{\Gamma \vdash B}{\Gamma \cup \{A\} \vdash B}$$

which expresses that a proof may not use all of the hypotheses. Gentzen's remaining rules give types to proofs according to propositions built as functor terms, each functor corresponding to a propositional connective. The main idea of his system is that inference rules should not be arbitrary,

but should follow the functor structure, in explaining in a uniform fashion how to *introduce* a functor, and how to *eliminate* it. For instance, minimal logic is obtained with $\Phi = \{\rightarrow\}$, and the rules of $\rightarrow$ *–intro* and $\rightarrow$ *–elim*, that is:

$$Abs \quad : \quad \frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \rightarrow B}$$

$$App \quad : \quad \frac{\Gamma \vdash A \rightarrow B \quad \Delta \vdash A}{\Gamma \cup \Delta \vdash B}$$

Now, the β-reduction of λ-calculus corresponds to cut-elimination, i.e. to proof-simplification. Reducing a redex corresponds to eliminating a detour in the demonstration, using an intermediate lemma. But now we have termination of this normalization process, that is the relation $\rightarrow$ is Nœtherian on valid proofs. This result is usually called *strong normalization* in proof theory. A full account of this theory is given in Stenlund [150].

Minimal logic can then be extended by adding more functors and corresponding inference rules. For instance, conjunction $\wedge$ is taken into account by the intro rule:

$$Pair \quad : \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \cup \Delta \vdash A \wedge B}$$

which, from the types point of view, may be considered as product formation, and by the two elim rules:

$$Fst \quad : \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$Snd \quad : \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

corresponding to the two projection functions. This corresponds to building-in a λ-calculus with pairing. Generalizing the notion of redex (cut) to the configuration of a connective intro, immediately followed by elim of the same connective, we get new computation rules:

$$Fst(Pair(x, y)) \quad \triangleright \quad x$$

$$Snd(Pair(x, y)) \quad \triangleright \quad y$$

and the Nœtherian property of $\triangleright$ still holds. We shall not develop further Gentzen's system. We just remark:

(a) More connectives, such as disjunction, can be added in a similar fashion. It is also possible to give rules for quantifiers, although we prefer to differ this topic until we consider dependent bindings.

(b) Gentzen originally considered natural deduction systems for meta-mathematical reasons, namely to prove their consistency. He considered another presentation of sequent inference rules, the L system, which possesses the subformula property (i.e. the result type of every operator is formed of subterms of the argument types), and is thus trivially consistent. Strong normalization in this context was the essential technical tool to establish the equivalence of the L and the N systems. Of course, according to Gödel's theorem, this does not establish *absolute* consistency of the logic, but relativizes it to a carefully identified troublesome point, the proof of termination of some reduction relation. This has the additional advantage to provide a hierarchy of strength of inference systems, classified according to the ordinal necessary to consider for the termination proof.

(c) All this development concerns so called *intuitionistic* logic, where operators (inference rules) are deterministic. It is possible to generalize the inference systems to *classical* logic, using a generalized notion of sequent $\Gamma \vdash \Delta$, where the right part Δ is also a set of propositions. It is possible to explain the composition of such non-deterministic operators, which leads to Gentzen's systems NK and LK (Klassical logic!). Remark that the analogue of the unification theorem above gives then precisely Robinson's resolution principle for general clauses [140].

(d) The categorical viewpoint fits nicely these developments. This point of view is completely developed in Szabo [152]. The specially important connections between λ-calculus, natural deduction proofs and cartesian closed categories are investigated in [98,121,87,143,35,68]. Further readings on natural deduction proof theory are Prawitz [130] and Dummett [41]. The connection with recursion theory is developed in Kleene [82] and an algebraic treatment of these matters is given in Rasiowa-Sikorski [134].

4.5 Programming languages, recursion

The design of programming languages such as ALGOL 60 was greatly influenced by λ-calculus. In 1966 Peter Landin wrote a landmark article setting the stage for coherent design of powerful functional languages in the λ-calculus tradition [89]. The core language of his proposal, ISWIM (If you see what I mean!) meant λ-calculus, with syntactically sugared versions of the β-redex $([x]M\ N)$, namely *let* $x = N$ *in* M and M *where* $x = N$ respectively. His language followed the static binding rules of λ-calculus. For instance, after the declarations:

$$let\ f\ x\ =\ x + y\ where\ y = 1;$$

$$let\ y\ =\ 2;$$

the evaluation (reduction) of expression $(f\ 1)$ leads to value 2, as expected. Note that in contrast languages such as LISP [107], although bearing some similarity with the λ-notation, implement rather *dynamic* binding, which would result in the example above in the incorrect result 3. This discrepancy has led to heated debates which we want to avoid here, but we remark that static binding is generally considered safer and leads to more efficient implementations where compilation is consistent with interpretation. However, ISWIM is not completely faithful to λ-calculus in one respect: its implementation does not follow the outside-in normal order of evaluation corresponding to the standardization theorem. Instead it follows the inside-out applicative order of evaluation demanding the arguments to be evaluated before a procedure is called. In the ALGOL terminology, ISWIM follows *call by value* instead of *call by name*.

The development of natural deduction as typed λ-calculus fits the design of an ISWIM-based language with a type discipline. We shall call this language ML , which stands for "meta-language", in the spirit of LCF's ML [54,53]. For instance, we get a core $\mathtt{ML}_0$ by considering minimal logic, with $\rightarrow$ interpreted as functionality, and further constant functors added for basic types such as *triv*, *bool*, *int* and *string*.

Adding products we get a language $\mathtt{ML}_1$ where types reflect an intuitionistic predicate calculus with $\rightarrow$ and $\wedge$. We may define functions on a pattern argument formed by pairing, such as:

$$let\ fst(x, y)\ =\ x$$

and the categorical analogue are the so-called *cartesian closed categories* (CCCs). Adding sums lead to Bi-CCC's with co-product. The corresponding $\mathtt{ML}_2$ primitives are inl, inr, outl, outr and isl, with obvious meaning. So far all computations terminate, since the corresponding reduction relations are Nœtherian.

However such a programming language is too weak for practical use, since recursion is missing. Adding recursion operators may be done in a stratified manner, as presented in Gödel's system T [51], or in a completely general way in $\mathtt{ML}_3$, where we allow a "letrec" construct permitting arbitrary recursive definitions, such as:

$$letrec\ fact\ n\ =\ if\ n = 0\ then\ 1\ else\ n * (fact\ (n-1))$$

But then we loose the termination of computations, since it is possible to write un-founded definitions such as

$$letrec\ absurd\ x\ =\ absurd\ x.$$

Furthermore, because ML follows the applicative order of evaluation we may get looping computations in cases where a λ-calculus normal form exists, such as for

$$let\ \ f\ x\ =\ 0\ \ in\ \ f\ (absurd\ x).$$

4.6 Polymorphism

We have polymorphic operators (inference rules) at the meta level. It seems a good idea to push polymorphism to the object level, for functions defined by the user as λ-expressions. To this end, we introduce bindings for type variables. This idea of type quantification corresponds to allowing proposition quantifiers in our propositional logic. First we allow a universal quantifier in prenex position. That is, with $T_0 = T(\Phi, V)$, we now introduce *type schemas* in $T_1 = T_0 \cup \forall\alpha\ T_1$, $\alpha \in V$. A (type) term in T_1 has thus both free and bound variables, and we write $FV(M)$ and $BV(M)$ for the sets of free (respectively bound) variables.

We now define *generic instanciation*.
Let $\tau = \forall\alpha_1 ... \alpha_m \cdot \tau_0 \in T_1$ and $\tau' = \forall\beta_1 ... \beta_n \cdot \tau_0' \in T_1$. We define $\tau' \geq_G \tau$ iff $\tau_0' = \sigma(\tau_0)$ with $D(\sigma) \subseteq \{\alpha_1, ..., \alpha_m\}$ and $\beta_1 \notin FV(\tau)$ $(1 \leq i \leq n)$. Remark that $\geq$ acts on FV whereas $\geq_G$ acts on BV. Also note

$$\tau' \geq_G \tau \ \Rightarrow\ \sigma(\tau') \geq_G \sigma(\tau)$$

We now present the Damas-Milner inference system for polymorphic λ-calculus [39]. In what follows, a sequent hypothesis A is assumed to be a list of specifications $x_i : \tau_i$, with $\tau_i \in T_1$, and we write $FV(A) = \bigcup_i FV(\tau_i)$.

$$TAUT\ :\ A \vdash x\ :\ \alpha \quad (x : \alpha \in A)$$

$$INST\ :\ \frac{A \vdash M\ :\ \alpha}{A \vdash M\ :\ \beta} \quad \alpha \leq_G \beta)$$

$$GEN\ :\ \frac{A \vdash M\ :\ \tau}{A \vdash M\ :\ \forall\alpha \cdot \tau} \quad (\alpha \notin FV(A)$$

$$APP\ :\ \frac{A \vdash M\ :\ \tau' \to \tau \quad A \vdash N\ :\ \tau'}{A \vdash (M\ N)\ :\ \tau}$$

$$ABS\ :\ \frac{A \cup \{x : \tau'\} \vdash M\ :\ \tau}{A \vdash [x]M\ :\ \tau' \to \tau}$$

$$LET\ :\ \frac{A \vdash M\ :\ \tau' \quad A \cup \{x : \tau'\} \vdash N\ :\ \tau}{A \vdash let\ x = M\ in\ N\ :\ \tau}$$

For instance, it is an easy exercise to show that

$$\vdash \mathit{let}\ i\ =\ [x]x\ \mathit{in}\ (i\ i)\ :\ \alpha \to \alpha.$$

The above system may be extended without difficulty by other functors such as product, and by other ML contructions such as letrec. Actually every ML compiler contains a typechecker implementing implicitly the above inference system. For instance, with the unary functor *list* and the following ML primitives: [] : $(\mathit{list}\ \alpha)$, $\mathit{cons} : \alpha \times (\mathit{list}\ \alpha)$ (written infix as a dot), $\mathit{hd} : (\mathit{list}\ \alpha) \to \alpha$ and $\mathit{tl} : (\mathit{list}\ \alpha) \to (\mathit{list}\ \alpha)$, we may define recursively the map functional as:

$$\mathit{letrec}\ \mathit{map}\ f\ l\ =\ \mathit{if}\ l = []\ \mathit{then}\ []\ \mathit{else}\ (f\ (\mathit{hd}\ l))\ \cdot\ \mathit{map}\ f\ (\mathit{tl}\ l)$$

and we get as its type:

$$\vdash\ \mathit{map} : (\alpha \to \beta) \to (\mathit{list}\ \alpha) \to (\mathit{list}\ \beta).$$

Of course the ML compiler does not implement directly the inference system above, which is non-deterministic because of rules $INST$ and GEN. It uses unification instead, and thus computes deterministically a principal type, which is minimum with respect to $\leq_G$:

Milner's Theorem. Every typable expression of the polymorphic λ-calculus possesses a principal type, minimum with respect to generic instanciation.

We obtain ML_4 by restricting ML_3 to the type system above. ML_4 is a strongly typed programming language, where type inference is possible because of the above theorem: the user need not write type specifications. The compiler of the language does more than typechecking, since it actually performs a proof synthesis. Types disappear at run time, but because of the type analysis no dynamic checks are needed to inforce the consistency of data operations, and this allows fast execution of ML programs. ML is a generic name for languages of the ML family. For instance, by adding to ML_4 exceptions, abstract data types (permitting in particular user-defined functors) and references, one gets approximately the meta-language of the LCF proof assistant [54]. By adding record type declarations (i.e. labeled sums and products) one gets L. Cardelli's ML [20]. By adding constructor types, pattern-matching and concrete syntax, we get the Le_ML language under development in the Formel project [33]. A more complete language, including modules, is under design as Standard ML [112]. It is to be hoped than less than 700 iterations of the language design will be necessary before the ultimate Standard ML is agreed upon [89]! Current research topics on the design of ML -like languages are incorporation of object-oriented features allowing subtypes, remanent data structures and bitmap operations [21], and "lazy evaluation" permitting streams and ZF expressions [162,118].

Note on the relationship between ML and λ-calculus. First, ML uses so-called call by value implementation of procedure call, corresponding to innermost reduction, as opposed to the outermost regime of the standard reduction. Lazy evaluation permits standard reductions, but closures (i.e. objects of a functional type $\alpha \to \beta$) are *not* evaluated. Finally, types in ML serve for insuring the integrity of data operations, but still allow infinite computations by non-terminating recursions.

4.7 The limits of ML 's polymorphism

Consider the following ML definition:

$$\mathit{letrec}\ \mathit{power}\ n\ f\ u\ =\ \mathit{if}\ n = 0\ \mathit{then}\ u$$

$$\mathit{else}\ f\ (\mathit{power}\ (n-1)\ f\ u)$$

of type $nat \to (\alpha \to \alpha) \to (\alpha \to \alpha)$. This function, which associates to natural n the polymorphic iterator mapping function f to the n-th power of f, may be considered a coercion operator between ML 's internal naturals and Church's representation of naturals in pure λ-calculus [23]. Let us recall briefly this representation. Integer 0 is represented as the projection term $[f]\,[u]\,u$. Integer 1 is $[f]\,[u]\,(f\ u)$. More generally, n is represented as the functional $\bar{n}$ iterating a function f to its n-th power:

$$\bar{n} \;=\; [f]\,[u]\,(f\ (f\ \ldots(f\ u)\ldots))$$

and the arithmetic operators may be coded respectively as:

$$n + m \;=\; [f]\,[u]\,(n\ f\ (m\ f\ u))$$

$$n \times m \;=\; [f]\,(n\ (m\ f))$$

$$n^m \;=\; (m\ n)$$

For instance, with $\bar{2} \;=\; [f]\,[u]\,(f\ (f\ u))$, we check that $\bar{2} \times \bar{2}$ converts to its normal form $\bar{4}$.

We would like to consider a type

$$NAT \;=\; \forall\alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha)$$

and be able to type the operations above as functions of type $NAT \to NAT \to NAT$. However the notion of polymorphism found in ML does not support such a type, it allows only the weaker

$$\forall\alpha \cdot ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\alpha \to \alpha) \to (\alpha \to \alpha))$$

which is inadequate, since it forces the *same* generic instanciation of NAT in the two arguments.

Warning. These preliminary notes are very sketchy from now on. A future version will cover the topics below in greater depth.

4.8 Girard's second order λ-calculus.

The example above suggests using the universal type quantifier *inside* type formulas. We thus consider a functor alphabet based on one binary $\to$ constructor and one quantifier $\forall$. We shall now consider a λ-calculus with such types, which we shall call *second-order λ-calculus*, owing to the fact that the type language is now a second-order propositional logic, with propositional variables explicitly quantified. Such a calculus was proposed by J.Y. Girard [49,50], and independently discovered by J. Reynolds [136].

Girard proved the main properties of the calculus:

Girard's theorem. Second-order λ-calculus admits strong normalization.

Corollary. Second-order natural deduction is consistent.

Girard used this last result to show the consistency of analysis.

Second-order λ-calculus is a very powerful language. Most usual data structures may be represented as types. Furthermore, it captures a large class of total recursive functions (precisely, all the functions *prouvably* total in second-order arithmetic). It may seriously be considered as a candidate for the foundations of powerful programming languages, where *recursion* is replaced by *iteration*. But the price we pay by extending polymorphism in this drastic fashion is that the notion of principal type is lost. Type synthesis is possible only in easy cases, and thus in general the programmer has to specify the types of its data.

Further discussions on the second-order λ-calculus may be found in [108,46,91,7].

5 Dependent types

5.1 Quantification

So far we have dealt only with types as propositions of some (intuitionistic) propositional logic. We shall now consider stronger logics, where it is possible to have statements depending upon variables that are λ-bound. We shall continue our identification of propositions and types, and thus consider a first-order statement such as $\forall x \in E \cdot P(x)$ as a product-forming type $\Pi_{x \in E} P(x)$.

We shall call such types *dependent*, in that it is now possible to declare a variable of a type which depends on the binding of some previously bound variable. Let us first of all remark that such types are absolutely necessary for practical programming purposes. For instance, a matrix manipulation procedure should have a declaration prefix of the type:

$$[n : nat]\,[matrix : array(n)]$$

where the second type *depends* on the dimension parameter. PASCAL programmers know that the lack of such declarations in the language is a serious hindrance.

We shall not develop first-order notions here, and shall rather jump directly to calculi based on higher-order logic.

5.2 Martin-Löf's Intuitionistic Theory of Types

P. Martin-Löf has been developing for the last 10 years a higher-order intuitionist logic based on a theory of types, allowing dependent sums and products [104,105,106]. His theory is not explicitly based on λ-calculus, but it is formulated in the spirit of natural deduction, with introduction and elimination rules for the various type constructors. Consistency is inferred from semantic considerations, with a model theory giving an analysis of the normal forms of elements of a type, and of the equality predicate for each type.

Martin-Löf's system has been advocated as a good candidate for the description and validation of computer programs, and is an active topic of research by the Göteborg Programming Methodology group [117,119,120]. A particularly ambitious implementation of Martin-Löf's system and extensions is under way at Cornell University, under the direction of R. Constable [25,26,132].

5.3 de Bruijn's AUTOMATH languages

The mathematical language AUTOMATH has been developed and implemented by the Eindhoven group, under the direction of prof. N.G. de Bruijn [14,16,18]. AUTOMATH is a λ-calculus with types that are themselves λ-expressions. It is based on the natural idea that λ-binding and universal instanciation are similar substitution operations. Thus in AUTOMATH there is only one binding operation, used both for parameter abstraction and product instanciation. The meta-theory of the various languages of the AUTOMATH family are investigated in [113,38,75]. The most notable success of the AUTOMATH effort has been the translation and mechanical validation of Landau's Grundlagen [74].

5.4 A Calculus of Constructions.

AUTOMATH established the correct linguistic foundations for higher-order natural deduction. Unfortunately, it did not allow Girard's second-order types, and probably for this reason was never considered under the programming language aspect. Th. Coquand showed that a slight extension of

the notation allowed the incorporation of Girard's types to AUTOMATH in a natural manner [27]. Coquand showed by a strong normalization theorem that the formalism is consistent. Experiments with an implementation of the calculus showed that it is well adapted to expressing naturally and concisely mathematical proofs and computer algorithms [29]. Variations on this calculus are under development [30,31].

Conclusion

We have presented in these notes a uniform account of logic and computation theory, based on proof theory notions, and most importantly on the Curry-Howard correspondance between propositions and types [37,59].

These notes are based on a course given at the Advanced School of Artificial Intelligence, Vigneu, France, in July 1985. An extended version is in preparation.

References

[1] A. Aho, J. Hopcroft, J. Ullman. "The Design and Analysis of Computer Algorithms." Addison-Wesley (1974).

[2] P. B. Andrews. "Resolution in Type Theory." Journal of Symbolic Logic **36,3** (1971), 414–432.

[3] P. B. Andrews, D. A. Miller, E. L. Cohen, F. Pfenning. "Automating higher-order logic." Dept of Math, University Carnegie-Mellon, (Jan. 1983).

[4] H. Barendregt. "The Lambda-Calculus: Its Syntax and Semantics." North-Holland (1980).

[5] E. Bishop. "Foundations of Constructive Analysis." McGraw-Hill, New-York (1967).

[6] E. Bishop. "Mathematics as a numerical language." Intuitionism and Proof Theory, Eds. J. Myhill, A.Kino and R.E.Vesley, North-Holland, Amsterdam, (1970) 53–71.

[7] C. Böhm, A. Berarducci. "Automatic Synthesis of Typed Lambda-Programs on Term Algebras." Unpublished manuscript, (June 1984).

[8] R.S. Boyer, J Moore. "The sharing of structure in theorem proving programs." Machine Intelligence **7** (1972) Edinburgh U. Press, 101–116.

[9] R. Boyer, J Moore. "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory." 5th International Joint Conference on Artificial Intelligence, (1977) 511–519.

[10] R. Boyer, J Moore. "A Computational Logic." Academic Press (1979).

[11] R. Boyer, J Moore. "A mechanical proof of the unsolvability of the halting problem." Report ICSCA-CMP-28, Institute for Computing Science, University of Texas at Austin (July 1982).

[12] R. Boyer, J Moore. "Proof Checking the RSA Public Key Encryption Algorithm." Report ICSCA-CMP-33, Institute for Computing Science, University of Texas at Austin (Sept. 1982).

[13] R. Boyer, J Moore. "Proof checking theorem proving and program verification." Report ICSCA-CMP-35, Institute for Computing Science, University of Texas at Austin (Jan. 1983).

[14] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[15] N.G. de Bruijn. "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[16] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[17] N.G. de Bruijn. "Some extensions of Automath: the AUT-4 family." Internal Automath memo M10 (Jan. 1974).

[18] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[19] M. Bruynooghe. "The Memory Management of PROLOG implementations." Logic Programming Workshop. Ed. Tarnlund S.A (July 1980).

[20] L. Cardelli. "ML under UNIX." Bell Laboratories, Murray Hill, New Jersey (1982).

[21] L. Cardelli. "Amber." Bell Laboratories, Murray Hill, New Jersey (1985).

[22] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic **5,1** (1940) 56–68.

[23] A. Church. "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J. (1941).

[24] A. Colmerauer, H. Kanoui, R. Pasero, Ph. Roussel. "Un système de communication homme-machine en francais." Rapport de recherche, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille (1973).

[25] R.L. Constable, J.L. Bates. "Proofs as Programs." Dept. of Computer Science, Cornell University. (Feb. 1983).

[26] R.L. Constable, J.L. Bates. "The Nearly Ultimate Pearl." Dept. of Computer Science, Cornell University. (Dec. 1983).

[27] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[28] Th. Coquand, G. Huet. "A Theory of Constructions." Preliminary version, presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (June 84).

[29] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[30] Th. Coquand, G. Huet. "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Colloque de Logique, Orsay (Juil. 1985).

[31] Th. Coquand, G. Huet. "A Calculus of Constructions." To appear, JCSS (1986).

[32] J. Corbin, M. Bidoit. "A Rehabilitation of Robinson's Unification Algorithm." IFIP 83, Elsevier Science (1983) 909-914.

[33] G. Cousineau, P.L. Curien and M. Mauny. "The Categorical Abstract Machine." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 50-64.

[34] P.L. Curien. "Combinateurs catégoriques, algorithmes séquentiels et programmation applicative." Thèse de Doctorat d'Etat, Université Paris VII (Dec. 1983).

[35] P. L. Curien. "Categorical Combinatory Logic." ICALP 85, Nafplion, Springer-Verlag LNCS 194 (1985).

[36] P.L. Curien. "Categorical Combinators, Sequential Algorithms and Functional Programming." Pitman (1986).

[37] H. B. Curry, R. Feys. "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[38] D. Van Daalen. "The language theory of Automath." Ph. D. Dissertation, Technological Univ. Eindhoven (1980).

[39] Luis Damas, Robin Milner. "Principal type-schemas for functional programs." Edinburgh University (1982).

[40] P.J. Downey, R. Sethi, R. Tarjan. "Variations on the common subexpression problem." JACM **27,4** (1980) 758–771.

[41] Dummett. "Elements of Intuitionism." Clarendon Press, Oxford (1977).

[42] F. Fages. "Formes canoniques dans les algèbres booléennes et application à la démonstration automatique en logique de premier ordre." Thèse de 3ème cycle, Univ. de Paris VI (Juin 1983).

[43] F. Fages. "Associative-Commutative Unification." Submitted for publication (1985).

[44] F. Fages, G. Huet. "Unification and Matching in Equational Theories." CAAP 83, l'Aquila, Italy. In Springer-Verlag LNCS **159** (1983).

[45] P. Flajolet, J.M. Steyaert. "On the Analysis of Tree-Matching Algorithms." in Automata, Languages and Programming 7th Int. Coll., Lecture Notes in Computer Science **85** Springer Verlag (1980) 208–219.

[46] S. Fortune, D. Leivant, M. O'Donnell. "The Expressiveness of Simple and Second-Order Type Structures." Journal of the Assoc. for Comp. Mach., **30,1**, (Jan. 1983) 151–185.

[47] G. Frege. "Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought." (1879). Reprinted in From Frege to Gödel, J. van Heijenoort, Harvard University Press, 1967.

[48] G. Gentzen. "The Collected Papers of Gerhard Gentzen." Ed. E. Szabo, North-Holland, Amsterdam (1969).

[49] J.Y. Girard. "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland (1970) 63-92.

[50] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[51] K. Gödel. "Uber eine bisher noch nicht benutze Erweitrung des finiten Standpunktes." Dialectica, **12** (1958).

[52] W. D. Goldfarb. "The Undecidability of the Second-order Unification Problem." Theoretical Computer Science, **13**, (1981) 225–230.

[53] M. Gordon, R. Milner, C. Wadsworth. "A Metalanguage for Interactive Proof in LCF." Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh (Sept. 1977).

[54] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[55] W. E. Gould. "A Matching Procedure for Omega Order Logic." Scientific Report 1, AFCRL 66-781, contract AF19 (628)-3250 (1966).

[56] J. Guard. "Automated Logic for Semi-Automated Mathematics." Scientific Report 1, AFCRL (1964).

[57] J. Herbrand. "Recherches sur la théorie de la démonstration." Thèse, U. de Paris (1930). In: Ecrits logiques de Jacques Herbrand, PUF Paris (1968).

[58] C. M. Hoffmann, M. J. O'Donnell. "Programming with Equations." ACM Transactions on Programming Languages and Systems, **4,1** (1982) 83–112.

[59] W. A. Howard. "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[60] G. Huet. "Constrained Resolution: a Complete Method for Type Theory." Ph.D. Thesis, Jennings Computing Center Report 1117, Case Western Reserve University (1972).

[61] G. Huet. "A Mechanization of Type Theory." Proceedings, 3rd IJCAI, Stanford (Aug. 1973).

[62] G. Huet. "The Undecidability of Unification in Third Order Logic." Information and Control **22** (1973) 257–267.

[63] G. Huet. "A Unification Algorithm for Typed Lambda Calculus." Theoretical Computer Science, **1.1** (1975) 27–57.

[64] G. Huet. "Résolution d'équations dans des langages d'ordre 1,2, ... ω." Thèse d'Etat, Université Paris VII (1976).

[65] G. Huet. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[66] G. Huet. "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm." JCSS **23,1** (1981) 11-21.

[67] G. Huet. "Initiation à la Théorie des Catégories." Polycopié de cours de DEA, Université Paris VII (Nov. 1985).

[68] G. Huet. "Cartesian Closed Categories and Lambda-Calculus." Category Theory Seminar, Carnegie-Mellon University (Dec. 1985).

[69] G. Huet, J.M. Hullot. "Proofs by Induction in Equational Theories With Constructors." JACM **25,2** (1982) 239–266.

[70] G. Huet, J.J. Lévy "Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems." Rapport Laboria 359, IRIA (Aug. 1979).

[71] G. Huet, D. Oppen. "Equations and Rewrite Rules: a Survey." In Formal Languages: Perspectives and Open Problems, Ed. Book R., Academic Press (1980).

[72] J.M. Hullot "Compilation de Formes Canoniques dans les Théories Equationnelles." Thèse de 3ème cycle, U. de Paris Sud (Nov. 80).

[73] Jean Pierre Jouannaud, Helene Kirchner. "Completion of a set of rules modulo a set of equations." (April 1984).

[74] L.S. Jutting. "A translation of Landau's "Grundlagen" in AUTOMATH." Eindhoven University of Technology, Dept of Mathematics (Oct. 1976).

[75] L.S. van Benthem Jutting. "The language theory of Λ_∞, a typed λ-calculus where terms are types." Unpublished manuscript (1984).

[76] G. Kahn, G. Plotkin. "Domaines concrets." Rapport Laboria 336, IRIA (Déc. 1978).

[77] J. Ketonen, J. S. Weening. "The language of an interactive proof checker." Stanford University (1984).

[78] J. Ketonen. "EKL-A Mathematically Oriented Proof Checker." 7th International Conference on Automated Deduction, Napa, California (May 1984). Springer-Verlag LNCS **170.**

[79] J. Ketonen. "A mechanical proof of Ramsey theorem." Stanford Univ. (1983).

[80] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[81] S.C. Kleene. "On the interpretation of intuitionistic number theory." J. Symbolic Logic **31** (1945).

[82] S.C. Kleene. "On the interpretation of intuitionistic number theory." J. Symbolic Logic **31** (1945).

[83] J.W. Klop. "Combinatory Reduction Systems." Ph. D. Thesis, Mathematisch Centrum Amsterdam (1980).

[84] D. Knuth, P. Bendix. "Simple word problems in universal algebras". In: Computational Problems in Abstract Algebra, J. Leech Ed., Pergamon (1970) 263–297.

[85] D.E. Knuth, J. Morris, V. Pratt. "Fast Pattern Matching in Strings." SIAM Journal on Computing **6,2** (1977) 323–350.

[86] G. Kreisel. "On the interpretation of nonfinitist proofs, Part I, II." JSL **16** (1952, 1953).

[87] J. Lambek. "From Lambda-calculus to Cartesian Closed Categories." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[88] J. Lambek and P. J. Scott. "Aspects of Higher Order Categorical Logic." Contemporary Mathematics **30** (1984) 145–174.

[89] P. J. Landin. "The next 700 programming languages." Comm. ACM **9,3** (1966) 157–166.

[90] Philippe Le Chenadec. "Formes canoniques dans les algèbres finiment présentées." Thèse de 3ème cycle, Univ. d'Orsay (Juin 1983).

[91] D. Leivant. "Polymorphic type inference." 10th ACM Conference on Principles of Programming Languages (1983).

[92] D. Leivant. "Structural semantics for polymorphic data types." 10th ACM Conference on Principles of Programming Languages (1983).

[93] J.J. Lévy. "Réductions correctes et optimales dans le λ-calcul." Thèse d'Etat, U. Paris VII (1978).

[94] S. MacLane. "Categories for the Working Mathematician." Springer-Verlag (1971).

[95] D. MacQueen, G. Plotkin, R. Sethi. "An ideal model for recursive polymorphic types." Proceedings, Principles of Programming Languages Symposium, Jan. 1984, 165–174.

[96] D. B. MacQueen, R. Sethi. "A semantic model of types for applicative languages." ACM Symposium on Lisp and Functional Programming (Aug. 1982).

[97] E.G. Manes. "Algebraic Theories." Springer-Verlag (1976).

[98] C. Mann. "The Connection between Equivalence of Proofs and Cartesian Closed Categories." Proc. London Math. Soc. **31** (1975) 289–310.

[99] A. Martelli, U. Montanari. "Theorem proving with structure sharing and efficient unification." Proc. 5th IJCAI, Boston, (1977) p 543.

[100] A. Martelli, U. Montanari. "An Efficient Unification Algorithm." ACM Trans. on Prog. Lang. and Syst. **4,2** (1982) 258–282.

[101] William A. Martin. "Determining the equivalence of algebraic expressions by hash coding." JACM **18,4** (1971) 549–558.

[102] P. Martin-Löf. "A theory of types." Report 71-3, Dept. of Mathematics, University of Stockholm, Feb. 1971, revised (Oct. 1971).

[103] P. Martin-Löf. "About models for intuitionistic type theories and the notion of definitional equality." Paper read at the Orléans Logic Conference (1972).

[104] P. Martin-Löf. "An intuitionistic Theory of Types: predicative part." Logic Colloquium 73, Eds. H. Rose and J. Sepherdson, North-Holland, (1974) 73–118.

[105] P. Martin-Löf. "Constructive Mathematics and Computer Programming." In Logic, Methodology and Philosophy of Science 6 (1980) 153–175, North-Holland.

[106] P. Martin-Löf. "Intuitionistic Type Theory." Studies in Proof Theory, Bibliopolis (1984).

[107] J. Mc Carthy. "Recursive functions of symbolic expressions and their computation by machine." CACM 3,4 (1960) 184–195.

[108] N. McCracken. "An investigation of a programming language with a polymorphic type structure." Ph.D. Dissertation, Syracuse University (1979).

[109] D.A. Miller. "Proofs in Higher-order Logic." Ph. D. Dissertation, Carnegie-Mellon University (Aug. 1983).

[110] D.A. Miller. "Expansion tree proofs and their conversion to natural deduction proofs." Technical report MS-CIS-84-6, University of Pennsylvania (Feb. 1984).

[111] R. Milner. "A Theory of Type Polymorphism in Programming." Journal of Computer and System Sciences 17 (1978) 348–375.

[112] R. Milner. "A proposal for Standard ML ." Report CSR-157-83, Computer Science Dept., University of Edinburgh (1983).

[113] R.P. Nederpelt. "Strong normalization in a typed λ calculus with λ structured types." Ph. D. Thesis, Eindhoven University of Technology (1973).

[114] R.P. Nederpelt. "An approach to theorem proving on the basis of a typed λ-calculus." 5th Conference on Automated Deduction, Les Arcs, France. Springer-Verlag LNCS 87 (1980).

[115] G. Nelson, D.C. Oppen. "Fast decision procedures based on congruence closure." JACM 27,2 (1980) 356–364.

[116] M.H.A. Newman. "On Theories with a Combinatorial Definition of "Equivalence"." Annals of Math. 43,2 (1942) 223–243.

[117] B. Nordström. "Programming in Constructive Set Theory: Some Examples." Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portmouth, New Hampshire (Oct. 1981) 141–154.

[118] B. Nordström. "Description of a Simple Programming Language." Report 1, Programming Methodology Group, University of Goteborg (Apr. 1984).

[119] B. Nordström, K. Petersson. "Types and Specifications." Information Processing 83, Ed. R. Mason, North-Holland, (1983) 915–920.

[120] B. Nordström, J. Smith. "Propositions and Specifications of Programs in Martin-Löf's Type Theory." BIT 24, (1984) 288–301.

[121] A. Obtulowicz. "The Logic of Categories of Partial Functions and its Applications." Dissertationes Mathematicae 241 (1982).

[122] M.S. Paterson, M.N. Wegman. "Linear Unification." J. of Computer and Systems Sciences **16** (1978) 158–167.

[123] L. Paulson. "Recent Developments in LCF : Examples of structural induction." Technical Report No 34, Computer Laboratory, University of Cambridge (Jan. 1983).

[124] L. Paulson. "Tactics and Tacticals in Cambridge LCF." Technical Report No 39, Computer Laboratory, University of Cambridge (July 1983).

[125] L. Paulson. "Verifying the unification algorithm in LCF." Technical report No 50, Computer Laboratory, University of Cambridge (March 1984).

[126] L. C. Paulson. "Constructing Recursion Operators in Intuitionistic Type Theory." Tech. Report 57, Computer Laboratory, University of Cambridge (Oct. 1984).

[127] G.E. Peterson, M.E. Stickel. "Complete Sets of Reduction for Equational Theories with Complete Unification Algorithms." JACM **28,2** (1981) 233–264.

[128] T. Pietrzykowski, D.C. Jensen. "A complete mechanization of ω-order type theory." Proceedings of ACM Annual Conference (1972).

[129] T. Pietrzykowski. "A Complete Mechanization of Second-Order Type Theory." JACM **20** (1973) 333–364.

[130] D. Prawitz. "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[131] D. Prawitz. "Ideas and results in proof theory." Proceedings of the Second Scandinavian Logic Symposium (1971).

[132] PRL staff. "Implementing Mathematics with the NUPRL Proof Development System." Computer Science Department, Cornell University (May 1985).

[133] W. V. Quine. "The problem of simplifying truth functions." Amer. Math. Monthly bf 59,8 (1952) 521–531.

[134] H. Rasiowa, R. Sikorski "The Mathematics of Metamathematics." Monografie Matematyczne tom **41**, PWN, Polish Scientific Publishers, Warszawa (1963).

[135] J. C. Reynolds. "Definitional Interpreters for Higher Order Programming Languages." Proc. ACM National Conference, Boston, (Aug. 72) 717–740.

[136] J. C. Reynolds. "Towards a Theory of Type Structure." Programming Symposium, Paris. Springer Verlag LNCS **19** (1974) 408–425.

[137] J. C. Reynolds. "Types, abstraction, and parametric polymorphism." IFIP Congress'83, Paris (Sept. 1983).

[138] J. C. Reynolds. "Polymorphism is not set-theoretic." International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).

[139] J. C. Reynolds. "Three approaches to type structure." TAPSOFT Advanced Seminar on the Role of Semantics in Software Development, Berlin (March 1985).

[140] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle." JACM **12** (1965) 32–41.

[141] J. A. Robinson. "Computational Logic: the Unification Computation." Machine Intelligence **6** Eds B. Meltzer and D.Michie, American Elsevier, New-York (1971).

[142] D. Scott. "Constructive validity." Symposium on Automatic Demonstration, Springer-Verlag Lecture Notes in Mathematics, **125** (1970).

[143] D. Scott. "Relating Theories of the Lambda-Calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[144] J.R. Shoenfield. "Mathematical Logic." Addison-Wesley (1967).

[145] R.E. Shostak "Deciding Combinations of Theories." JACM **31,1** (1985) 1–12.

[146] J. Smith. "Course-of-values recursion on lists in intuitionistic type theory." Unpublished notes, Göteborg University (Sept. 1981).

[147] J. Smith. "The identification of propositions and types in Martin-Lof's type theory : a programming example." International Conference on Foundations of Computation Theory, Borgholm, Sweden, (Aug. 1983) Springer-Verlag LNCS **158**.

[148] R. Statman. "Intuitionistic Propositional Logic is Polynomial-space Complete." Theoretical Computer Science **9** (1979) 67–72, North-Holland.

[149] R. Statman. "The typed Lambda-Calculus is not Elementary Recursive." Theoretical Computer Science **9** (1979) 73–81.

[150] S. Stenlund. "Combinators λ-terms, and proof theory." Reidel (1972).

[151] M.E. Stickel "A Complete Unification Algorithm for Associative-Commutative Functions." JACM **28,3** (1981) 423–434.

[152] M.E. Szabo. "Algebra of Proofs." North-Holland (1978).

[153] W. Tait. "A non constructive proof of Gentzen's Hauptsatz for second order predicate logic." Bull. Amer. Math. Soc. **72** (1966).

[154] W. Tait. "Intensional interpretations of functionals of finite type I." J. of Symbolic Logic **32** (1967) 198–212.

[155] W. Tait. "A Realizability Interpretation of the Theory of Species." Logic Colloquium, Ed. R. Parikh, Springer Verlag Lecture Notes **453** (1975).

[156] M. Takahashi. "A proof of cut-elimination theorem in simple type theory." J. Math. Soc. Japan **19** (1967).

[157] G. Takeuti. "On a generalized logic calculus." Japan J. Math. **23** (1953).

[158] G. Takeuti. "Proof theory." Studies in Logic **81** Amsterdam (1975).

[159] R. E. Tarjan. "Efficiency of a good but non linear set union algorithm." JACM **22,2** (1975) 215 225.

[160] R. E. Tarjan, J. van Leeuwen. "Worst-case Analysis of Set Union Algorithms." JACM **31,2** (1985) 245–281.

[161] A. Tarski. "A lattice-theoretical fixpoint theorem and its applications." Pacific J. Math. **5** (1955) 285–309.

[162] D.A. Turner. "Miranda: A non-strict functional language with polymorphic types." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 1–16.

[163] R. de Vrijer "Big Trees in a λ-calculus with λ-expressions as types." Conference on λ-calculus and Computer Science Theory, Rome, Springer-Verlag LNCS **37** (1975) 252–271.

[164] D. Warren "Applied Logic - Its use and implementation as a programming tool." Ph.D. Thesis, University of Edinburgh (1977).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique