Basic Proof Theory with Applications to Computation

Stanley S. Wainer

Leeds University UK

These lecture notes extend and revise an earlier joint paper: Wainer and Wallen (1992). The intention is to introduce some of the most fundamental concepts and results of proof theory, and to illustrate their relevance to the theory of computation. Each lecture contains one main theorem, given in its simplest and most basic form. The aim is to convey the essential ideas behind their proofs, keeping the syntactic detail to a minimum so as not to obscure their underlying structure. Some of these results are used elsewhere in this volume, sometimes implicitly and sometimes explicitly, but often in more sophisticated forms. As with car sales, the hope is that once having driven the basic version, the reader will quickly appreciate the more streamlined models.

1 Completeness of First Order Logic.

Classical first order predicate calculus (PC) is formulated here in the style of Tait (1968) with *finite sets* of formulas for sequents. It is kept "pure" (i.e. without function symbols) merely for the sake of technical simplicity. Later, in lecture 3, it will be refined to cope with *multiset* sequents in order to illuminate the rôle of the so-called structural inferences of *contraction* and *weakening* in proof-theoretic arguments.

The language of PC consists of

- Individual variables: $x_0, x_1, x_2, \ldots$;
- Predicate symbols: $P_0, \bar{P}_0, P_1, \bar{P}_1, \ldots$ in complementary pairs;
- Logical symbols: $\vee$ (or), $\wedge$ (and), $\exists$ (some), $\forall$ (all);
- Brackets for unique readability.

Formulas $A, B, \ldots$ are built up from atoms $P(x_{i_1}, \ldots, x_{i_k})$ and their complements $\bar{P}(x_{i_1}, \ldots, x_{i_k})$, by applying $\vee$, $\wedge$, $\exists x$ and $\forall x$.

Note that negation $\neg$ and implication $\rightarrow$ are not included as basic logical symbols. Negation is *defined* by De Morgan's Laws: $\neg P \equiv \bar{P}$; $\neg \bar{P} \equiv P$; $\neg(A \vee B) \equiv \neg A \wedge \neg B$; $\neg(A \wedge B) \equiv \neg A \vee \neg B$; $\neg \exists x A \equiv \forall x \neg A$; $\neg \forall x A \equiv \exists x \neg A$. Thus $\neg\neg A$ is just A. Implication $A \rightarrow B$ is *defined* to be $\neg A \vee B$. The reason for presenting logic in this way is that we will later want to exploit the duality between $\vee$ and $\wedge$, and between $\exists$ and $\forall$. The price paid is that we cannot present intuitionistic logic in this way, since De Morgan's Laws are not intuitionistically valid.

Derivability in PC. Rather than deriving single formulas we shall derive finite sets of them $\Gamma = \{A_1, A_2, \ldots, A_n\}$ meaning "A_1 or A_2 or ... or A_n". Γ, A denotes $\Gamma \cup \{A\}$. Γ, Δ denotes $\Gamma \cup \Delta$ etcetera.

The Proof-Rules of PC are (with any Γ):

$$\text{(Axioms)} \quad \Gamma, P(x_{i_1}, \ldots, x_{i_k}), \bar{P}(x_{i_1}, \ldots, x_{i_k})$$

$$(\vee)\ \frac{\Gamma, A_0, A_1}{\Gamma, (A_0 \vee A_1)} \qquad (\wedge)\ \frac{\Gamma, A_0 \quad \Gamma, A_1}{\Gamma, (A_0 \wedge A_1)}$$

$$(\exists)\ \frac{\Gamma, A(x')}{\Gamma, \exists x A(x)} \qquad (\forall)\ \frac{\Gamma, A(x')}{\Gamma, \forall x A(x)}\ \ x' \text{ not free in } \Gamma$$

$$(\text{Cut})\ \frac{\Gamma, C \quad \Gamma, \neg C}{\Gamma}\ \ C \text{ is the "cut formula".}$$

We write $\vdash_{PC} \Gamma$ to mean there is a PC-derivation of Γ from axioms.

Example.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\bar{P}(x')\ ,\ \forall x P(x)\ ,\ \bar{P}(x)\ ,\ P(x')}{(\bar{P}(x') \vee \forall x P(x))\ ,\ \bar{P}(x)\ ,\ P(x')}}{\exists x(\bar{P}(x) \vee \forall x P(x))\ ,\ \bar{P}(x)\ ,\ P(x')}}{\exists x(\bar{P}(x) \vee \forall x P(x))\ ,\ \bar{P}(x)\ ,\ \forall x P(x)}}{\exists x(\bar{P}(x) \vee \forall x P(x))\ ,\ (\bar{P}(x) \vee \forall x P(x))}}{\exists x(\bar{P}(x) \vee \forall x P(x))}$$

Note that $\{\exists x(\bar{P}(x) \vee \forall x P(x))\ ,\ \exists x(\bar{P}(x) \vee \forall x P(x))\}$ can be contracted to $\{\exists x(\bar{P}(x) \vee \forall x P(x))\}$ in the final inference because as *sets* of formulas they are the same.

Exercises.

- Show by induction on the "build-up" of the formula A, that for all Γ and all A, $\vdash_{PC} \neg A, A$.
- Show that if $\vdash_{PC} \Gamma, (A_0 \wedge A_1)$ then $\vdash_{PC} \Gamma, A_0$ and $\vdash_{PC} \Gamma, A_1$. (Hint: use induction on the height of the given derivation.)
- Similarly show that if $\vdash_{PC} \Gamma, \forall x A(x)$ then $\vdash_{PC} \Gamma, A(x')$. (Remember that one can always rename a variable to something "new".)
- Show that if $\vdash_{PC} \Gamma$ and $\Gamma \subset \Delta$ then $\vdash_{PC} \Delta$.

The Semantics of PC. An *interpretation* of PC gives a fixed meaning to all the formulas. It consists of a structure $\mathcal{M} = \langle M, P_0^M, P_1^M, P_2^M, \ldots \rangle$ where M is some non-empty set and each

P_k^M is a chosen relation on M which gives a fixed meaning to the predicate symbol P_k. Thus with respect to a given interpretation, and a given assignment $x_{i_1} := m_1, ..., x_{i_n} := m_n, ...$ of elements of M to the free variables, a formula $A(x_{i_1}, ..., x_{i_n})$ makes a statement about $\mathcal{M}$ which is either true or false. If it works out true under *all* possible interpretations $\mathcal{M}$ and *all* possible assignments of elements of M to its free variables, then A is said to be (logically or universally) valid. A finite set of formulas $\{A_1, \ldots, A_k\}$ is valid if the disjunction of its members is.

Theorem 1.1 (Completeness Theorem - Gödel 1930)

$$\vdash_{PC} \Gamma \textit{ if and only if } \Gamma \textit{ is valid.}$$

Proof.

For *soundness*: $\vdash_{PC} \Gamma \Rightarrow \Gamma$ is valid; simply note that the axioms are valid and each of the rules preserves validity.

For *adequacy*: $\not\vdash_{PC} \Gamma \Rightarrow \Gamma$ not valid; we try to construct a derivation tree for Γ by successively taking it to bits using the $(\vee)$, $(\wedge)$, $(\exists)$, $(\forall)$ rules backwards. We do not use Cut! Since we are assuming that Γ is not derivable, this procedure must fail to produce a derivation, and out of the failure we can construct an interpretation in which Γ is false. Hence Γ is not valid. It goes thus:

First write out Γ as an ordered sequence of formulas, starting with all the atoms if there are any. Let A denote the first non-atomic formula in the sequence and Δ the rest of Γ, thus

$$\Gamma = \text{atoms}, A, \Delta.$$

Now take A to bits using whichever one of the rules $(\vee)$, $(\wedge)$, $(\exists)$, $(\forall)$ applies. This produces one or (in the case of $\wedge$) two new sequences of formulas Γ' as follows:

- If $A \equiv (A_0 \vee A_1)$ then $\Gamma' = \text{atoms}, A_0, A_1, \Delta$ with A_0, A_1 reversed if A_1 is atomic and A_0 isn't;
- If $A \equiv (A_0 \wedge A_1)$ then $\Gamma'_i = \text{atoms}, A_i, \Delta$ for each $i = 0, 1$;
- If $A \equiv \forall x A_0(x)$ then $\Gamma' = \text{atoms}, A_0(x_j), \Delta$;

- If $A \equiv \exists x A_0(x)$ then $\Gamma' = \text{atoms}, A_0(x_k), \Delta, \exists x A_0(x)$;

where, in the $\forall$ case x_j is any new variable not already used previously in this iterated process, and in the $\exists$ case x_k is the first variable in the list $x_0, x_1, x_2, \ldots$ which has not already been used at a previous stage to witness the same formula $\exists x A_0(x)$.

Repeat this process to form $\Gamma, \Gamma', \Gamma'', \ldots$ and notice that each time, Γ (considered as a set) follows from Γ' by applying the corresponding rule. Note also that A will repeatedly come back under attention if it is of existential form. In this way we develop what looks like a derivation-tree for Γ with branching at applications of the $(\wedge)$ rule. But assuming Γ is not derivable in PC, there must be at least one branch on this tree – call it $\mathcal{B}$ – which either (a) terminates in a sequence of atoms only, but not a logical axiom, or (b) goes on forever!

From $\mathcal{B}$ we construct a "counter-interpretation",

$$\mathcal{M} = \left\langle N, P_0^M, P_1^M, P_2^M, \ldots \right\rangle$$

where $N = \{0, 1, 2, 3, \ldots\}$ and the relations P_j^M are defined as follows :

$P_j^M(i_1, \ldots, i_n) \Leftrightarrow_{\text{Def}}$ the atom $P_j(x_{i_1}, \ldots, x_{i_n})$ does not occur on $\mathcal{B}$.

Claim: Under this interpretation $\mathcal{M}$ and the assignment $x_i := i$ to each free variable, every formula A occurring on the branch $\mathcal{B}$ is false.

Proof of Claim. By induction on the build-up of formulas A occurring on $\mathcal{B}$, noticing that as the sequence $\Gamma, \Gamma', \Gamma'', \ldots$ is developed, every non-atomic formula on $\mathcal{B}$ will eventually "come under attention" as the first non-atomic formula at some stage.

(i) $A \equiv P_j(x_{i_1}, \ldots, x_{i_n})$ is false by definition.

(ii) $A \equiv \bar{P}_j(x_{i_1}, \ldots, x_{i_n})$ is false as its complement $P_j(x_{i_1}, \ldots, x_{i_n})$ cannot be on $\mathcal{B}$ (otherwise $\mathcal{B}$ would terminate in an axiom) and therefore $P_j(x_{i_1}, \ldots, x_{i_n})$ is true by definition.

(iii) $A \equiv A_0 \vee A_1$. Since A comes under attention at some stage along branch $\mathcal{B}$, both A_0 and A_1 also occur on $\mathcal{B}$. So by the induction hypothesis, both are false and hence so is A.

(iv) $A \equiv A_0 \wedge A_1$. Again, since A must come under attention at some stage, either A_0 or A_1 occurs on $\mathcal{B}$. So one of them is false and hence so is A.

(v) $A \equiv \forall x A_0(x)$. In this case $A_0(x_j)$ is also on $\mathcal{B}$ for one of the variables x_j. So $A_0(x_j)$ is false under the given interpretation and hence so is A, because the assignment $x := j$ fails to satisfy $A_0(x)$.

(vi) $A \equiv \exists x A_0(x)$. Then by the construction of $\mathcal{B}$, A comes under attention infinitely often and each time a "new" $A_0(x_k)$ is introduced. Therefore *every one* of

$$A_0(x_0), A_0(x_1), A_0(x_2), A_0(x_3), \ldots$$

occurs on $\mathcal{B}$, and they are all false. Hence A is false since there is no witnessing number which satisfies $A_0(x)$.

This completes the proof of the claim.

Now since the set Γ we started with occurs at the root of branch $\mathcal{B}$, it is false under the given interpretation and therefore is not valid. This completes the proof of the theorem.

Corollary 1.2 (Cut-Elimination Theorem)
If Γ is derivable in PC then it is derivable without use of Cut.

Semantic Proof. If $\vdash_{PC} \Gamma$ then by the soundness of PC, Γ is valid. But the proof of adequacy actually shows that if Γ is not derivable using only the rules $\vee, \wedge, \exists, \forall$, then Γ is not valid. Since Γ is valid, it must therefore be derivable without Cut.

Cut-Elimination for Theories. Suppose one wanted to make PC-derivations from certain additional *non-logical* axioms $NLAX$ describing a particular data type, for example the natural numbers N described by the Peano axioms plus the principle of induction. Then

$$NLAX \vdash_{PC} A$$

would be equivalent to requiring

$$\vdash_{PC} NLAX \rightarrow A.$$

Although this latter derivation has a cut-free proof in PC, we nevertheless need Cut in order to derive the formula A itself from $NLAX$ as follows:

$$\frac{NLAX \quad NLAX \rightarrow A}{A}$$

Thus in the presence of non-logical axioms, we cannot expect to have (full) Cut-Elimination. Often, however, we will be able to use more constructive Cut-Elimination methods in order to keep the Cuts down to "manageable levels". We shall see later how this can be done, in lectures 3 and 4.

2 Natural Deduction and Typed λ-Calculus.

In natural deduction ND a single formula is proved at a time, rather than a finite set of them as in the previous section. One starts with assumptions and builds derivations using the ND-rules which now come in pairs – an introduction and an elimination rule for each logical symbol. We shall concentrate on the minimal set of connectives $\wedge$, $\rightarrow$ and $\forall$. Thus for example the $\wedge$-rules are labelled $(\wedge I)$ and $(\wedge E)$. In the $(\rightarrow I)$-rule, one or more occurrences of an assumption A used in proving B, may be discharged or cancelled upon deriving $A \rightarrow B$. The discharge of A is denoted by enclosing it in brackets thus $[A]$.

The Curry-Howard Correspondence, see Howard (1980).

Each ND-derivation of a formula A in the logic based on $\wedge$, $\rightarrow$, $\forall$ has an associated 'formulas as types' representation, as a typed λ-expression t^A built up according to the rules displayed on the next page, where the individual variables z and terms a of the logic are assigned the 'ground-type' 0. Nowadays one usually signifies the type of a term t by writing $t : A$ instead of t^A but we shall stick to using the superscript, occasionally suppressing it altogether when the context makes the type clear.

	ND-Rules	λ-Expressions
(Assume)	A	Variable x^A
$(\wedge I)$	$\dfrac{\begin{array}{cc}\vdots t_0 & \vdots t_1 \\ A & B\end{array}}{A \wedge B}$	$\langle t_0^A, t_1^B \rangle^{A \wedge B}$
$(\wedge E)_0$	$\dfrac{\begin{array}{c}\vdots t \\ A \wedge B\end{array}}{A}$	$(t^{A \wedge B} 0)^A$
$(\wedge E)_1$	$\dfrac{\begin{array}{c}\vdots t \\ A \wedge B\end{array}}{B}$	$(t^{A \wedge B} 1)^B$
$(\rightarrow I)$	$\dfrac{\begin{array}{c}[A] \\ \vdots t \\ B\end{array}}{A \rightarrow B}$	$(\lambda x^A . t^B)^{A \rightarrow B}$
$(\rightarrow E)$	$\dfrac{\begin{array}{cc}\vdots t & \vdots s \\ A \rightarrow B & A\end{array}}{B}$	$(t^{A \rightarrow B} s^A)^B$
$(\forall I)$	$\dfrac{\begin{array}{c}\vdots t \\ A(z)\end{array}}{\forall x . A(z)}$	$(\lambda z^0 . t^{A(z)})^{\forall z . A}$
$(\forall E)$	$\dfrac{\begin{array}{c}\vdots t \\ \forall z . A(z)\end{array}}{A(a)}$	$(t^{\forall z . A} a^0)^{A(a)}$

In $(\forall I)$, z should not be free in uncancelled assumptions.

Example. The ND-derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall z.(A(z)\to B(z))]}{A(z)\to B(z)} \quad \dfrac{[\forall z.A(z)]}{A(z)}}{B(z)}}{\forall z.B(z)}}{\forall z.A(z)\to\forall z.B(z)}}{\forall z.(A(z)\to B(z))\to(\forall z.A(z)\to\forall z.B(z))}$$

has corresponding λ-expression:

$$\lambda x^{\forall z(A(z)\to B(z))}.\lambda y^{\forall z A(z)}.\lambda z^0.((xz)(yz))^{B(z)}.$$

Note how in the example each representation can be decoded from the other.

Exercises. Construct ND-derivations for the following formulas and write out the λ-expressions that correspond to your derivations.

1. $((A\to B)\wedge(B\to C))\to(A\to C)$
2. $\forall z.A(z)\to(\forall z.B(z)\to\forall z.(A(z)\wedge B(z)))$.

2.1 Embedding Classical Logic in ND.

First introduce a new symbol $\bot$ for falsity and define negation by $\neg A\equiv A\to\bot$, disjunction by $A\vee B\equiv\neg(\neg A\wedge\neg B)$, and existence by $\exists z.A\equiv\neg\forall z.\neg A$. The system CND is then obtained by adding the classical falsity rule:

$$\dfrac{\begin{array}{c}[\neg A]\\ \vdots\\ \bot\end{array}}{A}$$

Note that this is equivalent to adding all "stability axioms" $\neg\neg A\to A$. It is in fact sufficient to add these for atomic A only.

Exercises. Show that the following are derived rules of CND:

$$\frac{A_i}{A_0 \vee A_1} \qquad\qquad \frac{A(a)}{\exists z.A(z)}$$

$$\frac{A_0 \vee A_1 \qquad \begin{array}{c}[A_0]\\ \vdots\\ C\end{array} \quad \begin{array}{c}[A_1]\\ \vdots\\ C\end{array}}{C} \qquad\qquad \frac{\exists z.A(z) \qquad \begin{array}{c}[A(z)]\\ \vdots\\ C\end{array}}{C}$$

where in the final $(\exists E)$ rule, the variable z is not free in any other assumptions upon which C depends.

Theorem 2.1 *If* $\vdash_{PC} A_1, A_2, \ldots, A_n$ *then* $\vdash_{CND} A_1 \vee A_2 \vee \ldots \vee A_n$.

The proof is left as a laborious exercise! Check all of the ways in which $\vdash_{PC} A_1, A_2, \ldots, A_n$.

2.2 Normalization for ND.

Notice that an introduction followed immediately by the corresponding elimination is an unnecessary detour. The derivation may be "reduced" to an equivalent one in which the introduction/elimination pair is removed. Normalization is the process of continued reduction which eliminates all such unnecessary detours from a proof.

We consider typed λ-expressions rather than their corresponding ND-derivations. The "one-step" reduction rules are:

$$\begin{array}{rrcl} (\wedge) & \langle t_0^{A_0}, t_1^{A_1}\rangle i & \Rightarrow & t_i^{A_i} \\ (\rightarrow) & (\lambda x^A.t^B)^{A\rightarrow B} s^A & \Rightarrow & t[s/x]^B \\ (\forall) & (\lambda z^0.t^{A(z)})^{\forall z A} a^0 & \Rightarrow & t^{A(a)} \end{array}$$

where, in the $\wedge$-reduction $i = 0, 1$.

The expressions which can be reduced in one step as above are called *redexes*. A λ-expression (or ND-derivation) is said to be in *normal form* if it contains no redexes (or introduction/elimination pairs).

Theorem 2.2 (Normalization Theorem) *Every λ-expression or ND-derivation reduces to a normal form.*

Define the *rank* of a typed λ-expression to be the maximum of all the "heights" of formulas/types A such that there is a redex $r^A s$ occurring in it. By the height of a formula we mean just the height of its formation tree, i.e. the least number greater than the heights of its immediate subformulas. If the expression contains no redexes its rank is 0.

The Normalization Theorem follows from:

Theorem 2.3 (The Reduction Lemma) *Every λ-expression t of rank $k+1$ can be reduced to another one t_1 of the same type but with rank $\le k$.*

Proof. By induction on the "height" of the expression t:

Variables: $t \equiv x^A$. Then t already has rank 0.

Introductions: $t \equiv \langle t', t'' \rangle$ or $t \equiv \lambda x^A.t'$ or $t \equiv \lambda z^0.t'$. Then t reduces to $t_1 \equiv \langle t'_1, t''_1 \rangle$ or $t_1 \equiv \lambda x^A.t'_1$ or $t_1 \equiv \lambda z^0.t'_1$ and in each of these cases t_1 has rank $\le k$ by the induction hypothesis.

Eliminations: $t \equiv r^A s$. First reduce r to r_1 with rank $\le k$ and reduce s to s_1 with rank $\le k$, using the induction hypothesis. Then t reduces to $r_1^A s_1$. If this is not a redex it has rank $\le k$ as required. If it is a redex it falls under one of the following cases:

1. $r_1^A \equiv \langle r_{10}^{A_0}, r_{11}^{A_1} \rangle$ and $s \equiv i$ where $i = 0$ or 1. Therefore $r_1^A s_1 \Rightarrow r_{1i}^{A_i}$ with rank $\le k$ and hence t reduces to $t_1 \equiv r_{1i}^{A_i}$ with rank $\le k$.
2. $r_1^A \equiv \lambda z^0.r_2^{B(z)}$ and $s_1 \equiv a^0$. Therefore $r_1^A s_1 \Rightarrow r_2^{B(a)}$ with rank $\le k$ and hence t reduces to $t_1 \equiv r_2^{B(a)}$ with rank $\le k$.
3. $r_1^A \equiv \lambda x^B.r_2^C$ and $s_1 \equiv s_1^B$. Therefore $r_1^A s_1 \Rightarrow r_2[s_1/x]^C$ with rank $\le k$ (?) Hence t reduces to $t_1 \equiv r_2[s_1/x]$ with rank $\le k$.

To complete the proof, we must answer (?) in part (3): why is $r_2[s_1/x]$ of rank $\le k$? The situation is this :

- r_2^C has rank $\le k$ and contains x^B free.
- s_1^B has rank $\le k$.
- $A \equiv B \to C$ has height $\le k+1$ so B has height $\le k$.

Now to check that $r_2[s_1/x]$ has rank $\leq k$, we consider all possible forms of r_2. But the only way in which the rank of r_2 could possibly be changed by substitution of s_1 for x, would be if new redexes were thus created. This could only happen if r_2 contained applicative subterms of the form $(x^B u)$ and s_1^B were of the introductory form $\langle , \rangle$ or λ or λ^0. But then the rank of such a newly created redex would just be the length of B which is still $\leq k$ by the remark above. This completes the proof.

Remark. What we have proved above is Weak Normalization (there is a terminating reduction sequence). Strong Normalization says that every reduction sequence terminates, and the Church-Rosser Property shows that the resulting normal form is unique (modulo changes of bound variable). For these see Girard, Lafont and Taylor (1989).

The Complexity of Reduction $t \mapsto t_1$**.** Let $|t|$ denote the height of the λ-expression t, or (equivalently) the height of its corresponding ND-derivation tree. We want to estimate a function F such that

$$|t_1| \leq F(|t|).$$

Notice that the worst that can happen in reducing t to t_1 occurs in case (3) where s_1 is substituted for x in r_2. Obviously $|r_2[s_1/x]| \leq |r_2| + |s_1|$. So as $|t|$ increases from n to $n+1$, in the worst case $|t_1|$ might be doubled. Thus F must have the property

$$F(n) + F(n) \leq F(n+1)$$

and therefore F is exponential $F(n) = 2^n$.

Consequently if t has rank k then it reduces to normal form t^* after k applications of the Reduction Lemma and hence the complexity of normalization is super-exponential :

$$|t^*| \leq 2^{2^{2^{\cdot^{\cdot^{2^{|t|}}}}}} \left.\vphantom{2^{2^{2^{2}}}}\right\} k \text{ times}$$

This should be compared with the Cut-Elimination result of the next section. Cut-Elimination and Normalization for the $(\wedge)$, $(\rightarrow)$, $(\forall)$ fragment of PC are analogues.

2.3 Consequences of Normalization.

Normal forms of ND-Derivations. A *branch* of an ND-derivation starts with any assumption, and traces down until it hits either the "end" formula (main branch), or the "minor" premise of an $\rightarrow$-elimination (side branch).

In a normal derivation each branch consists of a sequence of Eliminations followed by a sequence of Introductions.

Subformula Property. In a normal derivation of a formula A from assumptions $A_1, A_2, \ldots, A_n$, every formula which occurs must be a subformula of A or of one of the A_i.

Theorem 2.4 (Herbrand's Theorem) *Given any ND-derivation of a Σ_1-formula $\exists z.A(z) \equiv \forall z.\neg A(z) \rightarrow\perp$, where A is quantifier-free, we can find individual terms $a_1, a_2, \ldots, a_n$ such that:*

$$\vdash_{ND} A(a_1) \vee A(a_2) \vee \ldots \vee A(a_n).$$

Proof. Normalize the given derivation of $\forall z.\neg A(z) \rightarrow\perp$. This yields a normal derivation of $\perp$ from (say n) occurrences of the assumption $\forall z.\neg A(z)$. Each branch must start with an $\forall E$:

$$\frac{\forall z.\neg A(z)}{\neg A(a_i)}\,.$$

Now replace each one of these by an $\wedge$-elimination:

$$\frac{\neg A(a_1) \wedge \neg A(a_2) \wedge \cdots \wedge \neg A(a_n)}{\neg A(a_i)}$$

to obtain a derivation of $\perp$ from the assumption $\neg A(a_1) \wedge \cdots \wedge \neg A(a_n)$. Then by $\rightarrow$-introduction we get $\neg A(a_1) \wedge \cdots \wedge \neg A(a_n) \rightarrow\perp$ which is just $A(a_1) \vee A(a_2) \vee \cdots \vee A(a_n)$.

3 Sequent Calculus and Cut Elimination.

Let the "sequent" $\underline{A} \vdash B$ stand for *there is a ND-derivation of B from the assumptions $\underline{A} = A_1, A_2, \ldots, A_k$.* Note that $\underline{A}$ now stands for a finite *multiset* of formulas, possibly with repetitions, possibly empty, and for the time being B is either a *single* formula or nothing (meaning "false").

Under this interpretation, the rules for forming ND-derivations translate quite straightforwardly into the rules of Gentzen's intuitionistic "Sequent Calculus" LJ, where introduction rules introduce formulas on the right of $\vdash$ and elimination rules introduce formulas on the left of $\vdash$.

Examples. The axioms are

$$\underline{A}, B \vdash B$$

the $(\wedge)$ rules become

$$\frac{\underline{A} \vdash B_0 \quad \underline{A'} \vdash B_1}{\underline{A}, \underline{A'} \vdash B_0 \wedge B_1} \qquad \frac{\underline{A}, C_i \vdash B}{\underline{A}, C_0 \wedge C_1 \vdash B}$$

the $(\neg)$ rules are

$$\frac{\underline{A}, B \vdash}{\underline{A} \vdash \neg B} \qquad \frac{\underline{A} \vdash B}{\underline{A}, \neg B \vdash}$$

the $(\forall)$ rules become

$$\frac{\underline{A} \vdash B(x)}{\underline{A} \vdash \forall x.B(x)} \qquad \frac{\underline{A}, C(t) \vdash B}{\underline{A}, \forall x.C(x) \vdash B}$$

and the $(\exists)$ rules are

$$\frac{\underline{A} \vdash B(t)}{\underline{A} \vdash \exists x.B(x)} \qquad \frac{\underline{A}, C(x) \vdash B}{\underline{A}, \exists x.C(x) \vdash B}$$

where, in the $\forall$-introduction and $\exists$-elimination rules, the variable x is not free in the conclusion, and in the $(\forall)$-elimination and $(\exists)$-introduction rules, t is any term.

In addition, if one is only concerned with the logical truth of a sequent, and does not wish to keep a record of the number of times an assumption is used, then this is reflected by the so-called *structural rules* of Contraction:

$$\frac{\underline{A}, C, C \vdash B}{\underline{A}, C \vdash B}$$

and Weakening:

$$\frac{\underline{A} \vdash B}{\underline{A}, C \vdash B} \qquad \frac{\underline{A} \vdash}{\underline{A} \vdash B}.$$

The cut-rule

$$\frac{\underline{A} \vdash C \quad \underline{A'}, C \vdash B}{\underline{A}, \underline{A'} \vdash B}$$

corresponds in ND to an introduction of C followed by its use (elimination) in deriving B. Thus normalization for ND corresponds to cut-elimination for LJ.

For intuitionistic logic the restriction that at most one formula occurs on the right of $\vdash$ is the crucial point, as it prevents derivations such as

$$\cfrac{\cfrac{\cfrac{B \vdash B}{\vdash \neg B, B}}{\neg\neg B \vdash B}}{\vdash \neg\neg B \to B}$$

A cut-free derivation of a sequent $\vdash B$ must have a right-introduction as its final rule. Thus cut-elimination yields the following properties of intuitionistic logic:

Disjunction Property.
If $\vdash B_0 \vee B_1$ then either $\vdash B_0$ or $\vdash B_1$.

Existence Property.
If $\vdash \exists z.B(z)$ then $\vdash B(t)$ for some term t.

3.1 Classical Sequent Calculus LK.

LK is obtained by allowing sequences of formulas $\underline{B}$ to occur also on the right hand side of $\vdash$. Thus a sequent is now of the form

$$A_1, A_2, \ldots, A_k \vdash B_1, B_2, \ldots, B_m$$

and has the intended meaning (A_1 and ... and A_k) implies (B_1 or ... or B_m).

The rules of LK are generalized versions of the LJ-rules, but now weakening and contraction are allowed on the right as well, and (for example) the left ($\vee$) rule is generalized to

$$\frac{\underline{A}, C_0 \vdash \underline{B} \quad \underline{A'}, C_1 \vdash \underline{B'}}{\underline{A}, \underline{A'}, (C_0 \vee C_1) \vdash \underline{B}, \underline{B'}}$$

Since more than one formula can occur on the right of $\vdash$ in a LK-derivation, notice that we can now derive $\vdash \neg\neg B \to B$. We can no longer guarantee that in a cut-free derivation $\vdash B$ in LK, the last rule applied is a logical one. It might just as well have been a contraction from $\vdash B, B$, etc. Thus, for example, the Existence Property is lost in classical logic, and gets replaced by versions of Herbrand's Theorem.

The Tait-style system PC of classical predicate logic used in section 1 is really a simplified version of LK and is easily obtained from it by :

1. using the $\neg$-rules to pass from $\underline{A} \vdash \underline{B}$ to $\vdash \neg\underline{A}, \underline{B}$.
2. using De Morgan's Laws to remove $\neg$ and $\to$ in favour of their "definitions" as in section 1.
3. thus obtaining a system of right-sequents $\vdash \underline{A}', \underline{B}$ with only right-hand introduction rules for ($\vee$), ($\wedge$), ($\exists$) and ($\forall$).
4. finally removing the need for structural rules by replacing multisets $\underline{A}$ with their corresponding finite sets Γ.

What remains is PC.

3.2 Cut Elimination.

In the rest of this section we shall develop a syntactic proof of the Cut-Elimination Theorem (Gentzen's Hauptsatz 1934), but in a particularly simple though informative "linear-style" context which displays the crucial ingredients of all such proofs. The system we shall consider is to be called MPC (standing for "multiplicative" version of PC). It is formed from PC simply by reinterpreting the *finite sets* of formulas Γ as *finite multisets*, and allowing the two premises of the ($\wedge$) and cut rules to have different contexts (side formulas), which are then joined together in the conclusion. There are no Contraction or Weakening rules.

Thus MPC is essentially the so-called multiplicative fragment of Girard's Linear Logic (see e.g. Girard, Lafont and Taylor (1989)), though we continue to use the usual $\wedge$ and $\vee$ symbols instead of the 'tensor' product and sum.

The Proof-Rules of MPC are (with any multisets Γ, Γ'):

$$\text{(Axioms)} \quad \Gamma, P(x_{i_1}, \ldots, x_{i_k}), \bar{P}(x_{i_1}, \ldots, x_{i_k})$$

$$(\vee) \ \frac{\Gamma, A_0, A_1}{\Gamma, (A_0 \vee A_1)} \qquad (\wedge) \ \frac{\Gamma, A_0 \quad \Gamma', A_1}{\Gamma, \Gamma', (A_0 \wedge A_1)}$$

$$(\exists) \ \frac{\Gamma, A(x')}{\Gamma, \exists x A(x)} \qquad (\forall) \ \frac{\Gamma, A(x')}{\Gamma, \forall x A(x)} \quad x' \text{ not free in } \Gamma$$

$$\text{(Cut)} \ \frac{\Gamma, C \quad \Gamma', \neg C}{\Gamma, \Gamma'} \quad C \text{ is the “cut formula”.}$$

Cut elimination in MPC takes on a particularly simple form since the reduction and elimination of cuts from a proof *decreases* the size of a proof (in contrast to the situation in both Classical and Intuitionistic Logic where the multiplicities inherent in the use of structural rules create new complexities). This respects the idea that cuts are "indirections" in a proof. If a proof makes recourse to indirections, one should expect its size to exceed that of a "direct" proof. On the other hand, if having derived a sequent once it may nevertheless be used *more than once* within a derivation, we might expect the introduction of the indirection to lead to a decrease in size. Consequently, cuts may be used to shorten proofs in the presence of contraction.

The size, height and cut-rank of MPC-derivations.

We write $d \vdash \Gamma$ to denote that d is a MPC-derivation of the multiset Γ. If the final rule applied in d has premise(s) Γ_0 (Γ_1) then we denote the(ir) immediate subderivation(s) d_0 (d_1).

The size $s(d)$ and height $h(d)$ of a derivation are defined recursively by

$$s(d) = s(d_0) + s(d_1) + 1 \quad \text{and} \quad h(d) = \max(h(d_0), h(d_1)) + 1$$

with d_1 omitted if the final rule in d has only one premise, and with $s(d) = h(d) = 1$ if d is just an axiom.

The cut-rank $r(d)$ is defined to be the maximum "height" of all cut-formulas appearing in the derivation d (recall that the height of a formula is the least number greater then the heights of all its subformulas). If the derivation d is cut-free then $r(d) = 0$.

Substitution. If $d(x) \vdash \Gamma(x)$ denotes a proof d of sequent Γ with variable x free, and x' is a variable free for x in d, then $d(x')$ denotes the derivation obtained from d by substitution of x' for x. Substitution has no effect on the size, height or cut-rank of a proof.

The main technical tool in the Cut-Elimination argument is the following:

Theorem 3.1 (Cut-Reduction Lemma) *If $d \vdash \Gamma, C$ and $e \vdash \Delta, \neg C$ both with cut-rank $< r =$ the height of formula C, then there is a derivation of Γ, Δ with*

$$\textit{cut-rank} < r, \quad \textit{size} \leq s(d) + s(e), \quad \textit{height} \leq h(d) + h(e).$$

Proof. By induction on $s(d) + s(e)$.

Case 1. Either C is a side formula of the last inference of d or $\neg C$ is a side formula of the last inference of e. By the duality between C and $\neg C$, we may, without loss of generality, assume the former.

If d is just an axiom then, since C is a side-formula, Γ and hence Γ, Δ are axioms, so the required bounds on size, height and cut-rank hold automatically. Otherwise d has one or two immediate subderivations of the form:

$$d_0 \vdash \Gamma', C \qquad d_1 \vdash \Gamma''$$

where, since we are dealing now with multisets of formulas, only one of these premises contains the distinguished occurrence of the formula C (this is crucial to our argument).

Now since $s(d_0) + s(e) < s(d) + s(e)$ we can apply the induction hypothesis to $d_0 \vdash \Gamma', C$ in order to obtain a derivation $d_0' \vdash \Gamma', \Delta$ with cut-rank $< r$, $s(d_0') \leq s(d_0) + s(e)$ and $h(d_0') \leq h(d_0) + h(e)$. Then by re-applying the final inference rule of d to the subderivations d'_0 and d_1 (and possibly renaming a free variable if d comes from d_0 by a $\forall$ rule) we obtain the desired derivation of Γ, Δ with cut-rank $< r$ and size

$$= s(d'_0) + s(d_1) + 1 \leq s(d_0) + s(e) + s(d_1) + 1 = s(d) + s(e)$$

and height

$$= \max(h(d'_0), h(d_1)) + 1 \leq \max(h(d_0) + h(e), h(d_1)) + 1 \leq h(d) + h(e).$$

Note that if the distinguished occurence of C appeared in both subderivations of d then the calculation of the size bound would no longer work.

Case 2. C is the principal formula of (i.e. the formula actually "proved" in) the final inference of d and $\neg C$ is the principal formula of the final inference of e. There are six cases according to the structure of C, which are reduced by duality to three.

Suppose C is atomic and $\neg C$ occurs in Γ so that $d \vdash \Gamma, C$ is an axiom, and suppose $e \vdash \Delta, \neg C$ is an axiom by virtue of the fact that C occurs in Δ. Then Γ, Δ is an axiom and the result holds automatically.

Suppose $C \equiv C_0 \vee C_1$, so $\neg C \equiv \neg C_0 \wedge \neg C_1$. Then the immediate subderivations of d and e are :

$$d_0 \vdash \Gamma, C_0, C_1 \quad e_0 \vdash \Delta', \neg C_0 \quad e_1 \vdash \Delta'', \neg C_1$$

where $\Delta = \Delta', \Delta''$. So by two successive cuts on C_0 and C_1 (both of height less than r) we obtain the desired derivation

$$\dfrac{\dfrac{\Gamma, C_0, C_1 \quad \Delta', \neg C_0}{\Gamma, \Delta', C_1} \quad \Delta'', \neg C_1}{\Gamma, \Delta}$$

with cut-rank $< r$. Furthermore we can easily calculate its size

$$= s(d_0) + s(e_0) + 1 + s(e_1) + 1 = s(d_0) + s(e) + 1 = s(d) + s(e)$$

and height

$$= \max(h(d_0) + 1, h(e_0) + 1, h(e_1)) + 1 \leq \max(h(d), h(e)) + 1 \leq h(d) + h($$

This completes the disjunctive case.

Finally suppose $C \equiv \exists x C_0(x)$, so $\neg C \equiv \forall x \neg C_0(x)$. Then the immediate subderivations of d and e are :

$$d_0 \vdash \Gamma, C_0(x') \quad e_0 \vdash \Delta, \neg C_0(y)$$

where y is not free in Δ. So by substituting x' for y throughout e_0 and then applying a cut on $C_0(x')$ (of height less than r) we again obtain the desired derivation

$$\frac{\Gamma, C_0(x') \quad \Delta, \neg C_0(x')}{\Gamma, \Delta}$$

with cut-rank $< r$, and again we can calculate its size

$$= s(d_0) + s(e_0) + 1 \leq s(d) + s(e)$$

and height

$$= \max(h(d_0), h(e_0)) + 1 = \max(h(d), h(e)) \leq h(d) + h(e).$$

This completes the proof.

Theorem 3.2 (Cut-Elimination for MPC) *If $d \vdash \Gamma$ with cut rank $r > 0$, there is a derivation $d' \vdash \Gamma$ with strictly smaller cut-rank such that $s(d') < s(d)$ and $h(d') \leq 2^{h(d)}$.*
Hence by iterating this, there is a cut-free derivation of Γ with size $< s(d)$ and height $\leq 2_r(h(d))$ where $2_0(m) = m$ and $2_{k+1}(m) = 2^{2_k(m)}$.

Proof. By induction on $s(d)$. Assume that the last inference of d is a cut of rank r (the result follows immediately from the induction hypothesis in the other cases; note that d cannot be an axiom because its cut-rank is non-zero). The immediate subderivations are of the form:

$$d_0 \vdash \Gamma', C \quad d_1 \vdash \Delta', \neg C$$

where the height of the cut formula C is r and $\Gamma = \Gamma', \Delta'$. By the induction hypothesis on d_0 and d_1 we get $d_0' \vdash \Gamma', C$ and $d_1' \vdash \Delta', \neg C$ with ranks $< r$, sizes $< s(d_0)$ and $< s(d_1)$ respectively, and heights $\leq 2^{h(d_0)}$ and $\leq 2^{h(d_1)}$ respectively. The Cut-Reduction Lemma on d_0' and d_1' then yields a derivation $d' \vdash \Gamma$

with rank strictly less than r, size $\leq s(d_0') + s(d_1') < s(d)$ and height $\leq 2^{h(d_0)} + 2^{h(d_1)} \leq 2^{h(d)}$. This completes the proof.

Thus in MPC the elimination of cuts reduces the size of proofs, but increases their height super-exponentially (as with normalization for ND).

Existence property for MPC. Because there is no contraction rule, MPC also admits a result more normally associated with constructive logics, namely, that proofs of existential statements yield witnesses. For if $\exists x A(x)$ is derivable in MPC then it has a cut-free proof and so $A(t)$ must have been derived for some term t. Note however that because of the form of the disjunction rule in MPC, we do not obtain the disjunction property in the same way.

Extending Cut-Elimination to PC. To extend the theorem to PC, we must reinterpret the Γ's and Δ's as *finite sets* of formulas (not multisets) so as to recover the effects of Contraction and Weakening. But then it is easy to check that essentially the same proof as above applies *provided* we omit all mention of "size". The calculations of "height" still work however. Thus a PC-derivation of height h and positive cut-rank r can be reduced to one of smaller cut-rank at the expense of an increase in height no greater than 2^h.

For further detailed discussions of Cut Elimination and much more relevant material, see for example Girard (1987), Kleene (1952), Schwichtenberg (1977) and Tait (1968).

4 Σ_1-Induction and Primitive Recursion.

Herbrand's Theorem provides a method for extracting or synthesizing algorithms (the terms $a_1, ..., a_n$) which "witness" existential theorems of logic. Program synthesis is concerned with this process, but in the more general context of applied logics such as Formal (Peano) Arithmetic PA.

PA can be formalized in PC by adding a distinguished constant 0 and function symbol S (successor), together with additional non-logical axioms defining given "elementary" relations and functions. For example the axioms for $+$ would be (all substitution instances of):

$$\begin{array}{lll} \Gamma, & (x+y \neq z), & (x+Sy = Sz) \\ \Gamma, & (x+0=x) & \\ \Gamma, & (x+y\neq z), & (x+y\neq z'),\ (z=z') \end{array}$$

Finally the Induction Rule is added:

$$\frac{\Gamma, A(0) \quad \Gamma, \neg A(x), A(Sx)}{\Gamma, A(y)}$$

where x is not free in Γ and y may be substituted by any term.

We shall concern ourselves here, only with a certain subsystem of PA, in which the Induction Rule is restricted to Σ_1-formulas:

$$A(x) \equiv \exists z_1, \ldots, \exists z_n . B(x, z_1, \ldots, z_n)$$

where B is quantifier-free or at worst contains only bounded universal quantifiers. This subsystem is denoted $(\Sigma_1 - IND)$.

4.1 Cut-Elimination.

In $(\Sigma_1 - IND)$ we can carry out Cut-Reduction, but only down as far as Σ_1-formulas, because then the new rule of induction gets in the way so that Cut-Reduction comes unstuck at the point:

$$\frac{\Gamma, \neg A(y) \qquad \dfrac{\Gamma, A(0) \quad \Gamma, \neg A(x), A(Sx)}{\Gamma, A(y)}}{\Gamma}$$

Henceforth we assume this Cut-Reduction to have been completed, so at worst, only Σ_1 cut-formulas C remain.

4.2 Semantics.

Let $A(x_1, \ldots, x_k)$ be a Σ_1-formula:

$$A(x_1, \ldots, x_k) \equiv \exists z_1, \ldots, \exists z_\ell \,. B(x_1, \ldots, x_k, z_1, \ldots, z_\ell)$$

that is, a specification that given inputs $x_1, \ldots, x_k$ there are outputs $z_1, \ldots, z_\ell$ satisfying B.

Then given an assignment of numbers $m_1, \ldots, m_k$ to the free variables $x_1, \ldots, x_k$, write

$$m \models A(m_1, \ldots, m_k)$$

to mean there are numbers $n_1, \ldots, n_\ell < m$ such that in the standard model $\mathcal{N}$ of arithmetic, $B(m_1, \ldots m_k, n_1, \ldots, n_\ell)$ is true.

If $\Gamma(x_1, \ldots, x_k) = \{A_1, \ldots, A_n\}$ is a set of Σ_1-formulas containing the free variables $x_1, \ldots, x_k$, write

$$m \models \Gamma(m_1, \ldots, m_k)$$

to mean that $m \models A_i(m_1, \ldots, m_k)$ for some $i = 1, .., n$.

Then, given a function $F : N^k \to N$, write $F \models \Gamma$ to mean that for all assignments $x_1 := m_1, x_2 := m_2, \ldots, x_k := m_k$,

$$F(m_1, \ldots, m_k) \models \Gamma(m_1, \ldots, m_k).$$

Note on "persistence."

1. $m \leq m'$ and $m \models A(m_1, \ldots, m_k) \Rightarrow m' \models A(m_1, \ldots, m_k)$.
2. $F \leq F'$ and $F \models \Gamma(x_1, \ldots, x_k) \Rightarrow F' \models \Gamma(x_1, \ldots, x_k)$.

4.3 A Basic Theorem.

The following is an old and fundamental result, due to Kreisel, Parsons (1972), Mints (1973) and others. It underlies many present-day generalisations and displays, in a simple context, the clear connections between inductive proofs and recursive programs.

Theorem 4.1 *If Γ is a set of Σ_1 -formulas and $(\Sigma_1 - IND) \vdash \Gamma$ then there is an increasing "primitive recursive" function F such that $F \models \Gamma$.*

Corollary 4.2 *If $(\Sigma_1 - IND) \vdash \forall x.\exists z.B(x,z)$ then there is a primitive recursive function f such that $B(n, f(n))$ holds for every $n \in N$.*

Corollary 4.3 (Incompleteness) *The non-primitive recursive Ackermann Function is not provably "specifiable" in $(\Sigma_1 - IND)$.*

Proof of Theorem. Proceed by induction on the length of the $(\Sigma_1 - IND)$-derivation of Γ, with a case-distinction according to which rule is applied last:

The axioms are true and quantifier-free, so any F will do for them.

The $(\vee)$ and $(\wedge)$ rules are trivial; for example, suppose

$$\frac{\Gamma, A_0 \quad \Gamma, A_1}{\Gamma, (A_0 \wedge A_1)}$$

Then by the induction hypothesis we have $F_i \models \Gamma, A_i$ for each $i = 0, 1$, so it suffices to choose $F = \max(F_0, F_1)$.

The $\forall$-rule is also trivial since Γ contains only Σ_1- formulas and so universal quantifiers will only occur in bounded contexts and we are therefore concerned merely that their truth is preserved.

For the $\exists$-rule:

$$\frac{\Gamma, A(t)}{\Gamma, \exists z.A(z)},$$

we have, by the induction hypothesis, an F_0 such that $F_0 \models \Gamma, A(t)$. So in this case we can choose $F = F_0 + t$.

For the Cut-rule we can (crucially) assume that the cut-formula C is in Σ_1-form, say $C \equiv \exists z.B$. Then $\neg C \equiv \forall z.\neg B$ and so the application of Cut looks like this with the free variables $\vec{x}$ displayed:

$$\frac{\Gamma(\vec{x}),\ \forall z.\neg B(\vec{x}, z) \quad \Gamma(\vec{x}),\ \exists z.B(\vec{x}, z)}{\Gamma(\vec{x})}$$

But the left premise now contains a $\forall z$ which must be removed in order to continue the proof. Fortunately an earlier exercise on $\forall$-inversion comes to our aid, allowing the proof of $\Gamma(\vec{x}), \forall z.\neg B(\vec{x}, z)$ to be replaced by a proof of $\Gamma(\vec{x}), \neg B(\vec{x}, y)$ which is no longer than the original proof, but contains a new variable y. Applying the induction hypothesis to this and the right premise of the Cut, we obtain primitive recursive functions F_0 and F_1 such that

$$\begin{aligned} F_0(\vec{x}, y) &\models \Gamma(\vec{x}),\ \neg B(\vec{x}, y) \\ F_1(\vec{x}) &\models \Gamma(\vec{x}),\ \exists z.B(\vec{x}, z) \end{aligned}$$

So define by composition:

$$F(\vec{x}) \;=\; F_0(\vec{x}, F_1(\vec{x})).$$

We now have to verify that $F(\vec{x}) \models \Gamma(\vec{x})$ for all values of $\vec{x}$. Suppose that under a given assignment $\vec{x} := \vec{m}$ we have $F(\vec{m}) \not\models \Gamma(\vec{m})$. Then by persistence, since $F_1(\vec{m}) \leq F(\vec{m})$ we have $F_1(\vec{m}) \not\models \Gamma(\vec{m})$ and therefore (i) $F_1(\vec{m}) \models B(\vec{m}, k)$ for some $k < F_1(\vec{m})$. Similarly, since $F_0(\vec{m}, k) \leq F(\vec{m})$ we must also have (ii) $F_0(\vec{m}, k) \models \neg B(\vec{m}, k)$. But $B(\vec{m}, k)$ and $\neg B(\vec{m}, k)$ cannot both be true - contradiction! Hence $F(\vec{m}) \models \Gamma(\vec{m})$ for all assignments $\vec{m}$.

Finally, consider an application of the Σ_1-Induction Rule:

$$\frac{\Gamma, A(0) \quad \Gamma, \neg A(x), A(Sx)}{\Gamma, A(x)}$$

where $A(x) \equiv \exists z.B(x, z)$ and x is not free in Γ. We have suppressed any other parameters which may occur free in A since they play no active role in what follows. Then we have a proof of

$$\Gamma,\ \exists z.B(0, z)$$

and also, using $\forall$-inversion again, a proof of

$$\Gamma,\ \neg B(x, y),\ \exists z.B(Sx, z)\,.$$

By the induction hypothesis we have increasing primitive recursive functions F_0, F_1 such that:

$$\begin{aligned} F_0 &\models \Gamma,\ \exists z.B(0, z) \\ F_1(x, y) &\models \Gamma,\ \neg B(x, y),\ \exists z.B(Sx, z). \end{aligned}$$

Now define F by primitive recursion from F_0 and F_1:

$$F(0) = F_0 \quad \text{and} \quad F(x+1) = F_1(x, F(x)).$$

Then we must verify $F(x) \models \Gamma, \exists z.B(x, z)$ for all values of x. To do this fix $x = m$ and proceed by induction on n to show that for all n,

$$F(n) \models \Gamma, \exists z.B(n, z).$$

The basis $n = 0$ is immediate and the induction step from n to $n + 1$ is very similar to the verification of the Cut-case above. It is left as an exercise!

Note the relationships between Cut and Composition and between Induction and Recursion. The converse of the above, that every primitive recursive function is provably specifiable in (Σ_1-IND), will be evident from the work of the next section.

5 The Logic of Primitive Recursion.

This is some joint work from Sieg and Wainer (1994). In lecture 4 we saw that the primitive recursive functions are those which can be proved to terminate in the fragment of arithmetic with induction restricted to existential (Σ_1) formulas, and more generally as shown in Sieg (1991), in the fragment with (Π_2) - induction provided any side assumptions are less complex, i.e. at worst Σ_2. This is an "extensional" result, characterizing a certain class of number-theoretic *functions* .

What we are looking for here is something more "intensional", i.e. a logic which allows us to distinguish between different kinds of primitive recursive *programs* according to the structure of their respective termination proofs. Preferably it should provide a clear correspondence between proofs and programs, and also at the higher level between proof-transformations and program- transformations, so that "program-complexity" is measurable directly in terms of "proof complexity".

Much recent work on implementation already illustrates the potential applicability of proof-transformation as a means to synthesize and analyze useful program-transformations. However our present concern lies rather in the general proof-theoretic principles which underly such applications. Thus we will be very restrictive in considering only programs over the natural numbers N since they already serve to illustrate the essential logical features, but with the least amount of syntactic "fuss". Feferman (1992) and Tucker and Zucker (1992) show how the ideas in section 4 can be extended and applied fruitfully to more general, abstract inductive data types which arise naturally in computer science.

The logic we arrive at below is a strictly "linear" one (no contraction, no weakening and no exchange !) obtained simply by analyzing just what one needs to prove termination of primitive recursive definitions. The absence of exchange rules means that two cut rules are needed - an ordinary one and another one which we call "call-by-value cut" for reasons which will be obvious. It then turns out that, in the appropriate setting, the transformation from recursive to tail-recursive programs is precisely call-by-value-cut-elimination !

5.1 Primitive Recursive Programs.

Definitions. A *primitive recursive* program is one in which every defining equation has one of the five forms "zero", "successor", "projection", "explicit definition" and "primitive recursion" as follows :

$$
\begin{array}{llcl}
(Z) & f_i(x) & = & 0 \\
(S) & f_i(x) & = & x+1 \\
(P) & f_i(\vec{x}) & = & x_j \\
(E) & f_i(\vec{x}) & = & t(f_0,\ldots,f_{i-1}\,;\vec{x}) \\
(PR_0) & f_i(0,\vec{x}) & = & f_{i_0}(\vec{x}) \\
(PR_1) & f_i(z+1,\vec{x}) & = & f_{i_1}(z,\vec{x},f_i(z,\vec{x}))
\end{array}
$$

where in the (E) scheme t is some term built up from the previously defined functions, and in the (PR) scheme $i_0, i_1 < i$.

A *generalized primitive recursive* program is one in which the primitive recursion equation (PR_1) is generalized to allow substitution of terms for the parameters $\vec{x}$ in the recursive call $f_i(z,\vec{x})$ as follows :

$$(GPR_1) \quad f_i(z{+}1,\vec{x}) \;=\; f_{i_1}(z,\vec{x},f_i(z,f_{i_2}(z,\vec{x}),\ldots,f_{i_{k+1}}(z,\vec{x})))$$

where $i_0, i_1, i_2, \ldots, i_{k+1} < i$.

A primitive *tail recursive* program is one in which generalized primitive recursion is allowed, but only in the following restricted context, where the recursive call on $f_i(z,\ldots)$ is the final function-call made in the evaluation of $f_i(z+1,\vec{x})$:

$$(TR_1) \quad f_i(z+1,\vec{x}) \;=\; f_i(z,f_{i_1}(z,\vec{x}),\ldots,f_{i_k}(z,\vec{x}))\,.$$

Remark. Tail recursive programs

$$
\begin{array}{lcl}
f(0,x) & = & g(x) \\
f(z+1,x) & = & f(z,h(z,x))
\end{array}
$$

are "efficient" since they can immediately be recast as while-loops:

while $z \neq 0$ **do** $z := z-1;\ x := h(z,x)$ **od** ; $f := g(x)$.

The following transformations are either explicit or implicit in the classic R. Péter (1967) which contains a wealth of information on the reduction of various kinds of recursions to simpler forms.

Theorem 5.1 *Every generalized primitive recursive program can be transformed into a primitive tail recursive program defining the same function. Every primitive tail recursive program can be transformed into an ordinary primitive recursive program defining the same function.*

Proof. (i) A generalized primitive recursion (i.e. with parameter substitution) such as

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & h(z,x,f(z,p(z,x))) \end{array}$$

can be transformed into a tail recursive program as follows (note however that three tail recursions seem to be needed - the two given here plus another one implicitly used in order to define the "modified minus" from the predecessor)

$$\begin{array}{llcl} (TR_0) & f_0(0,z,x) & = & x \\ (TR_1) & f_0(n+1,z,x) & = & f_0(n,z\dot{-}1,p(z\dot{-}1,x)) \\ (E) & f_1(n,z,x,y) & = & h(z\dot{-}(n+1),f_0(n,z,x),y) \\ (TR_0) & f_2(0,z,x,y) & = & y \\ (TR_1) & f_2(n+1,z,x,y) & = & f_2(n,z,x,f_1(n,z,x,y)) \\ (E) & f_3(z,x) & = & f_2(z,z,x,g(f_0(z,z,x)) \end{array}$$

The devoted (!) reader with a taste for intricate inductions might now like to verify that

$$\forall z\,\forall x\;(\,f_3(z,x) \;=\; f(z,x)\,)\,.$$

Hint : one needs first to check the following identities

$$f_1(n+1,z+1,x,y) \;=\; f_1(n,z,p(z,x),y)$$

$$f_2(n+1,z+1,x,y) \;=\; h(z,x,f_2(n,z,p(z,x),y))$$

and then a further induction on z yields the desired result.

(ii) A primitive tail recursion such as

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & f(z,p(z,x)) \end{array}$$

can be transformed into an ordinary primitive recursion as follows

$$\begin{array}{llcl} (PR_0) & f_0(0,z,x) & = & x \\ (PR_1) & f_0(n+1,z,x) & = & p(z\dot{-}n,f_0(n,z,x)) \\ (E) & f_1(z,x) & = & g(f_0(z,z\dot{-}1,x))\,. \end{array}$$

The verification needs a preliminary induction on n to show

$$f_0(n+1,z,x) \;=\; f_0(n, z \dot{-} 1, p(z,x))$$

and then by a further induction on z,

$$\forall z\, \forall x\, (\, f_1(z,x) \;=\; f(z,x)\,)\,.$$

Notice that the above program - equivalences are all provable by inductions on quantifier - free equational formulas, or on universally quantified equational formulas, i.e. Π_1 formulas.

We are now going to devise a logic exactly tailored to proofs about primitive recursive and generalized primitive recursive programs.

5.2 The Logic of Primitive Recursion (LPR).

Formulas $A, B, C, \ldots$ will be either atoms of the form $f(\vec{x}) \simeq y$ with y a variable, meaning $f(\vec{x})$ is defined with value y, or Σ_1 -formulas $\exists y(f(\vec{x}) \simeq y)$ or Π_2 -formulas $\forall \vec{x} \exists y(f(\vec{x}) \simeq y)$.

The axioms are of two kinds, the principal ones being purely relational sequents or "logic programs" describing the order of evaluation of individual equations in a primitive recursive program, thus for example

$$f_0(\vec{x}) \simeq y_0, f_1(\vec{x}, y_0) \simeq y_1, \ldots, f_m(\vec{x}, y_0, \ldots, y_{m-1}) \simeq y_m \vdash f(\vec{x}) \simeq y_m$$

describes an explicit definition

$$f(\vec{x}) \;=\; f_m(\vec{x}, f_0(\vec{x}), f_1(\vec{x}, f_0(\vec{x})), \ldots)\,.$$

The other axioms simply express that the zero, successor and projection functions are defined :

$$(N - Ax) \quad \vdash \exists y(0 \simeq y)\,, \;\vdash \exists y(x+1 \simeq y)\,, \;\vdash \exists y(x \simeq y)\,.$$

The logic rules are the sequent rules for $\exists$ and $\forall$:

$$(\exists \vdash)\; \frac{\ldots,\, A(y)\,,\, \ldots \vdash B}{\ldots,\, \exists y A(y)\,,\, \ldots \vdash B} \qquad (\vdash \exists)\; \frac{\ldots \vdash B(y')}{\ldots \vdash \exists y B(y)}$$

$$(\forall \vdash)\; \frac{\ldots,\, A(x')\,,\, \ldots \vdash B}{\ldots,\, \forall x A(x)\,,\, \ldots \vdash B} \qquad (\vdash \forall)\; \frac{\ldots \vdash B(x)}{\ldots \vdash \forall x B(x)}$$

with the usual "eigenvariable" conditions on ($\exists\vdash$) and ($\vdash\forall$), i.e. the quantified variable can not occur free in the "side formulas".

In addition there are two cut rules :

$$(C)\ \frac{\vdash C \quad C, \ldots \vdash B}{\ldots \vdash B} \qquad (CVC)\ \frac{\vdash C \qquad \ldots, C, \ldots \vdash B}{\ldots \quad \ldots \vdash B}$$

and the induction rule :

$$(IND)\ \frac{\vdash B(0) \quad B(z) \vdash B(z+1)}{\vdash B(z)} \, .$$

Note. What you see is all there is ! The dots ... denote arbitrary finite sequences of assumptions and the logic is strictly linear in the sense that there are no hidden structural rules - no Contraction, no Weakening, and furthermore no Exchange ! Hence the need for two Cut rules, the second of which applies a cut "in context" and is called a "call by value" cut for reasons which will shortly become obvious. Note also that there are no other assumptions in the induction rule besides the induction hypothesis $B(z)$.

Definition. Call a recursive program defining a function f *provably recursive* or *terminating* in a given logic L if

$$L \vdash \ \forall \vec{x}\, \exists y\, (\, f(\vec{x}) \simeq y\,)\, .$$

Obviously the more restrictive the logic, the more restricted will be the class of recursive programs we can prove to terminate in it. The aim here is to impose simple logics on the equation calculus in such a way that there is a clear and precise structural correspondence between termination proofs and known subclasses of recursive programs. We concentrate here on primitive recursive programs, though the ideas have a wider range of application.

Definition. LPR($\exists$) and LPR($\forall\exists$) denote the logics restricted to Σ_1 and Π_2 formulas respectively. LPR($\forall\exists$)-(CVC) denotes the logic LPR($\forall\exists$) without call-by-value cuts.

Theorem 5.2 *Primitive Recursive* $\equiv$ *LPR($\exists$) - terminating.*
Generalized Primitive Recursive $\equiv$ *LPR($\forall\exists$) - terminating.*
Primitive Tail Recursive $\equiv$ *LPR($\forall\exists$)-(CVC) - terminating.*

Proof. We do not give a completely detailed proof here, but sufficient to display the basic relationships.

(i) That primitive recursive programs are LPR($\exists$) - terminating is easily seen. Suppose for example that f is defined explicitly from g and h by

$$f(x) \; = \; g(h(x))$$

where g and h are already assumed to be LPR($\exists$) - verifiable. Then the starting axiom is

$$h(x) \simeq y \; , \; g(y) \simeq z \; \vdash \; f(x) \simeq z \; .$$

By ($\vdash \exists$) followed by ($\exists \vdash$) we then obtain

$$h(x) \simeq y \; , \; \exists z \; (g(y) \simeq z) \; \vdash \; \exists z \; (f(x) \simeq z) \; .$$

From this and the assumption $\exists z \; (g(y) \simeq z)$ we then have by a call by value cut (CVC),

$$h(x) \simeq y \; \vdash \; \exists z \; (f(x) \simeq z)$$

and then by ($\exists \vdash$),

$$\exists y \; (h(x) \simeq y) \; \vdash \; \exists z \; (f(x) \simeq z) \; .$$

Thus by the assumption $\exists y \; (h(x) \simeq y)$ and an ordinary cut (C),

$$\vdash \; \exists z \; (f(x) \simeq z) \; .$$

Note how the eigenvariable conditions on ($\exists \vdash$) rules completely determine the order of events in the above proof, so that the call by value cut was essential.

As a further example, suppose f is defined primitive recursively from g and h as follows ;

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & h(z,x,f(z,x)) \; . \end{array}$$

where $\vdash \; \exists y \; (g(x) \simeq y)$ and $\vdash \; \exists u \; (h(z,x,y) \simeq u)$ are assumed. Then the starting axioms are

$$g(x) \simeq y \; \vdash \; f(0,x) \simeq y$$

and

$$f(z,x) \simeq y \ , \ h(z,x,y) \simeq u \ \vdash \ f(z+1,x) \simeq u \ .$$

Concentrating on the induction step first, we have by $(\vdash \exists)$ and $(\exists \vdash)$,

$$f(z,x) \simeq y \ , \ \exists u \ (h(z,x,y) \simeq u) \ \vdash \ \exists y \ (f(z+1,x) \simeq y) \ .$$

Then by a call by value cut,

$$f(z,x) \simeq y \ \vdash \ \exists y \ (f(z+1,x) \simeq y)$$

and by $(\exists \vdash)$,

$$\exists y \ (f(z,x) \simeq y) \ \vdash \ \exists y \ (f(z+1,x) \simeq y) \ .$$

Applying $(\vdash \exists)$, $(\exists \vdash)$ and an ordinary cut to the first axiom we easily obtain $\vdash \ \exists y \ (f(0,x) \simeq y)$, and so by the induction rule we have

$$\vdash \ \exists y \ (f(z,x) \simeq y)$$

as required.

(ii) Next we show why LPR($\forall\exists$) - terminating programs are generalized primitive recursive. Suppose we had a proof of

$$\vdash \ \forall x \ \exists y \ (f(z,x) \simeq y)$$

by induction on z. The induction step would therefore be

$$\forall x \ \exists y \ (f(z,x) \simeq y) \ \vdash \ \forall x \ \exists y \ (f(z+1,x) \simeq y) \ .$$

This deduction presumably used some recursive calls on "given" functions, so let us assume it came about by means of one ordinary cut on a function p and a call by value cut on a function h from :

$$\forall x \exists u (p(z,x) \simeq u), \quad \forall x \exists y (f(z,x) \simeq y), \forall x \forall y \exists v (h(z,x,y) \simeq v) \\ \vdash \forall x \exists y (f(z+1,x) \simeq y) \ .$$

The eigenvariable conditions place heavy restrictions on how this could have been derived. Essentially it must have come about by applying $(\exists \vdash)$, $(\forall \vdash)$, $(\vdash \forall)$, in that order (!) to :

$$p(z,x) \simeq u \ , \ \ \forall x \ \exists y \ (f(z,x) \simeq y) \ , \ \forall x \ \forall y \ \exists v \ (h(z,x,y) \simeq v) \\ \vdash \ \exists y \ (f(z+1,x) \simeq y) \ .$$

Stripping away the quantifiers prefixing $f(z,x) \simeq y$ we now see that this would have come from

$$\begin{array}{ll} p(z,x) \simeq u,\ f(z,u) \simeq y, & \forall x\, \forall y\, \exists v\, (h(z,x,y) \simeq v) \\ & \vdash\ \forall x\, \exists y\, (f(z+1,x) \simeq y) \end{array}$$

by applying $(\exists \vdash)$ and then $(\forall \vdash)$ with u as witnessing variable (the only other possible witnessing variables would have been z or x but these are less general). Now we can strip away the quantifier prefix on $h(z,x,y) \simeq v$ to see that this last line would have come about by applying $(\exists \vdash)$ and $(\forall \vdash)$ to :

$$p(z,x) \simeq u\ ,\ f(z,x) \simeq y\ ,\ h(z,x,y) \simeq v\ \vdash\ \exists y\, (f(z+1,x) \simeq y)\ .$$

Finally, this would have arisen by $(\vdash \exists)$ from the axiom :

$$p(z,x) \simeq u\ ,\ f(z,x) \simeq y\ ,\ h(z,x,y) \simeq v\ \vdash\ f(z+1,x) \simeq v$$

describing a generalized primitive recursion :

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & h(z,x,f(z,p(z,x)))\ . \end{array}$$

By reversing the above we also obtain the converse, that every generalized primitive recursion is LPR($\forall\exists$) - terminating. Note that if we took apart an LPR($\exists$) - inductive proof in a similar way then we would be prevented (by the absence of the $\forall \vdash$ rule) from substituting $p(z,x)$ for the variable x and so an ordinary primitive recursive program would be the only possible result. Hence the converse to part (i).

(iii) The only other crucial thing to note is that if call by value cuts were disallowed in the derivation in part (ii) above, then the h function could not appear and so the extracted program would have to be a tail recursion :

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & f(z,p(z,x))\ . \end{array}$$

This completes the proof.

Theorem 5.3 *Hence the transformation from generalized primitive recursive programs to primitive tail recursive programs corresponds exactly to the elimination of call by value cuts in LPR($\forall\exists$).*

Remarks.

A careful analysis of the above termination proofs in LPR should convince the reader of the close correspondence between the proof - structure and the computation - structure of the given program. By reading the termination proof in a goal - directed way, one sees how the order of $\forall\exists$ - eliminations exactly reflects the intended order of evaluation.

Although the transformation to tail recursion corresponds to elimination of call by value cuts in LPR($\forall\exists$), the actual transformation itself takes place at the equational rather than the logical level, as given by Theorem 5.1. Thus most of the complexity of the transformation is tied up in the Π_1 - inductive proofs of program - equivalence associated with 5.1, rather than in the structural complexity of changing call by value cuts into ordinary ones, since this only amounts to an implicit use of the exchange rule to swap the order of cut - formulas in a sequent ! However it is Theorem 5.1 that tells us this is indeed possible, and furthermore what the new exchanged cut formulas should be.

It should be clear by now that the form of the induction rule severely restricts the kinds of recursion that can be verified in the given logic. The simple form we have used so far, in which the induction step requires just one use of the premise $B(x)$ to derive $B(x+1)$, limits the corresponding forms of verifiable recursions to those in which only *one* recursive call is made. If we wish to verify a recursion with two recursive calls, then the linear - style logic requires an induction rule in which the premise $B(x)$ of the induction step is explicitly written twice ! In this way the logic reflects the fine structural distinctions between various kinds of recursive programs. To illustrate, we consider some well known examples below.

5.3 Example : The Minimum Function.

Colson (1989) points out that the minimum function $\min(x,y)$ cannot be computed by an ordinary primitive recursive program in time $O(\min(x,y))$. This is essentially because one of the variables would have to be chosen as the recursion variable, and the other one would then remain unchanged throughout the course of the recursion, so the number of computation steps - irrespective of the additional subsidiary functions needed to define it - would still be at least either x or y. He notes however that it can be computed in time $O(\min(x,y))$ by a generalized primitive recursion, say on y, with the predecessor $x \dot{-} 1$ substituted for the parameter x, thus

$$\begin{array}{lcl} \min(x,0) & = & 0 \\ \min(x,y+1) & = & \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } \min(x \dot{-} 1, y) + 1 \end{array}$$

and he comments that this should really be regarded as a higher type "functional" form of recursion.

In our sense, *the efficiency is gained by virtue of a necessary increase in the quantifier complexity of the inductive termination proof, from Σ_1 up to Π_2.*

Note also the use of the "cases" function here. But this can be verified easily by a degenerate form of our induction rule, in which the premise $B(x)$ of the induction step is not used.

5.4 Example : Nested Recursion.

A typical example of nested recursion, requiring two calls on the induction hypothesis in its termination proof, would be

$$\begin{array}{lcl} f(0,n) & = & g(n) \\ f(m+1,n) & = & h(m,n,f(m,f(m,n))) \ . \end{array}$$

The LPR derivation of the induction step in the termination proof for f begins with

$$f(z,x) \simeq y_0,\ f(z,y_0) \simeq y_1,\ h(z,x,y_1) \simeq y_2 \vdash\ f(z+1,x) \simeq y_2$$

and then by quantifier rules and a call-by-value cut on the formula $\forall u \exists y (h(z,x,u) \simeq y)$ we obtain

$$\forall x\ \exists y\ (f(z,x) \simeq y),\ \forall x\ \exists y\ (f(z,x) \simeq y) \vdash\ \forall x\ \exists y\ (f(z+1,x) \simeq y)$$

Since LPR does not allow contraction, the only way in which we can now derive

$$\vdash\ \forall x\ \exists y\ (f(z,x) \simeq y)$$

is by an extended induction rule :

$$\frac{\vdash B(0) \qquad B(z)\ ,\ B(z) \vdash B(z+1)}{\vdash B(z)}$$

which explicitly allows two uses of the induction hypothesis.

The lesson is of course, that each new form of recursion must carry its own new form of induction in LPR. However it is well known that we can in this case still transform the recursion to a primitive recursion and thereby bring the termination proof back into the original logic LPR(∀∃).

5.5 Example : Ackermann Function.

The following more complex nested recursion over the lexicographic ordering of pairs $(m,n) \in N^2$:

$$\begin{array}{lcl} F(0,n,k) & = & k+2^n \\ F(m+1,0,k) & = & F(m,k,k) \\ F(m+1,n+1,k) & = & F(m+1,n,F(m+1,n,k)) \end{array}$$

defines an alternative version of the Ackermann Function. For each fixed m, $F_m(n,k) = F(m,n,k)$ is a primitive recursive function of n, k (given by a nested recursion on n similar to the example above). However as a function of all three variables m, n, k, F is no longer primitive recursive.

Clearly, in order to prove termination of F in LPR, we need an "outer" induction on x with induction step

$$\forall y \forall z \exists u (F(x,y,z) \simeq u) \ \vdash\ \forall y \forall z \exists u (F(x+1,y,z) \simeq u)\ .$$

But this requires an "inner" induction on y whose basis is

$$\forall y \forall z \exists u (F(x,y,z) \simeq u) \ \vdash\ \forall z \exists u (F(x+1,0,z) \simeq u)$$

and whose induction step requires two calls on the induction hypothesis as in the last example:

$$\begin{array}{l} \forall z \exists u (F(x+1,y,z) \simeq u)\ ,\ \ \forall z \exists u (F(x+1,y,z) \simeq u) \\ \qquad\qquad\qquad\qquad \vdash\ \forall z \exists u (F(x+1,y+1,z) \simeq u)\ . \end{array}$$

Thus the induction rule needed in this case has the form

$$\frac{C \vdash B(0) \quad B(y), B(y) \vdash B(y+1)}{C \vdash B(y)}$$

with a side formula (additional assumption) C in the base case.

It is the occurrence of this side formula in the base case which leads us outside the realm of primitive recursion. Without it we could allow any fixed number of calls on the induction hypothesis, and still be sure that only primitive recursive functions could be proved to terminate. The above recursive definition of F may be transformed to a tail recursion, but now over a transfinite well-ordering of order-type ω^ω. It is a general feature that recursions may be transformed to tail recursions, but at the cost of a (possibly) exponential increase in the order type of the recursion ordering needed; see Fairtlough and Wainer (1992).

6 Gödel's Primitive Recursive Functionals.

The characterization of primitive recursion given in lectures 4, 5 above depended on the fact that a proof of a Σ_1- formula in Σ_1-Arithmetic essentially only involves Σ_1-formulas throughout. But what happens in full *Peano Arithmetic* (PA) where there is no restriction on the logical complexity of the formula A in the induction rule, and hence no possibility of restricting the cut-formulas ? In PA the proof of a formula like $\forall\vec{x}\,\exists y\, C_f(\vec{x}, y)$ may thus involve inductions and cuts of greater logical complexity. So how might the provably recursive functions of PA be characterized ?

One way to attack this problem would be to try to re-express *all* formulas in the logical form $\forall\vec{x}\,\exists y\, B(\vec{x}, y)$ with B 'bounded', and then find a suitable class of functions F such that

$$PA \vdash \forall\vec{x}\,\exists y\, B(\vec{x}, y) \implies F(\vec{x}) \models \exists y\, B(\vec{x}, y).$$

This was the approach taken by Gödel[1958] in his 'Dialectica' interpretation of arithmetic, and we shall briefly describe the main idea in this final lecture. As in section 3, the general method applies equally well (if not better!) to intuitionistic arithmetic, but we continue to work in classical style following e.g. Shoenfield (1967) (the reader should also consult the references to Schwichtenberg and Girard).

But how can an arithmetical formula with arbitrary quantifier complexity be reduced to the 2- quantifier form $\forall\vec{x}\,\exists y\, B(\vec{x}, y)$? The secret is to allow the variables to range not just over the 'ground type' of natural numbers, but over 'higher' function-types so that an $\forall\exists$ quantifier prefix can be transformed into $\exists\forall$ by 'Skolemization' thus

$$\forall\vec{x}\,\exists y\, B(\vec{x}, y) \;\equiv\; \exists z\,\forall\vec{x}\, B(\vec{x}, z(\vec{x})).$$

Systematic application of this idea - as below - will then transform any arithmetical formula A into a corresponding 'generalized' formula

$$A^* \;\equiv\; \forall\vec{x}\,\exists\vec{y}\, B(\vec{x}, \vec{y})$$

where B contains only bounded numerical quantifiers, and we can then hope to associate with each theorem A of full Peano Arithmetic, a higher-type functional F which 'satisfies' A^* in the sense that the following is 'true' :

$$\forall\vec{x}\, B(\vec{x}, F(\vec{x})).$$

Then by analogy with section 3 we would hope to classify F as being 'primitive recursive' in some generalized higher-type sense. As we shall see, the analogy is quite strong !

Note. It is convenient here to assume that the underlying logic of PA is supplied in 'natural deduction' form, rather than the Tait-style calculus used earlier. Thus single formulas are proved at a time, using the usual classical natural deduction rules for $\neg, \wedge, \rightarrow, \forall$ with arithmetical axioms and (unrestricted) induction rule formulated in the obvious way.

Definition. The simple types are generated from the ground type 0 of natural numbers, by repeated applications of '$\rightarrow$' so as to build function-types. We use $\sigma, \tau, \ldots$ to denote arbitrary types. The special types $0 \rightarrow 0$, $(0 \rightarrow 0) \rightarrow (0 \rightarrow 0)$, $((0 \rightarrow 0) \rightarrow (0 \rightarrow$

$0)) \to ((0 \to 0) \to (0 \to 0))$ etcetera are sometimes just denoted '1', '2', '3', ... A type $\sigma_1 \to (\sigma_2 \ldots (\sigma_k \to \tau))$ will generally be denoted $\sigma_1, \sigma_2, \ldots, \sigma_k \to \tau$.

Each type σ will have a stock of variables $x^\sigma, y^\sigma, \ldots$ which are to be thought of as ranging over the set N^σ where $N^0 = N$ and $N^{\sigma \to \tau}$ is the collection of all functions from N^σ to N^τ. From these and any typed constants we can build applicative terms : thus if x is of type $\sigma_1, \ldots, \sigma_k \to \tau$ and $y_1, \ldots, y_k$ are of types $\sigma_1, \ldots, \sigma_k$ then

$$x(y_1, y_2, \ldots, y_k) \equiv x(y_1)(y_2) \ldots (y_k)$$

is a term of type τ. In what follows we shall not always give the type of a variable explicitly, but it is to be understood that whatever terms we write are properly typed.

Definition. With each arithmetical formula A is associated a *generalized* formula

$$A^* \equiv \forall \vec{x}\, \exists \vec{y}\, B(\vec{x}, \vec{y})$$

defined as follows :

$$\begin{array}{lcl}
(A_0 \wedge A_1)^* & \equiv & \forall \vec{x}\, \exists \vec{y}\, (B_0(\vec{x}, \vec{y}) \wedge B_1(\vec{x}, \vec{y})) \\
(\forall z A(z))^* & \equiv & \forall z \forall \vec{x}\, \exists \vec{y}\, B(z, \vec{x}, \vec{y}) \\
(\neg A)^* & \equiv & \neg A^* \\
 & \equiv & \neg \forall \vec{x}\, \exists \vec{y}\, B(\vec{x}, \vec{y}) \\
 & \equiv & \neg \exists \vec{z} \forall \vec{x}\, B(\vec{x}, \vec{z}(\vec{x})) \\
 & \equiv & \forall \vec{z}\, \exists \vec{x}\, \neg B(\vec{x}, \vec{z}(\vec{x})) \\
(A_0 \to A_1)^* & \equiv & \neg A_0^* \vee A_1^* \\
 & \equiv & \forall \vec{z}, \vec{w}\, \exists \vec{x}, \vec{y}\, (B_0(\vec{x}, \vec{z}(\vec{x})) \to B_1(\vec{w}, \vec{y})).
\end{array}$$

Definition. The *primitive recursive functionals* are those which can be defined from the constants 'zero' and 'successor' of types 0 and $0 \to 0$ respectively, by applying the following schemes :

Explicit definitions.

$$F(x_1, \ldots, x_k) = t(x_1, \ldots, x_k)$$

where t is an applicative term built up from the displayed variables and previously defined primitive recursive functionals. If the types of $x_1, \ldots, x_k$ are $\sigma_1, \ldots, \sigma_k$ and t is of type τ then the type of F is $\sigma_1, \ldots, \sigma_k \rightarrow \tau$.

Primitive recursion.

$$\begin{cases} F(0) & = G \\ F(n+1) & = H(n, F(n)) \end{cases}$$

where G has a type σ, H has type $0, \sigma \rightarrow \sigma$ and F has type $0 \rightarrow \sigma$.

Theorem 6.1 (Gödel) *If A is a theorem of Peano Arithmetic with generalized formula*

$$A^* \equiv \forall \vec{x} \, \exists y_1 \ldots \exists y_m \, B(\vec{x}, y_1, \ldots y_m)$$

then there is a sequence $\vec{F} = F_1, \ldots, F_m$ of primitive recursive functionals of the appropriate types, which satisfies A^* in the sense that for all values of $\vec{x}$ in $N^{\sigma_1} \times \ldots \times N^{\sigma_k}$ the following holds :

$$B(\vec{x}, F_1(\vec{x}), \ldots, F_m(\vec{x})).$$

Proof. We only consider the two main cases in proving A - cut and induction - the others being fairly straightforward. If $\vec{F} = F_1, \ldots, F_m$ then we shall write $\vec{F}(\vec{x})$ for the sequence of values $F_1(\vec{x}), \ldots, F_m(\vec{x})$.

For the Cut case, which in Natural Deduction takes the form of implies-elimination :

$$\frac{C \quad C \rightarrow A}{A}$$

assume inductively that sequences of primitive recursive functionals $\vec{H}$ and $\vec{G} = \vec{G}_0, \vec{G}_1$ have already been constructed so as to satisfy

$$C^* \equiv \forall \vec{u} \, \exists \vec{v} \, D(\vec{u}, \vec{v})$$

and

$$(C \rightarrow A)^* \equiv \forall \vec{z}, \vec{x} \, \exists \vec{u}, \vec{y} \, (D(\vec{u}, \vec{z}(\vec{u})) \rightarrow B(\vec{x}, \vec{y})).$$

Then for all $\vec{u}$ we have

$$D(\vec{u}, \vec{H}(\vec{u}))$$

and for all $\vec{z}$ and $\vec{x}$ we have

$$D(\vec{G}_0(\vec{z},\vec{x}),\vec{z}(\vec{G}_0(\vec{z},\vec{x}))) \;\rightarrow\; B(\vec{x},\vec{G}_1(\vec{z},\vec{x})).$$

Now we can unify the two D-formulas by setting $\vec{z} = \vec{H}$ and $\vec{u} = \vec{G}_0(\vec{H},\vec{x})$. Hence we obtain for all values of $\vec{x}$,

$$B(\vec{x},\vec{G}_1(\vec{H},\vec{x})).$$

Therefore the desired sequence of primitive recursive functionals $\vec{F}$ satisfying A^* is given explicitly by composing $\vec{G}_1$ on $\vec{H}$ thus :

$$\vec{F}(\vec{x}) \;=\; \vec{G}_1(\vec{H},\vec{x})\,.$$

For the induction case :

$$\frac{A(0) \quad A(n) \rightarrow A(n+1)}{\forall n\, A(n)}$$

suppose inductively that we already have sequences of primitive recursive functionals $\vec{G}$ satisfying

$$A(0)^* \;\equiv\; \forall\vec{x}\,\exists\vec{y}\,B(0,\vec{x},\vec{y})$$

and $\vec{H} = \vec{H}_0, \vec{H}_1$ satisfying

$$(A(n) \rightarrow A(n{+}1))^* \;\equiv\; \forall\vec{z},\vec{x}\,\exists\vec{u},\vec{y}\,(B(n,\vec{u},\vec{z}(\vec{u})) \rightarrow B(n{+}1,\vec{x},\vec{y})).$$

Then for all $\vec{x}$,

$$B(0,\vec{x},\vec{G}(\vec{x}))$$

and for all $\vec{x}$ and $\vec{z}$,

$$B(n,\vec{H}_0(n,\vec{z},\vec{x}),\vec{z}(\vec{H}_0(n,\vec{z},\vec{x}))) \;\rightarrow\; B(n+1,\vec{x},\vec{H}_1(n,\vec{z},\vec{x})).$$

We now have to construct a sequence of primitive recursive functionals $\vec{F}$ satisfying $\forall n A^*$, i.e. such that for all numbers n and all $\vec{x}$,

$$B(n,\vec{x},\vec{F}(n)(\vec{x})).$$

So define $\vec{F}$ by primitive recursion from $\vec{G}$ and $\vec{H}_1$ as follows :

$$\vec{F}(0) \;=\; \vec{G} \quad\text{and}\quad \vec{F}(n+1) \;=\; \vec{H}_1(n,\vec{F}(n)).$$

Then we prove that for all $\vec{x}$, $B(n, \vec{x}, \vec{F}(n)(\vec{x}))$ by induction on n. The base case $n = 0$ is immediate from the property of $\vec{G}$ above. For the induction step from n to $n+1$ assume that for all $\vec{u}$, $B(n, \vec{u}, \vec{F}(n)(\vec{u}))$. In particular this holds for $\vec{u} = \vec{H}_0(n, \vec{F}(n), \vec{x})$ so if we set $\vec{z} = \vec{F}(n)$ the premise of the above implication defining $\vec{H}$ is satisfied. But the conclusion is then just

$$B(n+1, \vec{x}, \vec{H}_1(n, \vec{F}(n), \vec{x}))$$

and by the definition of $\vec{F}(n+1)$ this is equivalent to

$$B(n+1, \vec{x}, \vec{F}(n+1)(\vec{x}))$$

as required. This completes the induction case.

Theorem 6.2 *The provably recursive functions of full Peano Arithmetic are exactly the primitive recursive functionals of function-type* $0, \ldots, 0 \to 0$.

Proof. We only prove one half of the theorem here, that every provably recursive function of PA is Gödel primitive recursive. The other half is beyond the scope of these lectures. So suppose f is provably recursive in PA. Then there is a 'bounded' formula C_f describing the computation of f such that

$$PA \vdash \forall \vec{x} \exists y \, C_f(\vec{x}, y).$$

But this formula is already in 'generalized' form and the variables are all of type 0. Therefore by the theorem above, there is a primitive recursive functional F of type $0, \ldots, 0 \to 0$ such that for all $\vec{n} = n_1, \ldots, n_k \in N$ we have

$$C_f(\vec{n}, F(\vec{n})).$$

The function f can thus be defined primitive recursively from F as follows :

$$f(\vec{n}) = \mathrm{Val}(F(\vec{n})).$$

Examples of primitive recursive functionals. At each level k we can define G of special type $k+1$ and H of type $0 \to k+2$

by explicit definitions as follows, where n, x, f, g are variables of types $0, k-1, k, k+1$ respectively :

$$G(f)(x) \;=\; x \quad \text{and} \quad H(n)(g)(f)(x) \;=\; f(g(f)(x))\,.$$

Then we can define the iterator It of type $0 \to k+1$ by primitive recursion thus :

$$It(0) \;=\; G \quad \text{and} \quad It(n+1) \;=\; H(n)(It(n))$$

and it is easy to see that

$$It(n)(f)(x) \;=\; f(It(n-1)(f)(x)) \;=\; \ldots \;=\; f^n(x).$$

Now with $k=1$ define I_2 from It by

$$I_2(f)(n) \;=\; It(n)(f)(n)$$

so that

$$I_2(f)(n) \;=\; f^n(n).$$

Then starting with the successor s of type $0 \to 0$ we can define a hierarchy of functions $F_m : N \to N$:

$$F_0 \;=\; s \quad \text{and} \quad F_{m+1} \;=\; I_2(F_m).$$

Therefore F_m is the result of iterating I_2, starting with the successor, and we can express this neatly using the 'next level' iterator It of type $0 \to 3$:

$$F_m \;=\; I_2^m(s) \;=\; It(m)(I_2)(s).$$

Hence the Ackermann-Péter function F can be defined by :

$$F(m,n) \;=\; It(m)(I_2)(s)(n)$$

and this is the first point at which we see the power of 'higher type' primitive recursion, since the use of the type-2 functional I_2 is crucial here in defining the number-theoretic function F. For if we could define F without using higher types then it would be primitive recursive in the ordinary sense, and we know this is not the case.

We can next define

$$I_3(g)(f)(n) = It(n)(g)(f)(n)$$

and move up to the iterator of type $0 \rightarrow 4$ so as to define a new number-theoretic function

$$F'(m,n) = It(m)(I_3)(I_2)(s)(n)$$

which is far far bigger than F, and we can continue in this way to define

$$F''(m,n) = It(m)(I_4)(I_3)(I_2)(s)(n)$$

etcetera.

All of these functions $F, F', F'', \ldots$ are provably recursive in PA and the use of higher type levels in defining them reflects the use of inductions of ever greater logical complexity in their termination proofs. Another way to analyse complex inductive proofs is to use transfinite ordinals instead of higher types. See for instance Buchholz and Wainer (1987), Schwichtenberg (1977) and Schwichtenberg and Wainer (1995).

REFERENCES.

(1) P. Aczel, H. Simmons, S. Wainer (Eds), *Proof Theory*, Cambridge 1992.

(2) Buchholz W. and Wainer S.S. *Provably Computable Functions and the Fast Growing Hierarchy.* in S.G. Simpson (Ed.), Contemporary Mathematics Vol. 65, 'Logic and Combinatorics', American Math. Society 1987, 179-198.

(3) L. Colson, *About Primitive Recursive Algorithms*, in Proceedings ICALP '89, Springer Lecture Notes in Computer Science 372, 194-206.

(4) M.V. Fairtlough and S.S. Wainer, *Ordinal Complexity of Recursive Definitions*, Information and Computation Vol. 99, 1992, 123-153.

(5) S. Feferman, *Logics for Termination and Correctness of Functional Programs II, Logics of Strength PRA*, in "Proof Theory", ref. (1) above, 195-225.

(6) J.Y. Girard, Y. Lafont, P. Taylor, *Proofs and Types*, Cambridge 1989.

(7) J.Y. Girard, *Proof Theory and Logical Complexity*, Bibliopolis 1987.

(8) W. Howard, *The Formulae as Types Notion of Construction* , in J.R. Hindley and J.P. Seldin (Eds.) : To H.B. Curry - Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press 1980.

(9) S.C. Kleene, *Introduction to Metamathematics* , North Holland 1952.

(10) G. Mints, *Quantifier - Free and One - Quantifier Systems* , J. Soviet Math. Vol. 1, 1973, 71-84.

(11) C. Parsons, *On n-Quantifier Induction.* J. Symbolic Logic Vol. 37, 1972, 466-482.

(12) R. Péter, *Recursive Functions*, Academic Press 1967.

(13) H.E. Rose, *Subrecursion - Functions and Hierarchies* , Oxford 1984.

(14) H. Schwichtenberg, *Proof Theory - Some Applications of Cut Elimination* , in J. Barwise (Ed.) : Handbook of Mathematical Logic, North Holland 1977, 867-896.

(15) H. Schwichtenberg and S.S. Wainer, *Ordinal Bounds for Programs*, in P. Clote and J. Remmel (Eds.) : "Feasible Mathematics II", Birkhäuser Progress in Computer Science and Applied Logic Vol. 13, 1995, 387-406.

(16) J.R. Shoenfield, *Mathematical Logic* , Addison Wesley 1967.

(17) W. Sieg, *Herbrand Analyses*, Archive Math Logic Vol.30, 1991, 409-441.

(18) W. Sieg and S.S. Wainer, *Program Transformation and Proof Transformation* , in E. Börger, Y. Gurevich, K. Meinke (Eds.) : "Computer Science Logic '93", Springer Lecture Notes in Computer Science 832, 1994, 305-317 .

(19) W.W. Tait, *Normal Derivability in Classical Logic* , in J. Barwise (Ed.) "The Syntax and Semantics of Infinitary Languages", Springer Lecture Notes in Math 72, 1968, 204-236.

(20) J.V. Tucker and J.Y. Zucker, *Provably Computable Functions on Abstract Data Types* , in "Proof Theory", ref. (1) above.

(21) S.S. Wainer and L. Wallen, *Basic Proof Theory*, in "Proof Theory", ref. (1) above, 3-26.