FOUNDATIONS OF LOGIC PROGRAMMING
COS333 Study Booklet: Dept. of Computer Science: Univ. of Pretoria

This study booklet covers only **Part B** of course COS333.[1]
Separate study material exists for the other Parts A, C, D.

Advice on Reading!
Please *read 'ahead'* of the lectures, such as to know what comes.
Read **5 pages per day**, in a day-by-day flow without interrupt.

Contents

[1] *The material presented in this course is based on lectures publicly taught by Professor Klaus Indermark at the RWTH Aachen (Germany), which were subsequently translated into the English language, and thereafter slightly modified such as to fit the purpose of course COS333 at the University of Pretoria. All the contents of this study booklet is traditional standard knowledge of computer science, and is therefore nobody's intellectual property. Thus this booklet is free for use, not copyright-protected, and* **not for sale**. As with all books, also these Lecture Notes might possibly contain some 'typo' errors, ambiguities, etc. If you spot any mistake, or anything that 'looks strange' to you, then please do not hesitate to contact your lecturer about the problem which you have spotted.

1 INTRODUCTION: Why Logic Programming?

Logic Programming is a "natural" generalisation of Functional Programming: LP is —mathematically— based on *Relations*, and Relations are generalisations of Functions; (every Function is a Relation, but not every Relation is a Function).

Thus: whereas in FP there is still one sense of direction: Input $\Longrightarrow$ Output (Function), in LP the difference between Input and Output vanishes. In LP, *Input and Output become interchangeable*: Input $\Longleftrightarrow$ Output (Relation).

By the way: The mathematical theory of Relations thus also provides a conceptual "bridge" between Logic Programming and Relational Database Systems which are based on Relations, too, (though their operational mode is technically different).

From a *syntactical* point of view, Logic Programs also have some similarity with Formal Grammars (in the Chomsky hierarchy) and Term Replacement Systems, which are all characterized by their typical LHS :— RHS structure (LHS = Left Hand Side, RHS = Right Hand Side), and are thus 'evaluated' (at run-time) in a similar manner like these.

Even more so than it was already the case in Functional Programming, in Logic Programming we mostly *specify* the problem that we would like to see solved, and we get the solving implementation (algorithm) so-to-say 'for free' (by the abstract machine that interprets our specification).

Basic Features of logic-oriented programming (LOP) languages are thus:
— *Relations* as fundamental objects
— Definition of those relations through *formulae of logics*
— *Facts* and *Rules* as the basis of a computable *formal calculus*
— Computation through *Resolution* and *Unificaton*

Advantages: The description of a problem is already a program, such that "algorithmic thinking" is no longer dominant in this computational paradigm.

Disadvantages: *Large Search-Spaces*to be explored (by the built-in resolution- and unification "engine"), as well as *low* run-time *efficiency* (because of the

inherent relational Non-Determinism).

Classification of LOP Languages: Depending on whether their internal search engine proceeds from *Premises* to *Implications*, or (vice versa) from Implications to Premises, we can basically classify all LOP Languages into forward-chaining versus backward-chaining.

Examples:

— OPS5: forward-chaining

— PROLOG: backward-chaining

Possibility of Concurrency and Parallelism: Due to their high level of abstraction and absence of stepwise algorithms, LOP Languages lend themselves quite well to implementations on Concurrent and Parallel Computers or Transputers with many processors. Concurrent Prolog / Parlog are examples of such programming languages. The parallelisation of Logic Programming can thus somewhat 'mitigate' its conceptually inherent inefficiency.

Practical APPLICATIONS particularly in Artificial Intelligence:

— Knowledge Representation and Expert Systems: description of "worlds"

— *Deductive* Data-Bases (DDB): finite description of *infinite* data sets.[2]

CLASSICAL LITERATURE on LP (just a few books out of many):

- J.W. Lloyd, *Foundations of Logic Programming*, Springer, 1987$^{(2)}$.
  - with particular emphasis on the Mathematical Fixpoint Semantics
- J.H. Gallier, *Logic for Computer Science*, Wiley, 1987.
  - with particular emphasis on Machine-Automated Theorem-Proving on the basis of a Formal-Logical Calculus
- W.F. Clocksin & C.S. Mellish, *Programming in Prolog*, Springer, 1987$^{(3)}$.
  - for a long time "the" reference book to Prolog

[2] A classical Relational Data-Base (RDB) must list all its data tuples and tables *explicitly* and can thus only represent a *finite* set of data.

- L. Sterling & E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT-Press, 1986.
  - very comprehensive collection of sophisticated practical programming tricks and techniques in Prolog
- H. Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT-Press, 1991.
  - how Logic Programs are interpreted by an imperative computer
- M. Ben-Ari, *Mathematical Logic for Computer Science*, Springer, $2004^{(2)}$.
  - A nice and concise "all-you-need"-book, which covers many topics

2 LOGIC: History and Basic Concepts

Antiquity: Logic as "practical philosophy" of truth-finding (*Aristotle*).

Middle Age: *Scholastic* Logic comprising chains of Syllogisms, with practical application also in Legal and Theological disputes.

Modernity: *Leibniz* began with first steps towards the *formalization* of Logic and also introduces for the first time a binary 0-1-System.

Boole, with his Boolean Algebra, provides a conceptual "bridge" between the 0-1-Number-System and the True/False-values of classical Logic. This lead to the notion of *Mathematical* Logic (in contrast to the older Philosophical Logic).

Frege, with his seminal "Begriffsschrift", makes the decisive step forward towards our contemporary syntax of formal logic notations. Frege also pointed out, most clearly, the difference between Syntax and Semantics.

Hilbert still believed that every problem of classical and modern Mathematics could be represented and tackled with help of some suitable formalized Logical Calculus.

Gödel proved —against Hilbert— that there exist mathematical problems of such kind that they cannot be solved (or proven) by any formal logical calculus whatsoever! Gödel's fundamental result thus also

provides a conceptual bridge to the Non-Computable Functions in the Theory of Turing Machines, (Halting Problem, etc.)!

Propositional Logic (PL): abstracts away from individual entities (and their relations) about which something is being asserted. "Sentences", which can be true or false, are the basic "atomic" concepts of this most simple type of Logic.[3]

Predicate Logic of First Order (PL1): also known as First-Order-Logic (FOL) makes more detailed assertions about "worlds" including *individuals*, which can be distinguished, and their relations. Those can be interpreted as elements of Algebraic Structures including Sorts (Types), Terms, and Operations.[4]

FOL-Formulae are expressed as/by Predicates over Terms to express relational dependencies, plus the simple logical operators from PL (see above), plus existential ($\exists$) as well as universal ($\forall$) Quantification of Individuals.

Higher-Order and Modal Logics: also exist for various highly specialised areas of application, but are NOT part of this course COS333.

Logic Programming: *Horn*-Logic as a specifically *restricted* variant of FOL, with *Resulution* (instead of a Hilbert- or Gentzen-calculus) used as its formal computable Calculus.[5]

3 PROPOSITIONAL LOGIC and Resolution

3.1 SYNTAX and SEMANTICS

Definition: *Syntax of PL.*
Let $\mathbf{A} := \{\mathsf{A}_i \mid i \in \mathbb{N}\}$ be a countable set of Variables, and $V \subseteq \mathbf{A}$. Then

[3] Do not confuse PL (Propositional Logic) with LP (Logic Programming).

[4] Recall the COS333 Study Booklet for COS333 Part A on Functional Programming!

[5] In formal logic, the word "calculus" does NOT have the same meaning as the word "calculus" which you know from pre-university school mathematics! In pre-university school mathematics "calculus" means specifically the operations with Numbers, whereas in formal logic "calculus" means —more generally— the operation with any kinds of abstract Formulae.

the set $\mathrm{PLF}(V)$ of PL-Formulae over V is inductively (recursively) defined as follows:[6]

— $V :\subseteq \mathrm{PLF}(V)$ *// comment: each Variable is an (atomic) Formula.*
— if $F \in \mathrm{PLF}(V)$, then also $\neg F \in \mathrm{PLF}(V)$.
— if both $F, G \in \mathrm{PLF}(V)$, then also $(F \wedge G) \in \mathrm{PLF}(V)$.
— if both $F, G \in \mathrm{PLF}(V)$, then also $(F \vee G) \in \mathrm{PLF}(V)$.

Abbreviations ("syntactic sugar") for any $F, G \in \mathrm{PLF}$:

$(F \to G) := (\neg F \vee G)$ *// comment: "implication"*
$(F \leftrightarrow G) := ((F \to G) \wedge (G \to F))$ *// comment: "equivalence"*
$\vee_{i=1}{}^{n} F_i := (((\cdots (F_1 \vee F_2) \vee F_3) \cdots \vee F_n)$
$\wedge_{i=1}{}^{n} F_i := (((\cdots (F_1 \wedge F_2) \vee F_3) \cdots \wedge F_n)$
$\mathbf{1} := (F \vee \neg F)$ *// comment: "tautology, trivial truth"*
$\mathbf{0} := (F \wedge \neg F)$ *// comment: "antinomy, contradiction, inconsistency"*[7]

Definition: *Truth-Value-SEMANTICS for PL*
Let $\mathbb{B} := \{0, 1\}$ the (binary / dual / dichotomic) set of two (Boolean) Truth Values,[8] and let the mapping-function $\alpha : V \longrightarrow \mathbb{B}$ provide a basic truth-interpretation of the (atomic) variables.[9] Then α can be *continued* to $\bar{\alpha} : \mathrm{PLF}(V) \longrightarrow \mathbb{B}$ as follows:

— $\bar{\alpha}(\mathsf{A}_i) := \alpha(\mathsf{A}_i)$ for all atomic variables $\mathsf{A}_i \in V$
— $\bar{\alpha}(F \wedge G) := 1$ if and only if $\bar{\alpha}(F) = 1$ **and** $\bar{\alpha}(G) = 1$
— $\bar{\alpha}(F \vee G) := 1$ if and only if $\bar{\alpha}(F) = 1$ **or** $\bar{\alpha}(G) = 1$ (or both)
— $\bar{\alpha}(\neg F) := 1$ if and only if $\bar{\alpha}(F) = 0$ *// "Tertium non datur"*[10]

[6] Singular: one formul**a**, Plural: many formula**e**.

[7] Do not confuse the syntactic formula **1** with the semantic truth-value 1 (true)! Likewise, do not confuse the syntactic formula **0** with the semantic truth-value 0 (false)! To link the one with the other one, i.e.: to provide a bridge from syntaxt to semantics, we first need an interpreting mapping!

[8] In modern Logics there also exist semantic systems which provide *more* than two Truth-Values, for example three: 0, 1, and "U" (with "U" representing "unknown"). There even exist special semantic systems which provide *infinitely many* Truth-Values, such as: 1, 2, 3, 4, 5,... However we do not consider such systems in COS333.

[9] Do *not* confuse the *logic implication* arrow $(F \to G)$ with the *functional mapping* arrow $\alpha : V \longrightarrow \mathbb{B}$.!

[10] This semantic rule reflects a fundamental axiom of classical Aristotelian Logic since the times of Antiquity. In modern logics, however, there also exists other systems in which this semantic rule is *not* presupposed. In such kind of Logics it would thus *not* be allowed to conclude that F is false if not-F is true. Such kind of Logics are called "intuitionist" or "constructivist" Logics and were advocated in the 1920s —against Hilbert— by Brouwer and Heyting. In this course COS333, however, we do not discuss such "intuitionist" Logics any further, (see a book on the History of Mathematics for an in-depth discussion).

Definition: *MODEL, CONSISTENCY (Satisfiability), VALIDITY*
Given a formula $F \in \mathrm{PLF}(V)$ and a truth-value-mapping (Interpretation) $\alpha : V \longrightarrow \mathbb{B}$.

- If $\bar{\alpha}(F) = 1$, then we write "$\alpha \models F$", and we speak: "α *is a Model* for (of) F", or: "F *holds* under the Interpretation α", or: "α *fulfills* F", or: "α *satisfies* F".
- $F \in \mathrm{PLF}(V)$ is called *consistent* (or *satisfiable*) if there exists at least one Model for F. Otherwise F is called *inconsistent* (or contradictory, or an antinomy).
- $F \in \mathrm{PLF}(V)$ is called *valid* (or tautologically true) if and only if $\alpha \models F$ FOR ALL (or for *any possible*) Interpretations $\alpha : V \longrightarrow \mathbb{B}$. This case we denote as "$\models F$".

Obviously we have $\models \mathbf{1}$, with Formula $\mathbf{1}$ as defined above :)

THEOREM! For all $F \in \mathrm{PLF}(V)$, $\models F$ if and only if $\neg F$ is an antinomy.
(This Theorem is a basis for Resolution in Logic Programming)

Semantic Notion of Implication (Consequence): results, model-theoretically, from the above-defined notion of general Validity ("for all...") as follows.

Definition: *CONSEQUENCE SET*
Let there be $\Phi \subseteq \mathrm{PLF}(V)$ and a formula $F \in \mathrm{PLF}(V)$.[11] Then we say "F is a consequence of Φ", and write $\Phi \models F$,[12] IF, in ALL Models of Φ, F will be true too. In other words, FOR ALL ($\alpha : V \longrightarrow \mathbb{B}$): IF α satisfies EACH formula $G \in \Phi$, AND $\Phi \models F$, THEN α also satisfies F. The set $\mathbf{CONS(\Phi)} := \{F \in \mathrm{PLF}(V) \mid \Phi \models F\}$ is called the *Consequence Set of* Φ.[13]

THEOREM! For any selection of formulae $F_1, F_2, \ldots, F_k, F \in \mathrm{PLF}(V)$:
$\{F_1, F_2, \ldots, F_k\} \models F$ **if and only if** $(\wedge_{i=1}{}^{k} F_i) \wedge \neg F$ is an Antinomy.

[11] Later, in the practice of LP, Φ shall represent a program ('world'), and F shall represent a 'query'.
[12] Note that the symbol "$\models$" is used in an 'over-loaded' manner in two different contexts: in $\alpha \models F$ whereby α is a truth-value-assignment for Variables in V, as well as in $\Phi \models F$ whereby Φ is a sub-set of Formulae from $\mathrm{PLF}(V)$. Please do not get confused about such 'over-loading' of "$\models$".
[13] Later, this set will be used to define the **Declarative Semantics** of Logic Programs.

Comments on the Theorem of above:

- The Antinomy-Test for PL is generally very hard to compute, namely NP-Complete. However for *Horn*-Logic, which is the basis of Logic Programming, this test is *linear* $\mathcal{O}(n)$.[14]

  The problem "is $F \in \mathrm{CONS}(\{F_1, \ldots, F_k\})$?" is in principle *decideable* (computable) —though NP-Complete— because the problem "is $G := (\wedge_{i=1}{}^{k} F_i) \wedge \neg F$ an antinomy?" is decideable, too.

  The NP-Completeness of the Antinomy Test results from the fact that (in the Worst Case) 2^n *Truth Tables* would have to be checked in order to find out that FOR ALL $(\alpha : \{\mathsf{A}_1, \ldots, \mathsf{A}_n\} \longrightarrow \mathbb{B})$: $\bar{\alpha} = 0$.

  For practical purposes (LP) it is thus imperative to work on smaller Sub-Logics of PL, e.g.: only Horn-Formulae, in which the Antinomy-Test can be computed considerably faster (and without resorting to Truth-Tables).

- The theorem of above is important for LP in the following interpretation:

  $\Phi := \{F_1, \ldots, F_k\}$ will be a *Logic Program*,

  $\mathrm{CONS}(\Phi)$ will be its *Declarative Semantics*,[15]

  $F \in \mathrm{PLF}(V)$ will be a *Query* to Φ (a.k.a. 'Goal'),

  i.e.: "is $F \in \mathrm{CONS}(\Phi)$**?**"

COMPACTNESS THEOREM (a.k.a. *Finity Theorem*)
FOR ALL $\Phi \subseteq \mathrm{PLF}(V)$:
- Φ is satisfiable **if and only if** *every finite* subset of Φ is satisfiable
- Φ is *un*-satisfiable **if and only if** exists a *finite un*-satisfiable subset of Φ

This theorem will become important later when we make the transition from Propositional Logic (PL) to First Order (Predicate) Logic (FOL). The

[14] Please see a book on Complexity Theory, where the NP-Completeness for the Satisfiability Problem (SAT) in PL will be demonstrated as *the* "classical" Result in Complexity Theory — **Cook**'s Theorem.

[15] whereas its Operational Semantics will be provided by the Resolution Calculus — also remember Part A (Functional Programming) of this course COS333, where the differences between the concepts 'Denotational Semantics' and 'Operational Semantics' were discussed, too.

theorem guarantees us that we need not browse forever through infinite sets of formulae in order to find answers to our 'queries' in LP.

3.2 NORMAL FORMS

The formulae, which constitute a Logic Program, cannot be written in any arbitrary manner. They must be written in some standardised form, towards which we are now working. For this purpose, however, we must be able to guarantee that some formula in normalised form is equivalent to its corresponding formula in non-normalised form. Therefore we must first look at the definition of 'equivalence'. Thereafter we can speak about normalisation and normal forms.

Definition: *Semantic EQUIVALENCE*
Two syntactically different formulae $F, G \in \mathrm{PLF}(V)$ are called *semantically equivalent* **if and only if** FOR ALL interpretations $(\alpha : V \longrightarrow \mathbb{B})$: $\bar{\alpha}(F) = \bar{\alpha}(G)$. In this case we write: "$F \equiv G$".

Lemma! $F \equiv G$ **if and only if** the formula $F \leftrightarrow G$ represents a Tautology.

Important Note: Whereas "$F \leftrightarrow G$" *is* a Formula —Syntax!— in $\mathrm{PLF}(V)$, "$F \equiv G$" —Semantics!— is *not*.

Definition: *CONJUNCTIVE Normal Form*
A formula $F \in \mathrm{PLF}(V)$ is in conjunctive normal form, **CNF**, if and only if it is syntactically structured as $(\wedge_{i=n}{}^{n}(\vee_{j=1}{}^{m_i} L_{i,j}))$, whereby each $L_{i,j}$ is a **Literal** in $\mathcal{L}$.

Definition: *LITERALs*
The set of *positive* Literals, POS, is defined as: V — all Variables are Literals. The set of *negative* Literals, NEG, is defined as: $\{\neg \mathsf{A}_i \mid \mathsf{A}_i \in V\}$ — all negated Variables are Literals. The set of all Literals is defined as $\mathcal{L} := \mathrm{POS} \uplus \mathrm{NEG}$.

Theorem! FOR EACH $F \in \mathrm{PLF}(V)$, a semantically equivalent CNF F' exists *and* can be effectively constructed.

Definition: *HORN Formula*[16]
$F \in \mathrm{PLF}(V)$ is a *Horn formula* **if and only if** F is in CNF **and** every sub-formula $(\vee_{j=1}{}^{m_i}\ L_{i,j})_i$ in F contains MAXIMALLY ONE POSITIVE Literal.

Motivation: Because of their syntactic restriction, Horn formulae can be re-represented by using the two logical operators "$\wedge$" and "$\rightarrow$" only, *without* the operator "$\vee$". **Horn Logic** will thus capture only "positive knowledge", such as it could be explicitly collected in a database. Thereby, *sub-classes* of Horn formulae can be practically interpreted, in LP, as follows:

- NO Positive Literal at all: "**Query**" (to be asked)
- NO Negative Literal at all: "**Fact**" (as in database)
- With ONE Positive and SOME Negative Literal(s): "**Program**"

Example: The formula $H := (\mathsf{A} \vee \neg\mathsf{B}) \wedge (\neg\mathsf{C} \vee \neg\mathsf{A} \vee \mathsf{D}) \wedge (\neg\mathsf{A} \vee \neg\mathsf{B}) \wedge \mathsf{D} \wedge \neg\mathsf{E}$ is in CNF and is also a Horn formula. It can be re-written as
$H' := (\mathsf{A} \rightarrow \mathsf{B}) \wedge ((\mathsf{C} \wedge \mathsf{A}) \rightarrow \mathsf{D}) \wedge ((\mathsf{A} \wedge \mathsf{B}) \rightarrow \mathbf{0}) \wedge (\mathbf{1} \rightarrow \mathsf{D}) \wedge (\mathsf{E} \rightarrow \mathbf{0})$, with $H' \equiv H$.[17]

Important Note: Though every Horn formula is in CNF, *not* every formula in CNF is a Horn formula! There exist formulae which *cannot* be "horned", namely all such formulae which represent either "negative" or "un-precice" knowledge like "$\mathsf{A} \vee \mathsf{B}$"! Practically, however, Horn Logic is "good enough" for the purpose of LP, and —most importantly— the *Satisfiability Problem* in Horn Logic can be *efficiently* solved (whereas the Satisfiability Problem in general PL is NP-Complete).

[16] Mathematician and Logician **Alfred Horn**, 1918-2001, see *http://en.wikipedia.org/wiki/Alfred_Horn*
[17] Please re-study, in your own time, the basic *Boolean Laws* for Formulae-Rewriting.

3.3 RESOLUTION

The **idea** is the following: To figure out if a given formula is a Tautology (always true), we figure out if its Negation (in CNF) is an Antinomy. The "calculus", by means of which we can *effectively and efficiently* figure this out, will be the Resolution Calculus. Such a resolution calculus will then constitute the core of the "Reasoning *Engine*" which must do the algorithmic work "behind the scenes" of any given Logic Program (which is, as such, declarative, i.e.: non-algorithmic).

In the following we assume that all formulae are already in CNF.

For merely practical reasons (convenience and simplicity of notation) we want our Resolution Calculus to work on formulae represented in Clause Notation, which is defined in the following.

Definition: *CLAUSE*
A set $\mathbf{C} \subset \{\mathsf{A}_1, \neg\mathsf{A}_1, \mathsf{A}_2, \neg\mathsf{A}_2, \mathsf{A}_3, \neg\mathsf{A}_3, \ldots\}$ is a *clause* **if** $|\mathbf{C}| < \infty$ (finite). Moreover, $\emptyset := \{\}$ is the *empty* clause.

Example: $\{\mathsf{A}_1, \neg\mathsf{A}_2, \neg\mathsf{A}_3, \mathsf{A}_4, \mathsf{A}_5, \mathsf{A}_6\}$ is a clause.

Definition: *CLAUSE SET of a PL-Formula in CNF*
Given a formula F in PLF(V) and in CNF, such that $F = (\wedge_{i=n}{}^{n}(\vee_{j=1}{}^{m_i} L_{i,j}))$, then $\mathcal{C}(F) := \{\{L_{1,1}, \ldots, L_{1,m_n}\}, \ldots, \{L_{n,1}, \ldots, L_{n,m_n}\}\}$ is the *clause set* of F.

Note that $\mathcal{C}$ is a set of sets, whereas $\mathbf{C}$ is a set of atoms (literals)!

Example: Horn-formula $(\mathsf{A} \vee \neg\mathsf{B}) \wedge (\neg\mathsf{C} \vee \neg\mathsf{A} \vee \mathsf{D}) \wedge (\neg\mathsf{A} \vee \neg\mathsf{B}) \wedge \mathsf{D} \wedge \neg\mathsf{E}$
has the Clause Set $\mathcal{C} = \{\{\mathsf{A}, \neg\mathsf{B}\},\ \{\neg\mathsf{C}, \neg\mathsf{A}, \mathsf{D}\},\ \{\neg\mathsf{A}, \neg\mathsf{B}\},\ \{\mathsf{D}\},\ \{\neg\mathsf{E}\}\}$.

Further Remarks

- For all $F, G \in$ PLF(V) in CNF: $\mathcal{C}(F) = \mathcal{C}(G)$ **if and only if** $F \equiv G$
- *In theory the sequential order of clauses in a clause set is irrelevant*, however in the practice of LP this order is indeed relevant for the 'run' of the resolution algorithm.

- For all interpretations, $\bar{\alpha}(\emptyset) := 0$ (false).

Definition: *RESOLVENT OF TWO CLAUSES in One Resolution Step*
Let C_1, C_2 and R be clauses. R is called *resolvent of* C_1 and C_2 **if and only if** there exists a literal $L \in \mathrm{C}_1$, its negation $\bar{L} \in \mathrm{C}_2$, and
$R = (\mathrm{C}_1 \backslash \{L\}) \cup (\mathrm{C}_2 \backslash \{\bar{L}\})$
whereby $\bar{L} := \neg\mathsf{A}$ if $L = \mathsf{A}$,
otherwise $\bar{L} := \mathsf{A}$ if $L = \neg\mathsf{A}$.
We speak: *"R is resolved from* C_1 *and* C_2 *with regard to L"*, and we may denote this situation graphically as

$$\begin{array}{ccc} \mathrm{C}_1 & & \mathrm{C}_2 \\ & \backslash \quad / & \\ & \mathrm{R} & \end{array}$$

Note that such resolution *step*s are the basis of the *operational semantics* of the Resolution Calculus.[18]

Further Remarks

- The *choice* of an applicable resolution step is, in general, *not unique*.[19]
  FOR EXAMPLE:
  $\{X, \neg Y, Z\}$ and $\{Y, \neg Z\}$ can resolve, via Y, to: $\{X, Z, \neg Z\}$
  $\{X, \neg Y, Z\}$ and $\{Y, \neg Z\}$ can resolve, via Z, to: $\{X, \neg Y, Y\}$
- Complement-Literals $\{L\}$ and $\{\bar{L}\}$ always resolve to $\emptyset$

LEMMA! *SEMANTIC INVARIANCE of One-Step Resolution*
Let $\mathcal{C}$ be a Clause Set, finite and non-empty. For all clauses $\mathrm{C}_1, \mathrm{C}_2 \in \mathcal{C}$: **If**

$$\begin{array}{ccc} \mathrm{C}_1 & & \mathrm{C}_2 \\ & \backslash \quad / & \\ & \mathrm{R} & \end{array}$$

[18] For comparison, remember the One-Step Evaluation Relation "$\Longrightarrow$" on Computational Terms (COMP $\times$ COMP) in the Operational Semantics of Functional Programs, as discussed in Part A of this course COS333.

[19] Remember, again, the Computation-Independent Semantics in Functional Programming (Part A of this course), where we could also have more than one possibility of selecting the Term which we wanted to evaluate next.

then $\mathcal{C}$ is semantically equivalent to $\mathcal{C} \cup \{\mathrm{R}\}$

Comment to the Lemma: *Logical Deduction* (or Inference) *cannot create new information!* By resolution (which is a form of logical deduction or inference) we can only *reveal* information which was already implicitly "hidden" in what we had been given at the very beginning of the deduction process.

Definition: *Resolution CALCULUS*
Let $\mathcal{C}$ be a Clause Set.
$\mathbf{RES}(\mathcal{C}) := \mathcal{C} \cup \{R \mid R$ is One-Step-Resolvent of any two $\mathrm{C}_i, \mathrm{C}_j \in \mathcal{C}\}$, a.k.a "the first closure around $\mathcal{C}$" or "Breadth", is the **Resolvent Expansion** of $\mathcal{C}$.[20] On this basis, the *Resolution Calculus* is characterised inductively (recursively) by the following three properties:

— $\mathrm{RES}^0(\mathcal{C}) := \mathcal{C}$ ..... no resolution
— $\mathrm{RES}^{n+1}(\mathcal{C}) := \mathrm{RES}(\mathrm{RES}^n(\mathcal{C}))$ for $n \geq 0$ ..... "Depth"
— $\mathrm{RES}^*(\mathcal{C}) := \cup_{n \in \mathbb{N}} \mathrm{RES}^n(\mathcal{C})$ ..... "Resolution Closure of $\mathcal{C}$"

This Resolution Calculus is not the only formal deduction calculus in formal logics: other "classical" calculi, such as the *Hilbert* calculi or the *Gentzen* calculi, are also well-known. For our specific purpose of **LP**, however, the Resolution Calculus is the calculus of choice.[21]

LEMMA! $|\mathrm{RES}^*(\mathcal{C})| < \infty$, **if** $|\mathcal{C}| < \infty$

Consequence from the Lemma: $\mathcal{C} \equiv \mathrm{RES}^*(\mathcal{C})$, semantically equivalent.

THEOREM! *CORRECTNESS AND COMPLETENESS of the Resolution Calculus with regard to the Non-Satisfiability of PL-Formulae in CNF*
For any given Clause Set $\mathcal{C}$ and Basic Interpretation α,
$\bar{\alpha} \not\models \mathcal{C}$ **if and only if** $\emptyset \in \mathrm{RES}^*(\mathcal{C})$

[20] Remember again Part A (Functional Programming) of this course, where the total Reduction Semantics of a Function Scheme was also defined on the basis of the One-Step Semantics.

[21] Yet another formal deduction calculus, namely the *Hoare* calculus, will be introduced for the purpose of *Imperative* Programming (IP) later in Part C of this course COS333.

Consequence from the Theorem: For a formula F in CNF we can determine, by resolution, in a *finite number of steps* whether or not F is satisfiable. We compute RES steps, until $\emptyset$ appears in $\mathrm{RES}^*(\mathcal{C}(\mathcal{F}))$: in this case, F is an Antinomy. In case that $\emptyset \notin \mathrm{RES}^*(\mathcal{C}(\mathcal{F}))$, F can be satisfied (F has a model). In general, the $\mathcal{O}$-runtime-complexity of grows *exponentially* with the numbers of variables contained in F. Practically this calls for suitable Resolution *Strategies*,[22] by means of which $\emptyset$ could possibly be derived *faster*. This leads us to the sub-topic of *restricted* resolution. Such restrictions, for the sake of better search-efficiency in the Clause-space, could be achieved

— by Strategies to control the search-"path" in the possibility-space, or

— by *Restrictions* on the Type of Clauses to be admitted for resolution.

3.4 SLD Resolution for HORN Clauses

SLD Resolution stands for **Selective Linear Definite** clause resolution, and is the resolution technique of choice for LP. Note that the SLD technique is *not complete* when attempted at formulae which are *not* Horn formulae. "Not complete" means that the technique would possibly *fail to find* $\emptyset$ in $\mathrm{RES}^*(\mathcal{C})$, even in a case where $\emptyset \in \mathrm{RES}^*(\mathcal{C})$ exists in reality. To grasp the principle of SLD resolution, we must briefly recapitulate the different *classes* of Clauses which can possibly occur in LP:

DEFINITE Clauses a.k.a. Program Clauses represent positive Knowledge about the Rules and the Facts which constitute the "world" of a Logic Program:

FACTS are clauses of type $\{\mathsf{A}_0\}$

RULES are clauses of type $\{\neg\mathsf{A}_1, \neg\mathsf{A}_2, \ldots, \neg\mathsf{A}_n, \mathsf{A}_0\}$

NEGATIVE Clauses a.k.a. Goal Clauses or Queries have the form

$\{\neg\mathsf{A}_1, \neg\mathsf{A}_2, \ldots, \neg\mathsf{A}_n\}$

On this basis, SLD Resolution always begins with a Query. Through resolution with a Program Clause *one Positive Literal must vanish*, such that a new Negative Clause —called Sub Goal— must come into existence. This

[22] Remember Part A (Functional Programming) of this course, where we also distinguished different Evaluation *Strategies* for Terms as well as for entire Function Schemes.

Sub Goal is taken as the next Query, against which another Program Clause will be resolved, etc. This is a procedure of linear $\mathcal{O}$-runtime-complexity, linear in the size of the initially given Clause Set.

EXAMPLE
$\{\neg \mathsf{A}, \neg \mathsf{B}\}$ is a (negative) Goal Clause,
$\{\neg \mathsf{D}, \mathsf{B}\}$ is a (definite) Rule Clause,
Resolution of both, via B, yields $\{\neg \mathsf{A}, \neg \mathsf{D}\}$ as a new (negative) Sub Goal.

This example also explains why SLD Resolution (such as in the "engine" of the LP language PROLOG) is a form of *backward* chaining: in order to prove some Goal, we must first prove all its Sub Goals, and in order to prove a Sub Goal we must first prove all its Sub Sub Goals, etc., whereby the "flow direction" of the resolution procedure goes in the opposite direction "against" the Logical Implication arrow "$\rightarrow$" in the Definite Clauses.[23]

THEOREM! For *Horn* clauses, SLD resolution is *complete*.

4 PREDICATE LOGIC of First Order and Resolution

The "atoms" in PL represent only simple sentences such as, for example, $\mathsf{A} :=$ *"Peter's house is small"*. For the practical purpose of LP, however, such atomic sentences are not particularly interesting. In LP we want to be able to program also generalised situation schemas, such as, for example: *"Wherever two individuals x and y have the same father and the same mother, x and y are siblings"*. For such purposes we need another type of formal logic which is 'more powerful' in its expressions than the simple logic PL. The First Order Logic **FOL** is such a logic. As usual we must first understand its syntax and semantics, and thereafter we will see how FOL could be utilized in LP by means of its own Resolution Calculus.

[23] Remember that a Clause of the form $\{\neg \mathsf{A}_1, \neg \mathsf{A}_2, \ldots, \neg \mathsf{A}_n, \mathsf{A}_0\}$ is equivalent to the Implication formula $(\mathsf{A}_1 \wedge \mathsf{A}_2 \wedge \cdots \wedge \mathsf{A}_n) \rightarrow \mathsf{A}_0$ in PLF(V).

4.1 SYNTAX and SEMANTICS of FOL

Similar to Part A (Functional Programming) of this course COS333, we will start with the definition of the concept of a *signature*. Now, however, our signature will be based on **only one Sort** (whereas in Functional Programming we had *many* Sorts).[24] This one Sort is the "Anything" Sort of *Individual Items* (houses, cars, people, bananas, terms, words, whatever...) which can be captured by un-typed individual variables like x, y, z.[25]

4.1.1 Syntax

Definition: *SIGNATURE*
Let Ω and Π be sets, and let there be a mapping $\varrho : (\Omega \cup \Pi) \longrightarrow \mathbb{N}$ into the Natural Numbers. Then the Triple $\Sigma :=< \Omega, \Pi, \varrho >$ is called a **Signature**, whereby
— Ω is its set of *Operation* symbols,
— Π is its set of *Predicate* symbols,
— ϱ provides the "arity" (number of "input variables") for all the operation symbols as well as all the predicate symbols.

Comment on "Predicates": There is a deep mathematical as well as philosophical relationship between the notions of 'Set', 'Relation', 'Predicate', and 'Property', such that the conceptual borderline between these notions is often somewhat blurred. Predicates typically *express* properties (for example: "*is* blue", or: "*has* children"). Items which somehow share some common property are thereby related with each other, which can be expressed again by their membership in a suitable set. In other words, there is a 'semantic connection' between the field of Logic and the field of Set Theory in Discrete Mathematics.

[24] Multi-sorted Logics also exist, however we do not consider them in this course COS333.

[25] In FOL, these 'individual items' *cannot* be composite entities such as *Sets of* Things — otherwise we would already 'climb up' into Second Order Logic. For the purpose of LP, Second and Higher Order Logics are theoretically too complicated, but at the same time also not needed from a practical point of view. Thus we need not worry about Second and Higher Order Logics in this course COS333. From a philosophical point of view, FOL is fully compatible with philosophical 'school' of *Nominalism*, whereas the compatibility of Second Order Logic and Nominalism is philosophically disputed — for a brief overview see *http://en.wikipedia.org/wiki/Nominalism*

Further Notations and Conventions:
— $\Omega^{(n)} := \{f \in \Omega \mid \varrho(f) = n\}$
— $\mathrm{K} := \Omega^{(0)} := \{k \in \Omega \mid \varrho(k) = 0\}$ are the *constants*
— $\Pi^{(n)} := \{P \in \Pi \mid \varrho(P) = n\}$
— Nullary predicates $\Pi^{(0)}$ do **not** occur
— For elements of Ω we typically use symbols such as $f, f_i, g^{(n)}, h$
— For elements of Π we typically use symbols such as P, Q, R
— For elements of K we typically use symbols such as a, b, c, d, e

Definition: *TERMS — Σ-Terms, (a.k.a. Ω-Terms)*
Let $X \subseteq \{x_1, x_2, x_r, \ldots\}$ be a set of **Variables**, (only ONE Sort). Then the set $T_\Sigma(X)$ of *Σ-Terms over X*, or (Σ, X)-Terms, is defined inductively (recursively) as follows:
— $(X \cup K) \subseteq T_\Sigma(X)$ ..... *// Variables and Constants are Terms*
— **if** $f \in \Omega^{(n)}$ **and** $t_1, \ldots, t_n \in T_\Sigma(X)$, **then also** $f(t_1 \cdots t_n) \in T_\Sigma(X)$.

A special case of terms are *Ground* terms $T_\Sigma \subseteq T_\Sigma(X)$ which do *not* contain any variables, (we say that they are "variable-free"). Later we will also see that variables (in terms) can occur "bound" or "un-bound" ("free").[26]

EXAMPLE
A signature $\Sigma =< \Omega, \Pi, \varrho >$ is given by:
• $\Omega^{(0)} := \mathbb{N}$ (natural numbers as constants)
• $\Omega^{(2)} := \{+, -\}$
• $\Pi^{(2)} := \{<, >, =, \neq\}$
On this basis, the set of all Σ-Terms *contains* elements such as:
• $x_0, x_{12}, x_{27}, x_{33},$ ..... (variables)
• $7, 42, 117,$ ..... (numbers)
• $(3 + x_{17}), (((5 + 12) - (x_{18} + 6)) - x_{14}),$ ..... (arithmetic expressions)
On this basis, the set of all Well-Formed Formulae *contains* elements like:
• $8 < 15$ (semantically "false", but syntactically well-formed nevertheless)
• $(6 - x_3) = (12 + x_9)$
• $x_1 \neq x_2$

[26] Thus we must *not confuse* a "free variable" inside a term with a "variable-free" term.

- $(x_8 + 9) > (17 - (x_1 + 32))$
- $\exists x_0 : (x_0 < 1)$
- $\forall x_0 : (x_0 < (1 + x_0))$

Definition: Σ-*Formulae of FOL (Well-Formed Formulae WFF$_\Sigma$)*
The set $\mathbf{WFF_\Sigma(}X\mathbf{)}$ of **Well-Formed Σ-Formulae over** X, a.k.a. (Σ, X)-Formulae,[27] is defined inductively (recursively) as follows.

- *Atomic* Formulae (a.k.a. Prime Formulae):

  If $P \in \Pi^{(n)}, t_1, \ldots, t_n \in T_\Sigma(X)$, **then** $P(t_1, \ldots, t_n) \in \mathrm{WFF}_\Sigma(X)$.

- *Composite* Formulae (containing logical operators):

  If $F, G \in \mathrm{WFF}_\Sigma(X)$,

  then also $\neg F, (F \vee G), (F \wedge G), (F \to G), (F \leftrightarrow G) \in \mathrm{WFF}_\Sigma(X)$.

- *Particularised* (existentially quantified) Formulae:

  If $x \in X$ and $F \in \mathrm{WFF}_\Sigma(X)$, **then** also $(\exists x : F) \in \mathrm{WFF}_\Sigma(X)$,

- *Generalized* (all-quantified) Formulae:

  If $x \in X$ and $F \in \mathrm{WFF}_\Sigma(X)$, **then** also $(\forall x : F) \in \mathrm{WFF}_\Sigma(X)$,

In quantified formulae of type $(\mathcal{Q}x : F)$, where $\mathcal{Q} \in \{\exists, \forall\}$, any occurrence of x in F is *bound*, otherwise free. Note that one and the same variable can possibly occur bound as well as free in one and the same formula!

Abbreviation: "$F[x_1, \ldots, x_n]$" stands for "$F \in \mathrm{WFF}_\Sigma(\{x_1, \ldots, x_n\})$".

Example: *Bound and Free Variables*
$F := ((\forall x : x < x + 1) \wedge (\forall y : (y + x) = (x + y)))$
$\underline{\mathrm{free}}(F) = \{x\}$, $\underline{\mathrm{bound}}(F) = \{x, y\}$.

Definition: Σ-*SENTENCE*
A Σ-Formula **without any free variables** is called a Σ-Sentence.

[27] Singular: one formul**a**, plural: many formul**ae**.

4.1.2 Semantics

Definition: *OPERATIONS and RELATIONS on a Carrier Set A*
Let A be a non-empty set ("Carrier" or "Carrier Set"), and $n \in \mathbb{N}$.[28]

- $\mathbf{OPS}^{(n)}(A) := \{f \mid f : A^n \longrightarrow A\}$ is the set of Operations on A which are represented by the n-ary operation symbols (from Ω).
- $\mathbf{REL}^{(n)}(A) := \{r \mid r \subseteq A\}$ is the set of Relations in A,[29] represented by the n-ary predicate symbols (from Π).

Moreover,
$\mathbf{OPS}(A) := \cup_{n\in\mathbb{N}}(\mathbf{OPS}^{(n)}(A))$
$\mathbf{REL}(A) := \cup_{n\in\mathbb{N}}(\mathbf{REL}^{(n)}(A))$

Definition: *STRUCTURE*
Let $\Sigma =< \Omega, \Pi, \varrho >$ be a signature, and let A be a carrier set. Moreover, let there be mappings

- $\phi : \Omega \longrightarrow \mathrm{OPS}(A)$ with $\phi(\Omega^{(n)}) \subseteq \mathrm{OPS}^{(n)}(A)$
- $\psi : \Pi \longrightarrow \mathrm{REL}(A)$ with $\psi(\Pi^{(n)}) \subseteq \mathrm{REL}^{(n)}(A)$

Then $\underline{\mathcal{A} := < A, \phi, \psi >}$ is a Σ-**Structure**, (or simply: Structure).

Abbreviation: If some structure $\mathcal{A}$ has been provided without ambiguity, then we can also write "$f_{\mathcal{A}}$" or "f_A" instead of "$\phi(f)$", as well as "$P_{\mathcal{A}}$" or "P_A" instead of "$\psi(P)$".

Structures are very general mathematical objects, which are not yet specific for the purpose of LP. This is because nothing has been required about the actual contents of the carrier set A. For the purpose of LP, it will be helpful to work with more specialised structures, which are Algebras, and particularly the Algebra of Terms, because in declarative programming we are always dealing with terms in one or another form.[30]

[28] Do *not confuse* the Carrier Set "A" in FOL with an Atomic Proposition "A" in PL.
[29] Please see your own Handbook of Mathematics (Universal Algebra) for the definition of 'Relation'.
[30] Remember Functional Programming (Part A of COS333), where we also dealt with an Algebra of Terms.

Some Philosophical Remarks about the 'Links' between Functions, Relations, and Predicates: If we have some function $f(x) = y$, then all its input-output pairs (x, y) are members of the function's graph g, which thus constitutes a relation $g(x, y)$ with regard to f. This g is a set full of 'things', namely pairs. With a suitable predicate p we can *characterize* the "*property*" of "membership" in such a relation-set g, such that $p(x, y) \equiv$ TRUE for every pair $(x, y) \in g$. In the "old" classical logics such predicates p were regarded as *abstracta* —like, for example, "good-ness" or "happi-ness" or "lazi-ness", which do not take any input arguments— whereas in modern logics the predicates themselves are almost always regarded non-abstractly as Truth-producing Functions again, with some inputs and a Boolean output (True or False). Some Logic-Philosophers have therefore dubbed the Predicates in modern Logics "Predicate-Functions", in contrast to the "pure" classical abstract predicates. Thus, if "good-ness" would be a classical abstract predicate p, then "is-good(x)" would be its corresponding modern predicate or predicate-function $p(x)$. Thus we can now make a boolean function $f'(x, y) =$ True, **if** $f(x) = y$, which is the case if the pair $(x, y) \in g$. Thus, from a philosophical point of view, *Predicates* have a quite funny 'ontological status' in which they could be regarded **both** as Truth-producing *Functions* **as well as** as Representatives of *Relations*, which are *not* Functions, because functions have a clear sense of direction from input to output, whereas relations do not need to have such a direction. The definition of above —*OPERATIONS and RELATIONS*— regards the Predicates as Representatives of Relations, rather than as Predicate-Functions, though in the subsequent *MODEL* definition —see below— the characteristics of Predicates as Truth-producing Functions will also appear. Beheld from an even higher level of abstraction, one could even think of a Functional which produces a predicate-function from a classical abstract predicate and a suitable membership domain, e.g.: $\mathcal{F}(p, x, y) = p(x, y)$, to clarify the relationship between the old and the modern concept of 'predicate'. Even the set-theoretical "$\in$" operator itself, like in $(x, y) \in g$, on the basis of which the Truth of $p(x, y)$ was defined, could be regarded as a Predicate in its own right, ditto the "$=$" as in $f(x) = y$, etc. Latest at this point we should be able to see that the "borderline" between syntax and semantics is not always as sharp and accurate as we might want

to have it, because in order to *speak* about the Semantics of the Syntax of Something, we need yet another syntax again for the *language with which* we can possibly speak *about* that Semantics, etc.

Definition: *HERBRAND STRUCTURE*
The set T_Σ of (variable-free) Σ **Ground Terms** is called *Herbrand Universe.*[31]
It determines the *Herbrand Algebra* $\tilde{\mathcal{H}} := < T_\Sigma, \phi_H >$ with
— $\phi_H(a) := a$ .....(constant)
— $\phi_H(f)(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$ ..... (constructor)
whereby the terms themselves are regarded as their own semantics.[32]
A Σ structure $\mathcal{H} := \mathcal{T}_\Sigma := < T_\Sigma, \phi_H, \psi >$ is then a *Herbrand Structure* (abbr.: "H Structure").

Note that Σ determines one and only one H Algebra, however —because of ψ— an un-determined number of H Structures. In this specific Herbrand "world" we want to continue with our theory of LP. Next, however, for the notions of "validity" and "satisfiability" of formulae, we need to see how to organise the interpretation of free variables in FOL.

Definition: (Σ, X)*-INTERPRETATION (Term Semantics)*
Let $\mathcal{A}$ be a Σ-Structure and $\beta : X \longrightarrow A$ a basic interpretation of variables. $I :=< \mathcal{A}, \beta >$ is then a (Σ, X)-*Interpretation.* This determines the *Term Semantics*, $\beta_\mathcal{A}$: $T_\Sigma(X) \longrightarrow A$, as follows:
— $\beta_\mathcal{A}(x) := \beta(x)$ .....(variables)
— $\beta_\mathcal{A}(a) := \phi(a)$ .....(constants)
— $\beta_\mathcal{A}(f(t_1, \ldots, t_n)) := f_\mathcal{A}(\beta_\mathcal{A}(t_1), \ldots, \beta_\mathcal{A}(t_n))$
(homomorphic-inductive continuation: "substitute and evaluate")

Notation for Ground Terms: In case that $A \subseteq T_\Sigma$, $\beta_\mathcal{A}(t)$ is independent of β (because there are no variables $x \in X$). In this case we can write "$t_\mathcal{A}$" instead of "$\beta_\mathcal{A}(t)$".

[31] Mathematician and Logician **Jacques Herbrand**, 1908-1931, obtained his Doctoral Degree at the age of 22, and died one year later at the age of 23 during a mountain hike in the Alps: see *http://en.wikipedia.org/wiki/Herbrand*

[32] This "trick" is not possible with Predicates, because the 'nature' of Predicates is indetermined.

Definition: *MODEL (Consistency, Satisfiability) in FOL*
Given $F \in \mathrm{WFF}_\Sigma(X)$ and a (Σ, X)-Interpretation $I =< \mathcal{A}, \beta >$. We define, inductively via the term-form of F, whether $< \mathcal{A}, \beta >$ is a **model of** F (a.k.a. "F is consistent in $< \mathcal{A}, \beta >$", or "$< \mathcal{A}, \beta >$ satisfies F"), denoted as $< \mathcal{A}, \beta > \models F$, for the following cases:

- $< \mathcal{A}, \beta > \models P(t_1, \ldots, t_n)$ **if and only if** $P_{\mathcal{A}}(\beta_{\mathcal{A}}(t_1), \ldots, \beta_{\mathcal{A}}(t_n))$ is true
- $< \mathcal{A}, \beta > \models (G \wedge G')$ **if and only if** $< \mathcal{A}, \beta > \models G$ **and** $< \mathcal{A}, \beta > \models G'$
- $< \mathcal{A}, \beta > \models (G \vee G')$ **if and only if** $< \mathcal{A}, \beta > \models G$ **or** $< \mathcal{A}, \beta > \models G'$
- $< \mathcal{A}, \beta > \models (\neg G)$ **if NOT** $< \mathcal{A}, \beta > \models G$, i.e.: $< \mathcal{A}, \beta > \not\models G$
- $< \mathcal{A}, \beta > \models (G \rightarrow G')$: This case follows from the cases of above, taking into account that $(G \rightarrow G') \equiv (\neg G \vee G')$
- $< \mathcal{A}, \beta > \models (G \leftrightarrow G')$: This case follows from the cases of above, taking into account that $(G \leftrightarrow G') \equiv ((G \rightarrow G') \wedge (G' \rightarrow G))$
- $< \mathcal{A}, \beta > \models (\exists x : G)$ **if and only if** there exists at least one $a \in A$ and if exists at least one suitable $\tilde{\beta}$ with $\tilde{\beta}(x) = a$ such that $< \mathcal{A}, \tilde{\beta} > \models G$
- $< \mathcal{A}, \beta > \models (\forall x : G)$ **if and only if** for all $a \in A$ for which there is any suitable $\tilde{\beta}$ with $\tilde{\beta}(x) = a$: $< \mathcal{A}, \tilde{\beta} > \models G$

In the special case that F is a Σ-Sentence (variable-free), β is irrelevant. In this case the Model-property is independent of β, and we write "$\mathcal{A} \models F$" instead of "$< \mathcal{A}, \beta > \models F$".

Definition: *THEORY*

- $\mathbf{TH}(\mathcal{A}, \beta) := \{F \in \mathrm{WFF}_\Sigma(X) \mid < \mathcal{A}, \beta > \models F\}$, the set of all well-formed formulae for which $< \mathcal{A}, \beta >$ is a model, is called the **Theory of** $< \mathcal{A}, \beta >$.
- $\mathbf{TH}(\mathcal{A}) := \{F \in \mathrm{WFF}_\Sigma(X) \mid < \mathcal{A}, \beta > \models F$ for all $\beta\}$, the set of all well-formed formulae for which $< \mathcal{A}, \beta >$ is a model *independent of* β, is called the **(complete) Theory of** $\mathcal{A}$.[33]

[33]Such type of theories are *complete* in the sense that in them we can either find a (variable-free) sentence S, or its negation $\neg S$, which is —from a computational point of view— decideable.

LEMMA! Generalisation Given $F \in \mathrm{WFF}_\Sigma(X)$ with $\underline{\mathrm{free}}(F) = \{x_1, \ldots, x_n\}$. Also given the "Generalisation of F", namely $\tilde{F} := (\forall x_1 \cdots \forall x_n : F)$. Also given a (semantic) structure $\mathcal{A}$. Then $\tilde{F} \in \mathrm{TH}(\mathcal{A})$ **if and only if** $F \in \mathrm{TH}(\mathcal{A})$.

Definition: *SATISFIABILITY and (General) VALIDITY w.r.t. Theories*
A formula $F \in \mathrm{WFF}_\Sigma(X)$ is called

- *(generally) valid* **if and only if FOR ALL** structures $\mathcal{A}$: $F \in \mathrm{TH}(\mathcal{A})$.
- *satisfiable* **if and only if** there exists some Interpretation $I =< \mathcal{A}, \beta >$ such that $F \in \mathrm{TH}(\mathcal{A}, \beta)$.

Also note that F is valid if and only if $\neg F$ is an Antinomy (un-satisfiable).

Definition: *Semantic Notion of CONSEQUENCE in FOL*
Given a formula $F \in \mathrm{WFF}_\Sigma(X)$, and a set of formulae $\Phi \subseteq \mathrm{WFF}_\Sigma(X)$.[34] We say: "$F$ **is a consequence of** Φ", denoted as: $\Phi \models F$,[35] **if and only if FOR ALL** structures $\mathcal{A}$: if $\Phi \subseteq \mathrm{TH}(\mathcal{A})$, then also $F \in \mathrm{TH}(\mathcal{A})$.

Definition: *CONSEQUENCE SET*
In FOL, the set $\mathbf{CONS(\Phi)} := \{F \in \mathrm{WFF}_\Sigma(X) : \Phi \models F\}$ is called the **consequence set of** Φ.

Note that CONS({}) contains all (and only the) Tautologies, because the Tautologies are always true regardless of any given premises. Tautologies follow, so-to-say, "out of nothing".

LEMMA! Closedness of Theories For all $\mathcal{A}$: $\mathrm{CONS}(\mathrm{TH}(\mathcal{A})) = \mathrm{TH}(\mathcal{A})$.

In other words, we cannot "add" information to a theory by deducing logical consequences from it. All its logical consequences are already "implicitly hidden" in the theory itself. This insight will now lead us to a (more general) re-definition of the notion of "Theory", as follows.

[34] This Φ here in LP has *nothing* to do with the Φ from Part A (Functional Programming) of this course.
[35] The symbol "$\models$" is somewhat "overloaded" and can be used in slightly different contexts, as you have seen in the case of $< \mathcal{A}, \beta > \models F$.

Re-Definition: *THEORY*
A set of well-formed formulae $\Phi \subseteq \mathrm{WFF}_\Sigma(X)$ is a *theory* **if and only if** $\mathbf{CONS(\Phi) = \Phi}$.[36]

This very abstract definition of the notion of "theory" also entails the special case the case of inconsistent (self-contradictory or contradictory) theories which are defined as follows.

Definition: *INCONSISTENCY (of Theories)*
A theory Φ is **inconsistent** (or self-contradictory, or contradictory) if it contains at least one formula of type $(F \wedge \neg F)$.

LEMMA! From every inconsistent theory, *any* statement can be derived, i.e.: for any inconsistent theory $\Phi_?$, $\mathrm{CONS}(\Phi_?) = \mathrm{WFF}_\Sigma(X)$.[37]

Obviously we want to *avoid* such useless theories.

4.1.3 History-Excursion: Hilbert's Dream

One of the goals of German mathematician *David Hilbert*, the most prominent supporter of the Philosophy of Formalism in Meta-Mathematics, was to *axiomatise* classes of (mathematical) theories (such as: theory of sets, theory of geometry, etc.) logically. Together with his Hilbert-Calculus (for the automatic derivation of consequences) this would lead to decideable mathematical theories which could be automatically derived by some suitable computing machinery. As we have seen above, any *complete* theory Φ (containing either F or its negation $\neg F$) is computationally decideable, such that $\mathrm{CONS}(\Phi)$ would then be recursively enumerable. If this dream would come true, then a computer could —for example— compute all theorems of Set Theory, all the theorems of Geometry, etc., such that a human mathematician would no longer be needed for such tasks. It was the Austrian mathematician and logician *Kurt Gödel* who terminated Hilbert's dream. Gödel demonstrated that there exist mathematical theories which *cannot* be axiomatised in the form

[36] Fixpoint!
[37] This is because of "(false $\Rightarrow$ anything) = true" in the classical definition of Implication ("$\Rightarrow$").

of a suitable set of formulae Φ. For example: whereas *Mojżesz Presburger*'s Arithmetics based on $(\mathbb{N},+)$ is still decideable, already its extension to our usual Arithmetics based on $(\mathbb{N},+,\bullet)$ is no longer decideable. Consequently, no Hilbert-Calculus (and in fact no automated Calculus whatsoever) could be applied in order to mechanically enumerate all the true Theorems of Arithmetics — thank goodness for the Arithmeticians, who would otherwise have gone home unemployed! For our practical purpose of LP, however, we need not worry about this episode of history-of-science any further.

4.2 NORMAL FORMS

In the previous section on Propositional Logic (PL) we had already seen how Normal Forms have lead us to a useful Resolution Calculus for practical application in LP. Now we would like to (somehow) "re-use" the basic techniques of such a Resolution Calculus in the context of FOL, too. For this purpose, however, two "tricks" will be needed:

- On the one hand we need to "expand" the Resolution Calculus in order to accomodate the particular features of FOL (such as Predicates P, Q, R, and Variables x, y, z) which PL does not have.

- On the other hand, we must *restrict and simplify* our FOL-formulae themselves (similar to the restrictions on our PL-formulae previously), too, such that the Resolution Calculus can work *effectively and efficiently* on them.

It is presumed that students are already familiar, from some earlier *Introduction to Logics* course (or similar) with the following "chain" of Normalisations: $\mathrm{WFF}_\Sigma(X)$ formula $\succ$ *Prenex* form $\succ$ **Skolem** form $\succ$ *Clause* form.[38] These normal forms are briefly summarised as follows:

Prenex Form: All the $\forall$ and/or $\exists$ Quantors are 'drawn' to the 'front side' of a formula, such that the 'inner body' of the formula is quantor-free.

[38]HOMEWORK: *Recapitulate*, in your own time, those normal forms from your own Handbook of Logic (or any other suitable sources)! For the Norwegian logician **Thoralf Skolem** (1887-1963) see *http://en.wikipedia.org/wiki/Skolem*

Skolem Form: All free variables and all $\exists$-Quantors are removed. (The resulting formula is *satisfiability-equivalent* to its source formula, though not strictly equivalent.)

Clause Form: The remaining $\forall$-Quantors are lexically 'removed' (omitted), though it must *not* be forgotten that 'in reality' they are (implicitly) still there, and the quantor-free 'body' of the formula is brought into CNF.

Difference between Equivalence and Satisfiability-Equivalence: Given two formulae $F, F' \in \mathrm{WFF}_\Sigma(X)$, we remember that they are (strictly) *equivalent* ($\equiv$) under the following condition: **For every** Model M, which has the property $M \models F$, we **also** find $M \models F'$, **and vice versa**. The notion of Satisfiability-Equivalence ($\cong$), however, is *weaker* (less strict). It stipulates that $F \cong F'$ under the following condition:

- **If NO** Model exists such that $M \models F$, **then also NO** Model exists such that $M \models F'$, **and vice versa**.

Re-phrased positively the condition states: **If** there exists some Model M such that $M \models F$, **then** there exists also (another) Model M' with $M' \models F'$, **and vice versa** — under these circumstances $F \cong F'$ (satisfiability-equivalent).

If two formulae are (strictly) equivalent, then they are also satisfiability-equivalent, but *not* vice versa: From $(F \equiv F')$ logically follows $(F \cong F')$, but from $(F \cong F')$ does *not* necessarily follow $(F \equiv F')$.

This (weaker) notion of satisfiability-equivalence allows us to utilise some suitable *standard-models*, which are "easier to compute" in the practical world of Logic Programming. We will soon see that our previously introduced *Herbrand* 'world' will fit this purpose.

THEOREM! For *every* formula $F \in \mathrm{WFF}_\Sigma(X)$ it is effectively possible to construct a satisfiability-equivalent formula F' in Skolem Normal Form (SNF), and with an un-quantified 'body' in Clause Form (CNF), such that $F \cong F'$

A Satisfiability-TEST for a given formula F, done with ALL mathematically possible structural Interpretations $< \mathcal{A}, \beta >$, would not be feasible! Thank goodness, our Herbrand Interpretation can serve as a *Representative* of all other possible Interpretations, such that we *only* need to check if a given formula is Herbrand-satisfiable in order to know that it is satisfiable. This "trick" will simplify our problem considerably, on our way towards an effective and efficient Resolution Calculus for (a restricted variant of) FOL.

Definition: *HERBRAND MODEL with canonical Predicate Interpretation* Given $\Sigma =< \Omega, \Pi, \varrho >$ whereby $\Omega^{(0)} \neq \{\}$, i.e.: there is at least one constant a, and $\mathcal{A} = < A, \phi, \psi >$ as a Σ-Structure. Then $\mathcal{A}$ determines a *Herbrand Structure* $\mathcal{H}_\mathcal{A} := < T_\Sigma, \phi_H, \psi^H_\mathcal{A} >$ under the condition that for all predicates $P \in \Pi^{(n)}$ and for all variable-free terms $t_1, \ldots, t_n \in T_\Sigma$: $\psi^H_\mathcal{A}(P)(t_1, \ldots, t_n) := \psi(P)(t_{1\mathcal{A}}, \ldots, t_{n\mathcal{A}})$ as a standard-convention for Predicate Interpretation.[39]

THEOREM! For every sentence F in Skolem Normal Form: **If** $F \in \mathrm{TH}(\mathcal{A})$, i.e.: $\mathcal{A} \models F$, **then also** $F \in \mathrm{TH}(\mathcal{H}_\mathcal{A})$.

In other words, the theorem tells us that we are indeed allowed to use the standardised Herbrand model as a Representative of any other possible model. Our restriction to terms is *not* a handicap: on the contrary it will considerably simplify our satisfiability tests in FOL for the sake of LP.

Corollary! For any sentence F in Skolem Normal Form: **If** F is satisfiable (i.e.: has any model) at all, **then** F **also** possesses a Herbrand model, **and vice versa**.

The computational advantage of Herbrand models are: They are recursively *enumerable* (i.e. countable), whereas other models might possibly be *over*-countably large (and thus not feasible for computation in LP). This simplifies the computational task of seeking and finding *solutions* in Herbrand

[39] Because predicates, unlike terms, are not strongly typed, they cannot simply represent themselves semantically in the manner in which terms can do that. Therefore we are forced to choose a suitable semantics for predicates in the Herbrand world.

worlds by means of some Resolution Calculus (which would be utterly hopeless in unrestricted FOL).

Beheld from a perspective of Formal Language Theory, the expressiveness of the 'language' of Herbrand models stands 'between' the expressiveness of the 'language' of FOL and the expressiveness of the 'language' of PL. What Herbrand worlds and PL have in common is the absence of free variables x, y, z, such that we need not 'guess' what such free variables may possibly stand for. The difference between PL and Herbrand worlds is that a model of a PL-formula is always *finite*, whereas the Herbrand-model of a FOL-formula may possibly be *infinite* (though countable-infinite, not over-countable).

Definition: *HERBRAND EXPANSION of a Sentence*
Let $F = (\forall x_1 \cdots \forall x_n : F')$ be a sentence (without free variables) in Skolem Normal Form. Let "$[x/t]$" denote *substitution.*[40] The set of variable-free formulae $E(F) := \{F'_{[x_1/t_1 \ldots x_n/t_n]} \mid t_i \in T_\Sigma\}$ is then called the **Herbrand Expansion** of F.

Note that all the t_i in the definition of above are Ground Terms, such that no 'fresh' variables can be 'smuggled into' the Herbrand Expansion by means of the variable-term-substitutions. Thus, because no formula in $E(F)$ possesses any variable any more, the **PL** Resolution Calculus (from Sub-Section 3.3) is now applicable (again), whereby a closed FOL-Sentence of type $P_i(t_1, \ldots, t_n)$ corresponds to an atomic PL-Sentence of type A_i (which can be either true or false). With help of the already mentioned "Finity Theorem" —if the infinite set $E(F)$ is inconsistent, then there must exist some subset $\mathcal{E} \subset E(F)$ which is *finite* in size ($|\mathcal{E}| < \infty$) and which is *also* inconsistent— the existence of $\emptyset \in E(F)$ becomes at least partially decideable (semi-decideable).[41]

[40] Remember Part A (Functional Programming) of this course, where we had substitution in the context of Lambda expressions.

[41] Remember that the Decision Problem in PL is decideable (though very expensive, namely NP-Complete) because the number of possible $\{\text{True}, \text{False}\}$-Mappings for the Atoms A_i in PL-Formulae is only finite. Now, however, the Herbrand Expansion of a FOL-Formula is possibly infinite in size, such we have no guarantee of actually finding the finite subset $\mathcal{A}$ which contains the $\emptyset$ — if such exists at all in $E(F)$. Therefore, we now have only semi-decideability in FOL, not full decideability.

THEOREM! "Reduction" from FOL to PL. For every sentence F in Skolem Normal Form the following assertion holds: F is satisfiable **if and only if** its Herbrand Expansion $E(F)$ is satisfiable.

4.2.1 Gilmore's Algorithm

With the theorem of above, in combination with the "Finity Theorem" of PL, *Gilmore* constructed a Semi-Decision Algorithm for the Non-Satisfiability of any given FOL-Formula:

1. Transform a given $F \in \mathrm{WFF}_{\Sigma}(X)$ into a satisfiability-equivalent sentence F' in Skolem Normal Form.

2. Let $\{F_1, F_2, F_3, \ldots\}$ be an *explicit enumeration* of the Herbrand Expansion $E(F')$ of F'.[42]

3. FOR $(i = 1,\ i < \infty,\ i{+}{+})$ DO:

— *Check with PL-Resolution* whether $(F_1 \wedge \cdots \wedge F_i)$ is inconsistent

— IF ("$\emptyset$" detected) THEN HALT

Note, again, that Halting is *not* guaranteed, because of the semi-decideable nature of the problem. Also note that Gilmore's Algorithm is computationally very "expensive", because its "working formula" $(F_1 \wedge \cdots \wedge F_i)$ is growing in every i-Loop, and the *entire* Resolution Algorithm (from Sub-Section 3.3) is called again and again in *every* i-Loop of Gilmore's Algorithm!

THEOREM! Gilmore's Algorithm is correct. (This is because: F cannot be satisfied *if and only if* its Herbrand Expansion $E(F)$ cannot be satisfied.)

Corollary! The Set of all un-satisfiable (Σ, X)-Formulae in FOL is *recursively enumerable*.[43]

[42] Such an enumeration can be effectively produced, for example by systematically taking the lexical term-sizes $|t_i|$ into account: first the smallest possible terms of size 1 (such as constants a, b, c), then the next larger terms, etc.

[43] For the notion 'recursively enumerable', please see your own Handbook of Theoretical Computer Science, particularly: Theory of Computability, or Theory of Decideability. The issue is "linked" with the Halting Problem for Turing Machines.

Example: $F = (\forall x : \ (P(x) \wedge \neg P(f(x))))$ with $a, f \in \Omega$

Herbrand Expansion:
$E(F) = \{(P(a) \wedge \neg P(f(a))), (P(f(a)) \wedge \neg P(f(f(a)))), \ldots\}$

In Clause-Set-Form:
$E(F) = \{\{P(a)\}, \{\neg P(f(a))\}, \{P(f(a))\}, \{\neg P(f(f(a)))\}, \ldots\}$

Gilmore's Algorithm:
— try PL-Resolution on $\{P(a)\}$
— try PL-Resolution on $\{P(a)\}, \{\neg P(f(a))\}$
— try PL-Resolution on $\{P(a)\}, \underbrace{\{\neg P(f(a))\}, \{P(f(a))\}}_{\text{Antinomy "}\emptyset\text{" detected!}}$
— HALT

To "see intuitively" that this formally-mechanically derived result is indeed correct, let us give a *different* interpretation (not the Herbrand interpretation) to the formula F of above. In a context of number-arithmetics, let us define:
$P(x) :=$ "The number x is even"
$f(x) := x^2$
$x := 4$
Thus: $f(4) = 4^2 = 16$
Thus: ($P(4) =$ "4 is even" $\wedge$ **NOT** $P(16) =$ "16 is even")
Result: **FALSE** ← as found by Gilmore's algorithm in the Herbrand world

To "see intuitively" that it was really the **PL**-Resolution which was applied in this example, we can simply *re-name* the closed (variable-free) $P(\cdot)$-Statements (from FOL) of above into "atomic" PL-Statements as follows. $P(a) := \mathsf{A}_1$, $P(f(a)) := \mathsf{A}_2$, $P(f(f(a))) := \mathsf{A}_3$, etc.

4.3 RESOLUTION

Gilmore's Algorithm resorted to PL in order to test the satisfiability of FOL formulae. Next we want to study how we can implement 'proper' FOL res-

olution (without resorting to PL). Again, however, this will entail that we have to get —somehow— rid of the free variables in the FOL formulae with which we are dealing, because we cannot know if a formula will evaluate to 'True' or to 'False' if we do not know the Meaning (interpretation) of its free variables x, y, z, etc. The problem here is to detect *suitable* 'instances' for variable-substitution, on the basis of which we may hope to find "$\emptyset$" amongst variable-free clauses *as quickly as possible — faster* than with the tedious Algorithm of Gilmore, which systematically generated the entire Herbrand world in order to find a "$\emptyset$"-Solution.

Motivating Example: Given $\{P(x)\}, \ldots \{\neg Q(g(x))\}, \ldots \{\neg P(f(y))\}, \ldots$ We see that we could resolve $\{P(\_\_)\}$ with $\{\neg P(\_\_)\}$ if their arguments $(\_\_)$ were identical. Here we can indeed yield such identity, namely by the following substitution: $[x/f(y)]$. Then we can indeed resolve $\{P(f(y))\}$ with $\{\neg P(f(y))\}$. In this example, however, the substitution $[x/f(y)]$ also has a *side effect* on the Q-Clause, because the Q-Clause, too, contains an occurrence of the variable x. Therefore, $\{\neg Q(g(f(y)))\}$ would remain after the resolution of $P(\_\_)$ with $\neg P(\_\_)$. We say that the substitution $[x/f(y)]$ **unifies** $P(x)$ and $P(f(y))$ with each other. This leads is now to the topic of Unification, which will then become the key "ingredient" of a genuine FOL Resolution technique: If we are only able to "see" suitable unifications, then we need *not* "stubbornly" enumerate the entire Herbrand world any more.

4.3.1 Unification

Definition: *UNIFICATION*
Let $\mathrm{C} = \{L_1, \ldots, L_n\}$ a (Σ, X)-Clause (in FOL) i.e.: each $L_i \in \mathrm{C}$ is a (Σ, X)-Literal.[44] For $i = 1 \cdots k$, $x_i \in X$ (variables) and $t_i \in T_\Sigma(X)$ (terms). Then the substitution $s := [x_1/t_1, \ldots, x_k/t_k]$ is a *Unifier* of C **if and only if** $L_{1s} = \cdots = L_{ks}$, such that the resulting unified clause has the cardinality (size) $|\mathrm{C}_s| = 1$.[45] Moreover, s is called the *Most General Unifier* (**MGU**) of C, if for any other unifier s' there exists yet another substitution s'' such that the functional composition of substitutions yields $s' = (s \circ s'')$.

[44] FOL-Literals are Predicates or Negated Predicates, like $P(x)$, $\neg P(g(y, z))$, etc.
[45] Do not confuse the set-size 1 of above (i.e.: one element) with the logical Truth-Value 1.

Note that the MGU of C is, variable renamings notwithstanding, *uniquely* (non-ambiguously) determined. In other words: Every C can have *maximally one* MGU (if any at all). This is guaranteed by Robinson's following theorem:

Unification THEOREM! [Robinson] Every unifiable clause has its MGU.

Unification ALGORITHM Input: FOL-Clause C $= \{L_1, \ldots, L_n\}$.
Substitution $s := []$. *// initialisation: the empty substitution*
WHILE ($|C_s| > 1$) DO:

- Check all the Predicate-characters of the Literals $L_1, \ldots, L_n$:
  — IF different characters occur *// for example:* P, Q
  — THEN RETURN("C cannot be unified"). *// Exit!*
- Select any two Literals $L_i, L_j \in$ C with $i \neq j$
- Let x be a variable at position p inside L_i
- Let t be a term at the same position p inside L_j
- Check the internal composition of t:
  — IF x re-occurs in t *// Infinite Loop of Substitutions is looming!*
  — THEN RETURN("C cannot be unified in Finite Form"). *// Exit!*
- Apply $[x/t]$ in C wherever applicable
- $s := (s \circ [x/t])$ *// functional composition of substitutions*

// End of WHILE loop is reached: only ONE Literal remains in Clause
RETURN(s **"is the MGU of C")**. *// Halt*

Lemma! The unification algorithm of above is correct [Robinson].[46]

[46] For **John Alan Robinson** see *http://en.wikipedia.org/wiki/Robinson's_unification_algorithm*, also *http://en.wikipedia.org/wiki/Unification_(computer_science)*, as well as *http://en.wikipedia.org/wiki/Robinson's_Resolution_Algorithm*, as a starting point for your own further studies of these topics.

4.3.2 FOL Resolution

Definition: *FOL RESOLUTION*
Let C, C', R be FOL-Clauses over (Σ, X). R is called the **Resolvent** (more precisely: FOL-Resolvent) of C and C' if all the following three conditions are fulfilled:

1. There exist 'bound' re-namings (renaming substitutions) s, s' of variables, such that the variable-renamed Clauses $s(\mathrm{C})$ and $s'(\mathrm{C'})$ do not have any 'shared' variable names in common any longer.[47]

2. There exist literals $L_1, \ldots, L_m \in s(\mathrm{C})$, with $m \geq 1$, and literals $L'_1, \ldots, L'_n \in s'(\mathrm{C'})$, with $n \geq 1$, such that the clause C" $:= \{L_1, \ldots, L_m, L'_1, \ldots, L'_n\}$ is unifiable with MGU s_{mgu}.

3. $\mathrm{R} := s_{\mathrm{mgu}}(((s(\mathrm{C}))\backslash\{L_1, \ldots, L_m\}) \cup ((s'(\mathrm{C}'))\backslash\{L'_1, \ldots, L'_n\}))$.[48]

Again we can depict this situation graphically:

C C'
\ /
R

Example: $\underline{\mathrm{C} = \{P(f(x)), \neg Q(z), P(z)\}}$ and $\underline{\mathrm{C'} = \{\neg P(x), R(g(x), a)\}}$.
We observe that variable x occurs both in C and in C'.
This would lead to flawed side-effects of substitution.
Thus we first rename $[x/u]$ consistently in C', yielding:
$\mathrm{C} = \{P(f(x)), \neg Q(z), P(z)\}$ and $\mathrm{C'} = \{\neg P(u), R(g(u), a)\}$.
Now C and C' do not contain any shared variable names any more.
We "see immediatly" that $[z/f(x), u/f(x)]$ is a suitable unifier,
yielding: $\underline{\mathrm{R} = \{\neg Q(f(x)), R(g(u), a)\}}$.[49]

[47] Because a variable-term-substitution $[x/t]$ with the MGU has *side effects on all other literals in which the variable name x occurs*, it is important that the two clauses C and C' do not have any 'shared' variable name in common: Otherwise the MGU-application would introduce flaws!

[48] Obviously, some of the literals must be "positive" and some "negative" such that they can "cancel each other out" in a step of resolution, as we had seen it previously in the case of PL.

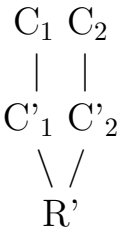
[49] Do *not confuse* the *Resolvent* R with the *Predicate* $R(.,.)$ in the example of above.

4.3.3 Relationship between FOL Resolution and PL Resolution

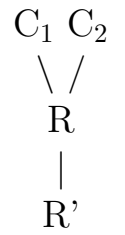
There is, indeed, a *systematic* relationship between FOL Resolution and PL Resolution, which is due to the fact that the 'languages' ℓ, which can be 'spoken' by these two logics, are so closely related with each other in a subset relation: $\ell_{\text{PL}} \subset \ell_{\text{FOL}}$. Therefore, PL Resolutions of *Ground Instances* of FOL clauses (in the Herbrand world) can be "lifted up" to proper FOL Resolutions. This is the statement made by the following famous "Lifting Lemma".

Lifting-LEMMA! Let C_1 and C_2 be FOL Clauses with Ground Instances (in the Herbrand world) C'_1 and C'_2. Moreover, let R' be the Resolvent of C'_1 and C'_2 in Propositional Logic (PL). Then there also exists a FOL Resolvent R for C_1 and C_2, such that R' is a Ground Instance of R.

Situation with Resolution in PL:



Situation with Resolution in FOL:



As we see intuitively from the pictures of above, the Lifting Lemma describes a kind of "*confluence*" situation,[50] whereby it does not matter whether we first to Resolution and then find a Ground Instance, or whether we first find Ground Instances and then do Resolution.

Resolution THEOREM of FOL! Let F be a FOL-Sentence in Skolem Normal Form, i.e. $F = (\forall x_1 \cdots \forall x_n : G)$, whereby G is in Conjunctive Normal Form,

[50] Remember Part A (Functional Programming) of this course COS333.

and let C_G be the Clause Set representation of G. Moreover, let RES be the Resolvent-Expansion as previously defined.[51] Then *F is un-satisfiable* **if and only if** the empty clause $\emptyset \in \mathrm{RES}^*(C_G)$.

Now we are well prepared to study the applications of all this in LP.

5 LOGIC PROGRAMS and LP-Computation

With the concepts we have learned so far, we know now the 'mathematical semantics' of a given set of formulae, but we do not know yet *how* to compute. For this purpose we will also need Evaluation Strategies which will guide us in the proper *choice* of Unifiers and Resolvents, we need to define the *operational* (procedural) semantics in addition to the mathematical semantics, etc.[52] Before we go into these details, however, we will briefly *recapitulate* the key concepts of a Logic Program.[53]

5.1 SYNTAX

Definition: *LOGIC PROGRAM*
A finite set of definite **(Σ, X)-Horn** Clauses is called a *Logic Program* over (Σ, X).[54] *Definite* Horn Clauses contain *exactly one* positive literal. Some program clause C is called a *Fact* if and only if $C = \{L\}$ — otherwise $C = \{L, \neg L_1, \ldots, \neg L_k\}$ (with $k > 0$) is called a *Rule*, whereby $L, L_1, \ldots, L_n$ are all FOL-literals. Implicitly all variables ($x \in X$) occuring in the literals are regarded as $\forall$-quantified.[55]

Remark on Facts: Facts need not necessarily be variable-free. For example the fact clause $\{\geq (0, x)\}$ can represent —in interpretation— at least the

[51] See Page 14 in these Lecture Notes.
[52] All of the following will thus be quite similar to Part A of this course, where we had studied evaluation strategies, procedural semantics, etc., in the context of Functional Programming.
[53] Also remember Pages 9, 11, 15, 16 of these Lecture Notes.
[54] Horn clauses can be build in FOL, too, not only in PL. Such as in PL, also FOL Horn Clauses possess maximally one positive Literal: see again Page 11 of these Lecture Notes for comparison.
[55] Remember: Skolem Normal Form.

entire set of natural numbers $\mathbb{N}_0$.[56]

Example: *Logic Program*

```
my_sister(Me, Si) ← female(Si),
                    mother_of(Me, Mo),
                    mother_of(Si, Mo),
                    female(Mo),
                    father_of(Me, Fa),
                    father_of(Si, Fa),
                    male(Fa),
                    couple(Mo, Fa).
```

Definition: *QUERY*
Let the formula F be a Logic Program over (Σ, X). A Query to F has generally the form $G_{y_1,\ldots,y_n}$, whereby G must be a well formed formula (WFF), and the variables $\{y_1, \ldots, y_n\} \supseteq \underline{\textbf{free}}(G)$ must be free. Implicitly, these free variables are regarded as $\exists$-quantified.

Example: *is it possible that my father is my sister?*

```
?- my_sister(Me,X), father_of(Me,X).
```
[57]

Now, to be 'useful', a query must be posed in such a manner that it will be possible to calculate its *semantic consequence* ($\models$) by means of *syntactic inference* (Resolution Calculus). This can be done in one of the following two manners:

1. *Do there exist Ground Terms* $t_1, \ldots, t_n \in T_\Sigma$
such that $G_{[y_1/t_1,\ldots,y_n/t_n]}$ is a semantic consequence of F*?*

Possible answers: yes / no.

2. If such Ground Terms exist, *which ones are they?*

[56] This goes well together with a common opinion according to which logic programs do only describe sets and relationships amongst them.

[57] Remember that all this is only syntax! The program cannot 'know' by itself the laws and constraints of biology and genetics as we can find them *outside* the Herbrand world of terms!

Possible answers: $\{t_1, \ldots, t_n, t'_1, \ldots, t'_n, t''_1, \ldots, t''_n, \ldots\}$
whereby this answer-set can *possibly* be *infinite* in size.

Ad 1. From a logical point of view, such a query is equivalent to the question: Is the formula $(F \wedge \neg(\exists y_1, \ldots, y_n : G))$ **un**-satisfiable? This is further equivalent to the question: Is the formula $(F \wedge (\forall y_1, \ldots, y_n : \neg G))$ **un**-satisfiable?[58] If G is represented as a Conjunction of Atomic formulae $G_1, \ldots, G_k$ (with $k > 0$), then $\neg G$ can be represented as a negative Clause $\{\neg G_1, \ldots, \neg G_k\}$ which can be 'tackled' with SLD Resolution (in combination with the clause set representation of F).

Ad 2. In the process of FOL Resolution (towards the empty clause $\emptyset$), unifying substitutions (of type $s \circ [x/t]$) are generated. As a side-effect, the "trace" of those substitutions will lead to the answers regarding suitable values for the free variables of the query clause. If such an answer still contains some un-substituted variable x, then such x must be regarded (implicitly) as $\forall$-quantified, representing 'anything'.

EXAMPLE: *Logic Program for Addition (Presburger-Arithmetics)*
Program:
$\{\mathsf{add}(x, 0, x)\}$.
$\{\mathsf{add}(x, y+1, z+1),\ \neg\mathsf{add}(x, y, z)\}$.
Query:
Which is a v, such that $\mathsf{add}(v, v, 2)$ *holds*?[59]
Resolution:

$\underbrace{\text{query and program clauses}}$	$\underbrace{\text{unifying substitutions}}$
$\{\neg\mathsf{add}(v, v, 2)\}\ \{\mathsf{add}(x, y+1, z+1),\ \neg\mathsf{add}(x, y, z)\}$	$[x/v, z/1, v/(y+1)]$
$\{\neg\mathsf{add}(y+1, y, 1)\}\ \{\mathsf{add}(x, 0, x)\}$	$[x/(y+1), y/0]$
$\emptyset$	

Answer: $s = ([v/(y+1)] \circ [y/0]) = \underline{[v/1]}$.[60]

[58] Law of quantification and negation in FOL: $(\neg\exists x : G) \equiv (\forall x : \neg G)$

[59] In Functional Programming (Part A of this course), such a "backwards" query is *not* possible! In FP we can only provide Input "1" and compute the Output $(1 + 1 = 2)$ in a "forward" manner.

[60] In this program $0, 1 \in \mathbb{N}$ are proper numbers, *not* the logical truth-values False/True. Moreover we assume in this program that these numbers are already *semantically interpreted* with $\mathbb{N} := A$, such that

In other words: In LP, every variable occuring in a query can be used in a *dual role*, as "Input parameter" as well as as "Output parameter". This is the semantics of *Relations* in LP (as opposed to the more specific Functions in FP).[61]

5.2 SEMANTICS

5.2.1 DECLARATIVE Semantics

Definition: *DECLARATIVE SEMANTICS of a Logic Program*
Let F be a Logic Program (as defined above) with a Goal Clause G containing (negative) Literals $L_1, \ldots, L_k$. The set $\mathcal{M}_d(F, G) := \{H \mid H$ is a **Ground Instance of** G with regard to $L_1, \ldots, L_k$, **and** H is a **Semantic Consequence** ($\models$) of $F\}$ is then the *Declarative Semantics* (Meaning) of F with regard to G.[62] Each of those Ground Instances reflects, in L_i, the variable substitutions (of type $[x/t]$) which represent the 'solution' for the Goal.

5.2.2 OPERATIONAL Semantics

The *idea* is the following. We derive (intermediate) computation-results by means of SLD resolution in a "backward chaining" mode of operation from goals to sub-goals, etc. Thereby we continue to make unifying variable substitutions on the basis of the already computed intermediate results. A computation will be a sequence of *configurations*, and a configuration will consist in a 'current' negative clause (sub-goal) together with an 'environment' which keeps track of the variable bindings to which we have already committed ourselves so far.[63] The start configuration contains the Goal and the empty substitution, whereas the final configuration (IF the process terminates) will contain the empty clause and the MGU as solution:

---

we can do the 'usual arithmetics' with them, such as $1 + 1 = 2$. If we would regard those numbers merely syntactically as constant terms in Σ, e.g.: $1 = \mathtt{1}()$, and with out any semantic interpretation of the $+$ operator in Σ, then we would *not* be able to carry out the additions of numbers which are shown in the example of above.

[61] Every Function is also a Relation, but *not* every Relation is also a Function.

[62] Note that this Semantics is defined 'per default' in the Herbrand World of Terms, *without* a Structural Mapping $\mathcal{A} =< A, \phi, \psi >$ into the 'World of Things'! For comparison see Page 20 of these Lecture Notes.

[63] For the 'environment' of variable bindings (substitutions), also remember Part A (FP) of this course.

$$\underbrace{(G, [])}_{\text{start config.}} \Longrightarrow^*_{\text{proc}} \underbrace{(\emptyset, [\text{MGU}])}_{\text{terminal config.}}$$

Important Note: Because of the MGU, which could still contain variables $x \in X$, there remains a 'gap' between the procedural (operational) and the declarative (denotational) semantics of a LP, because the declarative semantics was defined above as completely variable-free.

Definition: *CONFIGURATION of a Logic Program*
A configuration is a **Pair** (G, s) with a (negative) Goal clause $\{\neg L_1, \ldots, \neg L_k\}$, whereby all L_i of G are atomic (Σ, X)-Literals, and $s = [x_1/t_1, \ldots, x_n/t_n]$ a substitution (of variables by terms).

Definition: *Calculation (or Computation) STEP*
Given a Logic Program F. One **Calculation Step** $(G, s) \vdash_F (G', s')$ between two configurations is defined with regard to F if the following conditions are fulfilled:

- $G = \{\neg L_1, \ldots, \neg L_k\}$ with $k > 0$
- There exists in F some Rule Clause C $= \{P, \underline{\neg P_1, \ldots, \neg P_n}\}$ with $n \geq 0$ such that
  - The literals of G and C do not have any variable name x in common (which can always be achieved through a suitable re-naming of variables)
  - There exists some number $i \in \{1, \ldots, k\}$ such that L_i and P can be unified with s_{mgu} as their MGU
  - $G' = s_{\text{mgu}}(\{\neg L_1, \ldots, \neg L_{i-1}, \underline{\neg P_1, \ldots, \neg P_n}, \neg L_{i+1}, \ldots, \neg L_k\})$,[64]
  - $s' = (s \circ s_{\text{mgu}})$

Note the *double* **non**-determinism in this construction: at *Program level*, there could be *several possible clauses* C, C', C" (etc.) applicable, and at *Clause*

[64]Note how the sub-goal $\neg P_1, \ldots, \neg P_n$ has *replaced* the resolved literal L_i

level there could be *several possible literals* which could be used to resolve a (sub) goal. Because of this double non-determinism a whole computation-*tree* emerges, whereby *each path* through the branches of the tree, from the root to a leaf, represents a possible 'evaluation'. *Unlike* in Functional Programming where different evaluation paths must be *confluent* towards the same result (because of Functional Result-Determinism),[65] in Logic Programming the branches of the computation-tree are **not** confluent and can lead to different results (because of Relational Non-Determinism). Expressed in the vocabulary of a Set-theoretical interpretation of computation: For a function FUN there can be maximally *one* y such that $\text{FUN}(x) = y$, whereas for a relation REL there may be *many* y such that $\text{REL}(x) = y$ holds.

Moreover, it is generally possible to find in such computation trees branches of *infinite length* (due to Recursion), whereby the resulting goal clauses are growing and growing (by insertion of more and more sub-goals) without "$\emptyset$" ever being detected.

Definition: *CALCULATION (or Computation)*
A Calculation (or Computation) of a logic program F upon a goal query G is a (finite or possibly infinite) **sequence of configurations** $(G, [])$, (G_1, s_1), (G_2, s_2), ... , such that

- for every i in the sequence: $(G_i, s_i) \vdash_F (G_{i+1}, s_{i+1})$
- with $(\emptyset, s_{\text{mgu}})$ as the *final* (terminal) configuration **if and only if** the computation has been *successful*.

Thus there are three possible types of LP-Computations:
— 1. Terminating *and* successful,
— 2. Terminating *but not* successful,
— 3. Non-Terminating (and thus also not successful).

Definition: *PROCEDURAL (Operational) SEMANTICS of a Logic Program*
The procedural semantics of a LP consists of **all successful computation**

[65] Remember Part A of this course COS333.

results, which are *additionally* **ground-instantiated**.[66] This is the Set $\mathcal{M}_p(F,G) = \{H \mid H$ is a **Ground Instance of a Result** of a successful Calculation of F upon $G\}$.[67]

EXAMPLE: Given is the following LP (now in concrete *Prolog* syntax)
Program
$Q(x,z)$:– $Q(y,z), R(x,y)$.[68] // deduction (derivation) **rule**
$Q(x,x)$. // *general* **fact** (axiom): *with* variables
$R(\mathbf{b},\mathbf{c})$. // *particular* **fact** (axiom): *without* variables
Query
?– $Q(x,\mathbf{c})$.

Its Computation Tree (in table form) comprises the sub-trees and branches which are shown in the columns of Figure 1.

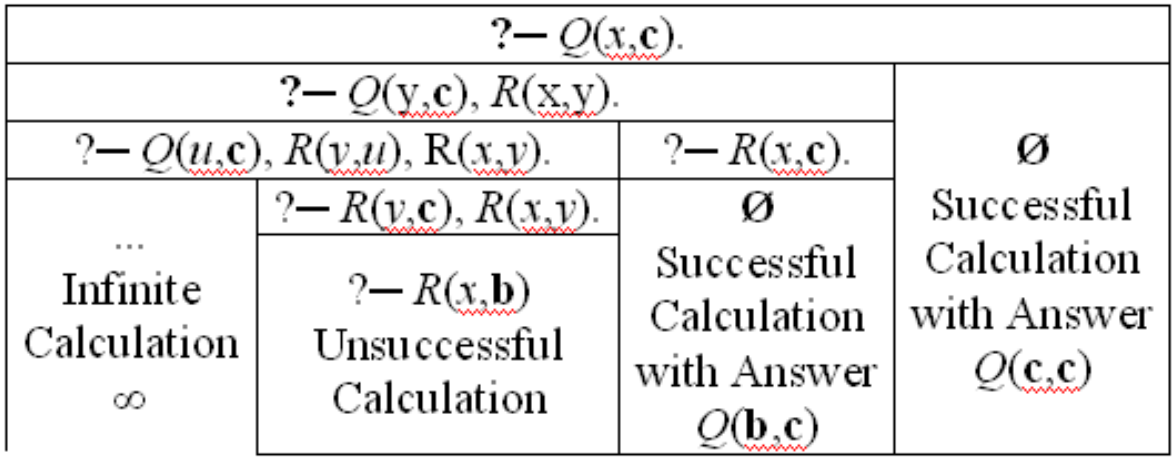


Figure 1: *Computation Tree (in table form) to the Example of above: three calculations are terminating, two are successful, one is non-terminating. Further un-successful calculations are branching off "side-ways" from the infinite path, but they are not shown in this figure.*

[66] This additional ground-instantiation condition is required to 'bridge the gap' between the Denotational Semantics, which is variable-free, and the final result of the LP computation which might possibly still contain un-instantiated variables in the MGU.

[67] To grasp the similarity and the difference between Operational and Denotational Semantics in LP, please compare this definition of $\mathcal{M}_p$ once again with the $\mathcal{M}_d$ definition of above.

[68] Note the *recursive* occurrence of predicate Q on the left-hand-side and on the right-hand-side of :–

THEOREM! [Clark] For every Logic Program F with some goal clause G, $\mathcal{M}_p(F,G) = \mathcal{M}_d(F,G)$.

Corollary! The procedural semantics is *correct and complete* with regard to the denotational (declarative) semantics.[69]

As we have seen above, there are two types of non-determinism in resolution, namely the choice of an applicable rule clause (if several rule clauses are applicable), and —*within* the chosen rule clause— the choice of an applicable literal L to be matched against its negation $\neg L$. These two types of non-determinism, which *both* have influence on the 'shape' of a Logic Program's Resolution Tree, have the following 'technical names'.

OR-Parallelism: This is the non-determinism of Type 1, whereby a rule clause is chosen.

AND-Parallelism: This is the non-determinism of Type 2, whereby a literal within a rule clause is chosen.

Note that the completeness property of SLD resolution *cannot* be harmed by AND-Parallelism, because all sub-goals of a goal must be fulfilled anyway (no matter in what sequence) before the goal itself can be fulfilled.

On the other hand, the OR-Parallelism *can* be harmful, especially when always those rule clauses are chosen which drive the computation deeper and deeper into an infinite branch of the computation tree, such that the other branches will would never be visited again at all.[70]

To tackle this problem, the 'engine' (interpreter) behind a Logic Programming system could attempt different 'tactics' of Tree-Traversal, such as, for example, Breadth-First traversal instead of Depth-First traversal,[71] or some

[69] Notwithstanding the possibility that an actual computation might 'run away' in a non-terminating infinite branch of the computation tree — keep in mind that the two semantics $\mathcal{M}_p(F,G)$ and $\mathcal{M}_p(F,G)$ are mathematically defined as *sets*. Indeed Clark's Theorem of above is typically proven set-theoretically, whereby it is first shown that $\mathcal{M}_p(F,G) \subset \mathcal{M}_d(F,G)$ (correctness), and secondly that $\mathcal{M}_p(F,G) \supset \mathcal{M}_d(F,G)$ (completeness).

[70] Please inform yourself from suitable sources, whether or not this problem can occur in the LP language PROLOG: What types of Tree Traversal strategies is implemented in the Resolution 'Engines' of the various available PROLOG versions?

[71] Remember Tree Traversal from your course on Algorithms and Data Structures: COS212.

'smart' combinations thereof. Thus: an *Evaluation Strategy*, which *deterministically* selects *one* computation path *out of many* available computation *path possibilities*,[72] is implemented in LP by a particular manner of Tree Traversal:

— Depth first search = rapid, but dangerous! (possibly $\nearrow^{\infty}$),
— Breadth first search = slow, but safe! (no branch will be omitted),
— Forward only search = stop when the first solution is found,
— Backtracking search = find all possible solutions.

Definition: *SLD TREE*
Let F be a logic program with goal clause G. The SLD Tree of F with regard to G is a finite or infnite tree, the **Nodes** of which are marked by **Configurations** such that the following conditions are fulfilled:

1. The **Root** is marked by $(G, [])$

2. If a **Node**, which is *not* the Root, is marked by some (G'', s'') then its **Parent** is marked by some (G', s') in such a manner that $(G', s') \vdash_F (G'', s'')$ can be calculated on one step.

3. Every Node, which is *not* a *Leaf*, has as many **Children** as is the size (cardinality) of the set of configurations which can be one-step-computed from that node.

For an illustrative example see Figure 1 of above.

5.2.3 FIXPOINT Semantics

Please remember Part A (Functional Programming) of this course, where the denotational semantics of a *recursive* Functional Program was defined in terms of a fixpoint of a function-generating Functional. Thus, in FP, the denotational semantics of recursive functions was indeed a fixpoint semantics. Now in LP, however, we had just defined the denotational semantics —see above— and that definition did not contain any reference to any fixpoint at all! Thus you may ask whether or not any fixpoint semantics of Logic Programs exists at all? The answer is YES, as shown in the subsequent paragraphs.

[72] Remember Part A (Functional Programming) of this course.

Let $\mathcal{F}$ be the **set of all atomic** Σ-formulae, i.e.: the variable-free prime-formulae over (Σ, X) whereby X is irrelevant. As we have already seen above, a Logic Program F over (Σ, X) yields a *set transformation* of type $\mathbf{TRANS}_F$**:** $\mathcal{P}(\mathcal{F}) \longrightarrow \mathcal{P}(\mathcal{F})$,[73] with

- $\mathrm{TRANS}_F(S) := \{L' \mid$ there exists a clause $\{L, \neg L_1, \ldots, \neg L_k\}$ in F (with $k \geq 0$) and thereof a variable-free ground instance $\{L', \neg L'_1, \ldots, \neg L'_k\}$ such that for all $i = 0, \ldots k$: $L'_i \in S\}$

In other words, TRANS_F describes the (set of) *logical implications* of F.

LEMMA! Under the above-defined conditions, the following assertions hold:

1. The pair $< \mathcal{P}(\mathcal{F}), \subseteq >$ is a **complete partial order**,[74] i.e.: every ordered ("directed") Sub-Set of $\mathcal{P}(\mathcal{F})$ possesses a Smallest Upper Bound.

2. The function TRANS_F is **monotonic function**,[75] i.e.: **If** $S \subseteq S'$, **then also** $\mathrm{TRANS}_F(S) \subseteq \mathrm{TRANS}_F(S')$.

3. The function TRANS_F is **continuous**,[76] such that for each ordered ("directed") sub-set $S \subseteq \mathcal{P}(\mathcal{F})$ with element-sets $S_i \in S$: $\mathrm{TRANS}_F(\cup_{S_i \in S}(S_i)) = \cup_{S_i \in S}(\mathrm{TRANS}_F(S_i))$.

Because of those three conditions holding, **Tarski's** *Fixpoint Theorem* is now *applicable* to the function TRANS, with the following result.[77] $\ldots\ldots\ldots \rightarrow$

[73] $\mathcal{P}(S)$ denotes the *Power-Set* of some given set S, i.e.: the set of all sub-sets of S. Please see your own Handbook of Set Theory for a precise definition of the concept of "Power-Set".

[74] Please see your own Handbook of Universal Algebra for the precise definition of the notion of "complete partial order", and also remember Part A (Functional Programming) of this course. For a very simple illustration of a complete partial order, also remember the data-structure HEAP from your course on algorithms and data structures, COS212.

[75] See your own Handbook of Universal Algebra and Function Theory for the precise definition of the concept of "monotonic function" (a.k.a. "monotone" function).

[76] Note that the definition of "continuous" in *Discrete* Universal Algebra is *different* from the definition of "continuous" in the domain of Analysis (differentiation, integration) of functions in the Real numbers $\mathbb{R}$.

[77] Tarski's Theorem, in its original form, was stated differenly. What you see above is already an *application* of Tarski's Theorem, not Tarski's Theorem in its 'pure' form. Please study Tarski's Fixpoint Theorem from your own Handbook of Mathematics.

Fixpoint THEOREM! The function TRANS_F, as defined above, possesses a least (smallest) fixpoint, namely $\mathbf{FIX}(\text{TRANS}_F) := \cup_{n\in\mathbb{N}}(\text{TRANS}^n{}_F(\{\}))$,[78] whereby $\text{TRANS}_F(\{\})$ contains only the *ground instances* of the positive *fact* clauses ("axioms") of program F.

Definition: *FIXPOINT SEMANTICS of a Logic Program*
Let F be a Logic Program together with a Goal $G = \{\neg L_1, \ldots, \neg L_k\}$. Its Fixpoint Semantics is defined as the set $\mathcal{M}_f(F,G) := \{H \mid H$ is a Ground Instance $(L'_1 \wedge \cdots \wedge L'_k)$ of $(L_1 \wedge \cdots \wedge L_k)$ whereby all $L'_i \in \text{FIX}(\text{TRANS}_F)\ \}$

THEOREM! $\mathcal{M}_f(F,G) = \mathcal{M}_p(F,G) = \mathcal{M}_d(F,G)$ for all Logic Programs F with Goal G.

5.3 UNIVERSALITY

In sub section we briefly address the question "how far we can get" with LP, particularly in comparison against Imperative Programming (IP).[79] Classically, in IP, we say that a programming languages is *universal* if it allows to *WHILE*-compute *every* Turing-*computable* function in the domain of *Arithmetics*. Thus the classical notion of universal computability is strongly connected with Arithmetics. Therefore, if we want to ask now whether or not LP is universal in this sense, we need to relate our question to some suitable LP language which does not only cover "pure" logic (as elaborated in the previous paragraphs), but which is *also* able to do Arithmetics. The LP language PROLOG, for example, is such a language. Only then we may possibly come to a conclusion concerning the universality of LP.

Definition: *Computation of Arithmetic Functions by Logic Programs*
A (possibly partial) **arithmetic function** $f : \mathbb{N}^n \rightharpoonup \text{IN}$ is represented by its I/O **graph** (relation) $\tilde{f} \subseteq \mathbb{N}^{n+1}$ such that $\tilde{f}(i_1, \ldots, i_n, o)$ if and only if

[78] As usual, $\text{TRANS}^n(\ldots) := \text{TRANS}(\text{TRANS}^{n-1}(\ldots))$. Similar to what we had seen in Part A (Functional Programming) of this course (with the function-generating Functional Φ) also here the result emerges so-to-say from "creation out of nothing". In Part A (FP), the "nothing" was the empty function $\lambda x.\ \bot$, whereas here in LP the "nothing" is the empty set {}.

[79] IP: Part C of this course COS333.

$f(i_1, \ldots, i_n) = o$. Thereby the output number $o \in \mathbb{N}$ is represented syntactically by the *term* $\underbrace{s(s(\cdots s(}_{(o \text{ many})} 0)\cdots))$, with s representing "successors" numbers.
On these preliminaries we say that a Logic Program F computes an arithmetic function $f : \mathbb{N}^n \rightharpoonup \mathrm{IN}$ **if an only if** there exists some predicate $P \in \Pi^{(n+1)}$ (with $P = \tilde{f}$) such that $f(i_1, \ldots, i_n) = o$ if and only if $P(i_1, \ldots, i_n, o)$ holds.

From a Computability-theoretical point of view,[80] the programs which can be written with imperative WHILE-Loops correspond the class of *μ-recursive* functions. Thus, if we want to know whether or not LP is "as powerful" as IP, we must ask whether or not μ-recursive functions are computable in LP.

Definition: *μ-RECURSIVE Function*
The class $\mathbf{REC} = \cup_{n \in \mathbb{N}}(\mathrm{REC}^{(n)})$ of all μ-recursive functions is the *smallest* class of arithmetic operations which has all of the following properties.[81]

1. **All basic functions are μ-recursive**, which are:
   - Constant Null $\in \mathrm{REC}^n$
     null: $\mathbb{N}^n \longrightarrow \mathbb{N}$, with $\mathsf{null}(i_1, \ldots, i_n) := 0$.
   - Successor $\in \mathrm{REC}^1$
     succ: $\mathbb{N} \longrightarrow \mathbb{N}$, with $\mathsf{succ}(i) := i + 1$.
   - j^{th} Projection $\in \mathrm{REC}^n$
     proj_j: $\mathbb{N}^n \longrightarrow \mathbb{N}$, with $\mathsf{proj}_j(i_1, \ldots, i_n) := i_j$ whereby $1 \leq j \leq n$.
2. **REC is closed** under *functional composition*,[82] *primitive recursion*,[83] and *minimalisation* as follows.
   - Functional Composition:

[80] Please see your own Handbook of Theoretical Computer Science, particularly Theory of Computability and Computable Functions.
[81] For comparison, look again at Section 3.1 of Part A (Functional Programming) of this course COS333, where the class of *Polynom* functions was defined on the basis of *constants*, *basic operations*, *projections*, and *functional composition*.
[82] Remember Part A (Functional Programming) of this course.
[83] Primitive Recursion corresponds to genuine down-counting and terminating FOR-loops in Imperative Programming, whereas μ-Recursion corresponds to possibly non-terminating WHILE-loops in IP.

Let there be $f \in \text{REC}^{(m)}$ and $f_1, \ldots, f_m \in \text{REC}^{(n)}$.
Then **also** $h := \mathsf{com}(f, f_1, \ldots, f_m) \in \text{REC}^{(m)}$, whereby
$h(\bar{i}) = f(f_1(\bar{i}), \ldots, f_n(\bar{i}))$ with $\bar{i} := (i_1, \ldots, i_n)$ as input vector

- Primitive Recursion:
  Let there be $f \in \text{REC}^{(n)}$ and $g \in \text{REC}^{(n+2)}$.
  Then **also** $h := \mathsf{pri}(f, g) \in \text{REC}^{(n+1)}$, whereby
  $h(\bar{i}, 0) := f(\bar{i})$ with input vector $\bar{i}$ as defined above, and
  $h(\bar{i}, (n+1)) := g(\bar{i}, n, h(\bar{i}, n))$
- Minimalisation:[84]
  Let there be $f \in \text{REC}^{(n+1)}$.
  Then **also** $h := \mathsf{min}(f) \in \text{REC}^{(n)}$, whereby
  $h(\bar{i}) := z$ **if and only if** $f(\bar{i}, z) = 0$, with $f(\bar{i}, y) > 0$ for all $0 \leq y < z$
  $h(\bar{i}) :=\perp$ **otherwise**.

Theoretical Computer Science tells us that this class of μ-recursive functions and the class of *Turing machines* are computationally equivalent, i.e.: anything that Turing machines can ever compute are the μ-recursive functions.

FUNDAMENTAL THEOREM of Logic Programming!

Every μ-Recursive Function f is computable by some Logic Program F_f.

6 WARREN's ABSTRACT MACHINE (WAM): Translation of Logic Programs into Stack Code

General Principle: Similar to the Abstract Stack Machines for Functional Programming, which you have already studied in Part A of this course COS333! However, instead of Function Evaluation (Part A), it now Unification and Resolution for which a translation to stack machine code must be found. In both cases (Functional Programming and Logic Programming) *recursion* may occur and must then be translated into *iteration* in the domain

[84] Minimalisation, as defined above, is a mathematically sound manner of describing IF-ELSE-Decisions: see Part A (Functional Programming) of this course for comparison. An alternative manner of describing such decisions is via the classical *Kronecker delta* function.

of the stack machine — recall that the stack machine is always "imperative" in its operationality.[85]

Declarative Program: *Recursion* $\Longrightarrow_{trans}$ Imperative Encoding: *Iteration*

Literature: Hassan Ait-Kaci, *Warren's Abstract Machine*, MIT-Press, 1991.

History

— Early 1970s: PROLOG interpreter coded in FORTRAN by Colmerauer
— Around 1980: PROLOG compiler for the DEC-10 machine by Warren
— 1983: Abstraction of the WAM makes PROLOG platform-independent.[86]

Structure of the WAM: The WAM is a unification machine with back-tracking such as to compute *all* possible solutions for a given goal. It is structured as a tuple $< D, I >$ with the following two main components:

D is a data memory structure to capture a goal term and its substitutions during the process of unification.

I is an instruction set which contains **many** instructions (commands) for:

- the execution of unification and resolution (*Logic* level), as well as
- *auxiliary* instructions to organize and update the WAM's memory structure (internal level).
  — The need for Backtracking makes this particularly complicated.[87]

Before such a machine can 'run'($\mathcal{E}[\![\ldots]\!]$ operational/evaluation semantics of machine commands/instructions), its memory structures must be 'fed' with the *translated form* of the Logic Program into which the user is interested.[88] This requires technical solutions to (at least) the following technical problems:

- Design of suitable memory *representations* of clauses and terms,[89]

[85] Remember Part A, Functional Programming, of this course COS333.
[86] The same idea was followed later with Java and its platform-independent virtual machine, JVM.
[87] For comparison: In the Abstract Stack Machines for Functional Programming (shown in Part A of this course) *no* backtracking was needed, because a *Function cannot have more than one output* value for any given input value.
[88] Remember the translations of Functional Programs into Stack Machine Code in Part A of this course.
[89] Remember your course on Algorithms and Data Structures: COS212.

- Translation of the LP goals (no positive literals),
- Translation of the LP rules (one positive literal),
- Translation of the LP facts (no negative literals),
- Design of suitable auxiliary instructions the execution of which will insert the translated clauses and terms into their proper 'slots' in the available memory.

Without going into any details, we mention here only that the Data Structure HEAP is used to contain the various parts of a translated Logic Program, and that a rather complicated STACK-similar Data Structure is used to maintain the variable-term-substitution *Environments*.[90] When back-tracking needs to be done (in search for further solutions in other branches of the SLD resolution tree of a Logic Program), then the most recently stacked Environments will be popped off the stack again up to a point from where the search can continue in a forward mode.

Illustration: *Internal WAM instructions to create Goal Terms on the Heap*

Internal WAM Command	Internal Operational Semantics
put_structure $f/n, X_i$	HEAP[H] :=<STR, H+1> HEAP[H+1] := f/n X_i := HEAP[H] H := H+2
set_variable X_i	HEAP[H] :=<REF, H> X_i := HEAP[H] H := H+1
set_value X_i	HEAP[H] := X_i H := H+1

[90] A similar *Environment*, for keeping information about variable-term-substitutions, we had already seen in the Abstract Stack Machines for Functional Programming in Part A of this course.

Illustration: *Internal WAM Program to construct a Heap Representation for the Goal Clause* "?-$\underline{p(Z, h(Z, Y), f(Y))}$"

Line 1: put_structure $h/2, X_3$
Line 2: set_variable X_2
Line 3: set_variable X_5
Line 4: put_structure $f/1$
Line 5: set_value X_5
Line 6: put_structure $p/3, X_1$
Line 7: set_value X_2
Line 8: set_value X_3
Line 9: set_value X_4

Further explanations:

Predicate $p(...)$ is of arity 3, therefore the notation $p/3$
Term $h(..)$ is of arity 2, therefore the notation $h/2$
Term $f(.)$ is of arity 1, therefore the notation $f/1$

The entire clause had previously been *de-composed* ("parsed") as follows:
$X_1 := p(Z, X_3, X_4)$
$X_4 := f(Y)$
$X_3 := h(Z, Y)$
$X_2 := Z$
$X_5 := Y$

Line 2, Line 3: "set_variable" is used when some variable is "seen" for the very first time, such that its value (interpretation) can still be freely chosen. Line 5, Line 7, Line 8, Line 9: "set_value" is used when a variable name has already been "seen" before and is now re-occurring, such that it *must* assume the *same* value as one of those variables which had been set previously with the "set_variable" command.

Illustration: *Resulting Heap, after the program from the previous page* (Line 1–Line 9) *has been carried out by the WAM*

Here the HEAP (Graph) is depicted in a Linear ARRAY Representation.[91] Note that *shared* occurrences of terms or variables are referenced *multiple* times, and that each "independent" variables is characterized by a pointer to itself.[92]

Adr.	*Contents*	*points to*
0	STR	1
1	$h/2$	
2	REF	2
3	REF	3
4	STR	5
5	$f/1$	
6	REF	3
7	STR	8
8	$p/3$	
9	REF	2
10	STR	1
11	STR	5

The "entry point" into this structure can be found at address **7**, whereby the H pointer to the next free cell contains the address **12**.[93]

[91] Remember COS212: Algorithms and Data Structures. Here, though, the Heap is a Graph, not a Tree.
[92] **Homework**: Onto the empty space of this page, *draw the Graph* which is represented by this Array.
[93] See the operational semantics of the heap construction commands on page 50 of these Lecture Notes.

For the execution of UNIFICATION, the WAM's instruction set contains a large number of further instructions and auxiliary instructions, including "unify_variable X_i", "get_structure $t/n, X_i$", for the de-referencing and binding of variables, etc.

For the purpose of Backtracking there are instructions such as "try", "unwind" and "proceed", whereas for the management of the ENVIRONMENT STACK stack there are further commands such as "call", "allocate", "deallocate", "store", etc.

Because the educational purpose of this course COS333 is the comparison of the 4 main programming paradigms —FP, LP, IP, OOP— with regard to their *formal* semantics at a *higher level of abstracion*, we need not look any deeper into the finer structures and operational details of the WAM any further in order to achieve the purpose of this course.

BACKLOOK to Part A) — Functional Programming: In this Part B) of this course, the similarities as well as the differences between Functional and Logic Programming should have become sufficiently clear to you. Both of them are instances of Declarative Programming, whereby Logic Programming may be regarded as a non-deterministic Generalisation of deterministic Functional Programming — or, vice versa, FP might be regarded as a Special Case of LP. Both of them are defined over the notions of Sigma-Algebras, and both of them are so "well compatible" with each other that Computer Scientists in the past have also created Hybrid Functional-Logic Programming Languages.[94] Last but not least it should now also be quite clear to you that the simple Rules of the SECD Machine for Tail-Recursive Functional Programs could be easily re-written in terms of the Rules of a Logic Programming Language such as PROLOG. For example, the SECD Machine's first State Transition Rule,[95]

1. $(S, \varrho, (a|C), D) \Longrightarrow ((S|a), \varrho, C, D)$ ..... if $a \in$ DATA

could be easily prototyped "the other way around" in PROLOG as:
secd_state(push_right(S,A),R,C,D) :- secd_state(S,R,push_left(A,C),D), data(A).
Thereby "secd_state" and "data" are Predicates (which can be true or false),

[94] See for example the Language LEFUN by **Hassan Ait-Kaci**.
[95] See again the Lecture Notes to Part A (FP) of this course.

whereas "push_left" and "push_right" are Term Operators to build up the stack data structure (which cannot be true or false as such).[96]

OUTLOOK to PART C) — Imperative Programming: In the following Part of this course we will see that the Formal Semantics of Imperative Programming Languages can be elegantly captured by means of yet another formal calculus, namely the **Hoare** Calculus, also known as *Hoare Logic*. Like any other formal-logical calculus, also the Hoare Logic is characterized by some "axioms", which state what is "given", plus some "rules", which state what can be logically deduced or derived or concluded from those given axioms. Having well understood Formal Logics after the study of this Part B of COS333, you will find it quite easy to understand Hoare Logics in the formal semantics of Imperative Programming in Part C, too.

*

[96] When looking at PROLOG, keep in mind that Predicates and Term-Operators *look* very similar from a merely syntactical point of view: both take various arguments into their parameters, however their *meaning* is rather different. One might perhaps say somewhat sloppily that predicates are "operators with a boolean output type" whereas the usual term operators are "non-boolean". However, because of the Set-Theoretical similarity between functions and relations, we can always construct a Function which outputs "true" if and only if some given elements are standing in some predicated Relation with each other. (Beheld from that perspective, one might even regard the most basic PL syntax operators $\wedge$, $\vee$, $\neg$ as some peculiar kind of "relation predicates", namely over the "Data Sort" $\{0, 1\}$, whereby the corresponding "relations" would be the Truth-producing parts (sub-tables) of the classical Truth-tables.)