Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

March 1, 2016

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Contents

Part I Preliminaries

1 Introduction 3
- 1.1 Structure and Meaning 3
  - 1.1.1 Syntax and Types 4
  - 1.1.2 Semantics 4
  - 1.1.3 Mathematical Models of Computation 6
  - 1.1.4 Operational Semantics 7
  - 1.1.5 Denotational Semantics 8
  - 1.1.6 Axiomatic Semantics 8
- 1.2 A Taste of Semantics Methods: Numerical Expressions 9
  - 1.2.1 An Informal Semantics 10
  - 1.2.2 A Small-Step Operational Semantics 11
  - 1.2.3 A Big-Step Operational Semantics (or Natural Semantics) . . 13
  - 1.2.4 A Denotational Semantics 15
  - 1.2.5 Semantic Equivalence 16
  - 1.2.6 Expressions with Variables 17
- 1.3 Applications of Semantics 18
  - 1.3.1 Language Design 18
  - 1.3.2 Implementation 18
  - 1.3.3 Analysis and Verification 19
  - 1.3.4 Synergy Between Different Semantics Approaches 19
- 1.4 Content Overview 20
  - 1.4.1 Induction and Recursion 22
  - 1.4.2 Semantic Domains 24
  - 1.4.3 Bisimulation 26
  - 1.4.4 Temporal and Modal Logics 26
  - 1.4.5 Probabilistic Systems 27
- 1.5 Chapters Contents and Reading Guide 27

2 Preliminaries 31
2.1 Notation 31
2.1.1 Basic Notation 31
2.1.2 Signatures and Terms 32
2.1.3 Substitutions 33
2.1.4 Unification Problem 33
2.2 Inference Rules and Logical Systems 35
2.3 Logic Programming 43
Problems 45

Part II IMP: a simple imperative language

3 Operational Semantics of IMP 51
3.1 Syntax of IMP 51
3.1.1 Arithmetic Expressions 52
3.1.2 Boolean Expressions 52
3.1.3 Commands 52
3.1.4 Abstract Syntax 53
3.2 Operational Semantics of IMP 54
3.2.1 Memory State 54
3.2.2 Inference Rules 55
3.2.3 Examples 59
3.3 Abstract Semantics: Equivalence of Expressions and Commands . . . 64
3.3.1 Examples: Simple Equivalence Proofs 65
3.3.2 Examples: Parametric Equivalence Proofs 66
3.3.3 Examples: Inequality Proofs 68
3.3.4 Examples: Diverging Computations 70
Problems 73

4 Induction and Recursion 75
4.1 Noether Principle of Well-founded Induction 75
4.1.1 Well-founded Relations 75
4.1.2 Noether Induction 81
4.1.3 Weak Mathematical Induction 82
4.1.4 Strong Mathematical Induction 83
4.1.5 Structural Induction 83
4.1.6 Induction on Derivations 86
4.1.7 Rule Induction 87
4.2 Well-founded Recursion 90
Problems 95

5 Partial Orders and Fixpoints 99
5.1 Orders and Continuous Functions 99
5.1.1 Orders 100
5.1.2 Hasse Diagrams 101
5.1.3 Chains 105

5.1.4 Complete Partial Orders . . . 105
5.2 Continuity and Fixpoints . . . 108
5.2.1 Monotone and Continuous Functions . . . 108
5.2.2 Fixpoints . . . 110
5.3 Immediate Consequence Operator . . . 113
5.3.1 The Operator $\widehat{R}$ . . . 113
5.3.2 Fixpoint of $\widehat{R}$ . . . 114
Problems . . . 117

6 Denotational Semantics of IMP . . . 121
6.1 λ-Notation . . . 121
6.1.1 λ-Notation: Main Ideas . . . 122
6.1.2 Alpha-Conversion, Beta-Rule and Capture-Avoiding Substitution . . . 125
6.2 Denotational Semantics of IMP . . . 127
6.2.1 Denotational Semantics of Arithmetic Expressions: The Function $\mathscr{A}$ . . . 128
6.2.2 Denotational Semantics of Boolean Expressions: The Function $\mathscr{B}$ . . . 129
6.2.3 Denotational Semantics of Commands: The Function $\mathscr{C}$ . . . 130
6.3 Equivalence Between Operational and Denotational Semantics . . . 135
6.3.1 Equivalence Proofs For Expressions . . . 135
6.3.2 Equivalence Proof for Commands . . . 136
6.4 Computational Induction . . . 143
Problems . . . 145

Part III HOFL: a higher-order functional language

7 Operational Semantics of HOFL . . . 151
7.1 Syntax of HOFL . . . 151
7.1.1 Typed Terms . . . 152
7.1.2 Typability and Typechecking . . . 156
7.2 Operational Semantics of HOFL . . . 159
Problems . . . 164

8 Domain Theory . . . 167
8.1 The Flat Domain of Integer Numbers $\mathbb{Z}_\perp$ . . . 167
8.2 Cartesian Product of Two Domains . . . 167
8.3 Functional Domains . . . 169
8.4 Lifting . . . 172
8.5 Function's Continuity Theorems . . . 174
8.6 Useful Functions . . . 177
Problems . . . 181

9 HOFL Denotational Semantics . . . 183
9.1 HOFL Semantic Domains . . . 183
9.2 HOFL Evaluation Function . . . 184
9.2.1 Constants . . . 184
9.2.2 Variables . . . 184
9.2.3 Binary Operators . . . 185
9.2.4 Conditional . . . 185
9.2.5 Pairing . . . 186
9.2.6 Projections . . . 186
9.2.7 Lambda Abstraction . . . 187
9.2.8 Function Application . . . 187
9.2.9 Recursion . . . 187
9.3 Continuity of Meta-language's Functions . . . 189
9.4 Substitution Lemma . . . 191
Problems . . . 192

10 Equivalence between HOFL denotational and operational semantics . 195
10.1 Completeness . . . 196
10.2 Equivalence (on Convergence) . . . 199
10.3 Operational and Denotational Equivalences of Terms . . . 201
10.4 A Simpler Denotational Semantics . . . 202
Problems . . . 203

Part IV Concurrent Systems

11 CCS, the Calculus for Communicating Systems . . . 209
11.1 Syntax of CCS . . . 214
11.2 Operational Semantics of CCS . . . 215
11.2.1 Action Prefix . . . 216
11.2.2 Restriction . . . 216
11.2.3 Relabelling . . . 216
11.2.4 Choice . . . 217
11.2.5 Parallel Composition . . . 217
11.2.6 Recursion . . . 218
11.2.7 CCS with Value Passing . . . 221
11.2.8 Recursive Declarations and the Recursion Operator . . . 222
11.3 Abstract Semantics of CCS . . . 224
11.3.1 Graph Isomorphism . . . 224
11.3.2 Trace Equivalence . . . 226
11.3.3 Bisimilarity . . . 227
11.4 Compositionality . . . 233
11.4.1 Bisimilarity is Preserved by Choice . . . 234
11.5 A Logical View to Bisimilarity: Hennessy-Milner Logic . . . 235
11.6 Axioms for Strong Bisimilarity . . . 238
11.7 Weak Semantics of CCS . . . 240

11.7.1 Weak Bisimilarity . . . 240
11.7.2 Weak Observational Congruence . . . 242
11.7.3 Dynamic Bisimilarity . . . 243
Problems . . . 244

12 Temporal Logic and μ-Calculus . . . 249
12.1 Temporal Logic . . . 249
12.1.1 Linear Temporal Logic . . . 250
12.1.2 Computation Tree Logic . . . 252
12.2 μ-Calculus . . . 254
12.3 Model Checking . . . 257
Problems . . . 258

13 π-Calculus . . . 261
13.1 Name Mobility . . . 261
13.2 Syntax of the π-calculus . . . 264
13.3 Operational Semantics of the π-calculus . . . 266
13.3.1 Action Prefix . . . 267
13.3.2 Choice . . . 268
13.3.3 Name Matching . . . 268
13.3.4 Parallel Composition . . . 268
13.3.5 Restriction . . . 269
13.3.6 Scope Extrusion . . . 269
13.3.7 Replication . . . 269
13.3.8 A Sample Derivation . . . 270
13.4 Structural Equivalence of π-calculus . . . 271
13.4.1 Reduction semantics . . . 271
13.5 Abstract Semantics of the π-calculus . . . 272
13.5.1 Strong Early Ground Bisimulations . . . 273
13.5.2 Strong Late Ground Bisimulations . . . 274
13.5.3 Strong Full Bisimilarities . . . 275
13.5.4 Weak Early and Late Ground Bisimulations . . . 276
Problems . . . 277

Part V Probabilistic Systems

14 Measure Theory and Markov Chains . . . 281
14.1 Probabilistic and Stochastic Systems . . . 281
14.2 Measure Theory . . . 282
14.2.1 σ-field . . . 282
14.2.2 Constructing a σ-field . . . 283
14.2.3 Continuous Random Variables . . . 285
14.2.4 Stochastic Processes . . . 289
14.3 Markov Chains . . . 289
14.3.1 Discrete and Continuous Time Markov Chain . . . 290
14.3.2 DTMC as LTS . . . 291

14.3.3 DTMC Steady State Distribution 293
14.3.4 CTMC as LTS 295
14.3.5 Embedded DTMC of a CTMC 296
14.3.6 CTMC Bisimilarity 296
14.3.7 DTMC Bisimilarity 298
Problems 299

15 Markov Chains with Actions and Non-determinism 303
15.1 Discrete Markov Chains With Actions 303
15.1.1 Reactive DTMC 304
15.1.2 DTMC With Non-determinism 306
Problems 309

16 PEPA - Performance Evaluation Process Algebra 311
16.1 From Qualitative to Quantitative Analysis 311
16.2 CSP 312
16.2.1 Syntax of CSP 312
16.2.2 Operational Semantics of CSP 313
16.3 PEPA 314
16.3.1 Syntax of PEPA 314
16.3.2 Operational Semantics of PEPA 316
Problems 321

Glossary 325

Solutions 327

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\cong$	CCS weak observational congruence (see Section 11.7.2)
$\approx_d$	CCS dynamic bisimilarity (see Definition 11.17)
$\overset{\circ}{\sim}_E$	π-calculus early bisimilarity (see Definition 13.3)
$\overset{\circ}{\sim}_L$	π-calculus late bisimilarity (see Definition 13.4)
$\sim_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\sim_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\overset{\bullet}{\approx}_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\overset{\bullet}{\approx}_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\mathrm{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.1.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)

DTMC	Discrete Time Markov Chain (see Definition **??**)
Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.5)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.5)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.1.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.10)
PI	set of partial injective functions on natural numbers (see Problem 5.11)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.3.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.11)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

Part I
Preliminaries

This part introduces the basic terminology, notation and mathematical tools used in the rest of the book.

Chapter 2
Preliminaries

A mathematician is a device for turning coffee into theorems.
(Paul Erdos)

Abstract In this chapter we fix some basic mathematical notation used in the rest of the book and introduce the important concepts of signature, logical system and goal-oriented derivation.

2.1 Notation

2.1.1 Basic Notation

As a general rule, we use capital letters, like D or X, to denote sets of elements and small letters, like d or x, for their elements, with membership relation $\in$. The set of natural numbers is denoted by $\mathbb{N} \stackrel{\text{def}}{=} \{0,1,2,...\}$, the set of integer numbers is denoted by $\mathbb{Z} \stackrel{\text{def}}{=} \{...,-2,-1,0,1,2,...\}$ and the set of boolean by $\mathbb{B} \stackrel{\text{def}}{=} \{\mathbf{true},\mathbf{false}\}$. We write $[m,n] \stackrel{\text{def}}{=} \{k \mid m \leq k \leq n\}$ for the interval of numbers from m to n, extremes included. If a set A is finite, we denote by $|A|$ its *cardinality,* i.e., the number of its elements. The emptyset is written $\varnothing$, with $|\varnothing| = 0$. We use the standard notation for set union, intersection, difference, cartesian product and disjoint union, which are denoted respectively by $\cup$, $\cap$, $\setminus$, $\times$ and $\uplus$. We write $A \subseteq B$ if all elements in A belong to B. We denote by $\wp(A)$ the powerset of A, i.e., the set of all subsets of A.

An indexed set of elements is written $\{e_i\}_{i\in I}$ and a family of sets is written $\{S_i\}_{i\in I}$. Set operations are extended to families of sets by writing, e.g., $\bigcup_{i\in I} S_i$ and $\bigcap_{i\in I} S_i$. If I is the interval set $[m,n]$, then we write also $\bigcup_{i=m}^{n} S_i$ and $\bigcap_{i=m}^{n} S_i$.

Given a set A, and a natural number k we denote by A^k the set of sequences of k (not necessarily distinct) elements in A. Such sequences are called *strings* and their concatenation is represented by juxtaposition. We denote by $A^* = \bigcup_{k\in\mathbb{N}} A^k$ the set of all finite (possibly empty) sequences over A. Given a string $w \in A^*$ we denote by $|w|$ its *length*, i.e., the number of its elements (including repetitions). The empty string is denoted ε, and we have $|\varepsilon| = 0$ and $A^0 = \{\varepsilon\}$ for any A.

A relation R between two sets A and B is a subset of $A \times B$. For $(a,b) \in R$ we write also aRb. A relation $f \subseteq A \times B$ can be regarded as a function if both the following properties are satisfied:

function: $\forall a \in A, \forall b_1, b_2 \in B.\ (a,b_1) \in f \wedge (a,b_2) \in f \ \Rightarrow\ b_1 = b_2$
total: $\forall a \in A, \exists b \in B.\ (a,b) \in f$

For such a function f, we write $f : A \to B$ and say that the set A is the *domain* of f, and B is its *codomain*. We write $f(a)$ for the unique element $b \in B$ such that $(a,b) \in f$, i.e., f can be regarded as the relation $\{(a, f(a)) \mid a \in A\} \subseteq A \times B$. The composition of two functions $f : A \to B$ and $g : B \to C$ is written $g \circ f : A \to C$, it is such that for any element $a \in A$ it holds $(g \circ f)(a) = g(f(a))$. A relation that satisfies the "function" property, but not necessarily the "total" property, is called *partial*. A partial function f from A to B is written $f : A \rightharpoonup B$.

2.1.2 Signatures and Terms

A one-sorted (or unsorted) *signature* is a set of function symbols $\Sigma = \{c, f, g, \ldots\}$ such that each symbol in Σ is assigned an *arity*, that is the number of arguments it takes. A symbol with arity zero is called a *constant*; a symbol with arity one is called *unary*; a symbol with arity two is called *binary*; a symbol with arity three is called *ternary*. For $n \in \mathbb{N}$, we let $\Sigma_n \subseteq \Sigma$ be the set of function symbols whose arity is n.

Given an infinite set of variables $X = \{x, y, z, \ldots\}$, the set $T_{\Sigma,X}$ is the set of terms over Σ and X, i.e., the set of all and only terms generated according to the following rules:

- each variable $x \in X$ is a term (i.e., $x \in T_{\Sigma,X}$),
- each constant $c \in \Sigma_0$ is a term (i.e., $c \in T_{\Sigma,X}$),
- if $f \in \Sigma_n$, and $t_1, \ldots, t_n$ are terms (i.e., $t_1, \ldots, t_n \in T_{\Sigma,X}$), then also $f(t_1, \ldots, t_n)$ is a term (i.e., $f(t_1, \ldots, t_n) \in T_{\Sigma,X}$).

For a term $t \in T_{\Sigma,X}$, we denote by $\mathrm{vars}(t)$ the set of variables occurring in t, and let $T_\Sigma \subseteq T_{\Sigma,X}$ be the set of terms with no variables, i.e., $T_\Sigma \overset{\text{def}}{=} \{t \in T_{\Sigma,X} \mid \mathrm{vars}(t) = \varnothing\}$.

Example 2.1. For example, take $\Sigma = \{0, succ, plus\}$ with 0 a constant, *succ* unary and *plus* binary. Then all of the following are terms:

- $0 \in T_\Sigma$
- $x \in T_{\Sigma,X}$
- $succ(0) \in T_\Sigma$
- $succ(x) \in T_{\Sigma,X}$
- $plus(succ(x), 0) \in T_{\Sigma,X}$
- $plus(plus(x, succ(y)), plus(0, succ(x))) \in T_{\Sigma,X}$

The set of variables of the above terms are respectively:

- $\mathrm{vars}(0) = \mathrm{vars}(succ(0)) = \varnothing$

- $\mathrm{vars}(x) = \mathrm{vars}(\mathit{succ}(x)) = \mathrm{vars}(\mathit{plus}(\mathit{succ}(x), 0)) = \{x\}$
- $\mathrm{vars}(\mathit{plus}(\mathit{plus}(x, \mathit{succ}(y)), \mathit{plus}(0, \mathit{succ}(x)))) = \{x, y\}$

Instead $\mathit{succ}(\mathit{plus}(0), x)$ is not a term: can you see why?

2.1.3 Substitutions

A *substitution* $\rho : X \to T_{\Sigma,X}$ is a function assigning terms to variables.

Since the set of variables is infinite while we are interested only in terms with a finite number of variables, we consider only substitutions that are defined as identity everywhere except on a finite number of variables. Such substitutions are written

$$\rho = [x_1 = t_1, \dots, x_n = t_n]$$

meaning that

$$\rho(x) = \begin{cases} t_i & \text{if } x = x_i \\ x & \text{otherwise} \end{cases}$$

We denote by $t\rho$, or sometimes by $\rho(t)$, the term obtained from t by simultaneously replacing each variable x with $\rho(x)$ in t.

Example 2.2. For example, consider the signature from Example 2.1, the term $t \stackrel{\text{def}}{=} \mathit{plus}(\mathit{succ}(x), \mathit{succ}(y))$ and the substitution $\rho \stackrel{\text{def}}{=} [x = \mathit{succ}(y), y = 0]$. We get:

$$t\rho = \mathit{plus}(\mathit{succ}(x), \mathit{succ}(y))[x = \mathit{succ}(y), y = 0] = \mathit{plus}(\mathit{succ}(\mathit{succ}(y)), \mathit{succ}(0))$$

We say that the term t is *more general* than the term t' if there exists a substitution ρ such that $t\rho = t'$. The "more general than" relation is reflexive and transitive, i.e., it defines a *pre-order*. Note that there are terms t and t', with $t \neq t'$, such that t is more general than t' and t' is more general than t.

We say that the substitution ρ is more general than the substitution ρ' if there exists a substitution ρ'' such that for any variable x we have that $\rho''(\rho(x)) = \rho'(x)$ (i.e., $\rho(x)$ is more general than $\rho'(x)$ as witnessed by ρ'').

2.1.4 Unification Problem

The unification problem, in its simplest formulation (syntactic, first-order unification), consists of finding a substitution ρ that identifies some terms pairwise.

Formally, given a set of potential equalities

$$G = \{ l_1 \stackrel{?}{=} r_1, \dots, l_n \stackrel{?}{=} r_n \}$$

where $l_i, r_i \in T_{\Sigma,X}$, we say that a substitution ρ is a solution of G if

$$\forall i \in [1,n].\ l_i\rho = r_i\rho.$$

The unification problem, consists in finding a *most general* substitution ρ.

We say that two sets of potential equalities G and G' are *equivalent* if they have the same set of solutions.

We denote by $\mathrm{vars}(G)$ the set of variables occurring in G, i.e.:

$$\mathrm{vars}(\{\, l_1 \overset{?}{=} r_1, ..., l_n \overset{?}{=} r_n \,\}) = \bigcup_{i=1}^{n} (\mathrm{vars}(l_i) \cup \mathrm{vars}(r_i))$$

Note that the solution does not necessarily exists, and when it exists it is not necessarily unique.

The unification algorithm takes as input a set of potential equalities G as the one above and applies some transformations until:

- either it terminates (no transformation can be applied any more) after having transformed the set G to an equivalent set of equalities

  $$G' = \{\, x_1 \overset{?}{=} t_1, ..., x_k \overset{?}{=} t_k \,\}$$

  where $x_1, ..., x_k$ are all distinct variables and $t_1, ..., t_k$ are terms with no occurrences of $x_1, ..., x_k$, i.e., such that $\{x_1, ..., x_k\} \cap \bigcup_{i=1}^{k} \mathrm{vars}(t_i) = \varnothing$: the set G' directly defines a most general solution

  $$[x_1 = t_1, ..., x_k = t_k]$$

  to the unification problem G;
- or it fails, meaning that the potential equalities cannot be unified.

In the following we denote by $G\rho$ the set of potential equalities obtained by applying the substitution ρ to all terms in G. Formally:

$$\{\, l_1 \overset{?}{=} r_1, ..., l_n \overset{?}{=} r_n \,\}\rho = \{\, l_1\rho \overset{?}{=} r_1\rho, ..., l_n\rho \overset{?}{=} r_n\rho \,\}$$

The unification algorithm tries to apply the following steps (the order is not important for the result, but it may affect complexity), to transform an initial set of potential equalities until no more steps can be applied or the algorithm fails:

delete:	$G \cup \{\, t \overset{?}{=} t \,\}$ is transformed to G
decompose:	$G \cup \{\, f(t_1, ..., t_m) \overset{?}{=} f(u_1, ..., u_m) \,\}$ is transformed to $G \cup \{\, t_1 \overset{?}{=} u_1, ..., t_m \overset{?}{=} u_m \,\}$
swap:	$G \cup \{\, f(t_1, ..., t_m) \overset{?}{=} x \,\}$ is transformed to $G \cup \{\, x \overset{?}{=} f(t_1, ..., t_m) \,\}$
eliminate:	$G \cup \{\, x \overset{?}{=} t \,\}$ is transformed to $G[x = t] \cup \{\, x \overset{?}{=} t \,\}$ if $x \in \mathrm{vars}(G) \wedge x \notin \mathrm{vars}(t)$
conflict:	$G \cup \{\, f(t_1, ..., t_m) \overset{?}{=} g(u_1, ..., u_h) \,\}$ leads to failure if $f \neq g \vee m \neq h$
occur chek:	$G \cup \{\, x \overset{?}{=} f(t_1, ..., t_m) \,\}$ leads to failure if $x \in \mathrm{vars}(\, f(t_1, ..., t_m)\,)$

Example 2.3. For example, if we start from

$$G = \{\, plus(succ(x), x) \stackrel{?}{=} plus(y, 0) \,\}$$

by applying rule **decompose** we obtain

$$\{\, succ(x) \stackrel{?}{=} y \,,\, x \stackrel{?}{=} 0 \,\}$$

by applying rule **eliminate** we obtain

$$\{\, succ(0) \stackrel{?}{=} y \,,\, x \stackrel{?}{=} 0 \,\}$$

finally, by applying rule **swap** we obtain

$$\{\, y \stackrel{?}{=} succ(0) \,,\, x \stackrel{?}{=} 0 \,\}$$

Since no further transformation is possible, we conclude that

$$\rho = [\, y = succ(0) \,,\, x = 0 \,]$$

is the most general unifier for G.

2.2 Inference Rules and Logical Systems

Inference rules are a key tool for defining syntax (e.g., which programs respect the syntax, which programs are well-typed) and semantics (e.g., to derive the operational semantics by induction on the syntax structure of the programs).

Definition 2.1 (Inference rule). Let $x_1, x_2, \dots, x_n, y$ be (well-formed) formulas. An *inference rule* is written, using inline notation, as

$$r \quad = \quad \{\underbrace{x_1, x_2, \dots, x_n}_{\text{premises}}\} \quad / \quad \underbrace{y}_{\text{conclusion}}$$

Letting $X = \{\, x_1, x_2, \dots, x_n \,\}$, equivalent notations are

$$r = \frac{X}{y} \qquad\qquad r = \frac{x_1 \quad \dots \quad x_n}{y}$$

The meaning of such a rule r is that if we can prove all the formulas $x_1, x_2, \dots, x_n$ in our logical system, then by exploiting the inference rule r we can also derive the validity of the formula y.

Definition 2.2 (Axiom). An *axiom* is an inference rule with empty premise:

$$r = \varnothing / y.$$

Equivalent notations are:

$$r = \frac{\varnothing}{y} \qquad\qquad r = \frac{}{y}$$

In other words, there are no preconditions for applying an axiom r, hence there is nothing to prove in order to apply the rule: in this case we can assume y to hold.

Definition 2.3 (Logical system). A *logical system* is a set of inference rules $R = \{ r_i \}_{i \in I}$.

Given a logical system, we can start by deriving obvious facts using axioms and then derive new valid formulas applying the inference rules to the formulas that we know to hold (used as premises). In turn, the newly derived formulas can be used to prove the validity of other formulas.

Example 2.4 (Some inference rules). The inference rule

$$\frac{x \in E \quad x \in E \quad x \oplus y = z}{z \in E}$$

means that, *if* x and y are two elements that belongs to the set E *and* the result of applying the operator $\oplus$ to x and y gives z as a result, *then* z must also belong to the set E.

The rule

$$\frac{}{2 \in E}$$

is an axiom, so we know that 2 belongs to the set E.

By composing inference rules, we build *derivations*, which explain how a logical deduction is achieved.

Definition 2.4 (Derivation). Given a logical system R, a *derivation* is written

$$d \Vdash_R y$$

where

- either $d = \varnothing/y$ is an axiom of R, i.e., $(\varnothing/y) \in R$;
- or there are some derivations $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ such that $d = (\{d_1, \ldots, d_n\}/y)$ and $(\{x_1, \ldots, x_n\}/y) \in R$.

The notion of derivation is obtained putting together different steps of reasoning according to the rules in R. We can see $d \Vdash_R y$ as a proof that, in the formal system R, we can derive y.

Let us look more closely at the two cases in Definition 2.4. The first case tells us that if we know that:

$$\left(\frac{\varnothing}{y}\right) \in R$$

i.e., if we have an axiom for deriving y in our inference system R, then

$$\left(\frac{\varnothing}{y}\right) \Vdash_R y$$

is a derivation of y in R.

The second case tells us that if we have already proved x_1 with derivation d_1, x_2 with derivation d_2 and so on, i.e.,

$$d_1 \Vdash_R x_1, \qquad d_2 \Vdash_R x_2, \qquad ..., \qquad d_n \Vdash_R x_n$$

and, in the logical system R, we have a rule for deriving y using $x_1, ..., x_n$ as premises, i.e.,

$$\left(\frac{x_1, ..., x_n}{y}\right) \in R$$

then we can build a derivation for y as follows:

$$\left(\frac{\{d_1, ..., d_n\}}{y}\right) \Vdash_R y$$

Summarising all the above:

- $(\varnothing/y) \Vdash_R y \quad$ if $(\varnothing/y) \in R$ (axiom)
- $(\{d_1, \ldots, d_n\}/y) \Vdash_R y \quad$ if $(\{x_1, \ldots, x_n\}/y) \in R$ and $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ (inference)

A derivation can roughly be seen as a tree whose root is the formula y we derive and whose leaves are the axioms we need. Correspondingly, we can define the height of a derivation tree as follows:

$$\mathit{height}(d) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } d = (\varnothing/y) \\ 1 + \max\{\mathit{height}(d_1), ..., \mathit{height}(d_n)\} & \text{if } d = (\{d_1, \ldots, d_n\}/y) \end{cases}$$

Definition 2.5 (Theorem). A *theorem* in a logical system R is a *well-formed formula* y for which there exists a proof, and we write $\Vdash_R y$.

In other words, y is a theorem in R if $\exists d. d \Vdash_R y$.

Definition 2.6 (Set of theorems in R). We let $I_R = \{y \mid \Vdash_R y\}$ be the set of all theorems that can be proved in R.

We mention two main approaches to prove theorems:

- *top-down* or *direct*: we start from theorems descending from the axioms and then prove more and more theorems by applying the inference rules to already proved theorems;
- *bottom-up* or *goal-oriented*: we fix a goal, i.e., a theorem we want to prove, and we try to deduce a derivation for it by applying the inference rules backward, until each needed premise is also proved.

In the following we will mostly follow the *bottom-up* approach, because we will be given a specific goal to prove.

Example 2.5 (Grammars as sets of inference rules). Every grammar can be presented equivalently as a set of inference rules. Let us consider the well-known grammar for strings of balanced parentheses. Recalling that ε denotes the empty string, we write:

$$S \quad ::= \quad S\,S \quad | \quad (S) \quad | \quad \varepsilon$$

We let L_S denote the set of strings generated by the grammar for the symbol S. The translation from production to inference rules is straightforward. The first production

$$S \quad ::= \quad S\,S$$

says that given any two strings s_1 and s_2 of balanced parentheses, their juxtaposition is also a string of balanced parentheses. In other words:

$$\frac{s_1 \in L_S \quad s_2 \in L_S}{s_1 s_2 \in L_S}\ (1)$$

Similarly, the second production

$$S \quad ::= \quad (S)$$

says that we can surround with brackets any string s of balanced parentheses and get again a string of balanced parentheses. In other words:

$$\frac{s \in L_S}{\texttt{(}\,s\,\texttt{)} \in L_S}\ (2)$$

Finally, the last production says that the empty string ε is just a particular string of balanced parentheses. In other words we have an axiom:

$$\frac{}{\varepsilon \in L_S}\ (3)$$

Note the difference between the placeholders s, s_1, s_2 and the symbol ε appearing in the rules above: the former can be replaced by any string to obtain a specific instance of rules (1) and (2), while the latter denotes a given string (i.e., rules (1) and (2) define rule schemes with many instances, while there is a unique instance of rule (3)).

For example, the rule

$$\frac{\texttt{)(} \in L_S \quad \texttt{((} \in L_S}{\texttt{)(((} \in L_S}\ (1)$$

is an instance of rule (1): it is obtained by replacing s_1 with `)(` and s_2 with `((`. Of course the string `)(((` appearing in the conclusion does not belong to L_S, but the rule instance is perfectly valid, because it says that "$)\,(((\in L_S$ *if* $)\,(\in L_S$ and $((\in L_S$": since the premises are false, the implication is valid even if we cannot draw the conclusion $)\,(((\in L_S$.

Let us see an example of valid derivation that uses some valid instances of rules (1) and (2).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \in L_S}\,(3)}{(\varepsilon) = () \in L_S}\,(2)}{(()) \in L_S}\,(2) \qquad \dfrac{\dfrac{}{\varepsilon \in L_S}\,(3)}{(\varepsilon) = () \in L_S}\,(2)}{(())() \in L_S}\,(1)$$

Reading the proof (from the leaves to the root): Since $\varepsilon \in L_S$ by axiom (3), then we know that $(\varepsilon) = () \in L_S$ by (2); if we apply again rule (2) we derive also $(()) \in L_S$ and hence $(())() \in L_S$ by (1). In other words $(())() \in L_S$ is a theorem.

Let us introduce a second formalisation of the same language (balanced parentheses) without using inference rules. To get an intuition, suppose we want to write an algorithm to check if the parentheses in a string are balanced. We can parse the string from left to right and count the number of unmatched, open parentheses in the prefix we have parsed. So, we sum 1 to the counter whenever we find an open parenthesis and subtract 1 whenever we find a closed parenthesis. If the counter is never negative, and it holds 0 when we have parsed the whole string, then the parentheses in the string are balanced.

In the following we let a_i denote the ith symbol of the string a. Let

$$f(a_i) = \begin{cases} 1 & \text{if } a_i = (\\ -1 & \text{if } a_i =) \end{cases}$$

A string of n parentheses $a = a_1 a_2 \ldots a_n$ is balanced if and only if both the following properties hold:

Property 1: $\forall m \in [0,n]$ we have $\sum_{i=1}^{m} f(a_i) \geq 0$
Property 2: $\sum_{i=1}^{n} f(a_i) = 0$

In fact, $\sum_{i=1}^{m} f(a_i)$ counts the difference between the number of open parentheses and closed parentheses that are present in the first m symbols of the string a. Therefore, the first property requires that in any prefix of the string a the number of open parentheses exceeds, or equals the number of closed ones; the second property requires that the string a has as many open parentheses than closed ones.

An example is shown below for the string $a = (())()$:

$$\begin{array}{rcccccc} m = & 1 & 2 & 3 & 4 & 5 & 6 \\ a_m = & (& (&) &) & (&) \\ f(a_m) = & 1 & 1 & -1 & -1 & 1 & -1 \\ \sum_{i=1}^{m} f(a_i) = & 1 & 2 & 1 & 0 & 1 & 0 \end{array}$$

Properties 1 and 2 are easy to check for any string and therefore define an useful procedure to decide if a string belongs to our language or not.

Next, we show that the two different characterisations of the language (by inference rules and by the counting procedure) of balanced parentheses are equivalent.

Theorem 2.1. *For any string of parentheses a of length n*

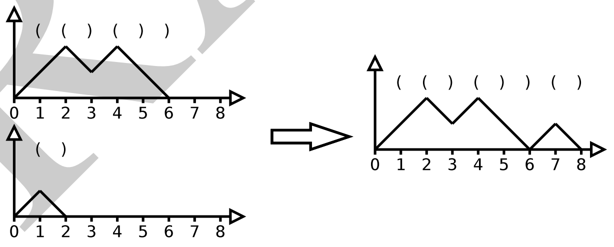
$$a \in L_S \iff \begin{cases} \sum_{i=1}^{m} f(a_i) \geq 0 & m = 0, 1 \ldots n \\ \sum_{i=1}^{n} f(a_i) = 0 \end{cases}$$

Proof. The proof is composed of two implications that we show separately:

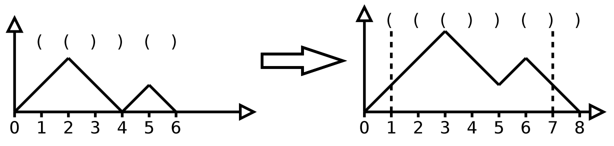
$\Rightarrow$) all the strings produced by the grammar satisfy the two properties;
$\Leftarrow$) any string that satisfy the two properties can be generated by the grammar.

Proof of $\Rightarrow$) To show the first implication, we proceed by *induction over the rules*: we assume that the implication is valid for the premises and we show that it holds for the conclusion. This proof technique is very powerful and will be explained in detail in Chapter 4.
The two properties can be represented graphically over the cartesian plane by taking m over the x-axis and the quantity $\sum_{i=1}^{m} f(a_i)$ over the y-axis. Intuitively, the graph start at the origin; it should never cross below the x-axis and it should end in $(n, 0)$.
Let us check that by applying any inference rule the properties 1 and 2 still hold.

Rule (1): The first inference rule corresponds to the juxtaposition of the two graphs and therefore the result still satisfies both properties (when the original graphs do).



Rule (2): The second rule corresponds to translate the graph upward (by 1 unit) and therefore the result still satisfies both properties (when the original graph does).

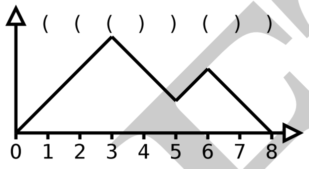


Rule (3): The third rule is just concerned with the empty string that trivially satisfies the two properties.

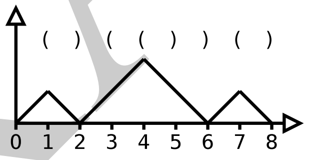
Since we have inspected all the inference rules, the proof of the first implication is concluded.

Proof of $\Leftarrow$) We need to find a derivation for any string that satisfies the two properties. Let a be such a generic string. (We only sketch this direction of the proof, that goes by induction over the length of the string a.) We proceed by case analysis, considering three cases:

1. If $n = 0$, $a = \varepsilon$. Then, by rule (3) we conclude that $a \in L_S$.
2. The second case is when the graph associated with a never touches the x-axis (except for its start and end points). An example is shown below:



   In this case we can apply rule (2), because we know that the parentheses opened at the beginning of a is only matched by the parenthesis at the very end of a.
3. The third and last case is when the graph touches the x-axis (at least) once in a point $(k,0)$ different from its start and its end. An example is shown below:



   In this case the substrings $a_1 \ldots a_k$ and $a_{k+1} \ldots a_n$ are also balanced and we can apply the rule (1) to their derivations to prove that $a \in L_S$. □

The last part of the proof outlines a goal-oriented strategy to build a derivation for a given string: We start by looking for a rule whose conclusion can match the goal we are after. If there are no alternatives, then we fail. If we have only one alternative we need to build a derivation for its premises. If there are more alternatives than one we can either explore all of them in parallel (*breadth-first* approach) or try one of them and back-track in case we fail (*depth-first*).

Suppose we want to find a proof for `(())()` $\in L_S$. We use the notation

$$\texttt{(())()} \in L_S \quad \nwarrow$$

1.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\varepsilon \in L_S,\quad \mathtt{(())()} \in L_S$
2.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{(} \in L_S,\quad \mathtt{())()} \in L_S$
3.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{((} \in L_S,\quad \mathtt{))()} \in L_S$
4.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{(()} \in L_S,\quad \mathtt{)()} \in L_S$
5.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{(())} \in L_S,\quad \mathtt{()} \in L_S$
6.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{(())(} \in L_S,\quad \mathtt{)} \in L_S$
7.	$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{(())()} \in L_S,\quad \varepsilon \in L_S$

Fig. 2.1: Tentative derivations for the goal $\mathtt{(())()} \in L_S$

to mean that we look for a goal-oriented derivation.

- Rule (1) can be applied in many different ways, by splitting the string $\mathtt{(())()}$ in all possible ways. We use the notation

$$\mathtt{(())()} \in L_S \quad \nwarrow \quad \varepsilon \in L_S,\ \mathtt{(())()} \in L_S$$

  to mean that we reduce the proof of $\mathtt{(())()} \in L_S$ to those of $\varepsilon \in L_S$ and $\mathtt{(())()} \in L_S$. Then we have all the alternatives in Figure 2.1 to inspect. Note that some alternatives are identical except for the order in which they list subgoals (1 and 7) and may require to prove the same goal from which we started (1 and 7). For example, if option 1 is selected applying depth-first strategy without any additional check, the derivation procedure might diverge. Moreover, some alternatives lead to goals we won't be able to prove (2, 3, 4, 6).
- Rule (2) can be applied in only one way:

$$\mathtt{(())()} \in L_S \quad \nwarrow \quad \mathtt{())(} \in L_S$$

- Rule (3) cannot be applied.

We show below a successful derivation, where the empty goal is written $\square$.

$\mathtt{(())()} \in L_S$	$\nwarrow$	$\mathtt{(())} \in L_S,\quad \mathtt{()} \in L_S$	by applying (1)
	$\nwarrow$	$\mathtt{(())} \in L_S,\quad \varepsilon \in L_S$	by applying (2) to the second goal
	$\nwarrow$	$\mathtt{(())} \in L_S$	by applying (3) to the second goal
	$\nwarrow$	$\mathtt{()} \in L_S$	by applying (2)
	$\nwarrow$	$\varepsilon \in L_S$	by applying (2)
	$\nwarrow$	$\square$	by applying (3)

We remark that in general the problem to check if a certain formula is a theorem is only *semidecidable* (not necessarily *decidable*). In this case the breadth-first strategy for goal-oriented derivation offers a semidecision procedure: if a derivation exists, then it will be found; if no derivation exists, the strategy may not terminate.

2.3 Logic Programming

We end this chapter by mentioning a particularly relevant paradigm based on goal-oriented derivation: *logic programming* and its Prolog incarnation. Prolog exploits depth-first goal-oriented derivations with backtracking.

Let $X = \{x, y, \dots\}$ be a set of variables, $\Sigma = \{ f, g, \dots \}$ a signature of function symbols (with given arities), $\Pi = \{ \mathsf{p}, \mathsf{q}, \dots \}$ a signature of predicate symbols (with given arities). As usual, we denote by Σ_n (respectively Π_n) the subset of function symbols (respectively predicate symbols) with arity n.

Definition 2.7 (Atomic formula). An *atomic formula* consists of a predicate symbol p of arity n applied to n terms with variables.

For example, if $\mathsf{p} \in \Pi_2$, $f \in \Sigma_2$ and $g \in \Sigma_1$, then $\mathsf{p}(\, f(g(x), x)\, , g(y)\,)$ is an atomic formula.

Definition 2.8 (Formula). A *formula* is a (possibly empty) conjunction of atomic formulas.

Definition 2.9 (Horn clause). A *Horn clause* is written $l\colon -r$ where l is an atomic formula, called the *head* of the clause, and r is a formula called the *body* of the clause.

Definition 2.10 (Logic program). A logic program is a set of Horn clauses.

The variables appearing in each clause can be instantiated with any term. A goal g is a formula whose validity we want to prove. The goal g can contain variables, which are implicitly existentially quantified.

Unification is used to "match" the head of a clause to an atomic formula of the goal we want to prove in the most general way (i.e., by instantiating the variables as little as possible). Before performing unification, the variables of the clause are renamed with fresh identifiers to avoid any clash with the variables already present in the goal.

Suppose we are given a logic program L and a goal $g = a_1, \dots, a_n$, where $a_1, \dots, a_n$ are atomic formulas. A derivation step $g \nwarrow_{\sigma'} g'$ is obtained by selecting a sub-goal a_i, a clause $l\colon -r \in L$ and a renaming ρ such that:

- $l\rho\colon -r\rho$ is a variant of the clause $l\colon -r \in L$ whose variables are fresh;
- the unification problem $\{ a_i \stackrel{?}{=} l\rho \}$ a most general solution σ;
- $\sigma' \stackrel{\text{def}}{=} \sigma_{|vars(a_i)}$;
- $g' \stackrel{\text{def}}{=} a_1, \dots, a_{i-1}, r\rho\sigma, a_{i+1}, \dots, a_n$.

If we can find a sequence of derivation steps

$$g \nwarrow_{\sigma_1} g_1 \nwarrow_{\sigma_2} g_2 \cdots g_{n-1} \nwarrow_{\sigma_n} \square$$

then we can conclude that the goal g is satisfiable and that the substitution $\sigma \stackrel{\text{def}}{=} \sigma_1 \cdots \sigma_n$ is a least substitutions for the variables in g such that $g\sigma$ is a valid theorem.

Example 2.6 (Sum in Prolog). Let us consider the logic program:

$$\begin{aligned} \mathsf{sum}(\,0\,,y\,,y\,) &:- . \\ \mathsf{sum}(\,s(x)\,,y\,,s(z)\,) &:-\ \mathsf{sum}(\,x\,,y\,,z\,). \end{aligned}$$

where $\mathsf{sum} \in \Pi_3$, $s \in \Sigma_1$, $0 \in \Sigma_0$ and $x, y, z \in X$.

Let us consider the goal $\mathsf{sum}(\,s(s(0))\,,s(s(0))\,,v\,)$ with $v \in X$.

There is no match against the head of the first clause, because 0 does not unify with $s(s(0))$.

We rename x, y, z in the second clause to x', y', z' and compute the unification of $\mathsf{sum}(\,s(s(0))\,,s(s(0))\,,v\,)$ and $\mathsf{sum}(\,s(x')\,,y'\,,s(z')\,)$. The result is the substitution (i.e., the most general unifier)

$$[x' = s(0), \quad y' = s(s(0)), \quad v = s(z')]$$

We then apply the substitution to the body of the clause, which will be added to the goal:

$$\mathsf{sum}(\,x'\,,y'\,,z'\,)[x' = s(0), y' = s(s(0)), v = s(z')] = \mathsf{sum}(\,s(0)\,,s(s(0))\,,z'\,)$$

If other subgoals were initially present, which may share variables with $\mathsf{sum}(\,s(s(0))\,,s(s(0))\,,v\,)$ then the substitution should have been applied to them too.

We write the derivation described above using the notation

$$\mathsf{sum}(\,s(s(0))\,,s(s(0))\,,v\,) \quad \nwarrow_{v=s(z')} \quad \mathsf{sum}(\,s(0)\,,s(s(0))\,,z'\,)$$

where we have recorded (as a subscript of the derivation step) the substitution applied to the variables originally present in the goal (just v in the example), to record the least condition under which the derivation is possible.

The derivation can then be completed as follows:

$$\begin{array}{lcl} \mathsf{sum}(\,s(s(0))\,,s(s(0))\,,v\,) & \nwarrow_{v=s(z')} & \mathsf{sum}(\,s(0)\,,s(s(0))\,,z'\,) \\ & \nwarrow_{z'=s(z'')} & \mathsf{sum}(\,0\,,s(s(0))\,,z''\,) \\ & \nwarrow_{z''=s(s(0))} & \square \end{array}$$

By composing the computed substitutions we get

$$\begin{aligned} z' &= s(z'') = s(s(s(0))) \\ v &= s(z') = s(s(s(s(0)))) \end{aligned}$$

This gives us a proof of the theorem

$$\mathsf{sum}(\,s(s(0))\,,s(s(0))\,,s(s(s(s(0))))\,)$$

Problems

2.1. Consider the alphabet $\{\mathtt{a},\mathtt{b}\}$ and the grammar

$$\begin{array}{lcl} A & ::= & \mathtt{a}\,A \mid \mathtt{a}\,B \\ B & ::= & \mathtt{b} \mid \mathtt{b}\,B \end{array}$$

1. Describe the form of the strings in the languages L_A and L_B.
2. Define the languages L_A and L_B formally.
3. Write the inference rules that correspond to the productions of the grammar.
4. Write the derivation for the string `a a a b b` both as a proof-tree and as a goal-oriented derivation.
5. Prove that the set of theorems associated with the inference rules coincide with the formal definitions you gave.

2.2. Consider the alphabet $\{\mathtt{0},\mathtt{1}\}$.

1. Give a context free grammar for the set of strings that contain an even number of `0` and `1`.
2. Write the inference rules that correspond to the productions of the grammar.
3. Write the derivation for the string `0 1 1 0 0 0` both as a proof-tree and as a goal-oriented derivation.
4. Prove that your logical systems characterises exactly the set of strings that contain an even number of `0` and `1`.

2.3. Consider the signature Σ such that $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ and $\Sigma_n = \varnothing$ for any $n \geq 2$.

1. Let $even \in \Pi_1$. What are the theorems of the logical system below?

$$\frac{}{even(0)}\ (1) \qquad \frac{even(x)}{even(s(s(x)))}\ (2)$$

2. Let $odd \in \Pi_1$.What are the theorems of the logical system below?

$$\frac{odd(x)}{odd(s(s(x)))}\ (1)$$

3. Let $leq \in \Pi_2$. What are the theorems of the logical system below?

$$\frac{}{leq(0,x)}\ (1) \qquad \frac{leq(x,y)}{leq(s(x),s(y))}\ (2)$$

2.4. Consider the signature Σ such that $\Sigma_0 = \mathbb{N}$, $\Sigma_2 = \{node\}$ and $\Sigma_n = \varnothing$ otherwise. Let $sum, eq \in \Pi_2$. What are the theorems of the logical system below?

$$\frac{}{sum(n,n)}\ n \in \mathbb{N}\ (1) \qquad \frac{sum(x,n) \quad sum(y,m)}{sum(node(x,y),k)}\ k = n+m\ (2) \qquad \frac{sum(x,n) \quad sum(y,n)}{eq(x,y)}\ (3)$$

2.5. Consider the signature Σ such that $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ and $\Sigma_n = \varnothing$ for any $n \geq 2$. Give two terms t and t', with $t \neq t'$, such that t is more general than t' and t' is also more general than t.

2.6. Consider the signature Σ such that $\Sigma_0 = \{a\}$, $\Sigma_1 = \{f,g\}$, $\Sigma_2 = \{h,l\}$ and $\Sigma_n = \varnothing$ for any $n \geq 3$. Solve the unification problems below:

1. $G_0 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(y,z), g(y) \stackrel{?}{=} g(l(a,a))\}$
2. $G_1 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(x,g(z))\}$
3. $G_2 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(y,f(z)), l(y,a) \stackrel{?}{=} l(a,z)\}$
4. $G_3 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(y,x) \stackrel{?}{=} h(g(a),f(g(z))), l(z,a) \stackrel{?}{=} l(a,z)\}$

2.7. Extend the logic program for computing the sum with the definition of:

1. a predicate *prod* for computing the product of two numbers;
2. a predicate *pow* for computing the power of a base to an exponent;
3. a predicate *div* that tells if a number can be divided by another number.

2.8. Extend the logic program for computing the sum with the definition of a binary predicate $\mathsf{fib}(N,F)$ that is true if F is the Nth Fibonacci number.

2.9. Suppose that a set of facts of the form $\mathsf{parent}(\mathsf{x},\mathsf{y})$ are given, meaning that x is a parent of y.

1. Define a predicate $\mathsf{brother}(X,Y)$ which holds true iff X and Y have a parent in common.
2. Define a predicate $\mathsf{cousin}(X,Y)$ which holds true iff X and Y are cousins.
3. Define a predicate $\mathsf{ancestor}(X,Y)$ which holds true iff X is an ancestor of Y.
4. If the set of basic facts is:

```
:- parent(alice,bob) .
:- parent(alice,carl) .
:- parent(bob,diana) .
:- parent(bob,ella) .
:- parent(carl,francisco) .
```

   which of the following goals can be derived?

```
   ?- brother(ella,francisco).
   ?- brother(ella,diana).
   ?- cousin(ella,francisco).
   ?- cousin(ella,diana).
   ?- ancestor(alice,ella).
   ?- ancestor(carl,ella).
```

2.10. Suppose that a set of facts of the form $\mathsf{arc}(\mathsf{x},\mathsf{y})$ are given to represent a directed, acyclic graph, meaning that there is an arc from x to y.

1. Define a predicate $\mathsf{path}(X,Y)$ which holds true iff there is a path from X to Y.

2. Suppose the acyclic requirement is violated, like in the graph

```
:- arc(a,b) .
:- arc(a,c) .
:- arc(b,c) .
:- arc(b,d) .
:- arc(c,d) .
:- arc(d,e) .
:- arc(d,f) .
:- arc(e,c) .
```

Does a goal-oriented derivation for a query, like the one below, necessarily lead to the empty goal? Why?

```
    ?- path(a,f).
```

2.11. Consider the Horn clauses that correspond to the following statements:

1. All jumping creatures are green.
2. All small jumping creatures are martians.
3. All green martians are intelligent.
4. Ngtrks is small and green.
5. Pgvdrk is a jumping martian.

Who is intelligent?[1]

[1] Taken from http://www.slideshare.net/SergeiWinitzki/prolog-talk.