Theory of Computation

Marco Bernardo

University of Urbino – Italy
Department of Pure and Applied Sciences
Section of Informatics and Mathematics

PhD Program in Research Methods in Science and Technology

Syllabus:

1. *Introduction to Informatics* 3
2. *Introduction to Computation* 15
3. *The Operational View: Turing Machines* 33
4. *The Functional View: Lambda Calculus* 47
5. *Computability for Functions, Sets, Problems* 69
6. *The Modeling View: Process Algebras* 91

1. Introduction to Informatics

Topics:

1.1 *Informatics: Science, Methodology, Technology* 4

1.2 *Socio-Economic Impact and Computational Thinking* 6

1.3 *A Brief History of Informatics* 9

1.4 *Why Theory and Methodology: Software Disasters* 14

1.1 Informatics: Science, Methodology, Technology

- What do we mean by informatics?

1.1 Informatics: Science, Methodology, Technology

- What do we mean by informatics?
- Informatics: discipline that studies automatic information processing.
- People perceive only technological aspects, while informaticians have to be aware of methodological and scientific aspects too.

1.1 Informatics: Science, Methodology, Technology

- What do we mean by informatics?
- Informatics: discipline that studies automatic information processing.
- People perceive only technological aspects, while informaticians have to be aware of methodological and scientific aspects too.
- Technological aspects – *Information & Communication Technology*: computers, operating systems, data bases, computer networks, ...

1.1 Informatics: Science, Methodology, Technology

- What do we mean by informatics?
- Informatics: discipline that studies automatic information processing.
- People perceive only technological aspects, while informaticians have to be aware of methodological and scientific aspects too.
- Technological aspects – *Information & Communication Technology*: computers, operating systems, data bases, computer networks, ...
- Methodological aspects – *Software Architecture & Engineering*: programming methodologies, languages, and environments, ...

1.1 Informatics: Science, Methodology, Technology

- What do we mean by informatics?
- Informatics: discipline that studies automatic information processing.
- People perceive only technological aspects, while informaticians have to be aware of methodological and scientific aspects too.
- Technological aspects – *Information & Communication Technology*: computers, operating systems, data bases, computer networks, ...
- Methodological aspects – *Software Architecture & Engineering*: programming methodologies, languages, and environments, ...
- Scientific aspects – *Computer Science*: theory of computation, algorithmics, automata theory, formal languages, ...

- What do we mean by computer?

- What do we mean by computer?
- Computer: set of programmable electromechanical components for inputting, storing, processing, and outputting information in the form of numbers, texts, images, audios, and videos.
- There are many sets of electromechanical components that can execute *several* functions (lifts, household appliances, vehicles, . . .) but only computers can execute a potentially *unlimited* number of functions, i.e., only computers are programmable machines!

- What do we mean by computer?
- Computer: set of programmable electromechanical components for inputting, storing, processing, and outputting information in the form of numbers, texts, images, audios, and videos.
- There are many sets of electromechanical components that can execute *several* functions (lifts, household appliances, vehicles, . . .) but only computers can execute a potentially *unlimited* number of functions, i.e., only computers are programmable machines!
- Hardware: set of electromechanical components that form a computer, i.e., computational resources that are made available (*physical parts*).
- Software: set of programs that can run on a computer, i.e., instructions given to its computational resources to carry out tasks (*immaterial parts* – name originated from Jacquard looms).

1.2 Socio-Economic Impact and Computational Thinking

- Socio-economic impact of informatics since 1950's:
  - Execute complicated calculations in a short time.
  - Process/retrieve/transmit large amounts of data in a short time.
  - Transfer repetitive or complex activities from people to machines.

1.2 Socio-Economic Impact and Computational Thinking

- Socio-economic impact of informatics since 1950's:
  - Execute complicated calculations in a short time.
  - Process/retrieve/transmit large amounts of data in a short time.
  - Transfer repetitive or complex activities from people to machines.
- The invention of printing in 1400 started to foster in the world a higher diffusion of knowledge.

1.2 Socio-Economic Impact and Computational Thinking

- Socio-economic impact of informatics since 1950's:
  - Execute complicated calculations in a short time.
  - Process/retrieve/transmit large amounts of data in a short time.
  - Transfer repetitive or complex activities from people to machines.
- The invention of printing in 1400 started to foster in the world a higher diffusion of knowledge.
- The industrial revolution in 1700 widened our physical capabilities through the introduction of automatic machines.

1.2 Socio-Economic Impact and Computational Thinking

- Socio-economic impact of informatics since 1950's:
  - Execute complicated calculations in a short time.
  - Process/retrieve/transmit large amounts of data in a short time.
  - Transfer repetitive or complex activities from people to machines.
- The invention of printing in 1400 started to foster in the world a higher diffusion of knowledge.
- The industrial revolution in 1700 widened our physical capabilities through the introduction of automatic machines.
- The digital transformation based on electronics and informatics is continuously and increasingly extending our cognitive capabilities through programmable devices like computers and smartphones and the propagation of digital data over Internet.

- Another socio-economic aspect is given by nonprofit communities of people developing and maintaining open-source software.
- Being open source (i.e., inspectable) is a technical prerequisite for being a free software.

- Another socio-economic aspect is given by nonprofit communities of people developing and maintaining open-source software.
- Being open source (i.e., inspectable) is a technical prerequisite for being a free software.
- Free Software Foundation founded by Richard Stallman (1985):
  - Freedom of executing a program for any purpose.
  - Freedom of studying a program and modifying it.
  - Freedom of distributing copies of a program to whoever needs it.
  - Freedom of improving a program and publicly distributing the improvements in such a way that everybody can benefit.
- Examples: GNU/Linux, Firefox, WordPress, LibreOffice, Moodle, ...

- Computational thinking is the cultural contribution of informatics, intended as a mental process aiming at problem solving.
- Expression coined by Jeannette Wing (2006).

- Computational thinking is the cultural contribution of informatics, intended as a mental process aiming at problem solving.
- Expression coined by Jeannette Wing (2006).
- Combination of:
  - Characteristic methods of informatics:
    - Algorithmic analysis of problems.
    - Digital representation of data.
    - Automation of solutions.
  - General intellectual capabilities:
    - Facing complexity.
    - Comparing alternatives.
- Computational variant of abstract thinking typical of mathematics: *software is immaterial!*

1.3 A Brief History of Informatics

1.3 A Brief History of Informatics

- Pioneers between 1600 and 1800 (mechanical technology):
  - Blaise Pascal (1623–1662):
    - Mechanical machine capable to do additions and subtractions (1642).

1.3 A Brief History of Informatics

- Pioneers between 1600 and 1800 (mechanical technology):
  - Blaise Pascal (1623–1662):
    - Mechanical machine capable to do additions and subtractions (1642).
  - Gottfried Wilhelm von Leibniz (1646–1716):
    - Mechanical machine including multiplications and divisions too (1672).
    - Binary system, characteristica universalis, calculus ratiocinator.

1.3 A Brief History of Informatics

- Pioneers between 1600 and 1800 (mechanical technology):
  - Blaise Pascal (1623–1662):
    - Mechanical machine capable to do additions and subtractions (1642).
  - Gottfried Wilhelm von Leibniz (1646–1716):
    - Mechanical machine including multiplications and divisions too (1672).
    - Binary system, characteristica universalis, calculus ratiocinator.
  - Charles Babbage (1791–1871):
    - Difference Engine (1822): tabulation of polynomial functions and hence approximation of logarithmic and trigonometric functions.
    - Analytical Engine (1834): archetype of modern computers both for its architecture and for its instruction set.

1.3 A Brief History of Informatics

- Pioneers between 1600 and 1800 (mechanical technology):
  - Blaise Pascal (1623–1662):
    - Mechanical machine capable to do additions and subtractions (1642).
  - Gottfried Wilhelm von Leibniz (1646–1716):
    - Mechanical machine including multiplications and divisions too (1672).
    - Binary system, characteristica universalis, calculus ratiocinator.
  - Charles Babbage (1791–1871):
    - Difference Engine (1822): tabulation of polynomial functions and hence approximation of logarithmic and trigonometric functions.
    - Analytical Engine (1834): archetype of modern computers both for its architecture and for its instruction set.
  - Ada Byron Lovelace (1815–1852):
    - Translation of and notes on the Analytical Engine project (1843).
    - Bernoulli numbers: *she was the first programmer of the history!*

- Some pioneers of 1900 (electronic technology):
  - Norbert Wiener (1894–1964):
    - Cybernetics: interdisciplinary study of natural and artificial systems in terms of control, feedback, and communication.

- Some pioneers of 1900 (electronic technology):
  - Norbert Wiener (1894–1964):
    - Cybernetics: interdisciplinary study of natural and artificial systems in terms of control, feedback, and communication.
  - Claude Shannon (1916–2001):
    - Connection between electrical circuits and George Boole algebra (1847).
    - Information theory: study of the encoding and the transmission of digital information, whose elementary unit he called bit.

- Some pioneers of 1900 (electronic technology):
  - Norbert Wiener (1894–1964):
    - Cybernetics: interdisciplinary study of natural and artificial systems in terms of control, feedback, and communication.
  - Claude Shannon (1916–2001):
    - Connection between electrical circuits and George Boole algebra (1847).
    - Information theory: study of the encoding and the transmission of digital information, whose elementary unit he called bit.
  - Alan Turing (1912–1954):
    - Turing machine (TM): first formalization of the concept of algorithm.
    - Universal Turing machine (UTM): vision of software in terms of computation scheme no longer hardwired.
    - Cryptography and artificial intelligence.

- Some pioneers of 1900 (electronic technology):
  - Norbert Wiener (1894–1964):
    - Cybernetics: interdisciplinary study of natural and artificial systems in terms of control, feedback, and communication.
  - Claude Shannon (1916–2001):
    - Connection between electrical circuits and George Boole algebra (1847).
    - Information theory: study of the encoding and the transmission of digital information, whose elementary unit he called bit.
  - Alan Turing (1912–1954):
    - Turing machine (TM): first formalization of the concept of algorithm.
    - Universal Turing machine (UTM): vision of software in terms of computation scheme no longer hardwired.
    - Cryptography and artificial intelligence.
  - John Von Neumann (1903–1957):
    - From plugboard computers to stored program computers (UTMs).
    - From serial decimal arithmetic to parallel binary arithmetic.
    - Game theory and quantum physics.

- Italian pioneers:
  - Adriano Olivetti (1901–1960):
    - Elea 9003 (1957): *first computer entirely based on transistors*, designed at Olivetti by Mario Tchou (1924–1961).
    - P101 (1964): *first personal computer of the history*, designed at Olivetti by Pier Giorgio Perotto (1930–2002).

- Italian pioneers:
  - Adriano Olivetti (1901–1960):
    - Elea 9003 (1957): *first computer entirely based on transistors*, designed at Olivetti by Mario Tchou (1924–1961).
    - P101 (1964): *first personal computer of the history*, designed at Olivetti by Pier Giorgio Perotto (1930–2002).
  - CEP – Calcolatrice Elettronica Pisana (1961):
    - First scientific computer designed in Italy, at the University of Pisa, an initiative promoted by Enrico Fermi and Adriano Olivetti.

- Italian pioneers:
  - Adriano Olivetti (1901–1960):
    - Elea 9003 (1957): *first computer entirely based on transistors*, designed at Olivetti by Mario Tchou (1924–1961).
    - P101 (1964): *first personal computer of the history*, designed at Olivetti by Pier Giorgio Perotto (1930–2002).
  - CEP – Calcolatrice Elettronica Pisana (1961):
    - First scientific computer designed in Italy, at the University of Pisa, an initiative promoted by Enrico Fermi and Adriano Olivetti.
  - Federico Faggin (1941):
    - Designed the first microprocessor of the history at Intel (1971).
    - Developed the first touchpads and touchscreens (late 1980's).

- Increasing computation speeds and storage capacities as well as decreasing costs and sizes of hw (ENIAC vs. modern computers):
  - Vacuum tubes $\rightarrow$ transistors $\rightarrow$ integrated circuits $\rightarrow$ VLSI $\rightarrow$ ...

- Increasing computation speeds and storage capacities as well as decreasing costs and sizes of hw (ENIAC vs. modern computers):
  - Vacuum tubes $\rightarrow$ transistors $\rightarrow$ integrated circuits $\rightarrow$ VLSI $\rightarrow$ ...
- Operating systems, data bases, computer networks (1969: Internet).
- Growing complexity of computing systems:
  - *Sequential* $\rightarrow$ *concurrent* $\rightarrow$ *distributed* $\rightarrow$ *decentralized* (DLT) $\rightarrow$ ...

- Increasing computation speeds and storage capacities as well as decreasing costs and sizes of hw (ENIAC vs. modern computers):
  - Vacuum tubes $\rightarrow$ transistors $\rightarrow$ integrated circuits $\rightarrow$ VLSI $\rightarrow$ ...
- Operating systems, data bases, computer networks (1969: Internet).
- Growing complexity of computing systems:
  - *Sequential* $\rightarrow$ *concurrent* $\rightarrow$ *distributed* $\rightarrow$ *decentralized* (DLT) $\rightarrow$ ...
- Mobile computing: devices not tied to physical locations (IoT).
- Global computing: abstraction of a single global computer accessible anywhere anytime (cloud, edge, fog).
- Autonomic computing: self-managing characteristics for adapting to unpredictable changes (sensors, actuators, policies, AI).

- Programming languages:
  - *Procedural imperative*: Fortran, Cobol, Algol, Basic, Pascal, C, ...
  - *Object-oriented imperative*: Simula, Smalltalk, C++, Java, C#, ...
  - *Declarative*: Lisp, Scheme, ML, Haskell, Prolog, ...
  - *Concurrent*: Modula, Ada, Occam, Erlang, Scala, ...

- Programming languages:
  - *Procedural imperative*: Fortran, Cobol, Algol, Basic, Pascal, C, ...
  - *Object-oriented imperative*: Simula, Smalltalk, C++, Java, C#, ...
  - *Declarative*: Lisp, Scheme, ML, Haskell, Prolog, ...
  - *Concurrent*: Modula, Ada, Occam, Erlang, Scala, ...
- Other languages:
  - *Script*: shell, Perl/Raku, Tcl/Tk, Python, PHP, JavaScript, Ruby, ...
  - *Query*: SQL, ...
  - *Markup*: HTML, XML, LaTeX, ...
  - *Modeling*: UML, AADL, ...

- Programming languages:
  - *Procedural imperative*: Fortran, Cobol, Algol, Basic, Pascal, C, ...
  - *Object-oriented imperative*: Simula, Smalltalk, C++, Java, C#, ...
  - *Declarative*: Lisp, Scheme, ML, Haskell, Prolog, ...
  - *Concurrent*: Modula, Ada, Occam, Erlang, Scala, ...
- Other languages:
  - *Script*: shell, Perl/Raku, Tcl/Tk, Python, PHP, JavaScript, Ruby, ...
  - *Query*: SQL, ...
  - *Markup*: HTML, XML, LaTeX, ...
  - *Modeling*: UML, AADL, ...
- Languages for reversible computing (lower power consumption).
- Languages for quantum computing (lower running time).

1.4 Why Theory and Methodology: Software Disasters

- Information and communication technology is *pervasive*.
- Malfunctioning, poor performance, security breaches, bad interfaces.
- These errors result in waste of time and money for software producers.
- Human losses and environmental damages if software is safety-critical!
- `https://en.wikipedia.org/wiki/List_of_software_bugs`

1.4 Why Theory and Methodology: Software Disasters

- Information and communication technology is *pervasive*.
- Malfunctioning, poor performance, security breaches, bad interfaces.
- These errors result in waste of time and money for software producers.
- Human losses and environmental damages if software is safety-critical!
- `https://en.wikipedia.org/wiki/List_of_software_bugs`
- Methodologies for guiding software design, development, deployment.
- Software testing does not guarantee the absence of errors.
- Software verification is needed:
  - Program annotation (Floyd-Hoare logic, separation logic) and deductive verification (preconditions-postconditions, invariants).
  - Program model, property formalization (modal logic, temporal logic), and model ckecking (including counterexample generation).

2. Introduction to Computation

Topics:

2.1 *Computational Problems and Algorithmic Solutions* 16

2.2 *A Classical Model of Computation* 19

2.3 *From Logicism to Incompleteness and Undecidability* 21

2.4 *Induction, Enumerations, Encodings* 26

2.1 Computational Problems and Algorithmic Solutions

- Given a computational problem, the theory of computation studies the existence of an algorithmic solution and, if any, the quantity of needed computational resources in terms of:
  - Running time.
  - Memory space.
  - Communication bandwidth.

2.1 Computational Problems and Algorithmic Solutions

- Given a computational problem, the theory of computation studies the existence of an algorithmic solution and, if any, the quantity of needed computational resources in terms of:
  - Running time.
  - Memory space.
  - Communication bandwidth.
- Computability theory: decidable vs. undecidable problems depending on the existence of an algorithmic solution or not.
- Complexity theory: tractable vs. intractable problems depending on polynomially or exponentially many resources.
- If a computational problem is decidable, there might be several different algorithms solving it *more or less efficiently* (insertsort, selectsort, bubblesort / quicksort, mergesort, heapsort).

- What do we mean by algorithm?

- What do we mean by algorithm?
- Algorithm: finite sequence of steps, represented in a way that is intelligible by an executor, that solve a computational problem in its generality (i.e., for all instances of its input data).
- Persian mathematician Muhammad ibn Musa al-Khwarizmi (780–850) is the author of one of the most ancient treatises of algebra (al-jabr) in which he described how to solve linear and quadratic equations.

- What do we mean by algorithm?
- Algorithm: finite sequence of steps, represented in a way that is intelligible by an executor, that solve a computational problem in its generality (i.e., for all instances of its input data).
- Persian mathematician Muhammad ibn Musa al-Khwarizmi (780–850) is the author of one of the most ancient treatises of algebra (al-jabr) in which he described how to solve linear and quadratic equations.
- Although the representation of an algorithm has a finite length, the duration of its execution can be unbounded (e.g., loops).
- The executor is not necessarily an electromechanical agent, can be a biological agent or a cyberphysical agent (e.g., recipes).
- Program: algorithm expressed in a specific programming language, whose executor will be a computer (software, application, code, ...).

- Computational problems are usually expressed in a combination of natural language and mathematical notation.
- *Decision problem*: computational problem with a yes/no answer.
- How to formalize algorithms?

- Computational problems are usually expressed in a combination of natural language and mathematical notation.
- *Decision problem*: computational problem with a yes/no answer.
- How to formalize algorithms?
- Syntax domain: every algorithm is a finite sequence of symbols taken from a finite alphabet (lexical and grammar rules).

- Computational problems are usually expressed in a combination of natural language and mathematical notation.
- *Decision problem*: computational problem with a yes/no answer.
- How to formalize algorithms?
- Syntax domain: every algorithm is a finite sequence of symbols taken from a finite alphabet (lexical and grammar rules).
- Semantic domain: set of mathematical objects through which the meaning of algorithms (i.e., what they compute) is expressed.

- Computational problems are usually expressed in a combination of natural language and mathematical notation.
- *Decision problem*: computational problem with a yes/no answer.
- How to formalize algorithms?
- Syntax domain: every algorithm is a finite sequence of symbols taken from a finite alphabet (lexical and grammar rules).
- Semantic domain: set of mathematical objects through which the meaning of algorithms (i.e., what they compute) is expressed.
- Interpretation function associating every algorithm with its meaning.

- Computational problems are usually expressed in a combination of natural language and mathematical notation.
- *Decision problem*: computational problem with a yes/no answer.
- How to formalize algorithms?
- Syntax domain: every algorithm is a finite sequence of symbols taken from a finite alphabet (lexical and grammar rules).
- Semantic domain: set of mathematical objects through which the meaning of algorithms (i.e., what they compute) is expressed.
- Interpretation function associating every algorithm with its meaning.
- The executor knows the language in which algorithms are written and is equipped with a memory to store input data and intermediate data.

2.2 A Classical Model of Computation

- How to formalize algorithm semantics?

2.2 A Classical Model of Computation

- How to formalize algorithm semantics?
- Discrete computation: at any time the executor is in one state from a *finite* set, state transitions occur at *different* instants, and data are represented *digitally*.

2.2 A Classical Model of Computation

- How to formalize algorithm semantics?
- Discrete computation: at any time the executor is in one state from a *finite* set, state transitions occur at *different* instants, and data are represented *digitally*.
- Deterministic computation: from the current state there is at most *one* possible next state (given by the current state plus input data).

2.2 A Classical Model of Computation

- How to formalize algorithm semantics?
- Discrete computation: at any time the executor is in one state from a *finite* set, state transitions occur at *different* instants, and data are represented *digitally*.
- Deterministic computation: from the current state there is at most *one* possible next state (given by the current state plus input data).
- Sequential computation: a *single* step at a time is executed.

2.2 A Classical Model of Computation

- How to formalize algorithm semantics?
- Discrete computation: at any time the executor is in one state from a *finite* set, state transitions occur at *different* instants, and data are represented *digitally*.
- Deterministic computation: from the current state there is at most *one* possible next state (given by the current state plus input data).
- Sequential computation: a *single* step at a time is executed.
- Alternative options: analog/quantum, nondeterministic/probabilistic, concurrent (shared memory/message passing/distributed ledger).

2.2 A Classical Model of Computation

- How to formalize algorithm semantics?
- Discrete computation: at any time the executor is in one state from a *finite* set, state transitions occur at *different* instants, and data are represented *digitally*.
- Deterministic computation: from the current state there is at most *one* possible next state (given by the current state plus input data).
- Sequential computation: a *single* step at a time is executed.
- Alternative options: analog/quantum, nondeterministic/probabilistic, concurrent (shared memory/message passing/distributed ledger).
- Each object in the semantic domain can be formalized as $f : \mathbb{N} \to \mathbb{N}$. In $f(a) = b$, number a represents input data, number b output data.
- Input and output *data can be encoded as natural numbers* because any datum is a *finite* sequence of symbols belonging to a *finite* set.

- How many algorithms are there?

- How many algorithms are there?
- The cardinality of the syntax domain is equal to the cardinality of $\mathbb{N}$.
- The algorithms in the syntax domain can be enumerated according to the lexicographic order or the shortlex order:
  - Choose a total order for the finitely many symbols of the alphabet.
  - Enumerate algorithms of the same length based on their symbols.
  - Enumerate algorithms of length k before algorithms of length $k + 1$.

- How many algorithms are there?
- The cardinality of the syntax domain is equal to the cardinality of $\mathbb{N}$.
- The algorithms in the syntax domain can be enumerated according to the lexicographic order or the shortlex order:
  - Choose a total order for the finitely many symbols of the alphabet.
  - Enumerate algorithms of the same length based on their symbols.
  - Enumerate algorithms of length k before algorithms of length $k + 1$.
- And how many functions over $\mathbb{N}$ are there?

- How many algorithms are there?
- The cardinality of the syntax domain is equal to the cardinality of $\mathbb{N}$.
- The algorithms in the syntax domain can be enumerated according to the lexicographic order or the shortlex order:
  - Choose a total order for the finitely many symbols of the alphabet.
  - Enumerate algorithms of the same length based on their symbols.
  - Enumerate algorithms of length k before algorithms of length $k+1$.
- And how many functions over $\mathbb{N}$ are there?
- The cardinality of $\{f \mid f : \mathbb{N} \to \mathbb{N}\}$ is greater than the one of $\mathbb{N}$!
- There are infinitely many functions not computable by any algorithm.
- Correspond to undecidable problems, i.e., computational problems for which there is no algorithm solving them.
- Existence of theoretical limits to what can be computed (1850-1950).

2.3 From Logicism to Incompleteness and Undecidability

- In 1800 logic needed mathematics to overcome the ambiguity of natural language in the same way as mathematics needed rigorous foundations and deductive mechanisms for proofs.

2.3 From Logicism to Incompleteness and Undecidability

- In 1800 logic needed mathematics to overcome the ambiguity of natural language in the same way as mathematics needed rigorous foundations and deductive mechanisms for proofs.
- George Boole introduced an algebraic view of logic (1847).

2.3 From Logicism to Incompleteness and Undecidability

- In 1800 logic needed mathematics to overcome the ambiguity of natural language in the same way as mathematics needed rigorous foundations and deductive mechanisms for proofs.
- George Boole introduced an algebraic view of logic (1847).
- Gottlob Frege thought that arithmetic were reducible to logic and proposed the use of logic to make proofs of theorems rigorous (1879):
  - Isolate intuitive or primitive elements as axioms.
  - Proceed from these axioms via inference rules *without gaps*.

2.3 From Logicism to Incompleteness and Undecidability

- In 1800 logic needed mathematics to overcome the ambiguity of natural language in the same way as mathematics needed rigorous foundations and deductive mechanisms for proofs.
- George Boole introduced an algebraic view of logic (1847).
- Gottlob Frege thought that arithmetic were reducible to logic and proposed the use of logic to make proofs of theorems rigorous (1879):
  - Isolate intuitive or primitive elements as axioms.
  - Proceed from these axioms via inference rules *without gaps*.
- This was the starting point of logicism:
  - All mathematical truths can be translated into logical truths, i.e., the vocabulary of mathematics is included in the vocabulary of logic.
  - All proofs of mathematical theorems can be recast as logical proofs *based on axioms and inference rules*, i.e., the set of theorems of mathematics is included in the set of theorems of logic.

- Georg Cantor founded set theory, showed the existence of a hierarchy of infinities, and proved that the real numbers are more numerous than the natural numbers through the diagonal method (1874).

- Georg Cantor founded set theory, showed the existence of a hierarchy of infinities, and proved that the real numbers are more numerous than the natural numbers through the diagonal method (1874).
- Richard Dedekind axiomatized the set of natural numbers (1888) in addition to formalizing the set of real numbers as well as a general notion of infinite set (equinumerosity to a proper subset).

- Georg Cantor founded set theory, showed the existence of a hierarchy of infinities, and proved that the real numbers are more numerous than the natural numbers through the diagonal method (1874).
- Richard Dedekind axiomatized the set of natural numbers (1888) in addition to formalizing the set of real numbers as well as a general notion of infinite set (equinumerosity to a proper subset).
- Giuseppe Peano simplified the natural number axiomatization (1889) yielding the currently used one (explicit induction principle, PA).

- Georg Cantor founded set theory, showed the existence of a hierarchy of infinities, and proved that the real numbers are more numerous than the natural numbers through the diagonal method (1874).
- Richard Dedekind axiomatized the set of natural numbers (1888) in addition to formalizing the set of real numbers as well as a general notion of infinite set (equinumerosity to a proper subset).
- Giuseppe Peano simplified the natural number axiomatization (1889) yielding the currently used one (explicit induction principle, PA).
- Bertrand Russell and Alfred Whitehead formally proved in their book "Principia Mathematica" the bulk of the mathematical knowledge of their time by using sheer symbolic manipulations (1913):
  - Only real numbers and set theory, including cardinals and ordinals.
  - Type theory to avoid set theory paradoxes ($A \triangleq \{B \mid B \notin B\} \in A$?).
  - A large amount of mathematical analysis could in principle be handled.
  - A further volume on the foundations of geometry had been planned.

- Logicism was becoming popular within (philosophy of) mathematics.
- David Hilbert launched a program to find a single formal procedure for deriving all theorems of mathematics, which amounts to manage to reduce all of mathematics in axiomatic form and to prove that this axiomatization does not lead to contradictions (1920).

- Logicism was becoming popular within (philosophy of) mathematics.
- David Hilbert launched a program to find a single formal procedure for deriving all theorems of mathematics, which amounts to manage to reduce all of mathematics in axiomatic form and to prove that this axiomatization does not lead to contradictions (1920).
- Hilbert claimed that "once a logical formalism is established, one can expect that a systematic, so-to-say *computational*, treatment of logic formulas is possible, which would somewhat correspond to the theory of equations in algebra".
- According to Hilbert, the complex theories of mathematics could be founded on simpler theories till the point in which all of mathematics is based on arithmetic, so that by proving the consistency of the latter one would have proven the consistency of the former.

- Kurt Gödel came up with his two incompleteness theorems (1931):
  1. In every axiomatizable theory expressive enough to form statements about what it can prove (reminiscent of autologies like "$A \in A$?"), there will always be true statements that the theory can express but cannot derive from its axioms through its inference rules.
  2. Every axiomatizable theory expressive enough to form statements about arithmetic will never be able to prove its own consistency.

- Kurt Gödel came up with his two incompleteness theorems (1931):
  1. In every axiomatizable theory expressive enough to form statements about what it can prove (reminiscent of autologies like "$A \in A$?"), there will always be true statements that the theory can express but cannot derive from its axioms through its inference rules.
  2. Every axiomatizable theory expressive enough to form statements about arithmetic will never be able to prove its own consistency.
- The first theorem means that in every sufficiently powerful theory there are theorems that cannot be proven within the theory itself. To prove them, it is necessary to resort to a more powerful theory, but this will suffer from the same limitation too.
- The second theorem points out that even a fundamental branch of mathematics like arithmetic raises incompleteness issues from the viewpoint of its computational treatment.

- Alonzo Church proved that predicate logic is not decidable: there is no algorithm able to establish in a finite amount of time whether an arbitrary predicate formula is satisfiable or not (1936).

- Alonzo Church proved that predicate logic is not decidable: there is no algorithm able to establish in a finite amount of time whether an arbitrary predicate formula is satisfiable or not (1936).
- Alan Turing proved that the halting problem is not decidable: there is no algorithm able to establish in a finite amount of time whether the execution of an arbitrary algorithm on an arbitrary instance of its input data terminates or not (1936).

- Alonzo Church proved that predicate logic is not decidable: there is no algorithm able to establish in a finite amount of time whether an arbitrary predicate formula is satisfiable or not (1936).
- Alan Turing proved that the halting problem is not decidable: there is no algorithm able to establish in a finite amount of time whether the execution of an arbitrary algorithm on an arbitrary instance of its input data terminates or not (1936).
- The existence in any deduction system of theorems that cannot be proven, as well as the existence of computational problems that cannot be solved by any algorithm, determined the impossibility of reducing mathematics to mechanistic deductions.
- The end of logicism, but not the end of mathematical logic, which is the calculus of informatics (like infinitesimal calculus for physics).

2.4 Induction, Enumerations, Encodings

- $\mathbb{N}$ is infinite because it is equinumerous to its proper subset $\mathbb{N} \setminus \{0\}$ as shown by the bijective function $succ(n) = n + 1$.
- Thus it cannot be defined by listing its elements.

2.4 Induction, Enumerations, Encodings

- $\mathbb{N}$ is infinite because it is equinumerous to its proper subset $\mathbb{N} \setminus \{0\}$ as shown by the bijective function $succ(n) = n + 1$.
- Thus it cannot be defined by listing its elements.
- Dedekind-Peano axiomatic definition of $\mathbb{N}$ based on set theory:
  1. There exists an element $0 \in \mathbb{N}$.
  2. There exists a total function $succ : \mathbb{N} \to \mathbb{N}$.
  3. For all $n \in \mathbb{N}$, $succ(n) \neq 0$.
  4. For all $n, n' \in \mathbb{N}$, if $n \neq n'$ then $succ(n) \neq succ(n')$.
  5. If M is a subset of $\mathbb{N}$ such that:
     - $0 \in M$;
     - for all $n \in \mathbb{N}$, $n \in M$ implies $succ(n) \in M$;

     then $M = \mathbb{N}$ (hence $\mathbb{N}$ is the smallest set closed w.r.t. 0 and $succ$).

2.4 Induction, Enumerations, Encodings

- $\mathbb{N}$ is infinite because it is equinumerous to its proper subset $\mathbb{N} \setminus \{0\}$ as shown by the bijective function $succ(n) = n + 1$.
- Thus it cannot be defined by listing its elements.
- Dedekind-Peano axiomatic definition of $\mathbb{N}$ based on set theory:
  1. There exists an element $0 \in \mathbb{N}$.
  2. There exists a total function $succ : \mathbb{N} \to \mathbb{N}$.
  3. For all $n \in \mathbb{N}$, $succ(n) \neq 0$.
  4. For all $n, n' \in \mathbb{N}$, if $n \neq n'$ then $succ(n) \neq succ(n')$.
  5. If M is a subset of $\mathbb{N}$ such that:
     - $0 \in M$;
     - for all $n \in \mathbb{N}$, $n \in M$ implies $succ(n) \in M$;

     then $M = \mathbb{N}$ (hence $\mathbb{N}$ is the smallest set closed w.r.t. 0 and $succ$).
- The elements of $\mathbb{N}$ are thus 0, $succ(0)$, $succ(succ(0))$, ... where $succ(0)$ is denoted by 1, $succ(succ(0))$ is denoted by 2, and so on.

- The last axiom is the induction principle, which is one of the most powerful tools of discrete mathematics for definitions and proofs.
- Allows entire arithmetic to be formally defined: Peano arithmetic.
- Let $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ be such that $pred(succ(n)) = n$ for all $n \in \mathbb{N}$ and $succ(pred(n)) = n$ for all $n \in \mathbb{N}_{\neq 0}$.

- The last axiom is the induction principle, which is one of the most powerful tools of discrete mathematics for definitions and proofs.
- Allows entire arithmetic to be formally defined: Peano arithmetic.
- Let $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ be such that $pred(succ(n)) = n$ for all $n \in \mathbb{N}$ and $succ(pred(n)) = n$ for all $n \in \mathbb{N}_{\neq 0}$.
- Formal definition of addition over $\mathbb{N}$:

$$m \oplus n = \begin{cases} m & \text{if } n = 0 \\ succ(m) \oplus pred(n) & \text{if } n \neq 0 \end{cases}$$

- The last axiom is the induction principle, which is one of the most powerful tools of discrete mathematics for definitions and proofs.
- Allows entire arithmetic to be formally defined: Peano arithmetic.
- Let $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ be such that $pred(succ(n)) = n$ for all $n \in \mathbb{N}$ and $succ(pred(n)) = n$ for all $n \in \mathbb{N}_{\neq 0}$.
- Formal definition of addition over $\mathbb{N}$:

$$m \oplus n = \begin{cases} m & \text{if } n = 0 \\ succ(m) \oplus pred(n) & \text{if } n \neq 0 \end{cases}$$

- Example: $5 \oplus 2 = 6 \oplus 1 = 7 \oplus 0 = 7$.

- The last axiom is the induction principle, which is one of the most powerful tools of discrete mathematics for definitions and proofs.
- Allows entire arithmetic to be formally defined: Peano arithmetic.
- Let $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ be such that $pred(succ(n)) = n$ for all $n \in \mathbb{N}$ and $succ(pred(n)) = n$ for all $n \in \mathbb{N}_{\neq 0}$.
- Formal definition of addition over $\mathbb{N}$:

$$m \oplus n = \begin{cases} m & \text{if } n = 0 \\ succ(m) \oplus pred(n) & \text{if } n \neq 0 \end{cases}$$

- Example: $5 \oplus 2 = 6 \oplus 1 = 7 \oplus 0 = 7$.
- Formal definition of order relations over $\mathbb{N}$:
  - $m \leq n$ iff there exists $m' \in \mathbb{N}$ such that $m \oplus m' = n$.
  - $m < n$ iff $m \leq n$ with $m \neq n$.
  - $m \geq n$ iff $n \leq m$.
  - $m > n$ iff $n < m$.

- Formal definition of subtraction over $\mathbb{N}$ ($m \geq n$):

$$m \ominus n = \begin{cases} m & \text{if } n = 0 \\ pred(m) \ominus pred(n) & \text{if } n > 0 \end{cases}$$

- Formal definition of subtraction over $\mathbb{N}$ ($m \geq n$):

$$m \ominus n = \begin{cases} m & \text{if } n = 0 \\ pred(m) \ominus pred(n) & \text{if } n > 0 \end{cases}$$

- Formal definition of multiplication over $\mathbb{N}$:

$$m \otimes n = \begin{cases} 0 & \text{if } n = 0 \\ m \oplus (m \otimes pred(n)) & \text{if } n > 0 \end{cases}$$

- Formal definition of subtraction over $\mathbb{N}$ $(m \geq n)$:

$$m \ominus n = \begin{cases} m & \text{if } n = 0 \\ pred(m) \ominus pred(n) & \text{if } n > 0 \end{cases}$$

- Formal definition of multiplication over $\mathbb{N}$:

$$m \otimes n = \begin{cases} 0 & \text{if } n = 0 \\ m \oplus (m \otimes pred(n)) & \text{if } n > 0 \end{cases}$$

- Formal definition of division over $\mathbb{N}$ $(n \neq 0)$:

$$m \oslash n = \begin{cases} 0 & \text{if } m < n \\ succ((m \ominus n) \oslash n) & \text{if } m \geq n \end{cases}$$

- Formal definition of subtraction over $\mathbb{N}$ $(m \geq n)$:

$$m \ominus n = \begin{cases} m & \text{if } n = 0 \\ pred(m) \ominus pred(n) & \text{if } n > 0 \end{cases}$$

- Formal definition of multiplication over $\mathbb{N}$:

$$m \otimes n = \begin{cases} 0 & \text{if } n = 0 \\ m \oplus (m \otimes pred(n)) & \text{if } n > 0 \end{cases}$$

- Formal definition of division over $\mathbb{N}$ $(n \neq 0)$:

$$m \oslash n = \begin{cases} 0 & \text{if } m < n \\ succ((m \ominus n) \oslash n) & \text{if } m \geq n \end{cases}$$

- Examples:
  - $5 \ominus 2 = 4 \ominus 1 = 3 \ominus 0 = 3.$
  - $5 \otimes 2 = 5 \oplus (5 \otimes 1) = 5 \oplus (5 \oplus (5 \otimes 0)) = 5 \oplus (5 \oplus 0) = 5 \oplus 5 = \ldots = 10.$
  - $5 \oslash 2 = succ((5 \ominus 2) \oslash 2) = \ldots = succ(3 \oslash 2) = succ(succ((3 \ominus 2) \oslash 2))$ $= \ldots = succ(succ(1 \oslash 2)) = succ(succ(0)) = succ(1) = 2.$

- The induction principle is the basis of recursive programming too.
- A complex problem is divided into simpler subproblems so that the solution is obtained by combining those of the subproblems.
- When the subproblems are of the *same nature as the original problem* we can adopt a recursive solution scheme:
  - Identify one or more base cases, for each of which we can directly obtain the solution to the problem.
  - Define one or more general cases through a set of subproblems of the same nature as the original one but closer to the base cases.

- The induction principle is the basis of recursive programming too.
- A complex problem is divided into simpler subproblems so that the solution is obtained by combining those of the subproblems.
- When the subproblems are of the *same nature as the original problem* we can adopt a recursive solution scheme:
  - Identify one or more base cases, for each of which we can directly obtain the solution to the problem.
  - Define one or more general cases through a set of subproblems of the same nature as the original one but closer to the base cases.
- Recursion is a very powerful tool:
  - Tackling problems not otherwise manageable (towers of Hanoi).
  - Solving problems more efficiently (binary search, mergesort, quicksort).
  - Defining data structures in a natural way (lists, trees).

- The induction principle is the basis of recursive programming too.
- A complex problem is divided into simpler subproblems so that the solution is obtained by combining those of the subproblems.
- When the subproblems are of the *same nature as the original problem* we can adopt a recursive solution scheme:
  - Identify one or more base cases, for each of which we can directly obtain the solution to the problem.
  - Define one or more general cases through a set of subproblems of the same nature as the original one but closer to the base cases.
- Recursion is a very powerful tool:
  - Tackling problems not otherwise manageable (towers of Hanoi).
  - Solving problems more efficiently (binary search, mergesort, quicksort).
  - Defining data structures in a natural way (lists, trees).
- ... How many functions from $\mathbb{N}$ to $\mathbb{N}$ are there?

- An infinite set A is countable iff it is equinumerous to $\mathbb{N}$.
- An enumeration of A is a bijective function from $\mathbb{N}$ to A (indexing).
- An encoding of A is a bijection from A to (an infinite subset of) $\mathbb{N}$.

- An infinite set A is countable iff it is equinumerous to $\mathbb{N}$.
- An enumeration of A is a bijective function from $\mathbb{N}$ to A (indexing).
- An encoding of A is a bijection from A to (an infinite subset of) $\mathbb{N}$.
- The union of two countable sets is countable,
  therefore $\mathbb{Z} = \mathbb{N} \cup \{-n \mid n \in \mathbb{N} \setminus \{0\}\}$ is countable.
- The Cartesian product of two countable sets is countable,
  therefore $\mathbb{Q} = \mathbb{Z} \times (\mathbb{N} \setminus \{0\})$ is countable.

- An infinite set A is countable iff it is equinumerous to $\mathbb{N}$.
- An enumeration of A is a bijective function from $\mathbb{N}$ to A (indexing).
- An encoding of A is a bijection from A to (an infinite subset of) $\mathbb{N}$.
- The union of two countable sets is countable, therefore $\mathbb{Z} = \mathbb{N} \cup \{-n \mid n \in \mathbb{N} \setminus \{0\}\}$ is countable.
- The Cartesian product of two countable sets is countable, therefore $\mathbb{Q} = \mathbb{Z} \times (\mathbb{N} \setminus \{0\})$ is countable.
- $|\mathbb{Z}| = |\mathbb{N}|$ is shown by the following encoding to the whole $\mathbb{N}$:
  - $f_{\mathbb{Z}}(0) = 0$.
  - $f_{\mathbb{Z}}(z) = 2 \cdot z - 1$ if $z > 0$.
  - $f_{\mathbb{Z}}(z) = 2 \cdot (-z)$ if $z < 0$.
- Positive integers are mapped to odd naturals whereas negative integers are mapped to even naturals:

$$\begin{array}{ccccccccccc} \cdots & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & \cdots \\ \cdots & 8 & 6 & 4 & 2 & 0 & 1 & 3 & 5 & 7 & \cdots \end{array}$$

- $|\mathbb{Q}| = |\mathbb{N}|$ is shown by the following encoding to $\mathbb{N} \setminus \{\sum_{i=0}^{k} i \mid k \in \mathbb{N}\}$:
  - $f_{\mathbb{Q}}(z, n) = \sum_{i=1}^{f_{\mathbb{Z}}(z)+n} i + n = \frac{(f_{\mathbb{Z}}(z)+n)\cdot(f_{\mathbb{Z}}(z)+n+1)}{2} + n$.
- Geometrical interpretation: moving in the nonnegative quadrant of the Cartesian plane along each of its diagonals parallel to $y = -x$ (the sum of the coordinates of the points of each diagonal is constant and corresponds to the number of points on the previous diagonal).

- $|\mathbb{Q}| = |\mathbb{N}|$ is shown by the following encoding to $\mathbb{N} \setminus \{\sum_{i=0}^{k} i \mid k \in \mathbb{N}\}$:
  - $f_{\mathbb{Q}}(z, n) = \sum_{i=1}^{f_{\mathbb{Z}}(z)+n} i + n = \frac{(f_{\mathbb{Z}}(z)+n)\cdot(f_{\mathbb{Z}}(z)+n+1)}{2} + n.$
- Geometrical interpretation: moving in the nonnegative quadrant of the Cartesian plane along each of its diagonals parallel to $y = -x$ (the sum of the coordinates of the points of each diagonal is constant and corresponds to the number of points on the previous diagonal).
- Rendering over $\mathbb{N} \times \mathbb{N}$ (the bottom line disappears when encoding $\mathbb{Q}$):

$$
\begin{array}{ccccccc}
\uparrow & \ddots & & & & & \\
3 & 9 & \ddots & & & & \\
2 & 5 & 8 & \ddots & & & \\
1 & 2 & 4 & 7 & \ddots & & \\
0 & 0 & 1 & 3 & 6 & \ddots & \\
 & 0 & 1 & 2 & 3 & \rightarrow &
\end{array}
$$

$$f_{\mathbb{N}\times\mathbb{N}}(n_1, n_2) = \sum_{i=1}^{n_1+n_2} i + n_2$$

- Proving $|\mathbb{R}| > |\mathbb{N}|$ through Cantor diagonal method:
  - Suppose by contradiction that $|\mathbb{R}_{]0,1[}| = |\mathbb{N}|$.
  - Then the numbers in $\mathbb{R}_{]0,1[}$ can be enumerated as follows:

$$\begin{array}{ccc} r_0 & = & 0.d_{0,0}d_{0,1}\ldots d_{0,n}\ldots \\ r_1 & = & 0.d_{1,0}d_{1,1}\ldots d_{1,n}\ldots \\ \ldots & \ldots & \ldots\ldots\ldots\ldots\ldots\ldots \\ r_n & = & 0.d_{n,0}d_{n,1}\ldots d_{n,n}\ldots \\ \ldots & \ldots & \ldots\ldots\ldots\ldots\ldots\ldots\ldots \end{array}$$

  - Let $r = 0.d_0d_1\ldots d_n\ldots$ where $d_k \neq d_{k,k}$ for all $k \in \mathbb{N}$ and at least one digit after the decimal point is different from 0.
  - It holds that $r \in \mathbb{R}_{]0,1[}$ but r does not occur in the enumeration, hence $|\mathbb{R}_{]0,1[}| > |\mathbb{N}|$.

- Proving $|\mathbb{R}| > |\mathbb{N}|$ through Cantor diagonal method:
  - Suppose by contradiction that $|\mathbb{R}_{]0,1[}| = |\mathbb{N}|$.
  - Then the numbers in $\mathbb{R}_{]0,1[}$ can be enumerated as follows:

$$\begin{array}{lll} r_0 & = & 0.d_{0,0}d_{0,1}\ldots d_{0,n}\ldots \\ r_1 & = & 0.d_{1,0}d_{1,1}\ldots d_{1,n}\ldots \\ \ldots & \ldots & \ldots\ldots\ldots\ldots\ldots\ldots \\ r_n & = & 0.d_{n,0}d_{n,1}\ldots d_{n,n}\ldots \\ \ldots & \ldots & \ldots\ldots\ldots\ldots\ldots\ldots \end{array}$$

  - Let $r = 0.d_0d_1\ldots d_n\ldots$ where $d_k \neq d_{k,k}$ for all $k \in \mathbb{N}$ and at least one digit after the decimal point is different from 0.
  - It holds that $r \in \mathbb{R}_{]0,1[}$ but r does not occur in the enumeration, hence $|\mathbb{R}_{]0,1[}| > |\mathbb{N}|$.
- $|\{f \mid f : \mathbb{N} \to \mathbb{N}\}| > |\mathbb{N}|$ can be proven in the same way, with each f_n viewed as $(0, f_n(0)), (1, f_n(1)), \ldots, (n, f_n(n)), \ldots$ and f such that $f(k) \neq f_k(k)$ for all $k \in \mathbb{N}$ not in the enumeration.
- Thus there are infinitely many functions that are *not* computable!

3. The Operational View: Turing Machines

Topics:

3.1 *Automata and Turing Machines* 34

3.2 *The Universal Turing Machine* 41

3.3 *Languages Recognized by Turing Machines* 43

3.4 *Functions Computed by Turing Machines* 46

3.1 Automata and Turing Machines

- An automaton is an abstract machine that at any time is in one state and evolves from a state to another in response to external stimuli like input data (one initial state, possibly several final states).
- Operational formalization of an algorithm or a computer.

3.1 Automata and Turing Machines

- An automaton is an abstract machine that at any time is in one state and evolves from a state to another in response to external stimuli like input data (one initial state, possibly several final states).
- Operational formalization of an algorithm or a computer.
- A finite-state automaton (FSA) is representable as a state-transition graph in which transitions are labeled with input/output information.
- Deterministic or nondeterministic depending on whether the labels of transitions departing from a state must be different or can be equal.

3.1 Automata and Turing Machines

- An automaton is an abstract machine that at any time is in one state and evolves from a state to another in response to external stimuli like input data (one initial state, possibly several final states).
- Operational formalization of an algorithm or a computer.
- A finite-state automaton (FSA) is representable as a state-transition graph in which transitions are labeled with input/output information.
- Deterministic or nondeterministic depending on whether the labels of transitions departing from a state must be different or can be equal.
- A pushdown automaton (PDA) is an FSA enriched with a stack-based memory in which previous input data can be stored or manipulated.

3.1 Automata and Turing Machines

- An automaton is an abstract machine that at any time is in one state and evolves from a state to another in response to external stimuli like input data (one initial state, possibly several final states).
- Operational formalization of an algorithm or a computer.
- A finite-state automaton (FSA) is representable as a state-transition graph in which transitions are labeled with input/output information.
- Deterministic or nondeterministic depending on whether the labels of transitions departing from a state must be different or can be equal.
- A pushdown automaton (PDA) is an FSA enriched with a stack-based memory in which previous input data can be stored or manipulated.
- A linear bounded automaton (LBA) is an FSA enriched with a tape of bounded length in which data can be read and written via a head.

- A Turing machine (1936) has:
  - A finite-state control unit.
  - A tape with infinitely many cells.
  - A head that reads and writes symbols and moves in both directions.
- First formalization of the notion of algorithm!

- A Turing machine (1936) has:
  - A finite-state control unit.
  - A tape with infinitely many cells.
  - A head that reads and writes symbols and moves in both directions.
- First formalization of the notion of algorithm!
- Equivalent formulations:
  - Several heads.
  - Several tapes (one for reading, one for writing, ...).
  - Multidimensional or semi-infinite tape.
  - The head can stay on the same cell instead of having to move.
  - Only two states or two input symbols (increase the other number).
- Writing capability (PDA, LBA), bidirectional movement (LBA), and tape unboundedness give TMs a lot of computational power!

- At the beginning of the computation:
  - The TM is in its initial state q_0.
  - The input, which is a finite-length sequence σ of input symbols, is placed on the tape.
  - The tape head is at the leftmost cell that holds an input symbol.
  - All the other tape cells contain a special symbol called blank (ƀ).

- At the beginning of the computation:
  - The TM is in its initial state q_0.
  - The input, which is a finite-length sequence σ of input symbols, is placed on the tape.
  - The tape head is at the leftmost cell that holds an input symbol.
  - All the other tape cells contain a special symbol called blank (ƀ).
- Based on its current state and the symbol currently under its head, at each step the TM will:
  - Evolve to the next state (may coincide with the current state).
  - Write a symbol in the current cell (may leave the same symbol).
  - Move the tape head left or right.

- At the beginning of the computation:
  - The TM is in its initial state q_0.
  - The input, which is a finite-length sequence σ of input symbols, is placed on the tape.
  - The tape head is at the leftmost cell that holds an input symbol.
  - All the other tape cells contain a special symbol called blank (ƀ).
- Based on its current state and the symbol currently under its head, at each step the TM will:
  - Evolve to the next state (may coincide with the current state).
  - Write a symbol in the current cell (may leave the same symbol).
  - Move the tape head left or right.
- The input alphabet is strictly included in the tape alphabet (ƀ).
- The computation is: discrete, deterministic, sequential.

- A TM is a tuple $Z = (Q, \Sigma, \Gamma, \delta, q_0)$ where:
  - Q is a *finite* set of states.
  - Σ is a *finite* set of input symbols.
  - Γ is a *finite* set of tape symbols, with $\Sigma \subsetneq \Gamma$.
  - $\delta : Q \times \Gamma \nrightarrow Q \times \Gamma \times D$ is a transition function, where $D = \{L, R\}$ is the set of directions in which the tape head can move.
  - $q_0 \in Q$ is the initial state.
- δ is a *partial* function, hence the computation terminates when δ does not associate anything with the current state and tape symbol.

- A TM is a tuple $Z = (Q, \Sigma, \Gamma, \delta, q_0)$ where:
  - Q is a *finite* set of states.
  - Σ is a *finite* set of input symbols.
  - Γ is a *finite* set of tape symbols, with $\Sigma \subsetneq \Gamma$.
  - $\delta : Q \times \Gamma \nrightarrow Q \times \Gamma \times D$ is a transition function, where $D = \{L, R\}$ is the set of directions in which the tape head can move.
  - $q_0 \in Q$ is the initial state.
- δ is a *partial* function, hence the computation terminates when δ does not associate anything with the current state and tape symbol.
- The instantaneous description of a TM is the finite sequence of symbols currently in the tape cells between the leftmost and the rightmost nonblanks, with the current state embedded just before the symbol currently under the head: $X_1 \dots X_{i-1} q X_i X_{i+1} \dots X_n$.
- Starting from the initial instantaneous description $q_0\,\sigma$, the computation may terminate or not.

- The moves of a TM can be described through relation $\vdash \subseteq ID \times ID$.
- If $\delta(q, X_i) = (q', X', L)$ then:
  - $1 < i < n$:
    $X_1 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash X_1 \dots X_{i-2}\, q' X_{i-1} X' X_{i+1} \dots X_n$.
  - $i = 1$:
    $q X_1 X_2 \dots X_n \vdash q' \text{b} X' X_2 \dots X_n$.
  - $i = n$:
    $X_1 \dots X_{n-1} q X_n \vdash X_1 \dots X_{n-2} q' X_{n-1} X'$.

- The moves of a TM can be described through relation $\vdash \subseteq ID \times ID$.
- If $\delta(q, X_i) = (q', X', L)$ then:
  - $1 < i < n$:
    $X_1 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash X_1 \dots X_{i-2}\, q' X_{i-1} X' X_{i+1} \dots X_n$.
  - $i = 1$:
    $q X_1 X_2 \dots X_n \vdash q' \mathfrak{b} X' X_2 \dots X_n$.
  - $i = n$:
    $X_1 \dots X_{n-1} q X_n \vdash X_1 \dots X_{n-2} q' X_{n-1} X'$.
- If $\delta(q, X_i) = (q', X', R)$ then:
  - $1 < i < n$:
    $X_1 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash X_1 \dots X_{i-1} X' q' X_{i+1} \dots X_n$.
  - $i = 1$:
    $q X_1 X_2 \dots X_n \vdash X' q' X_2 \dots X_n$.
  - $i = n$:
    $X_1 \dots X_{n-1} q X_n \vdash X_1 \dots X_{n-1} X' q' \mathfrak{b}$.
- A computation from id_0 to id_n, written $id_0 \vdash^* id_n$, is a sequence of instantaneous descriptions $(id_i)_{0 \le i \le n}$ such that $id_i \vdash id_{i+1}$ for all i.

- Example of TM computing the successor of a natural number:
  - $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
  - $\Gamma = \Sigma \cup \{\text{b}\}$.
  - $Q = \{q_0, q_1, q_2\}$ where:
    - q_0: TM moves from the leftmost digit to the rightmost one.
    - q_1: TM adds 1 and continues to the left in case of carry.
    - q_2: TM halts.

- Example of TM computing the successor of a natural number:
  - $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
  - $\Gamma = \Sigma \cup \{\text{ƀ}\}$.
  - $Q = \{q_0, q_1, q_2\}$ where:
    - q_0: TM moves from the leftmost digit to the rightmost one.
    - q_1: TM adds 1 and continues to the left in case of carry.
    - q_2: TM halts.
  - Matrix representation of transition function δ:

	0	1	2	3	4	5	6	7	8	9	ƀ
q_0	$\frac{0}{\frac{R}{q_0}}$	$\frac{1}{\frac{R}{q_0}}$	$\frac{2}{\frac{R}{q_0}}$	$\frac{3}{\frac{R}{q_0}}$	$\frac{4}{\frac{R}{q_0}}$	$\frac{5}{\frac{R}{q_0}}$	$\frac{6}{\frac{R}{q_0}}$	$\frac{7}{\frac{R}{q_0}}$	$\frac{8}{\frac{R}{q_0}}$	$\frac{9}{\frac{R}{q_0}}$	$\frac{\text{ƀ}}{\frac{L}{q_1}}$
q_1	$\frac{1}{\frac{L}{q_2}}$	$\frac{2}{\frac{L}{q_2}}$	$\frac{3}{\frac{L}{q_2}}$	$\frac{4}{\frac{L}{q_2}}$	$\frac{5}{\frac{L}{q_2}}$	$\frac{6}{\frac{L}{q_2}}$	$\frac{7}{\frac{L}{q_2}}$	$\frac{8}{\frac{L}{q_2}}$	$\frac{9}{\frac{L}{q_2}}$	$\frac{0}{\frac{L}{q_1}}$	$\frac{1}{\frac{L}{q_2}}$
q_2	—	—	—	—	—	—	—	—	—	—	—

- Example of TM verifying the balancing of a sequence of parentheses:
  - $\Sigma = \{(,)\}$.
  - $\Gamma = \Sigma \cup \{\text{b}, \text{X}, 0, 1\}$.
  - $Q = \{q_0, q_1, q_2, q_3\}$ where:
    - q_0: TM goes right until it deletes a right parenthesis if any (X).
    - q_1: TM goes left until it deletes the corresponding left par. if any (X).
    - q_2: balancing correct (1) or not (0) if no more left parentheses or not.
    - q_3: TM halts.

- Example of TM verifying the balancing of a sequence of parentheses:
  - $\Sigma = \{(,)\}$.
  - $\Gamma = \Sigma \cup \{\text{ƀ}, \mathsf{X}, 0, 1\}$.
  - $Q = \{q_0, q_1, q_2, q_3\}$ where:
    - q_0: TM goes right until it deletes a right parenthesis if any (X).
    - q_1: TM goes left until it deletes the corresponding left par. if any (X).
    - q_2: balancing correct (1) or not (0) if no more left parentheses or not.
    - q_3: TM halts.
  - Matrix representation of transition function δ:

	(	)	X	ƀ
q_0	$\frac{(}{\frac{R}{q_0}}$	$\frac{\mathsf{X}}{\frac{L}{q_1}}$	$\frac{\mathsf{X}}{\frac{R}{q_0}}$	$\frac{\text{ƀ}}{\frac{L}{q_2}}$
q_1	$\frac{\mathsf{X}}{\frac{R}{q_0}}$	—	$\frac{\mathsf{X}}{\frac{L}{q_1}}$	$\frac{0}{\frac{L}{q_3}}$
q_2	$\frac{0}{\frac{L}{q_3}}$	—	$\frac{\mathsf{X}}{\frac{L}{q_2}}$	$\frac{1}{\frac{L}{q_3}}$
q_3	—	—	—	—

3.2 The Universal Turing Machine

- A TM can be viewed as the formal description of an algorithm according to an operational style.
- It can be thought of as a computer with a *hardwired program*.

3.2 The Universal Turing Machine

- A TM can be viewed as the formal description of an algorithm according to an operational style.
- It can be thought of as a computer with a *hardwired program*.
- The universal Turing machine can simulate any TM on any input.
- It can be thought of as a computer with a *stored program*, which may thus be changed after its execution.

3.2 The Universal Turing Machine

- A TM can be viewed as the formal description of an algorithm according to an operational style.
- It can be thought of as a computer with a *hardwired program*.
- The universal Turing machine can simulate any TM on any input.
- It can be thought of as a computer with a *stored program*, which may thus be changed after its execution.
- UTM represents a computation scheme no longer hardwired.
- The first general-purpose electronic computers like ENIAC (1946) were programmed by setting up switches and connecting cables.
- From plugboard computers to stored program computers thanks to Von Neumann (1952).

- UTM acts as an *interpreter* of the instructions of the simulated TM.
- UTM needs to know:
  - The current contents of the tape of the TM being simulated.
  - The current state of the TM being simulated.
  - The symbol currently under the head of the TM being simulated.
  - All the tuples of the transition function of the TM being simulated.

- UTM acts as an *interpreter* of the instructions of the simulated TM.
- UTM needs to know:
  - The current contents of the tape of the TM being simulated.
  - The current state of the TM being simulated.
  - The symbol currently under the head of the TM being simulated.
  - All the tuples of the transition function of the TM being simulated.
- These pieces of information may reside on separate tapes of UTM.
- If on a single tape, special tape symbols delimit the various sections.
- Like in the fetch-decode-execute cycle of the processor of a computer, based on the current state and the symbol currently under the head of the TM being simulated, UTM repeatedly selects a tuple and updates the state, the tape contents, and the head position.

3.3 Languages Recognized by Turing Machines

- Automata theory is deeply intertwined with formal languages.
- A language L builds on:
  - A *finite* alphabet Σ of symbols (with associated phonemes).
  - A set of lexemes or words or strings each belonging to the set Σ^* of *finite* sequences of symbols of Σ (indeed $L \subseteq \Sigma^*$).
  - A *finite* set of grammar rules yielding phrases or sentences.
  - A semantic interpretation assigning meanings to phrases.

3.3 Languages Recognized by Turing Machines

- Automata theory is deeply intertwined with formal languages.
- A language L builds on:
  - A *finite* alphabet Σ of symbols (with associated phonemes).
  - A set of lexemes or words or strings each belonging to the set Σ^* of *finite* sequences of symbols of Σ (indeed $L \subseteq \Sigma^*$).
  - A *finite* set of grammar rules yielding phrases or sentences.
  - A semantic interpretation assigning meanings to phrases.
- A grammar is a tuple $G = (\Sigma, N, S, P)$ where
  - Σ is a *finite* alphabet of terminal symbols.
  - N is a *finite* set of nonterminal symbols (syntactic categories).
  - $S \in N$ is the start nonterminal symbol.
  - P is a *finite* set of productions each of the form $\alpha \to \beta$ (or $\alpha ::= \beta$) where $\alpha \in (\Sigma \cup N)^+$ while $\beta \in (\Sigma \cup N)^*$.
- The language generated by G is $L(G) = \{\sigma \in \Sigma^* \mid S \to^* \sigma\}$.

- If TM Z is enriched with an accepting state q_{a}, then we can define the language recognized by Z as $L(Z) = \{\sigma \in \Sigma^* \mid q_0\sigma \vdash^* id \nvdash, q_{\mathsf{a}} \in id\}$.
- Membership problem – given $\sigma \in \Sigma^*$, establish whether $\sigma \in L(Z)$:
  - If Z terminates on σ and q_{a} is reached, then $\sigma \in L(Z)$.
  - If Z terminates on σ and q_{a} is not reached, then $\sigma \notin L(Z)$.
  - If Z does not terminate on σ, then no answer can be provided.

- If TM Z is enriched with an accepting state q_{a}, then we can define the language recognized by Z as $L(Z) = \{\sigma \in \Sigma^* \mid q_0\sigma \vdash^* id \nvdash, q_{\mathsf{a}} \in id\}$.
- Membership problem – given $\sigma \in \Sigma^*$, establish whether $\sigma \in L(Z)$:
  - If Z terminates on σ and q_{a} is reached, then $\sigma \in L(Z)$.
  - If Z terminates on σ and q_{a} is not reached, then $\sigma \notin L(Z)$.
  - If Z does not terminate on σ, then no answer can be provided.
- The membership problem is only *semi-decidable* in general, because Z certainly halts only when the answer is yes.
- $L(Z)$ is thus a *recursively enumerable* set, i.e., a set for which there exists an enumeration of its elements.
- $L(Z)$ is instead a *recursive* set when Z terminates on any input, i.e., when Z ensures the decidability of the membership problem.
- The infinite set 2^{Σ^*} of all languages over Σ is not countable, hence only countably many languages are recursively enumerable.

- Language classification by Noam Chomsky (1956):
  - L_0 (basis for natural languages):
    - Generated by general grammars $G = (\Sigma, N, S, P)$.
    - Recursively enumerable languages – semi-decidability of $\sigma \in L$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by TMs.

- Language classification by Noam Chomsky (1956):
  - L_0 (basis for natural languages):
    - Generated by general grammars $G = (\Sigma, N, S, P)$.
    - Recursively enumerable languages – semi-decidability of $\sigma \in L$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by TMs.
  - L_1 (context-sensitive languages):
    - Generated by context-sensitive grammars $(\gamma_1 A \gamma_2 \rightarrow \gamma_1 \gamma \gamma_2)$.
    - Recursive languages $(a^n b^n c^n)$ – decidability of $\sigma \in L$ in $O(\alpha^{|\sigma|})$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by linear bounded automata (LBA).

- Language classification by Noam Chomsky (1956):
  - L_0 (basis for natural languages):
    - Generated by general grammars $G = (\Sigma, N, S, P)$.
    - Recursively enumerable languages – semi-decidability of $\sigma \in L$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by TMs.
  - L_1 (context-sensitive languages):
    - Generated by context-sensitive grammars $(\gamma_1 A \gamma_2 \rightarrow \gamma_1 \gamma \gamma_2)$.
    - Recursive languages $(a^n b^n c^n)$ – decidability of $\sigma \in L$ in $O(\alpha^{|\sigma|})$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by linear bounded automata (LBA).
  - L_2 (context-free languages – basis for programming languages):
    - Generated by context-free grammars $(A \rightarrow \gamma)$.
    - Recursive languages $(a^n b^n)$ – decidability of $\sigma \in L$ in $O(|\sigma|^2)$.
    - Undecidability of language equivalence.
    - Recognized by pushdown automata (PDA).

- Language classification by Noam Chomsky (1956):
  - L_0 (basis for natural languages):
    - Generated by general grammars $G = (\Sigma, N, S, P)$.
    - Recursively enumerable languages – semi-decidability of $\sigma \in L$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by TMs.
  - L_1 (context-sensitive languages):
    - Generated by context-sensitive grammars ($\gamma_1 A \gamma_2 \rightarrow \gamma_1 \gamma \gamma_2$).
    - Recursive languages ($a^n b^n c^n$) – decidability of $\sigma \in L$ in $O(\alpha^{|\sigma|})$.
    - Undecidability of language emptiness, finiteness, equivalence.
    - Recognized by linear bounded automata (LBA).
  - L_2 (context-free languages – basis for programming languages):
    - Generated by context-free grammars ($A \rightarrow \gamma$).
    - Recursive languages ($a^n b^n$) – decidability of $\sigma \in L$ in $O(|\sigma|^2)$.
    - Undecidability of language equivalence.
    - Recognized by pushdown automata (PDA).
  - L_3 (regular languages – representable through regular expressions):
    - Generated by linear grammars ($A \rightarrow \gamma$ with at most one $B \in N$ in γ).
    - Recursive languages ($a^n b^m$) – decidability of $\sigma \in L$ in $O(|\sigma|)$.
    - Recognized by finite-state automata (FSA).

3.4 Functions Computed by Turing Machines

- The function computed by TM Z is partial function $f_Z : ID_Z \nrightarrow ID_Z$.
- Z converges on input σ, written $Z \downarrow \sigma$, if $q_0\sigma \vdash^* id \nvdash (f_Z(q_0\sigma) = id)$, otherwise we say that Z diverges on σ, written $Z \uparrow \sigma$ ($f_Z(q_0\sigma)$ und.).

3.4 Functions Computed by Turing Machines

- The function computed by TM Z is partial function $f_Z : ID_Z \nrightarrow ID_Z$.
- Z converges on input σ, written $Z \downarrow \sigma$, if $q_0\sigma \vdash^* id \nvdash (f_Z(q_0\sigma) = id)$, otherwise we say that Z diverges on σ, written $Z \uparrow \sigma$ ($f_Z(q_0\sigma)$ und.).
- Using a suitable enumeration and a suitable encoding for the set ID_Z we get $f'_Z : \mathbb{N} \nrightarrow \mathbb{N}$ where $f'_Z(n) = enc_Z(f_Z(enum_Z(n)))$.
- f_Z and f'_Z are total iff Z always converges.
- Given an enumeration of TMs, the function computed by UTM U is $f'_U : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N}$ where $f'_U(n_1, n_2) = f'_{enum(n_1)}(n_2)$.

3.4 Functions Computed by Turing Machines

- The function computed by TM Z is partial function $f_Z : ID_Z \nrightarrow ID_Z$.
- Z converges on input σ, written $Z \downarrow \sigma$, if $q_0\sigma \vdash^* id \nvdash (f_Z(q_0\sigma) = id)$, otherwise we say that Z diverges on σ, written $Z \uparrow \sigma$ ($f_Z(q_0\sigma)$ und.).
- Using a suitable enumeration and a suitable encoding for the set ID_Z we get $f'_Z : \mathbb{N} \nrightarrow \mathbb{N}$ where $f'_Z(n) = enc_Z(f_Z(enum_Z(n)))$.
- f_Z and f'_Z are total iff Z always converges.
- Given an enumeration of TMs, the function computed by UTM U is $f'_U : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N}$ where $f'_U(n_1, n_2) = f'_{enum(n_1)}(n_2)$.
- A function over naturals is Turing computable iff there exists a TM that computes the considered function.
- Remind that not all functions over naturals are Turing computable: countably many are, *uncountably many are not*.
- Can we characterize the set of Turing-computable functions? Are there computable functions beyond Turing-computable ones?

4. The Functional View: Lambda Calculus

Topics:

4.1 *Syntax of Lambda Calculus* 48

4.2 *Semantics and Combinatory Logic* 51

4.3 *Recursive Functions via Fixed Points* 58

4.4 *Termination and Confluence* 62

4.5 *Lambda Calculus with Types* 66

4.1 Syntax of Lambda Calculus

- A function $f : A \to B$ is a subset of $A \times B$ in which for each $a \in A$ there exists *at most* one $b \in B$ such that $(a, b) \in f$, written $f(a) = b$.
- Extensional view: f is a set of ordered pairs.
- Intensional view: computational rules to obtain b from a.
- *Notational ambiguity*: is $f(x)$ the definition or an application of f?

4.1 Syntax of Lambda Calculus

- A function $f : A \to B$ is a subset of $A \times B$ in which for each $a \in A$ there exists *at most* one $b \in B$ such that $(a, b) \in f$, written $f(a) = b$.
- Extensional view: f is a set of ordered pairs.
- Intensional view: computational rules to obtain b from a.
- *Notational ambiguity*: is $f(x)$ the definition or an application of f?
- In 1932 Alonzo Church introduced λ-calculus as part of a general theory of functions and logic, to be a foundation of mathematics.
- Calculus able to capture the computational aspects of functions:
  - Formal system equipped with a syntax for generating terms.
  - Set of rewriting rules for transforming terms into other terms.
- Functional programming: Lisp, Scheme, ML, Haskell (also Python).

- The λ-calculus describes functions in their full generality, including recursive functions and higher-order functions.
- This is achieved through an extremely simple syntax, which clearly distinguishes between function definition and function application.

- The λ-calculus describes functions in their full generality, including recursive functions and higher-order functions.
- This is achieved through an extremely simple syntax, which clearly distinguishes between function definition and function application.
- The set Λ of λ-terms is generated by:

$$E \ ::= \ x \mid \lambda x \,.\, E \mid E\, E$$

where:

  - $x \in Var$, with Var being a countable set of variables.
  - $\lambda x \,.\, E$ is a λ-abstraction, i.e., a function definition, with λ called *binder* for *formal parameter* x within *function body* E (unary operator).
  - $E_1\, E_2$ is the application of E_1 to E_2 (binary operator).

- The λ-calculus describes functions in their full generality, including recursive functions and higher-order functions.
- This is achieved through an extremely simple syntax, which clearly distinguishes between function definition and function application.
- The set Λ of λ-terms is generated by:

$$E \ ::= \ x \mid \lambda x \,.\, E \mid E\, E$$

  where:
  - $x \in Var$, with Var being a countable set of variables.
  - $\lambda x \,.\, E$ is a λ-abstraction, i.e., a function definition, with λ called *binder* for *formal parameter* x within *function body* E (unary operator).
  - $E_1\, E_2$ is the application of E_1 to E_2 (binary operator).
- Precedence of application over λ-abstr.: $\lambda x \,.\, E_1\, E_2 = \lambda x \,.\, (E_1\, E_2)$.
- Right associativity of λ-abstraction: $\lambda x \,.\, \lambda y \,.\, E = \lambda x \,.\, (\lambda y \,.\, E)$.
- Left associativity of application: $E_1\, E_2\, E_3 = (E_1\, E_2)\, E_3$.

- Examples:
  - Identity function: $\lambda x \,.\, x$.
  - Selection of either argument: $\lambda x \,.\, \lambda y \,.\, x$ and $\lambda x \,.\, \lambda y \,.\, y$.
  - Successor and predecessor functions: $\lambda x \,.\, x + 1$ and $\lambda x \,.\, x - 1$.
  - Application of the successor function to value 2: $(\lambda x \,.\, x + 1)\, 2$.

- Examples:
  - Identity function: $\lambda x \,.\, x$.
  - Selection of either argument: $\lambda x \,.\, \lambda y \,.\, x$ and $\lambda x \,.\, \lambda y \,.\, y$.
  - Successor and predecessor functions: $\lambda x \,.\, x + 1$ and $\lambda x \,.\, x - 1$.
  - Application of the successor function to value 2: $(\lambda x \,.\, x + 1)\, 2$.
- Symbols 1, 2, $+$, $-$ are *not* admitted in Λ. How to encode them?
- Start with natural numbers, then arithmetical operations.

- Examples:
  - Identity function: $\lambda x \,.\, x$.
  - Selection of either argument: $\lambda x \,.\, \lambda y \,.\, x$ and $\lambda x \,.\, \lambda y \,.\, y$.
  - Successor and predecessor functions: $\lambda x \,.\, x + 1$ and $\lambda x \,.\, x - 1$.
  - Application of the successor function to value 2: $(\lambda x \,.\, x + 1)\, 2$.
- Symbols 1, 2, $+$, $-$ are *not* admitted in Λ. How to encode them?
- Start with natural numbers, then arithmetical operations.
- Church numerals based on Dedekind-Peano axioms:
  - $\underline{0} = \lambda s \,.\, \lambda z \,.\, z$ so that the representation of 0 returns z.
  - $\underline{1} = \lambda s \,.\, \lambda z \,.\, s\, z$ so that the representation of 1 returns a single application of s to z.
  - $\underline{2} = \lambda s \,.\, \lambda z \,.\, s\, (s\, z)$ so that the representation of 2 returns a double application of s to z.

    $\vdots$
  - $\underline{n} = \lambda s \,.\, \lambda z \,.\, s\, (\ldots (s\, z) \ldots)$ where the application of s is repeated $n \in \mathbb{N}_{\geq 1}$ times as if $\underline{n}$ were an *iterator* of s on z.

4.2 Semantics and Combinatory Logic

- A variable occurrence within a λ-term is *bound*/*free* depending on whether it is in the scope of a binder with the same name or not: x is bound in $\lambda x \,.\, x$, free in $\lambda y \,.\, x$.
- Term rewriting rules based on *syntactical substitutions of free vars*:
  - α-conversion: $\lambda x \,.\, E =_\alpha \lambda y \,.\, (E[{}^{y}/_{x}])$ provided that $y \notin fvar(E)$.
  - β-conversion: $(\lambda x \,.\, E)\, F =_\beta E[{}^{F}/_{x}]$.
  - η-conversion: $\lambda x \,.\, E\, x =_\eta E$ provided that $x \notin fvar(E)$.

4.2 Semantics and Combinatory Logic

- A variable occurrence within a λ-term is *bound*/*free* depending on whether it is in the scope of a binder with the same name or not: x is bound in $\lambda x \,.\, x$, free in $\lambda y \,.\, x$.
- Term rewriting rules based on *syntactical substitutions of free vars*:
  - α-conversion: $\lambda x \,.\, E =_\alpha \lambda y \,.\, (E[{}^{y}/_{x}])$ provided that $y \notin fvar(E)$.
  - β-conversion: $(\lambda x \,.\, E)\, F =_\beta E[{}^{F}/_{x}]$.
  - η-conversion: $\lambda x \,.\, E\, x =_\eta E$ provided that $x \notin fvar(E)$.
- The corresponding equivalence relations $\equiv_\alpha$, $\equiv_\beta$, $\equiv_\eta$ over Λ are *congruences* with respect to λ-abstraction and application, hence they can be applied *compositionally*.
- If $E_1 \equiv E_2$ for $\equiv \in \{\equiv_\alpha, \equiv_\beta, \equiv_\eta\}$, then:
  - $\lambda x \,.\, E_1 \equiv \lambda x \,.\, E_2$ for all $x \in Var$.
  - $E_1\, F \equiv E_2\, F$ and $F\, E_1 \equiv F\, E_2$ for all $F \in \Lambda$.

- α-conversion allows *bound* variable names and binders to be changed: $\lambda x \,.\, x =_\alpha \lambda y \,.\, y$, but $\lambda x \,.\, x\, z \neq_\alpha \lambda z \,.\, z\, z$ because $z \in fvar(x\, z)$.
- Convention: inside every λ-term, especially of the form $(\lambda x \,.\, E)\, F$, the names of variables bound to different occurrences of binders are always *different* from each other and from the names of free variables.

- α-conversion allows *bound* variable names and binders to be changed: $\lambda x \,.\, x =_\alpha \lambda y \,.\, y$, but $\lambda x \,.\, x\, z \neq_\alpha \lambda z \,.\, z\, z$ because $z \in \mathit{fvar}(x\, z)$.
- Convention: inside every λ-term, especially of the form $(\lambda x \,.\, E)\, F$, the names of variables bound to different occurrences of binders are always *different* from each other and from the names of free variables.
- β-conversion, with $(\lambda x \,.\, E)\, F$ called *redex* and $E[{}^{F}/_{x}]$ *contractum*, formalizes the application of a function $\lambda x \,.\, E$ to an argument F as the substitution of the *actual parameter* F for every occurrence of the *formal parameter* x that is free in the *function body* E.

- α-conversion allows *bound* variable names and binders to be changed: $\lambda x \,.\, x =_\alpha \lambda y \,.\, y$, but $\lambda x \,.\, x\, z \neq_\alpha \lambda z \,.\, z\, z$ because $z \in \mathit{fvar}(x\, z)$.
- Convention: inside every λ-term, especially of the form $(\lambda x \,.\, E)\, F$, the names of variables bound to different occurrences of binders are always *different* from each other and from the names of free variables.
- β-conversion, with $(\lambda x \,.\, E)\, F$ called *redex* and $E[{}^{F}/_{x}]$ *contractum*, formalizes the application of a function $\lambda x \,.\, E$ to an argument F as the substitution of the *actual parameter* F for every occurrence of the *formal parameter* x that is free in the *function body* E.
- η-conversion encodes the *extensional equality* between two functions, i.e., that two functions are equal iff they produce the same result whenever they are applied to the same argument.
- Equivalent to: if $E_1\, F \equiv_\beta E_2\, F$ for all $F \in \Lambda$, then $E_1 = E_2$.

- Reduction semantics when oriented from left to right:
  - β-reduction: $(\lambda x \,.\, E)\, F \longrightarrow_\beta E[{}^{F}/_{x}]$.
  - η-reduction: $\lambda x \,.\, E\, x \longrightarrow_\eta E$ provided that $x \notin fvar(E)$.
  - Reflexive and transitive closures up to α-conversion: $\longrightarrow_\beta^*$, $\longrightarrow_\eta^*$.

- Reduction semantics when oriented from left to right:
  - β-reduction: $(\lambda x \,.\, E)\, F \longrightarrow_\beta E[{}^{F}/_{x}]$.
  - η-reduction: $\lambda x \,.\, E\, x \longrightarrow_\eta E$ provided that $x \notin fvar(E)$.
  - Reflexive and transitive closures up to α-conversion: $\longrightarrow_\beta^*$, $\longrightarrow_\eta^*$.
- Currying: every function with several arguments is viewed as a function of a single argument that returns a function in which the remaining arguments are handled (right associativity).
- Not all the arguments have to be passed at once (partial evaluation).
- Example: $(\lambda y \,.\, \lambda x \,.\, x + y)\, 1 \longrightarrow_\beta (\lambda x \,.\, x + y)[{}^{1}/_{y}] = \lambda x \,.\, x + 1$.

- Reduction semantics when oriented from left to right:
  - β-reduction: $(\lambda x \,.\, E)\, F \longrightarrow_\beta E[{}^{F}/_{x}]$.
  - η-reduction: $\lambda x \,.\, E\, x \longrightarrow_\eta E$ provided that $x \notin fvar(E)$.
  - Reflexive and transitive closures up to α-conversion: $\longrightarrow_\beta^*$, $\longrightarrow_\eta^*$.
- Currying: every function with several arguments is viewed as a function of a single argument that returns a function in which the remaining arguments are handled (right associativity).
- Not all the arguments have to be passed at once (partial evaluation).
- Example: $(\lambda y \,.\, \lambda x \,.\, x + y)\, 1 \longrightarrow_\beta (\lambda x \,.\, x + y)[{}^{1}/_{y}] = \lambda x \,.\, x + 1$.
- Higher-order functions: a function can be passed as an argument to another function.
- Example: $(\lambda y \,.\, y\, 2)\, (\lambda x \,.\, x \cdot 3) \longrightarrow_\beta (y\, 2)[{}^{\lambda x \,.\, x \cdot 3}/_{y}] = (\lambda x \,.\, x \cdot 3)\, 2$ $\longrightarrow_\beta (x \cdot 3)[{}^{2}/_{x}] = 2 \cdot 3 = 6$.

- If $\underline{succ} = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y)$ then $\underline{succ}\,\underline{n} \longrightarrow_\beta^* \underline{n+1}$.
- Example: $\underline{succ}\,\underline{0} = (\lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))\,(\lambda s \,.\, \lambda z \,.\, z) \longrightarrow_\beta$
  $(\lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))[^{\lambda s \,.\, \lambda z \,.\, z}/_n] = \lambda x \,.\, \lambda y \,.\, x\,((\lambda s \,.\, \lambda z \,.\, z)\,x\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)[^x/_s]\,y) = \lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,(z[^y/_z]) = \lambda x \,.\, \lambda y \,.\, x\,y =_\alpha \underline{1}$.

- If $\underline{succ} = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y)$ then $\underline{succ}\,\underline{n} \longrightarrow_\beta^* \underline{n+1}$.
- Example: $\underline{succ}\,\underline{0} = (\lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))\,(\lambda s \,.\, \lambda z \,.\, z) \longrightarrow_\beta$
  $(\lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))[^{\lambda s \,.\, \lambda z \,.\, z}/_n] = \lambda x \,.\, \lambda y \,.\, x\,((\lambda s \,.\, \lambda z \,.\, z)\,x\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)[^x/_s]\,y) = \lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,(z[^y/_z]) = \lambda x \,.\, \lambda y \,.\, x\,y =_\alpha \underline{1}$.
- If $\underline{add} = \lambda m \,.\, \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, m\,x\,(n\,x\,y)$ then $\underline{add}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m+n}$.

- If $\underline{succ} = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y)$ then $\underline{succ}\,\underline{n} \longrightarrow_\beta^* \underline{n+1}$.
- Example: $\underline{succ}\,\underline{0} = (\lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))\,(\lambda s \,.\, \lambda z \,.\, z) \longrightarrow_\beta$
  $(\lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))[^{\lambda s \,.\, \lambda z \,.\, z}/_n] = \lambda x \,.\, \lambda y \,.\, x\,((\lambda s \,.\, \lambda z \,.\, z)\,x\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)[^x/_s]\,y) = \lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,(z[^y/_z]) = \lambda x \,.\, \lambda y \,.\, x\,y =_\alpha \underline{1}$.
- If $\underline{add} = \lambda m \,.\, \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, m\,x\,(n\,x\,y)$ then $\underline{add}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m+n}$.
- If $\underline{mult} = \lambda m \,.\, \lambda n \,.\, \lambda x \,.\, m\,(n\,x)$ then $\underline{mult}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m \cdot n}$.

- If $\underline{succ} = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y)$ then $\underline{succ}\,\underline{n} \longrightarrow_\beta^* \underline{n+1}$.
- Example: $\underline{succ}\,\underline{0} = (\lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))\,(\lambda s \,.\, \lambda z \,.\, z) \longrightarrow_\beta$
  $(\lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y))[^{\lambda s \,.\, \lambda z \,.\, z}/_n] = \lambda x \,.\, \lambda y \,.\, x\,((\lambda s \,.\, \lambda z \,.\, z)\,x\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)[^x/_s]\,y) = \lambda x \,.\, \lambda y \,.\, x\,((\lambda z \,.\, z)\,y) \longrightarrow_\beta$
  $\lambda x \,.\, \lambda y \,.\, x\,(z[^y/_z]) = \lambda x \,.\, \lambda y \,.\, x\,y =_\alpha \underline{1}$.
- If $\underline{add} = \lambda m \,.\, \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, m\,x\,(n\,x\,y)$ then $\underline{add}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m+n}$.
- If $\underline{mult} = \lambda m \,.\, \lambda n \,.\, \lambda x \,.\, m\,(n\,x)$ then $\underline{mult}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m \cdot n}$.
- If $\underline{exp} = \lambda m \,.\, \lambda n \,.\, m\,n$ then $\underline{exp}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{n^m}$.

- We now present three interesting λ-terms related to each other.
- If $\underline{I} = \lambda x \,.\, x$ then $\underline{I}\, E \longrightarrow_\beta^* E$ and in particular $\underline{I}\,\underline{I} \longrightarrow_\beta^* \underline{I}$: $\underline{I}\,\underline{I} =_\alpha (\lambda x \,.\, x)\,(\lambda x' .\, x') \longrightarrow_\beta x[^{\lambda x' .\, x'}/_x] = \lambda x' .\, x' =_\alpha \underline{I}$.

- We now present three interesting λ-terms related to each other.
- If $\underline{I} = \lambda x \,.\, x$ then $\underline{I}\, E \longrightarrow_\beta^* E$ and in particular $\underline{I}\,\underline{I} \longrightarrow_\beta^* \underline{I}$: $\underline{I}\,\underline{I} =_\alpha (\lambda x \,.\, x)\,(\lambda x' .\, x') \longrightarrow_\beta x[^{\lambda x' .\, x'}/_x] = \lambda x' .\, x' =_\alpha \underline{I}$.
- If $\underline{K} = \lambda x \,.\, \lambda y \,.\, x$ then $\underline{K}\, E = (\lambda x \,.\, \lambda y \,.\, x)\, E \longrightarrow_\beta \lambda y \,.\, E$ and moreover $\underline{K}\, E\, F \longrightarrow_\beta^* E$ because $y \notin fvar(E)$ otherwise we would have renamed λy in the substitution.

- We now present three interesting λ-terms related to each other.
- If $\underline{I} = \lambda x \,.\, x$ then $\underline{I}\,E \longrightarrow_\beta^* E$ and in particular $\underline{I}\,\underline{I} \longrightarrow_\beta^* \underline{I}$:
  $\underline{I}\,\underline{I} =_\alpha (\lambda x \,.\, x)\,(\lambda x' .\, x') \longrightarrow_\beta x[^{\lambda x' .\, x'}/_x] = \lambda x' .\, x' =_\alpha \underline{I}$.
- If $\underline{K} = \lambda x \,.\, \lambda y \,.\, x$ then $\underline{K}\,E = (\lambda x \,.\, \lambda y \,.\, x)\,E \longrightarrow_\beta \lambda y \,.\, E$
  and moreover $\underline{K}\,E\,F \longrightarrow_\beta^* E$ because $y \notin fvar(E)$
  otherwise we would have renamed λy in the substitution.
- If $\underline{S} = \lambda x \,.\, \lambda y \,.\, \lambda z \,.\, x\,z\,(y\,z)$ then in particular
  $\underline{S}\,\underline{K}\,\underline{K} =_\alpha (\lambda x \,.\, \lambda y \,.\, \lambda z \,.\, x\,z\,(y\,z))\,(\lambda x' .\, \lambda y' .\, x')\,(\lambda x'' .\, \lambda y'' .\, x'') \longrightarrow_\beta^* \underline{I}$.

- K and S are the basis of combinatory logic, introduced in 1924 by Moses Schönfinkel and reformulated in 1930 by Haskell Curry.
- Investigating foundations of mathematics by relying on operations instead of sets.

- K and S are the basis of combinatory logic, introduced in 1924 by Moses Schönfinkel and reformulated in 1930 by Haskell Curry.
- Investigating foundations of mathematics by relying on operations instead of sets.
- The set $\mathcal{L}$ of terms contains neither functions nor variable binders, but only the combinators K and S and their applications:

$$P \ ::= \ x \mid K \mid S \mid P\,P$$

- Reduction semantics for K, S, and application:
  - $K\,P\,Q \longrightarrow_{\mathcal{L}} P$.
  - $S\,P\,Q\,R \longrightarrow_{\mathcal{L}} P\,R\,(Q\,R)$.
  - If $P \longrightarrow_{\mathcal{L}} P'$ then $P\,Q \longrightarrow_{\mathcal{L}} P'\,Q$ and $Q\,P \longrightarrow_{\mathcal{L}} Q\,P'$.

- K and S are sufficient for implementing the λ-calculus, in particular λ-abstraction and β-reduction along with syntactical substitutions.
- Distinction between bound/free variable occurrences no longer applies to combinatory logic terms, occurrences are all free.

- K and S are sufficient for implementing the λ-calculus, in particular λ-abstraction and β-reduction along with syntactical substitutions.
- Distinction between bound/free variable occurrences no longer applies to combinatory logic terms, occurrences are all free.
- Define the following $\mathcal{L}$-terms that simulate λ-abstraction according to three cases based on the body of the function.
  - $\hat{\lambda}x\,.\,x = S\,K\,K$ for all $x \in Var$.
  - $\hat{\lambda}x\,.\,P = K\,P$ for all $x \in Var$ and $P \in \mathcal{L}$ such that $x \notin var(P)$.
  - $\hat{\lambda}x\,.\,P_1\,P_2 = S\,(\hat{\lambda}x\,.\,P_1)\,(\hat{\lambda}x\,.\,P_2)$ whenever $x \notin var(P_1) \cup var(P_2)$.
- Theorem: $(\hat{\lambda}x\,.\,P)\,Q \longrightarrow_{\mathcal{L}} P[{}^{Q}/_{x}]$ for all $x \in Var$ and $P, Q \in \mathcal{L}$.

- K and S are sufficient for implementing the λ-calculus, in particular λ-abstraction and β-reduction along with syntactical substitutions.
- Distinction between bound/free variable occurrences no longer applies to combinatory logic terms, occurrences are all free.
- Define the following $\mathcal{L}$-terms that simulate λ-abstraction according to three cases based on the body of the function.
  - $\hat{\lambda}x\,.\,x = S\,K\,K$ for all $x \in Var$.
  - $\hat{\lambda}x\,.\,P = K\,P$ for all $x \in Var$ and $P \in \mathcal{L}$ such that $x \notin var(P)$.
  - $\hat{\lambda}x\,.\,P_1\,P_2 = S\,(\hat{\lambda}x\,.\,P_1)\,(\hat{\lambda}x\,.\,P_2)$ whenever $x \notin var(P_1) \cup var(P_2)$.
- Theorem: $(\hat{\lambda}x\,.\,P)\,Q \longrightarrow_{\mathcal{L}} P[{}^{Q}/_{x}]$ for all $x \in Var$ and $P, Q \in \mathcal{L}$.
- Can we characterize the set of λ-definable functions?
  Relation between Turing-computable functions and λ-definable ones?

4.3 Recursive Functions via Fixed Points

- Functions are *anonymous* in λ-calculus, i.e., have no name.
- A recursive mathematical function $f = \cdots f \cdots$ can be expressed in λ-calculus only in a nonrecursive way through a higher-order function $\lambda f\,.\,\cdots f \cdots$ to which a fixed point combinator Ξ is then applied.

4.3 Recursive Functions via Fixed Points

- Functions are *anonymous* in λ-calculus, i.e., have no name.
- A recursive mathematical function $f = \cdots f \cdots$ can be expressed in λ-calculus only in a nonrecursive way through a higher-order function $\lambda f\,.\,\cdots f \cdots$ to which a fixed point combinator Ξ is then applied.
- A fixed point of a function $f : A \to A$ is $a \in A$ such that $a = f(a)$, i.e., an element of A invariant with respect to the application of f, i.e., a solution of the equation $x = f(x)$.
- Example: no fixed point for successor, several fixed points for identity.

4.3 Recursive Functions via Fixed Points

- Functions are *anonymous* in λ-calculus, i.e., have no name.
- A recursive mathematical function $f = \cdots f \cdots$ can be expressed in λ-calculus only in a nonrecursive way through a higher-order function $\lambda f\,.\,\cdots f \cdots$ to which a fixed point combinator Ξ is then applied.
- A fixed point of a function $f : A \to A$ is $a \in A$ such that $a = f(a)$, i.e., an element of A invariant with respect to the application of f, i.e., a solution of the equation $x = f(x)$.
- Example: no fixed point for successor, several fixed points for identity.
- The fixed point equation for a λ-term E is expressed as $F \equiv_{\beta\eta} E\,F$.
- $\Xi \in \Lambda$ is a fixed point combinator iff $\Xi\,E \equiv_{\beta\eta} E\,(\Xi\,E)$ for all $E \in \Lambda$.
- A fixed point combinator is a λ-term that allows the fixed point of *any* λ-term E to be computed by simply applying the former to the latter.

- Turing fixed point combinator:

$$\Theta \;=\; (\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,(\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))$$

- $\Theta\,E \longrightarrow_\beta (\lambda y\,.\,y\,((\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,(\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,y))\,E$
  $\longrightarrow_\beta E\,((\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))(\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,E) = E\,(\Theta\,E).$

- Turing fixed point combinator:

$$\Theta \;=\; (\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,(\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))$$

- $\Theta\,E \longrightarrow_\beta (\lambda y\,.\,y\,((\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,(\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,y))\,E$
  $\longrightarrow_\beta E\,((\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))(\lambda x\,.\,\lambda y\,.\,y\,(x\,x\,y))\,E) = E\,(\Theta\,E)$.
- Curry fixed point combinator:

$$Y \;=\; \lambda f\,.\,(\lambda x\,.\,f\,(x\,x))\,(\lambda x\,.\,f\,(x\,x))$$

- $Y\,E \longrightarrow_\beta (\lambda x\,.\,E\,(x\,x))\,(\lambda x\,.\,E\,(x\,x))$
  $\longrightarrow_\beta E\,((\lambda x\,.\,E\,(x\,x))\,(\lambda x\,.\,E\,(x\,x))) \equiv_\beta E\,(Y\,E)$ because in the second β-reduction $x \notin fvar(E)$ otherwise we would have renamed λx in either substitution $(\lambda x\,.\,f\,(x\,x))[^{E}/_{f}]$ of the first β-reduction.
- $Y\,E \not\longrightarrow_\beta^* E\,(Y\,E)$ because in the final step
  $E\,((\lambda x\,.\,E\,(x\,x))\,(\lambda x\,.\,E\,(x\,x))) \equiv_\beta E\,(Y\,E)$
  we have exploited the outcome of the initial β-reduction
  $Y\,E \longrightarrow_\beta (\lambda x\,.\,E\,(x\,x))\,(\lambda x\,.\,E\,(x\,x))$, not the definition of Y.

- Definition of factorial:
  - Let $\underline{test_0} = \lambda n \,.\, \lambda p \,.\, \lambda q \,.\, n\,(\underline{K}\,q)\,p$ where $\underline{K} = \lambda x \,.\, \lambda y \,.\, x$ (when applied to $\underline{n}$ it behaves as p if $\underline{n}$ is $\underline{0}$, q otherwise).
  - Let $\underline{F} = \lambda r \,.\, \lambda n \,.\, \underline{test_0}\; n\; \underline{1}\; (\underline{mult}\; n\; (r\; (\underline{pred}\; n)))$.
  - Then $\underline{fact} = \Theta\, \underline{F}$, so that $\underline{fact}\; \underline{n} \longrightarrow_\beta^* \underline{n!}$.

- Definition of factorial:
  - Let $\underline{test_0} = \lambda n\,.\,\lambda p\,.\,\lambda q\,.\,n\,(\underline{K}\,q)\,p$ where $\underline{K} = \lambda x\,.\,\lambda y\,.\,x$ (when applied to $\underline{n}$ it behaves as p if $\underline{n}$ is $\underline{0}$, q otherwise).
  - Let $\underline{F} = \lambda r\,.\,\lambda n\,.\,\underline{test_0}\,n\,\underline{1}\,(\underline{mult}\,n\,(r\,(\underline{pred}\,n)))$.
  - Then $\underline{fact} = \Theta\,\underline{F}$, so that $\underline{fact}\,\underline{n} \longrightarrow_\beta^* \underline{n!}$.
- Example of application of factorial:
  - $\underline{fact}\,\underline{2} = \Theta\,\underline{F}\,\underline{2}$
    $\longrightarrow_\beta^* \underline{F}\,(\Theta\,\underline{F})\,\underline{2}$
    $\longrightarrow_\beta^* \underline{test_0}\,\underline{2}\,\underline{1}\,(\underline{mult}\,\underline{2}\,(\Theta\,\underline{F}\,(\underline{pred}\,\underline{2})))$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\Theta\,\underline{F}\,\underline{1})$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\underline{F}\,(\Theta\,\underline{F})\,\underline{1})$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\underline{test_0}\,\underline{1}\,\underline{1}\,(\underline{mult}\,\underline{1}\,(\Theta\,\underline{F}\,(\underline{pred}\,\underline{1}))))$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\underline{mult}\,\underline{1}\,(\Theta\,\underline{F}\,\underline{0}))$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\underline{mult}\,\underline{1}\,(\underline{F}\,(\Theta\,\underline{F})\,\underline{0}))$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\underline{mult}\,\underline{1}\,(\underline{test_0}\,\underline{0}\,\underline{1}\,(\underline{mult}\,\underline{0}\,(\Theta\,\underline{F}\,(\underline{pred}\,\underline{0})))))$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,(\underline{mult}\,\underline{1}\,\underline{1})$
    $\longrightarrow_\beta^* \underline{mult}\,\underline{2}\,\underline{1}$
    $\longrightarrow_\beta^* \underline{2}$.

- The idea behind the encoding of the predecessor of $n \in \mathbb{N}_{\geq 1}$ is to generate the sequence of pairs $(0,0), (0,1), (1,2), \ldots, (n-1, n)$ and then return the first component of the last pair:
  - Let $\underline{pro}_{k,i} = \lambda x_1 . \ldots . \lambda x_k . x_i$ for $k \geq 1$ and $1 \leq i \leq k$.
  - Let $\underline{p}_{E_1, E_2} = \langle E_1; E_2 \rangle = \lambda z . z\, E_1\, E_2$ where $z \notin fvar(E_1) \cup fvar(E_2)$.
  - Let $\underline{gp} = \lambda p . \langle p\, \underline{pro}_{2,2}; \underline{succ}\, (p\, \underline{pro}_{2,2}) \rangle$.
  - Then $\underline{pred} = \lambda n . n\, \underline{gp}\, \langle \underline{0}; \underline{0} \rangle\, \underline{pro}_{2,1}$, so that $\underline{pred}\, \underline{n} \longrightarrow\!\!\!\!\!\rightarrow_{\beta}^{*} \underline{n-1}$.

- The idea behind the encoding of the predecessor of $n \in \mathbb{N}_{\geq 1}$ is to generate the sequence of pairs $(0,0), (0,1), (1,2), \ldots, (n-1, n)$ and then return the first component of the last pair:
  - Let $\underline{pro}_{k,i} = \lambda x_1 \,.\, \ldots \,.\, \lambda x_k \,.\, x_i$ for $k \geq 1$ and $1 \leq i \leq k$.
  - Let $\underline{p}_{E_1, E_2} = \langle E_1; E_2 \rangle = \lambda z \,.\, z\, E_1\, E_2$ where $z \notin fvar(E_1) \cup fvar(E_2)$.
  - Let $\underline{gp} = \lambda p \,.\, \langle p\, \underline{pro}_{2,2};\ \underline{succ}\, (p\, \underline{pro}_{2,2}) \rangle$.
  - Then $\underline{pred} = \lambda n \,.\, n\, \underline{gp}\, \langle \underline{0}; \underline{0} \rangle\, \underline{pro}_{2,1}$, so that $\underline{pred}\, \underline{n} \longrightarrow_\beta^* \underline{n-1}$.
- Subtraction and division can thus be recursively defined as follows:
  - Let $\underline{S} = \lambda r \,.\, \lambda m \,.\, \lambda n \,.\, \underline{test}_0\, n\, m\, (r\, (\underline{pred}\, m)\, (\underline{pred}\, n))$.
  - Then $\underline{subtr} = \Theta\, \underline{S}$, so that $\underline{subtr}\, \underline{m}\, \underline{n} \longrightarrow_\beta^* \underline{m-n}$.

- The idea behind the encoding of the predecessor of $n \in \mathbb{N}_{\geq 1}$ is to generate the sequence of pairs $(0,0), (0,1), (1,2), \ldots, (n-1, n)$ and then return the first component of the last pair:
  - Let $\underline{pro}_{k,i} = \lambda x_1 . \ldots . \lambda x_k . x_i$ for $k \geq 1$ and $1 \leq i \leq k$.
  - Let $\underline{p}_{E_1,E_2} = \langle E_1; E_2 \rangle = \lambda z . z\, E_1\, E_2$ where $z \notin fvar(E_1) \cup fvar(E_2)$.
  - Let $\underline{gp} = \lambda p . \langle p\, \underline{pro}_{2,2}; \underline{succ}\, (p\, \underline{pro}_{2,2}) \rangle$.
  - Then $\underline{pred} = \lambda n . n\, \underline{gp}\, \langle \underline{0}; \underline{0} \rangle\, \underline{pro}_{2,1}$, so that $\underline{pred}\, \underline{n} \longrightarrow_\beta^* \underline{n-1}$.
- Subtraction and division can thus be recursively defined as follows:
  - Let $\underline{S} = \lambda r . \lambda m . \lambda n . \underline{test}_0\, n\, m\, (r\, (\underline{pred}\, m)\, (\underline{pred}\, n))$.
  - Then $\underline{subtr} = \Theta\, \underline{S}$, so that $\underline{subtr}\, \underline{m}\, \underline{n} \longrightarrow_\beta^* \underline{m-n}$.
  - Let $\underline{test}_{\leq} = \Theta\, \underline{T}_{\leq}$ where $\underline{T}_{\leq} = \lambda r . \lambda m . \lambda n . \lambda p . \lambda q . \underline{test}_0\, m\, p\, (\underline{test}_0\, n\, q\, (r\, (\underline{pred}\, m)\, (\underline{pred}\, n)\, p\, q))$.
  - Let $\underline{D} = \lambda r . \lambda m . \lambda n . \underline{test}_{\leq}\, (\underline{succ}\, m)\, n\, \underline{0}\, (\underline{succ}\, (r\, (\underline{subtr}\, m\, n)\, n))$ where the initial test $m < n$ is implemented as $m + 1 \leq n$.
  - Then $\underline{div} = \Theta\, \underline{D}$, so that $\underline{div}\, \underline{m}\, \underline{n} \longrightarrow_\beta^* \underline{m : n}$.

4.4 Termination and Confluence

- β-reduction is not terminating as witnessed by $\underline{\omega} = \lambda x \,.\, x\, x$ because $\underline{\omega}\,\underline{\omega} = (\lambda x \,.\, x\, x)\,\underline{\omega} \longrightarrow_\beta (x\, x)[{}^{\underline{\omega}}/_{x}] = \underline{\omega}\,\underline{\omega} \longrightarrow_\beta \ldots$
- The absence of redexes in a λ-term provides a clear evidence of the *finality* of the term itself from a computational viewpoint, like the number obtained from an arithmetical expression.

4.4 Termination and Confluence

- β-reduction is not terminating as witnessed by $\underline{\omega} = \lambda x \,.\, x\, x$ because $\underline{\omega}\,\underline{\omega} = (\lambda x \,.\, x\, x)\,\underline{\omega} \longrightarrow_\beta (x\, x)[{}^{\underline{\omega}}/_{x}] = \underline{\omega}\,\underline{\omega} \longrightarrow_\beta \ldots$
- The absence of redexes in a λ-term provides a clear evidence of the *finality* of the term itself from a computational viewpoint, like the number obtained from an arithmetical expression.
- A λ-term is in normal form iff it contains no redexes, in which case it is generated by:

$$\begin{array}{rcl} N & ::= & x \mid \lambda x \,.\, N \mid x\, F \\ F & ::= & x \mid \lambda x \,.\, N \mid F\, F \end{array}$$

4.4 Termination and Confluence

- β-reduction is not terminating as witnessed by $\underline{\omega} = \lambda x \,.\, x\, x$ because $\underline{\omega}\,\underline{\omega} = (\lambda x \,.\, x\, x)\,\underline{\omega} \longrightarrow_\beta (x\, x)[{}^{\underline{\omega}}/_{x}] = \underline{\omega}\,\underline{\omega} \longrightarrow_\beta \ldots$
- The absence of redexes in a λ-term provides a clear evidence of the *finality* of the term itself from a computational viewpoint, like the number obtained from an arithmetical expression.
- A λ-term is in normal form iff it contains no redexes, in which case it is generated by:

$$\begin{array}{rcl} N & ::= & x \mid \lambda x \,.\, N \mid x\, F \\ F & ::= & x \mid \lambda x \,.\, N \mid F\, F \end{array}$$

- $E \in \Lambda$ admits normal form iff there exists $E' \in \Lambda$ in normal form such that $E \longrightarrow_\beta^* E'$.
- Not all λ-terms admit normal form, as shown by $\underline{\omega}\,\underline{\omega}$.
- Analogous to the divergence of a Turing machine on an input.

- β-reduction is not deterministic, i.e., it is not defined a priori a reduction strategy that, in the presence of several redexes, establishes to which redex to apply β-reduction first.
- In principle it is not guaranteed that the normal form is reached for a term owning it, nor that the normal form is unique when it exists.

- β-reduction is not deterministic, i.e., it is not defined a priori a reduction strategy that, in the presence of several redexes, establishes to which redex to apply β-reduction first.
- In principle it is not guaranteed that the normal form is reached for a term owning it, nor that the normal form is unique when it exists.
- In the presence of several redexes, one can choose between reducing the outermost or innermost ones w.r.t. the syntactical structure.
- In $(\lambda x \,.\, E)\, F$ one can apply β-reduction to the entire term before or after applying it to possible redexes inside E and inside F.

- β-reduction is not deterministic, i.e., it is not defined a priori a reduction strategy that, in the presence of several redexes, establishes to which redex to apply β-reduction first.
- In principle it is not guaranteed that the normal form is reached for a term owning it, nor that the normal form is unique when it exists.
- In the presence of several redexes, one can choose between reducing the outermost or innermost ones w.r.t. the syntactical structure.
- In $(\lambda x \,.\, E)\, F$ one can apply β-reduction to the entire term before or after applying it to possible redexes inside E and inside F.
- In the presence of several redexes at the same syntactical level, one can choose between reducing the leftmost or rightmost one.
- In $E_1\, E_2$, where E_1 is not a λ-abstraction, one can apply β-reduction to possible redexes inside E_1 first or inside E_2 first.

- Call by name: systematically reduce the leftmost redex among the outermost ones, which corresponds to passing the arguments to the function without evaluating them: $\underline{\omega}\,(\underline{I}\,\underline{I}) = (\lambda x\,.\,x\,x)\,(\underline{I}\,\underline{I})$ $\longrightarrow_\beta (\underline{I}\,\underline{I})\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.

- Call by name: systematically reduce the leftmost redex among the outermost ones, which corresponds to passing the arguments to the function without evaluating them: $\underline{\omega}\,(\underline{I}\,\underline{I}) = (\lambda x\,.\,x\,x)\,(\underline{I}\,\underline{I})$ $\longrightarrow_\beta (\underline{I}\,\underline{I})\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.
- Call by value: systematically reduce the leftmost redex among the innermost ones, which corresponds to evaluating the arguments before passing them to the function: $\underline{\omega}\,(\underline{I}\,\underline{I}) = \underline{\omega}\,((\lambda x\,.\,x)\,\underline{I})$ $\longrightarrow_\beta \underline{\omega}\,\underline{I} \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.

- Call by name: systematically reduce the leftmost redex among the outermost ones, which corresponds to passing the arguments to the function without evaluating them: $\underline{\omega}\,(\underline{I}\,\underline{I}) = (\lambda x\,.\,x\,x)\,(\underline{I}\,\underline{I})$ $\longrightarrow_\beta (\underline{I}\,\underline{I})\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.
- Call by value: systematically reduce the leftmost redex among the innermost ones, which corresponds to evaluating the arguments before passing them to the function: $\underline{\omega}\,(\underline{I}\,\underline{I}) = \underline{\omega}\,((\lambda x\,.\,x)\,\underline{I})$ $\longrightarrow_\beta \underline{\omega}\,\underline{I} \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.
- Strong confluence or diamond property holds iff for all $E \in \Lambda$ $E \longrightarrow_\beta E_1 \wedge E \longrightarrow_\beta E_2 \implies E_1 \longrightarrow_\beta E' \wedge E_2 \longrightarrow_\beta E'$.
- Weak confluence or Church-Rosser property holds iff for all $E \in \Lambda$ $E \longrightarrow_\beta^* E_1 \wedge E \longrightarrow_\beta^* E_2 \implies E_1 \longrightarrow_\beta^* E' \wedge E_2 \longrightarrow_\beta^* E'$.
- Strong confluence implies weak one but does *not* hold in λ-calculus as shown by the two reduction strategies above applied to $\underline{\omega}\,(\underline{I}\,\underline{I})$.

- Strip lemma (confluence property valid for λ-calculus): for all $E \in \Lambda$ $E \longrightarrow_\beta E_1 \wedge E \longrightarrow_\beta^* E_2 \implies E_1 \longrightarrow_\beta^* E' \wedge E_2 \longrightarrow_\beta^* E'$.
- Church-Rosser theorem: λ-calculus fulfills weak confluence.
- Corollary (normal form uniqueness): if $E \in \Lambda$ admits normal form, then this is unique up to α-conversion.
- Corollary (consistency): the β-equivalence theory is consistent, i.e., $E_1 \equiv_\beta E_2$ does not hold for all $E_1, E_2 \in \Lambda$.

- Strip lemma (confluence property valid for λ-calculus): for all $E \in \Lambda$ $E \longrightarrow_\beta E_1 \wedge E \longrightarrow_\beta^* E_2 \implies E_1 \longrightarrow_\beta^* E' \wedge E_2 \longrightarrow_\beta^* E'$.
- Church-Rosser theorem: λ-calculus fulfills weak confluence.
- Corollary (normal form uniqueness): if $E \in \Lambda$ admits normal form, then this is unique up to α-conversion.
- Corollary (consistency): the β-equivalence theory is consistent, i.e., $E_1 \equiv_\beta E_2$ does not hold for all $E_1, E_2 \in \Lambda$.
- When existing, the normal form is *always reached via call by name*: $(\lambda y \,.\, z)\,(\underline{\omega}\,\underline{\omega})$ β-reduces to the normal form z via call by name, while it β-reduces to itself and hence diverges via call by value.
- When leading to the normal form, the call-by-value strategy may instead turn out to be *more efficient* than the call-by-name strategy: in $(\lambda n \,.\, \underline{add}\, n\, n)\,(\underline{molt}\,\underline{5}\,\underline{4})$ the latter calculates $\underline{molt}\,\underline{5}\,\underline{4}$ twice.
- Combining the two strategies via graph sharing and rewriting.

4.5 Lambda Calculus with Types

- Types were introduced by Russell and Whitehead in their book "Principia Mathematica" to avoid logical paradoxes.
- Nowadays used in all modern programming languages.
- Types permit to classify terms so as to identify sets of terms with similar computational properties and prevent undesired behaviors.
- Kleene-Rosser paradox replicating Richard one, then Curry paradox.

4.5 Lambda Calculus with Types

- Types were introduced by Russell and Whitehead in their book "Principia Mathematica" to avoid logical paradoxes.
- Nowadays used in all modern programming languages.
- Types permit to classify terms so as to identify sets of terms with similar computational properties and prevent undesired behaviors.
- Kleene-Rosser paradox replicating Richard one, then Curry paradox.
- $\mathbb{T}$: countable set of type symbols that is closed with respect to the function type $\tau \to \sigma$, where the type operator $\to$ is right associative like λ-abstraction for consistency with currying.
- Two different kinds of type systems in λ-calculus:
  - Type systems à la Church: every λ-term is defined together with its type so that syntax and semantics include *type checking*.
  - Type systems à la Curry: the type of every λ-term is ascribed via formal rules that *infer the type* from the format of the term.

- The set $\Lambda_{\mathbb{T}}$ of typed λ-terms à la Church is generated by:

$$E \ ::= \ x^{\tau} \mid (\lambda x^{\tau} \,.\, E^{\sigma})^{\tau \to \sigma} \mid (E^{\tau \to \sigma} \ E^{\tau})^{\sigma}$$

- The semantic rules over $\Lambda_{\mathbb{T}}$ include type checking:
  - Typed α-conversion: $\lambda x^{\tau} \,.\, E^{\sigma} =_{\alpha} \lambda y^{\tau} \,.\, (E^{\sigma}[{}^{y^{\tau}}/_{x^{\tau}}])$ with $y^{\tau} \notin fvar(E^{\sigma})$.
  - Typed β-conversion: $(\lambda x^{\tau} \,.\, E^{\sigma}) \, F^{\tau} =_{\beta} E^{\sigma}[{}^{F^{\tau}}/_{x^{\tau}}]$.
  - Typed η-conversion: $\lambda x^{\tau} \,.\, E^{\sigma} \, x^{\tau} =_{\eta} E^{\sigma}$ with $x^{\tau} \notin fvar(E^{\sigma})$.

- The set $\Lambda_{\mathbb{T}}$ of typed λ-terms à la Church is generated by:

$$E \ ::= \ x^{\tau} \mid (\lambda x^{\tau} . E^{\sigma})^{\tau \to \sigma} \mid (E^{\tau \to \sigma} \, E^{\tau})^{\sigma}$$

- The semantic rules over $\Lambda_{\mathbb{T}}$ include type checking:
  - Typed α-conversion: $\lambda x^{\tau} . E^{\sigma} =_{\alpha} \lambda y^{\tau} . (E^{\sigma}[{}^{y^{\tau}}/_{x^{\tau}}])$ with $y^{\tau} \notin fvar(E^{\sigma})$.
  - Typed β-conversion: $(\lambda x^{\tau} . E^{\sigma}) \, F^{\tau} =_{\beta} E^{\sigma}[{}^{F^{\tau}}/_{x^{\tau}}]$.
  - Typed η-conversion: $\lambda x^{\tau} . E^{\sigma} \, x^{\tau} =_{\eta} E^{\sigma}$ with $x^{\tau} \notin fvar(E^{\sigma})$.
- Let $\Gamma = \{x_i : \tau_i \mid x_i \in Var, \tau_i \in \mathbb{T}, 1 \leq i \leq n, x_i \neq x_j \text{ for } i \neq j\}$ be a basis of type declarations for variables.
- Type inference rules à la Curry:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \cup \{x : \tau\} \vdash E : \sigma}{\Gamma \vdash (\lambda x . E) : \tau \to \sigma} \qquad \frac{\Gamma \vdash E_1 : \tau \to \sigma \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash (E_1 \, E_2) : \sigma}$$

- The set $\Lambda_{\mathbb{T}}$ of typed λ-terms à la Church is generated by:
$$E \ ::= \ x^{\tau} \mid (\lambda x^{\tau} \,.\, E^{\sigma})^{\tau \to \sigma} \mid (E^{\tau \to \sigma} \, E^{\tau})^{\sigma}$$
- The semantic rules over $\Lambda_{\mathbb{T}}$ include type checking:
  - Typed α-conversion: $\lambda x^{\tau} \,.\, E^{\sigma} =_{\alpha} \lambda y^{\tau} \,.\, (E^{\sigma}[{}^{y^{\tau}}/_{x^{\tau}}])$ with $y^{\tau} \notin fvar(E^{\sigma})$.
  - Typed β-conversion: $(\lambda x^{\tau} \,.\, E^{\sigma}) \, F^{\tau} =_{\beta} E^{\sigma}[{}^{F^{\tau}}/_{x^{\tau}}]$.
  - Typed η-conversion: $\lambda x^{\tau} \,.\, E^{\sigma} \, x^{\tau} =_{\eta} E^{\sigma}$ with $x^{\tau} \notin fvar(E^{\sigma})$.
- Let $\Gamma = \{x_i : \tau_i \mid x_i \in Var, \tau_i \in \mathbb{T}, 1 \leq i \leq n, x_i \neq x_j \text{ for } i \neq j\}$ be a basis of type declarations for variables.
- Type inference rules à la Curry:
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \cup \{x : \tau\} \vdash E : \sigma}{\Gamma \vdash (\lambda x \,.\, E) : \tau \to \sigma} \qquad \frac{\Gamma \vdash E_1 : \tau \to \sigma \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash (E_1 \, E_2) : \sigma}$$
- Theorem (equivalence of type systems à la Church and à la Curry): $E^{\tau} \in \Lambda_{\mathbb{T}}$ iff $\{x_i : \tau_i \mid x_i^{\tau_i} \in fvar(E^{\tau})\} \vdash |E^{\tau}| : \tau$ where $|E^{\tau}|$ is the λ-term obtained from E^{τ} by eliminating all types occurring in it.

- Every term in $\Lambda_{\mathbb{T}}$ is strongly normalizable, i.e., *any* reduction strategy applied to it terminates and produces its normal form.
- $\Lambda_{\mathbb{T}} \subsetneq \Lambda$, i.e., not all λ-terms can be typed: e.g., in $\underline{\omega} = \lambda x \,.\, x\, x$ x should simultaneously be of type $\tau \to \sigma$ and of type τ.

- Every term in $\Lambda_{\mathbb{T}}$ is strongly normalizable, i.e., *any* reduction strategy applied to it terminates and produces its normal form.
- $\Lambda_{\mathbb{T}} \subsetneq \Lambda$, i.e., not all λ-terms can be typed: e.g., in $\underline{\omega} = \lambda x \,.\, x\, x$ x should simultaneously be of type $\tau \to \sigma$ and of type τ.
- Not even the fixed point combinators Θ and Y are definable in $\Lambda_{\mathbb{T}}$.
- Since they are necessary for implementing recursion, $\Lambda_{\mathbb{T}}$ is extended with constants among which F_τ of type $(\tau \to \tau) \to \tau$ for each $\tau \in \mathbb{T}$, together with the following fixed point rule:
  - δ-conversion: $F_\tau\, E^{\tau\to\tau} =_\delta E^{\tau\to\tau}\, (F_\tau\, E^{\tau\to\tau})$.

- Every term in $\Lambda_{\mathbb{T}}$ is strongly normalizable, i.e., *any* reduction strategy applied to it terminates and produces its normal form.
- $\Lambda_{\mathbb{T}} \subsetneq \Lambda$, i.e., not all λ-terms can be typed: e.g., in $\underline{\omega} = \lambda x \,.\, x\, x$ x should simultaneously be of type $\tau \to \sigma$ and of type τ.
- Not even the fixed point combinators Θ and Y are definable in $\Lambda_{\mathbb{T}}$.
- Since they are necessary for implementing recursion, $\Lambda_{\mathbb{T}}$ is extended with constants among which F_τ of type $(\tau \to \tau) \to \tau$ for each $\tau \in \mathbb{T}$, together with the following fixed point rule:
  - δ-conversion: $F_\tau\, E^{\tau\to\tau} =_\delta E^{\tau\to\tau}\, (F_\tau\, E^{\tau\to\tau})$.
- Type inference rules à la Curry result in Curry-Howard isomorphism:
  - Types correspond to logic formulas (formulas-as-types analogy):
    - The type inference rule for application corresponds to modus ponens.
    - The types of combinatory logic operators correspond to Hilbert axioms: $\underline{I} : \tau \to \tau$, $\underline{K} : \tau \to \sigma \to \tau$, $\underline{S} : (\tau \to \sigma \to \rho) \to (\tau \to \sigma) \to \tau \to \rho$.
  - λ-terms of those types correspond to validity proofs of those formulas.
  - β-reduction corresponds to composition of proofs based on the cut rule.

5. Computability for Functions, Sets, Problems

Topics:

5.1 *Church-Turing Thesis* 70

5.2 *Primitive and General Recursive Functions* 72

5.3 *Recursive and Recursively Enumerable Sets* 80

5.4 *Decidable and Undecidable Problems* 83

5.5 *Tractable and Intractable Problems* 87

5.1 Church-Turing Thesis

- Can we characterize the set of Turing-computable functions?
- Can we characterize the set of λ-definable functions?

5.1 Church-Turing Thesis

- Can we characterize the set of Turing-computable functions?
- Can we characterize the set of λ-definable functions?
- General recursive functions are functions over naturals introduced by Kurt Gödel and Jacques Herbrand (1931).
- Stephen Kleene proved that general recursive functions are λ-definable (1936).
- Alan Turing proved that λ-definable functions in turn are Turing-computable (1937).
- Both Turing-computable functions and λ-definable functions coincide with general recursive functions.

- Are there computable functions beyond the previous ones?

- Are there computable functions beyond the previous ones?
- *Effective method*: finite sequence of instructions that can be unambiguously interpreted, whose execution always terminates in a finite amount of time by producing the correct outcome.
- The results of Kleene and Turing led to think that the set of functions computable via *any* effective method are equivalently characterized by Turing-computability, λ-definability, and general recursion.

- Are there computable functions beyond the previous ones?
- *Effective method*: finite sequence of instructions that can be unambiguously interpreted, whose execution always terminates in a finite amount of time by producing the correct outcome.
- The results of Kleene and Turing led to think that the set of functions computable via *any* effective method are equivalently characterized by Turing-computability, λ-definability, and general recursion.
- Church thesis: The functions over naturals that are computable through any effective method are exactly the λ-definable ones.
- Turing thesis: The functions over naturals that are computable through any effective method are exactly the Turing-computable ones.
- Theses, not conjectures, as the notion of effective method is informal.

5.2 Primitive and General Recursive Functions

- *Basic* primitive recursive functions:
  - Zero function: $0_k(x_1, \ldots, x_k) = 0$ for $k \geq 0$.
  - Successor function: $succ(x) = x + 1$.
  - Projection function: $pro_{k,i}(x_1, \ldots, x_k) = x_i$ for $k \geq 1$, $1 \leq i \leq k$.

5.2 Primitive and General Recursive Functions

- *Basic* primitive recursive functions:
  - Zero function: $0_k(x_1, \ldots, x_k) = 0$ for $k \geq 0$.
  - Successor function: $succ(x) = x + 1$.
  - Projection function: $pro_{k,i}(x_1, \ldots, x_k) = x_i$ for $k \geq 1$, $1 \leq i \leq k$.
- *Closure* with respect to:
  - Composition: if $h(y_1, \ldots, y_m)$ is primitive recursive and $g_i(x_1, \ldots, x_k)$ is primitive recursive for all $1 \leq i \leq m$, then $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$ is primitive recursive too.

5.2 Primitive and General Recursive Functions

- *Basic* primitive recursive functions:
  - Zero function: $0_k(x_1, \ldots, x_k) = 0$ for $k \geq 0$.
  - Successor function: $succ(x) = x + 1$.
  - Projection function: $pro_{k,i}(x_1, \ldots, x_k) = x_i$ for $k \geq 1$, $1 \leq i \leq k$.
- *Closure* with respect to:
  - Composition: if $h(y_1, \ldots, y_m)$ is primitive recursive and $g_i(x_1, \ldots, x_k)$ is primitive recursive for all $1 \leq i \leq m$, then $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$ is primitive recursive too.
  - Primitive recursion: if $h(x_1, \ldots, x_m)$ and $g(y, k, x_1, \ldots, x_m)$ are primitive recursive, then $f(k, x_1, \ldots, x_m)$ defined by letting:

$$\begin{aligned} f(0, x_1, \ldots, x_m) &= h(x_1, \ldots, x_m) \\ f(k+1, x_1, \ldots, x_m) &= g(f(k, x_1, \ldots, x_m), k, x_1, \ldots, x_m) \end{aligned}$$

    is primitive recursive too (h is the base case, g is the induction case).

- Encoding $\underline{f}$ in λ-calculus of function f: for all $k, n_1, \ldots, n_k, m \in \mathbb{N}$, if $f(n_1, \ldots, n_k) = m$ then $\underline{f}\,\underline{n}_1 \ \ldots \ \underline{n}_k \longrightarrow\!\!\!\!\!\!\beta^{*}\ \underline{m}$.

- Encoding $\underline{f}$ in λ-calculus of function f: for all $k, n_1, \ldots, n_k, m \in \mathbb{N}$, if $f(n_1, \ldots, n_k) = m$ then $\underline{f}\,\underline{n}_1 \ \ldots \ \underline{n}_k \longrightarrow_\beta^* \underline{m}$.
- Encoding of basic functions:
  - $\underline{0}_k = \lambda x_1 \,.\, \ldots \,.\, \lambda x_k \,.\, \lambda s \,.\, \lambda z \,.\, z$.
  - $\underline{succ} = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x\,(n\,x\,y)$.
  - $\underline{pro}_{k,i} = \lambda x_1 \,.\, \ldots \,.\, \lambda x_k \,.\, x_i$.

- Encoding $\underline{f}$ in λ-calculus of function f: for all $k, n_1, \ldots, n_k, m \in \mathbb{N}$, if $f(n_1, \ldots, n_k) = m$ then $\underline{f}\,\underline{n}_1 \ \ldots \ \underline{n}_k \longrightarrow\!\!\!\!\!\!\beta^{*}\ \underline{m}$.
- Encoding of basic functions:
  - $\underline{0}_k = \lambda x_1 . \ldots . \lambda x_k . \lambda s . \lambda z . z$.
  - $\underline{succ} = \lambda n . \lambda x . \lambda y . x\,(n\,x\,y)$.
  - $\underline{pro}_{k,i} = \lambda x_1 . \ldots . \lambda x_k . x_i$.
- Encoding of composition:
  - $\underline{f} = \lambda x_1 . \ldots . \lambda x_k . \underline{h}\,(\underline{g}_1 x_1 . \ldots . x_k) \ \ldots \ (\underline{g}_m x_1 . \ldots . x_k)$.

- Encoding $\underline{f}$ in λ-calculus of function f: for all $k, n_1, \ldots, n_k, m \in \mathbb{N}$, if $f(n_1, \ldots, n_k) = m$ then $\underline{f}\,\underline{n}_1\ \ldots\ \underline{n}_k \longrightarrow\!\!\!\!\!\!\beta^*\ \underline{m}$.
- Encoding of basic functions:
  - $\underline{0}_k = \lambda x_1 . \ldots . \lambda x_k . \lambda s . \lambda z . z$.
  - $\underline{succ} = \lambda n . \lambda x . \lambda y . x\,(n\,x\,y)$.
  - $\underline{pro}_{k,i} = \lambda x_1 . \ldots . \lambda x_k . x_i$.
- Encoding of composition:
  - $\underline{f} = \lambda x_1 . \ldots . \lambda x_k . \underline{h}\,(\underline{g}_1 x_1 . \ldots . x_k)\ \ldots\ (\underline{g}_m x_1 . \ldots . x_k)$.
- The idea behind the encoding of primitive recursion is to start from the triple corresponding to f evaluated at 0:

$$(0;\ h(x_1, \ldots, x_m);\ g(h(x_1, \ldots, x_m), 0, x_1, \ldots, x_m)) = (0;\ f(0, x_1, \ldots, x_m);\ f(1, x_1, \ldots, x_m))$$

  then generate a sequence of k triples where the second component of the last triple is the result to return for f evaluated at k:

$$\begin{array}{r} (1;\ f(1, x_1, \ldots, x_m);\ g(f(1, x_1, \ldots, x_m), 1, x_1, \ldots, x_m)) = (1;\ f(1, x_1, \ldots, x_m);\ f(2, x_1, \ldots, x_m)) \\ (2;\ f(2, x_1, \ldots, x_m);\ g(f(2, x_1, \ldots, x_m), 2, x_1, \ldots, x_m)) = (2;\ f(2, x_1, \ldots, x_m);\ f(3, x_1, \ldots, x_m)) \\ \ldots = \ldots \\ (k;\ f(k, x_1, \ldots, x_m);\ g(f(k, x_1, \ldots, x_m), k, x_1, \ldots, x_m)) = (k;\ f(k, x_1, \ldots, x_m);\ f(k+1, x_1, \ldots, x_m)) \end{array}$$

- Encoding of primitive recursion:
  - Let $\underline{t}_{E_1,E_2,E_3} = \langle E_1; E_2; E_3 \rangle = \lambda z \,.\, z\, E_1\, E_2\, E_3$ with $z \notin fvar(E_i)$ for $1 \leq i \leq 3$, where:
    - $\underline{t}_{E_1,E_2,E_3}\ \underline{pro}_{3,1} \longrightarrow_\beta^* E_1.$
    - $\underline{t}_{E_1,E_2,E_3}\ \underline{pro}_{3,2} \longrightarrow_\beta^* E_2.$
    - $\underline{t}_{E_1,E_2,E_3}\ \underline{pro}_{3,3} \longrightarrow_\beta^* E_3.$

- Encoding of primitive recursion:
  - Let $\underline{t}_{E_1,E_2,E_3} = \langle E_1; E_2; E_3 \rangle = \lambda z \,.\, z\, E_1\, E_2\, E_3$ with $z \notin fvar(E_i)$ for $1 \leq i \leq 3$, where:
    - $\underline{t}_{E_1,E_2,E_3}\, \underline{pro}_{3,1} \longrightarrow_\beta^* E_1$.
    - $\underline{t}_{E_1,E_2,E_3}\, \underline{pro}_{3,2} \longrightarrow_\beta^* E_2$.
    - $\underline{t}_{E_1,E_2,E_3}\, \underline{pro}_{3,3} \longrightarrow_\beta^* E_3$.
  - Let $\underline{h}$, $\underline{g}$ be the encodings of the primitive recursive functions h, g.
  - Let $\underline{gt} = \lambda t \,.$
    $\langle \underline{succ}\,(t\, \underline{pro}_{3,1});\ t\, \underline{pro}_{3,3};\ \underline{g}\,(t\, \underline{pro}_{3,3})\,(\underline{succ}\,(t\, \underline{pro}_{3,1}))\, x_1\ \ldots\ x_m \rangle$.
  - Then $\underline{f} = \lambda k \,.\, \lambda x_1 \,.\, \ldots \,.\, \lambda x_m \,.$
    $k\, \underline{gt}\, \langle \underline{0};\ \underline{h}\, x_1\ \ldots\ x_m;\ \underline{g}\,(\underline{h}\, x_1\ \ldots\ x_m)\, \underline{0}\, x_1\ \ldots\ x_m \rangle\, \underline{pro}_{3,2}$
    where k acts as an iterator of the triple generator.

- Many largely used functions are primitive recursive:
  - $1_k(x_1, \ldots, x_k) = succ(0_k(x_1, \ldots, x_k))$
    $2_k(x_1, \ldots, x_k) = succ(1_k(x_1, \ldots, x_k))$
    and so on.

- Many largely used functions are primitive recursive:
  - $1_k(x_1, \ldots, x_k) = succ(0_k(x_1, \ldots, x_k))$
    $2_k(x_1, \ldots, x_k) = succ(1_k(x_1, \ldots, x_k))$
    and so on.
  - $add(0, x) = pro_{1,1}(x)$
    $add(k + 1, x) = g(add(k, x), k, x)$
    where $g(y, k, x) = succ(pro_{3,1}(y, k, x))$.

- Many largely used functions are primitive recursive:
  - $1_k(x_1, \ldots, x_k) = succ(0_k(x_1, \ldots, x_k))$
    $2_k(x_1, \ldots, x_k) = succ(1_k(x_1, \ldots, x_k))$
    and so on.
  - $add(0, x) = pro_{1,1}(x)$
    $add(k+1, x) = g(add(k, x), k, x)$
    where $g(y, k, x) = succ(pro_{3,1}(y, k, x))$.
  - $mult(0, x) = 0_1(x)$
    $mult(k+1, x) = g(mult(k, x), k, x)$
    where $g(y, k, x) = add(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.

- Many largely used functions are primitive recursive:
  - $1_k(x_1, \ldots, x_k) = succ(0_k(x_1, \ldots, x_k))$
    $2_k(x_1, \ldots, x_k) = succ(1_k(x_1, \ldots, x_k))$
    and so on.
  - $add(0, x) = pro_{1,1}(x)$
    $add(k+1, x) = g(add(k, x), k, x)$
    where $g(y, k, x) = succ(pro_{3,1}(y, k, x))$.
  - $mult(0, x) = 0_1(x)$
    $mult(k+1, x) = g(mult(k, x), k, x)$
    where $g(y, k, x) = add(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.
  - $exp(0, x) = 1_1(x)$
    $exp(k+1, x) = g(exp(k, x), k, x)$
    where $g(y, k, x) = mult(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.

- Many largely used functions are primitive recursive:
  - $1_k(x_1, \ldots, x_k) = succ(0_k(x_1, \ldots, x_k))$
    $2_k(x_1, \ldots, x_k) = succ(1_k(x_1, \ldots, x_k))$
    and so on.
  - $add(0, x) = pro_{1,1}(x)$
    $add(k+1, x) = g(add(k, x), k, x)$
    where $g(y, k, x) = succ(pro_{3,1}(y, k, x))$.
  - $mult(0, x) = 0_1(x)$
    $mult(k+1, x) = g(mult(k, x), k, x)$
    where $g(y, k, x) = add(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.
  - $exp(0, x) = 1_1(x)$
    $exp(k+1, x) = g(exp(k, x), k, x)$
    where $g(y, k, x) = mult(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.
  - $fact(0) = 1_0$
    $fact(k+1) = g(fact(k), k)$
    where $g(y, k) = mult(pro_{2,1}(y, k), g'(y, k))$
    with $g'(y, k) = succ(pro_{2,2}(y, k))$.

- A function $f(k, x_1, \ldots, x_m)$ obtained by primitive recursion from $h(x_1, \ldots, x_m)$ and $g(y, k, x_1, \ldots, x_m)$ is *not inherently recursive*.
- The value of f at k can always be computed in an *iterative* way like in the encoding of primitive recursion in λ-calculus.

- A function $f(k, x_1, \ldots, x_m)$ obtained by primitive recursion from $h(x_1, \ldots, x_m)$ and $g(y, k, x_1, \ldots, x_m)$ is *not inherently recursive*.
- The value of f at k can always be computed in an *iterative* way like in the encoding of primitive recursion in λ-calculus.
- Iterative algorithm:
  - Initialize f' by letting $f' = h(x_1, \ldots, x_m)$.
  - Initialize f'' by letting $f'' = g(f', 0, x_1, \ldots, x_m)$.
  - For each i from 1 to k repeat:
    - Update f' by letting $f' = f''$.
    - Update f'' by letting $f'' = g(f', i, x_1, \ldots, x_m)$.
  - Return f'.
- It is thus reasonable to expect that not all the recursive functions can be expressed as primitive recursive functions.

- The *Ackermann function* is not primitive recursive:

$$\begin{aligned} A(0,0,y) &= y \\ A(0,x+1,y) &= A(0,x,y)+1 \\ A(1,0,y) &= 0 \\ A(k+2,0,y) &= 1 \\ A(k+1,x+1,y) &= A(k,A(k+1,x,y),y) \end{aligned}$$

- The *Ackermann function* is not primitive recursive:

$$\begin{aligned} A(0,0,y) &= y \\ A(0,x+1,y) &= A(0,x,y)+1 \\ A(1,0,y) &= 0 \\ A(k+2,0,y) &= 1 \\ A(k+1,x+1,y) &= A(k,A(k+1,x,y),y) \end{aligned}$$

- It is called *generalized exponential* because it grows more rapidly than any primitive recursive function:

$$\begin{aligned} A(0,x,y) &= x+y \\ A(1,x,y) &= x \cdot y \\ A(2,x,y) &= y^x \\ A(3,x,y) &= y^{y^{\cdot^{\cdot^{\cdot^{y}}}}} \quad \text{where the number of exponents } y \text{ is equal to } x \end{aligned}$$

- A function $f(x_1, \ldots, x_m)$ is general recursive iff it is obtained by *minimization* from a primitive recursive function $h(k, x_1, \ldots, x_m)$: $f(x_1, \ldots, x_m) = \min\{k \in \mathbb{N} \mid h(k, x_1, \ldots, x_m) = 0\}$.
- The Ackermann function can be expressed in that way.

- A function $f(x_1, \ldots, x_m)$ is general recursive iff it is obtained by *minimization* from a primitive recursive function $h(k, x_1, \ldots, x_m)$: $f(x_1, \ldots, x_m) = \min\{k \in \mathbb{N} \mid h(k, x_1, \ldots, x_m) = 0\}$.
- The Ackermann function can be expressed in that way.
- The idea behind the encoding of general recursion is to generate the sequence $h(0, x_1, \ldots, x_m), h(1, x_1, \ldots, x_m), \ldots$
  until we reach the first value equal to 0 in the sequence if any.
- If 0 is not encountered, the computation diverges.
- The function generating the considered sequence is the fixed point of a suitable higher-order function based on a test for 0.

- Encoding of general recursion:
  - Let $\underline{test_0} = \lambda n \,.\, \lambda p \,.\, \lambda q \,.\, n\,(\underline{K}\,q)\,p$ with $\underline{K} = \lambda x \,.\, \lambda y \,.\, x$
    so $\underline{test_0}\,\underline{0}\,E\,F \longrightarrow_\beta \underline{0}\,(\underline{K}\,F)\,E \longrightarrow_\beta E$
    while $\underline{test_0}\,\underline{n}\,E\,F \longrightarrow_\beta \underline{n}\,(\underline{K}\,F)\,E \longrightarrow_\beta \underline{n}\,(\lambda y \,.\, F)\,E \longrightarrow_\beta^* F$
    if $n > 0$ because $y \notin fvar(F)$.

- Encoding of general recursion:
  - Let $\underline{test_0} = \lambda n\,.\,\lambda p\,.\,\lambda q\,.\,n\,(\underline{K}\,q)\,p$ with $\underline{K} = \lambda x\,.\,\lambda y\,.\,x$
    so $\underline{test_0}\,\underline{0}\,E\,F \longrightarrow_\beta \underline{0}\,(\underline{K}\,F)\,E \longrightarrow_\beta E$
    while $\underline{test_0}\,\underline{n}\,E\,F \longrightarrow_\beta \underline{n}\,(\underline{K}\,F)\,E \longrightarrow_\beta \underline{n}\,(\lambda y\,.\,F)\,E \longrightarrow_\beta^* F$
    if $n > 0$ because $y \notin fvar(F)$.
  - Let $\underline{h}$ be the encoding of the primitive recursive function h.
  - Let $\underline{H} = \lambda r\,.\,\lambda k\,.\,\lambda x_1\,.\,\ldots\,.\,\lambda x_m\,.$
    $\underline{test_0}\,(\underline{h}\,k\,x_1\,\ldots\,x_m)\,k\,(r\,(\underline{succ}\,k)\,x_1\,\ldots\,x_m)$.
  - Then $\underline{f} = \lambda x_1\,.\,\ldots\,.\,\lambda x_m\,.\,\Theta\,\underline{H}\,\underline{0}\,x_1\,\ldots\,x_m$.
  - $\underline{f}\,\underline{n_1}\,\ldots\,\underline{n_m} = \Theta\,\underline{H}\,\underline{0}\,\underline{n_1}\,\ldots\,\underline{n_m} \longrightarrow_\beta^* \underline{H}\,(\Theta\,\underline{H})\,\underline{0}\,\underline{n_1}\,\ldots\,\underline{n_m} =$
    $\underline{test_0}\,(\underline{h}\,\underline{0}\,\underline{n_1}\,\ldots\,\underline{n_m})\,\underline{0}\,((\Theta\,\underline{H})\,(\underline{succ}\,\underline{0})\,\underline{n_1}\,\ldots\,\underline{n_m}) \longrightarrow_\beta^* \ldots$

5.3 Recursive and Recursively Enumerable Sets

- How to characterize sets that can be built algorithmically?
- Membership problem: for $A \subseteq \mathbb{N}$ and $n \in \mathbb{N}$, establish whether $n \in A$.
- The characteristic function of A is $\chi_A(n) = \begin{cases} 1 & \text{if } n \in A \\ 0 & \text{if } n \notin A \end{cases}$.
- The membership problem is solvable iff χ_A is a computable function.

5.3 Recursive and Recursively Enumerable Sets

- How to characterize sets that can be built algorithmically?
- Membership problem: for $A \subseteq \mathbb{N}$ and $n \in \mathbb{N}$, establish whether $n \in A$.
- The characteristic function of A is $\chi_A(n) = \begin{cases} 1 & \text{if } n \in A \\ 0 & \text{if } n \notin A \end{cases}$.
- The membership problem is solvable iff χ_A is a computable function.
- A is a recursive set iff there exists a TM whose computed function is total and coincides with χ_A.
- A is a recursively enumerable set iff it is the domain or codomain of a Turing-computable function.
- $2^{\mathbb{N}}$ is uncountable, while there are only countably many subsets of $\mathbb{N}$ that are recursively enumerable.

- Recursive sets correspond to total computable functions, while recursively enumerable sets correspond to all computable functions.
- Any finite set is recursive, with the corresponding TM ideally comparing the input value with each value in the set.

- Recursive sets correspond to total computable functions, while recursively enumerable sets correspond to all computable functions.
- Any finite set is recursive, with the corresponding TM ideally comparing the input value with each value in the set.
- The set of even (resp. odd) natural numbers is infinite and recursive, with the corresponding TM checking whether the rightmost digit is $0, 2, 4, 6, 8$ (resp. $1, 3, 5, 7, 9$).
- The set of prime natural numbers is infinite and recursive too, with the corresponding TM having to check finitely many values.

- Recursive sets correspond to total computable functions, while recursively enumerable sets correspond to all computable functions.
- Any finite set is recursive, with the corresponding TM ideally comparing the input value with each value in the set.
- The set of even (resp. odd) natural numbers is infinite and recursive, with the corresponding TM checking whether the rightmost digit is $0, 2, 4, 6, 8$ (resp. $1, 3, 5, 7, 9$).
- The set of prime natural numbers is infinite and recursive too, with the corresponding TM having to check finitely many values.
- Any recursive set is recursively enumerable.
- Consider an enumeration of the set $\mathcal{F}_{\mathrm{c}}$ of computable functions:
  - $A = \{n \in \mathbb{N} \mid f_n(n)\downarrow\}$ is recursively enumerable but not recursive.
  - $\overline{A} = \{n \in \mathbb{N} \mid f_n(n)\uparrow\}$ is not even recursively enumerable.

- A is recursively enumerable iff its elements can be algorithmically enumerated, i.e., $A = \emptyset$ or A is the codomain of a function g for which there exists a TM whose computed function is total and coincides with g.
- Union/intersection/complement of recursive sets is recursive.
- Union/intersection of recursively enumerable sets is rec. enumerable.

- A is recursively enumerable iff its elements can be algorithmically enumerated, i.e., $A = \emptyset$ or A is the codomain of a function g for which there exists a TM whose computed function is total and coincides with g.
- Union/intersection/complement of recursive sets is recursive.
- Union/intersection of recursively enumerable sets is rec. enumerable.
- Post theorem: if A and its complement $\overline{A}$ are recursively enumerable, then A is recursive (and vice versa).
- Rice theorem: for $F \subseteq \mathcal{F}_{\mathrm{c}}$ the set $A = \{n \in \mathbb{N} \mid f_n \in F\}$ is recursive iff $F = \emptyset$ or $F = \mathcal{F}_{\mathrm{c}}$, i.e., when $\emptyset \neq F \neq \mathcal{F}_{\mathrm{c}}$ establishing whether the function computed by an arbitrary TM is in F is not decidable.
- $A = \{n \in \mathbb{N} \mid f_n \in \mathcal{F}_{\mathrm{c}}\ \mathrm{total}\}$ is not recursively enumerable, hence any formalism computing only total functions cannot compute all of them!

5.4 Decidable and Undecidable Problems

- Membership problems are special cases of decision problems.
- A decision problem is a computational problem with a yes/no answer.

5.4 Decidable and Undecidable Problems

- Membership problems are special cases of decision problems.
- A decision problem is a computational problem with a yes/no answer.
- Examples of formulations of different related problems:
  - The existence of a path between two vertices is a decision problem.
  - Finding a path between those two vertices is not a decision problem.
  - Finding the shortest path between them is an optimization problem.

5.4 Decidable and Undecidable Problems

- Membership problems are special cases of decision problems.
- A decision problem is a computational problem with a yes/no answer.
- Examples of formulations of different related problems:
  - The existence of a path between two vertices is a decision problem.
  - Finding a path between those two vertices is not a decision problem.
  - Finding the shortest path between them is an optimization problem.
- Let I_{y} be the set of input data for which the answer is yes.
- A decision problem is said to be decidable when I_{y} is recursive and semi-decidable when I_{y} is recursively enumerable.
- A decision problem is said to be undecidable when it is not decidable.

- Problems about the function computed or language recognized by an arbitrary TM are undecidable in general due to Rice theorem.
- The object of study and the working tool coincide, both are TMs!

- Problems about the function computed or language recognized by an arbitrary TM are undecidable in general due to Rice theorem.
- The object of study and the working tool coincide, both are TMs!
- Every total transformation of TMs into TMs admits a fixed point, i.e., a TM computing the same function as the transformed TM.
- Kleene theorem: for every total function $t \in \mathcal{F}_c$ there exists $n \in \mathbb{N}$ such that $f_n = f_{t(n)}$.

- Problems about the function computed or language recognized by an arbitrary TM are undecidable in general due to Rice theorem.
- The object of study and the working tool coincide, both are TMs!
- Every total transformation of TMs into TMs admits a fixed point, i.e., a TM computing the same function as the transformed TM.
- Kleene theorem: for every total function $t \in \mathcal{F}_c$ there exists $n \in \mathbb{N}$ such that $f_n = f_{t(n)}$.
- To study whether the function computed by an arbitrary TM enjoys a certain property, we work with a TM acting as a verifier.
- If the function computed by the verifier has at least one fixed point, i.e., a TM invariant w.r.t. the transformation applied by the verifier, then the verifier cannot always establish whether the property is met.

- The proof of undecidability of a problem is usually by contradiction:
  - Applying Cantor diagonal method to the problem itself.
  - Showing that the decidability of the problem would lead to the decidability of a problem known to be undecidable.

- The proof of undecidability of a problem is usually by contradiction:
  - Applying Cantor diagonal method to the problem itself.
  - Showing that the decidability of the problem would lead to the decidability of a problem known to be undecidable.
- Examples of undecidable problems on TMs:
  - Halting problem: an arbitrary TM converges on an arbitrary input.
  - Equivalence problem: two arbitrary TMs compute the same function.
  - The function computed by an arbitrary TM is total/constant.
  - The language recognized by an arbitrary TM is $L_1/L_2/L_3$/finite/$\emptyset$.

- The proof of undecidability of a problem is usually by contradiction:
  - Applying Cantor diagonal method to the problem itself.
  - Showing that the decidability of the problem would lead to the decidability of a problem known to be undecidable.
- Examples of undecidable problems on TMs:
  - Halting problem: an arbitrary TM converges on an arbitrary input.
  - Equivalence problem: two arbitrary TMs compute the same function.
  - The function computed by an arbitrary TM is total/constant.
  - The language recognized by an arbitrary TM is $L_1/L_2/L_3$/finite/$\emptyset$.
- Examples of undecidable problems in other fields:
  - Satisfiability of an arbitrary predicate logic formula.
  - Reducibility of an arbitrary λ-term to a normal form.
  - Tenth Hilbert problem: an arbitrary Diophantine equation (polynomial with integer coefficients) has an integer solution.

- There are also some decision problems whose undecidability cannot even be proven.
- Examples:
  - Individual termination of a specific TM on a specific input.
  - Fermat last theorem: there are no $x, y, z \in \mathbb{Z} \setminus \{0\}$ and $n \in \mathbb{Z}$ with $n > 2$ such that $x^n + y^n = z^n$.
  - Planar graph coloring: four colors are sufficient to decorate the vertices of an arbitrary planar graph in such a way that no two adjacent vertices are of the same color.

- There are also some decision problems whose undecidability cannot even be proven.
- Examples:
  - Individual termination of a specific TM on a specific input.
  - Fermat last theorem: there are no $x, y, z \in \mathbb{Z} \setminus \{0\}$ and $n \in \mathbb{Z}$ with $n > 2$ such that $x^n + y^n = z^n$.
  - Planar graph coloring: four colors are sufficient to decorate the vertices of an arbitrary planar graph in such a way that no two adjacent vertices are of the same color.
- It does not mean that those problems are undecidable.
- Fermat last theorem was proven in 1995.
- The four color theorem was proven in 1977 in a computer-based way, thus four colors suffice for political geographic maps.

5.5 Tractable and Intractable Problems

- Every decidable decision problem admits an algorithmic solution.
- The computational complexity of an algorithm depends on its:
  - Running time.
  - Memory space.
  - Communication bandwidth.

5.5 Tractable and Intractable Problems

- Every decidable decision problem admits an algorithmic solution.
- The computational complexity of an algorithm depends on its:
  - Running time.
  - Memory space.
  - Communication bandwidth.
- Time is the most expensive resource as it is irreversible.
- Time complexity is the order of magnitude of the running time expressed as a function of the input data size n:
  - Constant: $O(1)$.
  - Logarithmic: $O(\log n)$.
  - Linear: $O(n)$.
  - Pseudolinear: $O(n \cdot \log n)$.
  - Polynomial: $O(n^k)$.
  - Exponential: $O(a^n)$.

- A decidable decision problem may have several algorithmic solutions.
- *Tractability*: how efficiently a decidable problem can be solved.
- A decidable decision problem is said to be tractable
  if it admits a polynomial-time algorithmic solution,
  otherwise it is said to be intractable.

- A decidable decision problem may have several algorithmic solutions.
- *Tractability*: how efficiently a decidable problem can be solved.
- A decidable decision problem is said to be tractable
  if it admits a polynomial-time algorithmic solution,
  otherwise it is said to be intractable.
- $\mathcal{P}$: class of decision problems that can be solved in polynomial time by a *deterministic* algorithm.
- $\mathcal{NP}$: class of decision problems that can be solved in polynomial time by a *nondeterministic* algorithm (it simultaneously explores all paths, can be exponentially many but each is solved in polynomial time).
- $\mathcal{P} \subseteq \mathcal{NP}$ but it is not known whether the inclusion is strict or not.
- $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ is one of the most important open problems in informatics.

- A decidable problem $P_1 \subseteq I_1 \times \{0,1\}$ reduces in polynomial time to a decidable problem $P_2 \subseteq I_2 \times \{0,1\}$ iff there exists a deterministic algorithm that computes in polynomial time a function $f : I_1 \to I_2$ such that $(i,s) \in P_1 \iff (f(i),s) \in P_2$ for all $i \in I_1$ and $s \in \{0,1\}$.
- The solution to each instance of P_1 can be obtained by solving the corresponding instance of P_2 computable in polynomial time.

- A decidable problem $P_1 \subseteq I_1 \times \{0,1\}$ reduces in polynomial time to a decidable problem $P_2 \subseteq I_2 \times \{0,1\}$ iff there exists a deterministic algorithm that computes in polynomial time a function $f : I_1 \to I_2$ such that $(i,s) \in P_1 \iff (f(i),s) \in P_2$ for all $i \in I_1$ and $s \in \{0,1\}$.
- The solution to each instance of P_1 can be obtained by solving the corresponding instance of P_2 computable in polynomial time.
- Cook theorem: any problem in $\mathcal{NP}$ reduces in polynomial time to the satisfiability problem.
- Corollary: if there exists a deterministic algorithm solving the satisfiability problem in polynomial time, then $\mathcal{P} = \mathcal{NP}$.
- *Satisfiability problem*: an arbitrary propositional logic formula in conjunctive normal form admits an assignment of truth values to its variables that makes the formula true.

- A problem in $\mathcal{NP}$ is $\mathcal{NP}$-complete iff the satisfiability problem reduces to it in polynomial time (corollary applies to each of them).
- Examples of $\mathcal{NP}$-complete problems other than satisfiability:
  - *Knapsack problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers c and z, establish whether there exists a subset of A whose elements sum up to a number between c and z.

- A problem in $\mathcal{NP}$ is $\mathcal{NP}$-complete iff the satisfiability problem reduces to it in polynomial time (corollary applies to each of them).
- Examples of $\mathcal{NP}$-complete problems other than satisfiability:
  - *Knapsack problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers c and z, establish whether there exists a subset of A whose elements sum up to a number between c and z.
  - *Bin packing problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers k and s, establish whether there exists a partition of A into k disjoint subsets $A_1, \ldots, A_k$ such that the sum of the elements in each A_i is not greater than s.

- A problem in $\mathcal{NP}$ is $\mathcal{NP}$-complete iff the satisfiability problem reduces to it in polynomial time (corollary applies to each of them).
- Examples of $\mathcal{NP}$-complete problems other than satisfiability:
  - *Knapsack problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers c and z, establish whether there exists a subset of A whose elements sum up to a number between c and z.
  - *Bin packing problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers k and s, establish whether there exists a partition of A into k disjoint subsets $A_1, \ldots, A_k$ such that the sum of the elements in each A_i is not greater than s.
  - *Clique problem*: given a graph G and a positive integer k, establish whether G has a complete subgraph with k vertices.

- A problem in $\mathcal{NP}$ is $\mathcal{NP}$-complete iff the satisfiability problem reduces to it in polynomial time (corollary applies to each of them).
- Examples of $\mathcal{NP}$-complete problems other than satisfiability:
  - *Knapsack problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers c and z, establish whether there exists a subset of A whose elements sum up to a number between c and z.
  - *Bin packing problem*: given a set $A = \{a_1, \ldots, a_n\}$ of positive integers and two positive integers k and s, establish whether there exists a partition of A into k disjoint subsets $A_1, \ldots, A_k$ such that the sum of the elements in each A_i is not greater than s.
  - *Clique problem*: given a graph G and a positive integer k, establish whether G has a complete subgraph with k vertices.
  - *Traveling salesman problem*: given a positive integer k and a graph G whose edges are each labeled with a positive integer cost, establish whether G has a cycle that traverses each of its vertices only once in which the sum of the costs of the edges is not greater than k.

6. The Modeling View: Process Algebras

Topics:

6.1 *Concurrency and Communication* 92

6.2 *Syntax of Process Calculi* 103

6.3 *Interleaving Semantics via Labeled Transition Systems* 111

6.4 *Computational Power of Process Calculi* 119

6.5 *Spectrum of Behavioral Equivalences* 124

6.6 *Strong Bisimilarity and Its Properties* 130

6.7 *Weak Bisimilarities and Their Properties* 148

6.8 *Truly Concurrent Semantics via Petri Nets* 157

6.9 *Truly Concurrent Semantics via Event Structures* 167

6.10 *Truly Concurrent Bisimilarities* 174

6.1 Concurrency and Communication

- Sequential computing (since 1930's):
  - A single step at a time is executed.
  - Imperative models: Turing machines and Von Neumann architecture.
  - Programming languages: Fortran, Cobol, Algol, Basic, Pascal, C, ...
  - Declarative models: Church λ-calculus and first-order logic.
  - Programming languages: Lisp, Scheme, ML, Haskell, Prolog, ...

6.1 Concurrency and Communication

- Sequential computing (since 1930's):
  - A single step at a time is executed.
  - Imperative models: Turing machines and Von Neumann architecture.
  - Programming languages: Fortran, Cobol, Algol, Basic, Pascal, C, ...
  - Declarative models: Church λ-calculus and first-order logic.
  - Programming languages: Lisp, Scheme, ML, Haskell, Prolog, ...
- Concurrent and distributed computing (since 1970's):
  - Several steps can be simultaneously executed.
  - Shared-memory model vs. message-passing model.
  - Primitives for software synchronization: semaphores, monitors, ...
  - Concurrent programming languages: Ada, Occam, Erlang, Scala, ...
  - Extensions of previously developed languages.

- Global computing (since 2000's): computation over infrastructures globally accessible via personal devices and offering uniform services.
- Abstraction of a global computer that we can use *anytime anywhere*.
- Development of large-scale general-purpose computing systems that hopefully have a dependably predictable behavior for the needs of a distributed and mobile world.

- Global computing (since 2000's): computation over infrastructures globally accessible via personal devices and offering uniform services.
- Abstraction of a global computer that we can use *anytime anywhere*.
- Development of large-scale general-purpose computing systems that hopefully have a dependably predictable behavior for the needs of a distributed and mobile world.
- Providing support for e-government and e-commerce (web services), resource sharing (cloud), ambient intelligence (IoT), ...
- Addressing issues that go beyond concurrent and distributed systems: mobility, ubiquity, dynamicity, interactivity, ...
- *"Computing is interaction!"* (Robin Milner, 1994).

- Concurrency and communication are essential in the design and deployment of modern computing systems.
- Any such system is composed of many interconnected parts that interact by exchanging information or simply synchronizing.
- Concurrency and communication are complementary notions:
  - Diversity: each part acts concurrently with (independently of) other parts.
  - Unity: achieved through communication among the various parts.

- Concurrency and communication are essential in the design and deployment of modern computing systems.
- Any such system is composed of many interconnected parts that interact by exchanging information or simply synchronizing.
- Concurrency and communication are complementary notions:
  - Diversity: each part acts concurrently with (independently of) other parts.
  - Unity: achieved through communication among the various parts.
- Computing systems featuring concurrency and communication are often required to possess a degree of reactivity to external stimuli and are usually nonterminating (like operating systems).
- Nondeterminism in the final result or in the computation can arise due to the different speeds of the parts, the interaction scheme among the parts, and the scheduling policies that are adopted.

- Assume that the computation state is the memory content at a certain point of the execution.
- The behavior of a sequential system can be formalized as a mathematical function that associates a final state (output) with every possible initial state (input).
- This input-output transformation approach for sequential systems is not applicable to communicating concurrent systems!

- Assume that the computation state is the memory content at a certain point of the execution.
- The behavior of a sequential system can be formalized as a mathematical function that associates a final state (output) with every possible initial state (input).
- This input-output transformation approach for sequential systems is not applicable to communicating concurrent systems!
- Cannot abstract from the intermediate states of the computation.
- Consider the following two sequential program fragments:

```
(1) X := 1;    (2) X := 0; X := X + 1;
```

- Assume that the computation state is the memory content at a certain point of the execution.
- The behavior of a sequential system can be formalized as a mathematical function that associates a final state (output) with every possible initial state (input).
- This input-output transformation approach for sequential systems is not applicable to communicating concurrent systems!
- Cannot abstract from the intermediate states of the computation.
- Consider the following two sequential program fragments:

```
(1) X := 1;    (2) X := 0; X := X + 1;
```

- When executed in isolation, they have the same effect (`X` becomes `1`).
- If the two fragments are executed concurrently, the final value of `X` is not necessarily 1, but can be either 1 or 2 (cannot be deterministically predicted).

- How to model and analyze communicating concurrent systems?
- Making no distinction of kind between systems and their parts enables uniform reasoning at different abstraction levels.
- Accomplished through the notion of process.

- How to model and analyze communicating concurrent systems?
- Making no distinction of kind between systems and their parts enables uniform reasoning at different abstraction levels.
- Accomplished through the notion of process.
- A process may be decomposed into subprocesses for a certain purpose or may be viewed as being atomic for other purposes.
- A process is a series of actions/events divided into:
  - Internal actions, possibly due to subprocesses communication.
  - Interactions with neighboring processes or the external environment.

- Any computing system features a structure and a behavior.
- A process is an abstraction of the behavior of a computing system.
- The behavior of a complex computing system can be defined as its entire capability of communication.

- Any computing system features a structure and a behavior.
- A process is an abstraction of the behavior of a computing system.
- The behavior of a complex computing system can be defined as its entire capability of communication.
- Black-box view: the behavior of a system is exactly what is observable and to observe a system is exactly to communicate with it.
- The notion of process focuses on the behavioral aspects of a system, while neglecting its structural and physical attributes.

- Consider a sequential system that:
  - either performs action a followed by action b and then terminates;
  - or performs action b followed by action a and then terminates.

- Consider a sequential system that:
  - either performs action a followed by action b and then terminates;
  - or performs action b followed by action a and then terminates.
- Consider another system that performs actions a and b in parallel:
  - either action a terminates first and action b terminates afterwards;
  - or action b terminates first and action a terminates afterwards.

- Consider a sequential system that:
  - either performs action a followed by action b and then terminates;
  - or performs action b followed by action a and then terminates.
- Consider another system that performs actions a and b in parallel:
  - either action a terminates first and action b terminates afterwards;
  - or action b terminates first and action a terminates afterwards.
- Structurally different, but behaviorally equivalent!
- A concurrent system behaves like a sequential one obtained by interleaving the actions executed by the parts of the former.
- Nondeterministic choice among all possible action sequencings.

- Is there a calculus for processes as basic as λ-calculus for functions?
- Concurrency theory started between late 1970's and early 1980's.

- Is there a calculus for processes as basic as λ-calculus for functions?
- Concurrency theory started between late 1970's and early 1980's.
- Process calculi/algebras constitute a fundamental branch:
  - Robin Milner, Tony Hoare, Matthew Hennessy, Rocco De Nicola, Ugo Montanari, Pierpaolo Degano, Ilaria Castellani, Ursula Goltz, Davide Sangiorgi, Kim Larsen, Rance Cleaveland, Scott Smolka, Jos Baeten, Jan Bergstra, Jan Willem Klop, Peter Weijland, Rob van Glabbeek, Frits Vaandrager, Jan Friso Groote, ...
  - CCS, π-calculus, CSP, ACP, LOTOS, ...
  - CWB, CWB-NC/PAC, FDR, μCRL, CADP, ...

- Process: actions or events representing the observable behavior.
- Calculus: system or method of calculation.
- Algebra: calculus of symbols combining according to certain laws.

- Process: actions or events representing the observable behavior.
- Calculus: system or method of calculation.
- Algebra: calculus of symbols combining according to certain laws.
- Generalization of formal languages and automata theory focusing on *system behavior* rather than language generation and recognition.
- Foundations of concurrent programming semantics.
- Support for model-based design of modern computing systems.

- Conceived for studying communicating concurrent systems and their various aspects: nondeterminism, mobility, probability, time, ...
- Linguistic counterpart of computational models such as Keller transition systems, Petri nets, Winskel event structures, ...

- Conceived for studying communicating concurrent systems and their various aspects: nondeterminism, mobility, probability, time, ...
- Linguistic counterpart of computational models such as Keller transition systems, Petri nets, Winskel event structures, ...
- Compositional modeling by means of several behavioral operators expressing *sequential*, *alternative*, *parallel* compositions of processes.
- System dynamics described through (recursive) behavioral equations.
- Process comparison through behavioral equivalences and preorders formalizing the notions of same behavior and behavior refinement.

- Conceived for studying communicating concurrent systems and their various aspects: nondeterminism, mobility, probability, time, ...
- Linguistic counterpart of computational models such as Keller transition systems, Petri nets, Winskel event structures, ...
- Compositional modeling by means of several behavioral operators expressing *sequential*, *alternative*, *parallel* compositions of processes.
- System dynamics described through (recursive) behavioral equations.
- Process comparison through behavioral equivalences and preorders formalizing the notions of same behavior and behavior refinement.
- Abstraction from certain unnecessary details of system behavior by distinguishing between *visible* and *invisible* actions.

- **Running example**: producer-consumer system.
- General description:
  - Three components: producer, finite-capacity buffer, consumer.
  - The producer deposits items into the buffer as long as the buffer capacity is not exceeded.
  - Stored items are then withdrawn from the buffer by the consumer according to some predefined discipline (like FIFO or LIFO).

- **Running example**: producer-consumer system.
- General description:
  - Three components: producer, finite-capacity buffer, consumer.
  - The producer deposits items into the buffer as long as the buffer capacity is not exceeded.
  - Stored items are then withdrawn from the buffer by the consumer according to some predefined discipline (like FIFO or LIFO).
- Specific scenario:
  - The buffer has only two positions.
  - Items are identical, hence the discipline is not important.

6.2 Syntax of Process Calculi

- Modeling languages for communicating concurrent systems.
- Compositionality: building complex models from simpler ones by means of suitable operators.
- Abstraction: ability to neglect some details of a model by considering them as invisible.
- Their basic ingredients are *actions* and *behavioral operators*.

6.2 Syntax of Process Calculi

- Modeling languages for communicating concurrent systems.
- Compositionality: building complex models from simpler ones by means of suitable operators.
- Abstraction: ability to neglect some details of a model by considering them as invisible.
- Their basic ingredients are *actions* and *behavioral operators*.
- $\mathcal{A}_{\mathrm{v}}$: countable set of visible action names.
- τ: invisible (or silent or unobservable) action.
- $\mathcal{A} = \mathcal{A}_{\mathrm{v}} \cup \{\tau\}$: set of all action names.
- $Relab = \{\varphi : \mathcal{A} \to \mathcal{A} \mid \varphi^{-1}(\tau) = \{\tau\}\}$: set of visibility-preserving action relabeling functions.

- Syntax of process terms:

$$
\begin{array}{rcll}
P & ::= & \underline{0} & \text{terminated process} \\
 & | & a\,.\,P & \text{action prefix} \quad (a \in \mathcal{A}) \\
 & | & P + P & \text{alternative composition} \\
 & | & P \parallel_S P & \text{parallel composition} \quad (S \subseteq \mathcal{A}_{\mathrm{v}}) \\
 & | & P \,/\, H & \text{hiding} \quad (H \subseteq \mathcal{A}_{\mathrm{v}}) \\
 & | & P \setminus L & \text{restriction} \quad (L \subseteq \mathcal{A}_{\mathrm{v}}) \\
 & | & P\,[\varphi] & \text{relabeling} \quad (\varphi \in \mathit{Relab}) \\
 & | & X & \text{process variable} \quad (X \in \mathit{Var}) \\
 & | & \mathrm{rec}\, X : P & \text{recursion} \quad (X \in \mathit{Var})
\end{array}
$$

- Syntax of process terms:

$$
\begin{array}{rcll}
P & ::= & \underline{0} & \text{terminated process} \\
& | & a\,.\,P & \text{action prefix} \quad (a \in \mathcal{A}) \\
& | & P + P & \text{alternative composition} \\
& | & P \parallel_S P & \text{parallel composition} \quad (S \subseteq \mathcal{A}_{\mathrm{v}}) \\
& | & P \,/\, H & \text{hiding} \quad (H \subseteq \mathcal{A}_{\mathrm{v}}) \\
& | & P \setminus L & \text{restriction} \quad (L \subseteq \mathcal{A}_{\mathrm{v}}) \\
& | & P\,[\varphi] & \text{relabeling} \quad (\varphi \in \mathit{Relab}) \\
& | & X & \text{process variable} \quad (X \in \mathit{Var}) \\
& | & \mathrm{rec}\, X : P & \text{recursion} \quad (X \in \mathit{Var})
\end{array}
$$

- Precedence: unary operators $>$ $+$ $>$ $\parallel$.
- Associativity: $+$ and $\parallel$ are left associative.

- $\underline{0}$ is a terminated process and hence cannot execute any action.
- $a\,.\,P$ can perform a and then behaves as P (action-based sequential composition).
- $P_1 + P_2$ behaves as P_1 or P_2 depending on the actions they enable.

- $\underline{0}$ is a terminated process and hence cannot execute any action.
- $a\,.\,P$ can perform a and then behaves as P (action-based sequential composition).
- $P_1 + P_2$ behaves as P_1 or P_2 depending on the actions they enable.
- The choice among several actions initially enabled by P_1 and P_2 is solved nondeterministically.
- The choice is *internal* if all the initially enabled actions are identical or some are τ, otherwise it can be affected by the *external environment*.

- $\underline{0}$ is a terminated process and hence cannot execute any action.
- $a\,.\,P$ can perform a and then behaves as P (action-based sequential composition).
- $P_1 + P_2$ behaves as P_1 or P_2 depending on the actions they enable.
- The choice among several actions initially enabled by P_1 and P_2 is solved nondeterministically.
- The choice is *internal* if all the initially enabled actions are identical or some are τ, otherwise it can be affected by the *external environment*.
- Action prefix and alternative composition are dynamic operators because they are discarded from the target process along with the executed action and the unselected subprocess respectively.

- Nil, action prefix, and alternative composition are taken from CCS.

- Nil, action prefix, and alternative composition are taken from CCS.
- CSP includes both action prefix, denoted by $a \rightarrow P$, and sequential composition, denoted by $P_1 \,;\, P_2$.
- CSP has two distinct alternative composition operators:
  - Internal choice: $P_1 \sqcap P_2$.
  - External choice: $P_1 \,\Box\, P_2$.

- Nil, action prefix, and alternative composition are taken from CCS.
- CSP includes both action prefix, denoted by $a \rightarrow P$, and sequential composition, denoted by $P_1 \,;\, P_2$.
- CSP has two distinct alternative composition operators:
  - Internal choice: $P_1 \sqcap P_2$.
  - External choice: $P_1 \,\Box\, P_2$.
- ACP features sequential composition $P_1 \cdot P_2$ instead of action prefix.
- As a consequence every action is a process too.
- Algebraic flavor: additive operation and multiplicative operation.

- $P_1 \parallel_S P_2$ behaves as P_1 running in parallel with P_2.
- Actions enabled by P_1 or P_2 whose name does not belong to S are executed autonomously by P_1 and by P_2.
- Synchronization is forced between any action enabled by P_1 and any action enabled by P_2 that have the same name belonging to S

($S = \emptyset$ means P_1 and P_2 fully independent, $S = \mathcal{A}_{\mathrm{v}}$ means P_1 and P_2 fully synchronized).

- $P_1 \parallel_S P_2$ behaves as P_1 running in parallel with P_2.
- Actions enabled by P_1 or P_2 whose name does not belong to S are executed autonomously by P_1 and by P_2.
- Synchronization is forced between any action enabled by P_1 and any action enabled by P_2 that have the same name belonging to S

  ($S = \emptyset$ means P_1 and P_2 fully independent, $S = \mathcal{A}_{\mathrm{v}}$ means P_1 and P_2 fully synchronized).
- P / H behaves as P but every action belonging to H is turned into τ

  (abstraction mechanism; can be used for preventing a process from communicating).
- $P \setminus L$ behaves as P but every action belonging to L is forbidden

  (same effect as $P \parallel_L \underline{0}$).
- $P\,[\varphi]$ behaves as P but every action is renamed according to φ

  (redundance avoidance; encoding of the previous two operators if φ is non-visibility-preserving or partial).

- $P_1 \parallel_S P_2$ behaves as P_1 running in parallel with P_2.
- Actions enabled by P_1 or P_2 whose name does not belong to S are executed autonomously by P_1 and by P_2.
- Synchronization is forced between any action enabled by P_1 and any action enabled by P_2 that have the same name belonging to S

  ($S = \emptyset$ means P_1 and P_2 fully independent, $S = \mathcal{A}_{\mathrm{v}}$ means P_1 and P_2 fully synchronized).
- P / H behaves as P but every action belonging to H is turned into τ

  (abstraction mechanism; can be used for preventing a process from communicating).
- $P \setminus L$ behaves as P but every action belonging to L is forbidden

  (same effect as $P \parallel_L \underline{0}$).
- $P\,[\varphi]$ behaves as P but every action is renamed according to φ

  (redundance avoidance; encoding of the previous two operators if φ is non-visibility-preserving or partial).
- They are static operators, parallel composition is also structural.

- Restriction and relabeling are taken from CCS.
- Hiding and (multiparty) parallel composition are taken from CSP.

- Restriction and relabeling are taken from CCS.
- Hiding and (multiparty) parallel composition are taken from CSP.
- In CCS (binary) parallel composition is denoted by $P_1 \mid P_2$.
- An action a can only synchronize with its coaction $\bar{a}$ yielding τ (binary because synchronization and hiding are mixed together).
- Restriction must be used to force synchronization: $(a \,.\, \underline{0} \mid \bar{a} \,.\, \underline{0}) \setminus \{a\}$.

- Restriction and relabeling are taken from CCS.
- Hiding and (multiparty) parallel composition are taken from CSP.
- In CCS (binary) parallel composition is denoted by $P_1 \mid P_2$.
- An action a can only synchronize with its coaction $\bar{a}$ yielding τ (binary because synchronization and hiding are mixed together).
- Restriction must be used to force synchronization: $(a \,.\, \underline{0} \mid \bar{a} \,.\, \underline{0}) \setminus \{a\}$.
- In ACP (multiparty) parallel composition is denoted by $P_1 \parallel P_2$.
- It is accompanied by a communication function establishing, for each action pair, the name of the action resulting from their synchronization (the three actions may be different).
- Partial function when there are actions that cannot synchronize.

- $\mathrm{rec}\, X : P$ behaves as P with every free occurrence of process variable $X \in Var$ being replaced by $\mathrm{rec}\, X : P$.
- Same as behavioral equation $B \stackrel{\Delta}{=} P$ for process constant $B \in Const$.

- $\mathrm{rec}\, X : P$ behaves as P with every free occurrence of process variable $X \in Var$ being replaced by $\mathrm{rec}\, X : P$.
- Same as behavioral equation $B \stackrel{\Delta}{=} P$ for process constant $B \in Const$.
- A process variable is said to occur *free* in a process term if it is not in the scope of a rec binder for that variable, otherwise it is said to be *bound* in that process term.
- A process term is said to be closed if all of its occurrences of process variables are bound, otherwise it is said to be open (resp. all of its occurrences of process constants are defined).

- $\mathrm{rec}\, X : P$ behaves as P with every free occurrence of process variable $X \in Var$ being replaced by $\mathrm{rec}\, X : P$.
- Same as behavioral equation $B \stackrel{\Delta}{=} P$ for process constant $B \in Const$.
- A process variable is said to occur *free* in a process term if it is not in the scope of a rec binder for that variable, otherwise it is said to be *bound* in that process term.
- A process term is said to be closed if all of its occurrences of process variables are bound, otherwise it is said to be open (resp. all of its occurrences of process constants are defined).
- A process term is said to be guarded iff all of its occurrences of process variables/constants are in the scope of action prefix operators.

- $\mathrm{rec}\, X : P$ behaves as P with every free occurrence of process variable $X \in Var$ being replaced by $\mathrm{rec}\, X : P$.
- Same as behavioral equation $B \stackrel{\Delta}{=} P$ for process constant $B \in Const$.
- A process variable is said to occur *free* in a process term if it is not in the scope of a rec binder for that variable, otherwise it is said to be *bound* in that process term.
- A process term is said to be closed if all of its occurrences of process variables are bound, otherwise it is said to be open (resp. all of its occurrences of process constants are defined).
- A process term is said to be guarded iff all of its occurrences of process variables/constants are in the scope of action prefix operators.
- $\mathbb{P}$: set of closed and guarded process terms, each of which is fully defined (closure) and enables finitely many actions (guardedness).

- **Running example** (process syntax):
  - Process constant names: nouns starting with an upper-case letter.
  - Action names: verbs composed of lower-case letters.
  - The only observable activities are deposits and withdrawals.
  - Visible actions: $deposit$ and $withdraw$.

- **Running example** (process syntax):
  - Process constant names: nouns starting with an upper-case letter.
  - Action names: verbs composed of lower-case letters.
  - The only observable activities are deposits and withdrawals.
  - Visible actions: $deposit$ and $withdraw$.
  - Structure-independent process algebraic description where the system state is the number of items in the buffer:

$$
\begin{aligned}
ProdCons_{0/2} &\stackrel{\Delta}{=} deposit \,.\, ProdCons_{1/2} \\
ProdCons_{1/2} &\stackrel{\Delta}{=} deposit \,.\, ProdCons_{2/2} + withdraw \,.\, ProdCons_{0/2} \\
ProdCons_{2/2} &\stackrel{\Delta}{=} withdraw \,.\, ProdCons_{1/2}
\end{aligned}
$$

  - Specification to which every correct implementation should conform.

6.3 Interleaving Semantics via Labeled Transition Systems

- Mathematical model in the form of a state transition graph that represents all computations and branching points.
- States are global as contain the description of the local states of the subprocesses composed in parallel.
- Computations are obtained by interleaving the actions executed by the subprocesses composed in parallel (all possible sequencings).

6.3 Interleaving Semantics via Labeled Transition Systems

- Mathematical model in the form of a state transition graph that represents all computations and branching points.
- States are global as contain the description of the local states of the subprocesses composed in parallel.
- Computations are obtained by interleaving the actions executed by the subprocesses composed in parallel (all possible sequencings).
- Keller labeled transition systems (1976) instead of Kripke structures to elicit transition-labeling actions instead of state properties.
- Process term $P \in \mathbb{P}$ is mapped to the LTS $[\![P]\!] = (\mathbb{P}, \mathcal{A}, \longrightarrow, P)$:
  - Each state corresponds to a process term into which P can evolve.
  - The initial state corresponds to P.
  - Each transition from a source state to a target state is labeled with the action that determines the corresponding state change.

- The transition relation $\longrightarrow \subseteq \mathbb{P} \times \mathcal{A} \times \mathbb{P}$ is the smallest subset of $\mathbb{P} \times \mathcal{A} \times \mathbb{P}$ that satisfies Plotkin-style operational semantic rules defined by induction on the syntactical structure of process terms.
- Derivation of one single transition at a time by applying the rules to the source state of the transition under construction.

- The transition relation $\longrightarrow \subseteq \mathbb{P} \times \mathcal{A} \times \mathbb{P}$ is the smallest subset of $\mathbb{P} \times \mathcal{A} \times \mathbb{P}$ that satisfies Plotkin-style operational semantic rules defined by induction on the syntactical structure of process terms.
- Derivation of one single transition at a time by applying the rules to the source state of the transition under construction.
- No rule for $\underline{0}$: $[\![\underline{0}]\!]$ has a single state and no transitions.
- Basic rule for action prefix: $a \,.\, P \xrightarrow{a} P$ (note that a disappears).

- The transition relation $\longrightarrow \subseteq \mathbb{P} \times \mathcal{A} \times \mathbb{P}$ is the smallest subset of $\mathbb{P} \times \mathcal{A} \times \mathbb{P}$ that satisfies Plotkin-style operational semantic rules defined by induction on the syntactical structure of process terms.
- Derivation of one single transition at a time by applying the rules to the source state of the transition under construction.
- No rule for $\underline{0}$: $[\![\underline{0}]\!]$ has a single state and no transitions.
- Basic rule for action prefix: $a \,.\, P \xrightarrow{a} P$ (note that a disappears).
- Inductive rules for all the other operators.
- Different formats for *dynamic* ($+$) and *static* ($\|$, $/$, $\setminus$, $[\,]$) operators.

- The transition relation $\longrightarrow \subseteq \mathbb{P} \times \mathcal{A} \times \mathbb{P}$ is the smallest subset of $\mathbb{P} \times \mathcal{A} \times \mathbb{P}$ that satisfies Plotkin-style operational semantic rules defined by induction on the syntactical structure of process terms.
- Derivation of one single transition at a time by applying the rules to the source state of the transition under construction.
- No rule for $\underline{0}$: $[\![\underline{0}]\!]$ has a single state and no transitions.
- Basic rule for action prefix: $a \,.\, P \xrightarrow{a} P$ (note that a disappears).
- Inductive rules for all the other operators.
- Different formats for *dynamic* ($+$) and *static* ($\|$, $/$, $\backslash$, $[\,]$) operators.
- $[\![P]\!]$ is finite state if inside P there are no recursive definitions that contain static operators.
- In that case $[\![P]\!]$ is also finitely branching, i.e., every state has finitely many outgoing transitions.

- Operational semantic rules for alternative composition:

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'}$$

- Reading order: left-bottom, left-top, right-top, right-bottom.
- Note that $+$ no longer occurs in the target processes P_1' and P_2'.

- Operational semantic rules for alternative composition:

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'}$$

- Reading order: left-bottom, left-top, right-top, right-bottom.
- Note that $+$ no longer occurs in the target processes P_1' and P_2'.
- If several actions are initially enabled, the choice among them is solved nondeterministically due to the absence of precise criteria or quantitative information (if-then-else, priority, probability, time race).
- The choice is internal if all initially enabled actions are identical or τ.
- Otherwise the choice can be influenced by the external environment.

- Operational semantic rules for parallel execution:

$$\frac{P_1 \xrightarrow{a} P_1' \qquad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2} \qquad \frac{P_2 \xrightarrow{a} P_2' \qquad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1 \parallel_S P_2'}$$

- These two rules result in the interleaving semantics.

- Operational semantic rules for parallel execution:

$$\frac{P_1 \xrightarrow{a} P_1' \qquad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2} \qquad \frac{P_2 \xrightarrow{a} P_2' \qquad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1 \parallel_S P_2'}$$

- These two rules result in the interleaving semantics.
- Operational semantic rule for synchronization:

$$\frac{P_1 \xrightarrow{a} P_1' \qquad P_2 \xrightarrow{a} P_2' \qquad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2'}$$

- Note that $\parallel_S$ still occurs in the target processes of all the three rules.

- The following process terms represent structurally different systems:

$$a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

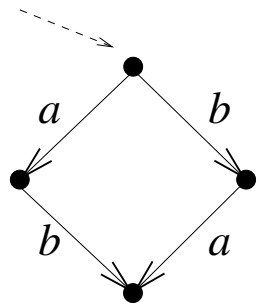
  but they are indistinguishable by an external observer.

- The following process terms represent structurally different systems:

$$a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

  but they are indistinguishable by an external observer.

- The interleaving semantics yields the same labeled transition system:



  up to processes associated with states (which are not observable):
  - Sequential: left $b\,.\,\underline{0}$, right $a\,.\,\underline{0}$, bottom $\underline{0}$.
  - Concurrent: left $\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$, right $a\,.\,\underline{0} \parallel_{\emptyset} \underline{0}$, bottom $\underline{0} \parallel_{\emptyset} \underline{0}$.

- Examples of transition derivations:
  - Starting from $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$:

$$\frac{a\,.\,b\,.\,\underline{0} \xrightarrow{a} b\,.\,\underline{0}}{a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0} \xrightarrow{a} b\,.\,\underline{0}}$$

- Examples of transition derivations:
  - Starting from $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$:

$$\frac{a\,.\,b\,.\,\underline{0} \xrightarrow{a} b\,.\,\underline{0}}{a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0} \xrightarrow{a} b\,.\,\underline{0}}$$

  - Starting from $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$:

$$\frac{a\,.\,\underline{0} \xrightarrow{a} \underline{0} \qquad a \notin \emptyset}{a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0} \xrightarrow{a} \underline{0} \parallel_{\emptyset} b\,.\,\underline{0}}$$

- Examples of transition derivations:
  - Starting from $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$:

$$\frac{a\,.\,b\,.\,\underline{0} \xrightarrow{a} b\,.\,\underline{0}}{a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0} \xrightarrow{a} b\,.\,\underline{0}}$$

  - Starting from $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$:

$$\frac{a\,.\,\underline{0} \xrightarrow{a} \underline{0} \qquad a \notin \emptyset}{a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0} \xrightarrow{a} \underline{0} \parallel_{\emptyset} b\,.\,\underline{0}}$$

  - Starting from $(a\,.\,\underline{0} + c\,.\,\underline{0}) \parallel_{\{c\}} (b\,.\,\underline{0} + c\,.\,\underline{0})$:

$$\frac{\dfrac{c\,.\,\underline{0} \xrightarrow{c} \underline{0}}{a\,.\,\underline{0} + c\,.\,\underline{0} \xrightarrow{c} \underline{0}} \qquad \dfrac{c\,.\,\underline{0} \xrightarrow{c} \underline{0}}{b\,.\,\underline{0} + c\,.\,\underline{0} \xrightarrow{c} \underline{0}} \qquad c \in \{c\}}{(a\,.\,\underline{0} + c\,.\,\underline{0}) \parallel_{\{c\}} (b\,.\,\underline{0} + c\,.\,\underline{0}) \xrightarrow{c} \underline{0} \parallel_{\emptyset} \underline{0}}$$

- Operational semantic rules for hiding, restriction, relabeling:

$$\frac{P \xrightarrow{a} P' \qquad a \in H}{P \,/\, H \xrightarrow{\tau} P' \,/\, H} \qquad \frac{P \xrightarrow{a} P' \qquad a \notin H}{P \,/\, H \xrightarrow{a} P' \,/\, H}$$

$$\frac{P \xrightarrow{a} P' \qquad a \notin L}{P \setminus L \xrightarrow{a} P' \setminus L} \qquad \frac{P \xrightarrow{a} P'}{P\,[\varphi] \xrightarrow{\varphi(a)} P'\,[\varphi]}$$

- Operational semantic rules for hiding, restriction, relabeling:

$$\frac{P \xrightarrow{a} P' \qquad a \in H}{P \,/\, H \xrightarrow{\tau} P' \,/\, H} \qquad \frac{P \xrightarrow{a} P' \qquad a \notin H}{P \,/\, H \xrightarrow{a} P' \,/\, H}$$

$$\frac{P \xrightarrow{a} P' \qquad a \notin L}{P \setminus L \xrightarrow{a} P' \setminus L} \qquad \frac{P \xrightarrow{a} P'}{P\,[\varphi] \xrightarrow{\varphi(a)} P'\,[\varphi]}$$

- Operational semantic rules for recursion (variables vs. constants):

$$\frac{P\{\mathrm{rec}\, X : P \hookrightarrow X\} \xrightarrow{a} P'}{\mathrm{rec}\, X : P \xrightarrow{a} P'} \qquad \frac{B \triangleq P \qquad P \xrightarrow{a} P'}{B \xrightarrow{a} P'}$$

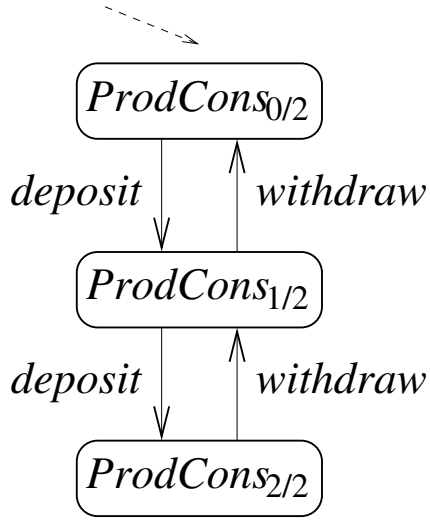
- **Running example** (process semantics):
  - Let us recall the structure-independent process algebraic description:

$$
\begin{aligned}
ProdCons_{0/2} &\triangleq deposit\,.\,ProdCons_{1/2} \\
ProdCons_{1/2} &\triangleq deposit\,.\,ProdCons_{2/2} + withdraw\,.\,ProdCons_{0/2} \\
ProdCons_{2/2} &\triangleq withdraw\,.\,ProdCons_{1/2}
\end{aligned}
$$

- **Running example** (process semantics):
  - Let us recall the structure-independent process algebraic description:

$$ProdCons_{0/2} \stackrel{\Delta}{=} deposit\,.\,ProdCons_{1/2}$$
$$ProdCons_{1/2} \stackrel{\Delta}{=} deposit\,.\,ProdCons_{2/2} + withdraw\,.\,ProdCons_{0/2}$$
$$ProdCons_{2/2} \stackrel{\Delta}{=} withdraw\,.\,ProdCons_{1/2}$$

  - Labeled transition system $[\![ProdCons_{0/2}]\!]$:



  - Obtained by mechanically applying the operational semantic rules for process constant, alternative composition, and action prefix.

6.4 Computational Power of Process Calculi

- Process calculi with the considered operators and recursion have the same computational power as Turing machines, λ-calculus, and general recursive functions.
- Proven by Robin Milner.

6.4 Computational Power of Process Calculi

- Process calculi with the considered operators and recursion have the same computational power as Turing machines, λ-calculus, and general recursive functions.
- Proven by Robin Milner.
- A Turing machine can be simulated by two stacks together with a finite-state control mechanism:
  - The first stack contains the symbols to the left of the head.
  - The second stack contains the symbols to the right of the head.
  - Reading a symbol from the tape corresponds to a pop operation.
  - Writing a symbol to the tape corresponds to a push operation.
  - Moving the head means pop on one stack and push on the other.
- How to model this in a process calculus?

- One process for the finite-state control mechanism together with one process for each stack.
- The interaction of the control mechanism with the two stacks can be represented through parallel composition.

- One process for the finite-state control mechanism together with one process for each stack.
- The interaction of the control mechanism with the two stacks can be represented through parallel composition.
- The finite-state control mechanism can be described by means of action prefix, alternative composition, and recursion, resulting in as many behavioral equations of the following form as there are states:

$$Q_i \stackrel{\Delta}{=} a_{i,j_1} . Q_{j_1} + a_{i,j_2} . Q_{j_2} + \ldots + a_{i,j_{n_i}} . Q_{j_{n_i}} \quad (1 \leq i \leq k)$$

- How to model a stack in a process calculus?

- Consider a finite set V of values that can be placed in the stack.
- The stack content σ is an element of V^*, which is ε when empty.
- Actions $push$ and pop for modeling stack operations along with $signal\_empty$ when attemping to pop from an empty stack.

- Consider a finite set V of values that can be placed in the stack.
- The stack content σ is an element of V^*, which is ε when empty.
- Actions $push$ and pop for modeling stack operations along with $signal\_empty$ when attemping to pop from an empty stack.
- The stack can be inductively specified as follows:

$$Stack_\varepsilon \triangleq \sum_{w \in V} push_w \,.\, Stack_w + signal\_empty \,.\, Stack_\varepsilon$$

$$Stack_{v::\sigma} \triangleq \sum_{w \in V} push_w \,.\, Stack_{w::v::\sigma} + pop_v \,.\, Stack_\sigma \quad (\sigma \in V^*)$$

- *Infinitely many* behavioral equations because V^* is countable!

- *Finite implementation* based on as many cells as there are values of V in the stack, which are created and linked together as needed:

$$Cell_{v_n} \widehat{} Cell_{v_{n-1}} \widehat{} \ldots \widehat{} Cell_{v_1} \widehat{} Empty$$

- In case of pop all remaining values are moved to the left by one cell.

- *Finite implementation* based on as many cells as there are values of V in the stack, which are created and linked together as needed:

$$Cell_{v_n} \hat{} Cell_{v_{n-1}} \hat{} \ldots \hat{} Cell_{v_1} \hat{} Empty$$

- In case of pop all remaining values are moved to the left by one cell.
- $|V| + 2$ equations are sufficient:

$$Empty \triangleq \sum_{w \in V} push_w \,.\, (Cell_w \hat{} Empty) + signal\_empty \,.\, Empty$$

$$Cell_v \triangleq \sum_{w \in V} push_w \,.\, (Cell_w \hat{} Cell_v) + pop_v \,.\, Changing \quad (v \in V)$$

$$Changing \triangleq \sum_{u \in V} subtop_u \,.\, Cell_u + bottom\_cell \,.\, Empty$$

- Definition of the left-associative linking operator $\widehat{\ }$:

$$P \widehat{\ } Q \triangleq P\left[{}^{p_u}/_{subtop_u}, {}^{e}/_{bottom\_cell}\right] \|_{\{p_u, e\}} Q\left[{}^{p_u}/_{pop_u}, {}^{e}/_{signal\_empty}\right]$$

where P stands for $Changing$ and Q stands for $Cell_u$ or $Empty$, while any action with subscript u is a shorthand for all such actions.

- Definition of the left-associative linking operator $\widehat{\ }$:

$$P \widehat{\ } Q \triangleq P\left[{}^{p_u}/_{subtop_u}, {}^{e}/_{bottom\_cell}\right] \|_{\{p_u, e\}} Q\left[{}^{p_u}/_{pop_u}, {}^{e}/_{signal\_empty}\right]$$

  where P stands for $Changing$ and Q stands for $Cell_u$ or $Empty$, while any action with subscript u is a shorthand for all such actions.
- Upon $push$ with the external environment, the leftmost cell spawns a new cell to the left for the new top value w of the stack.
- Upon pop with the external environment, the leftmost cell takes on the value of the cell to the right (subtop) and this behavior propagates till the rightmost cell, which becomes empty (no garbage collection).

- Definition of the left-associative linking operator $\widehat{\ }$:

$$P \widehat{\ } Q \triangleq P\,[{}^{p_u}/_{subtop_u}, {}^{e}/_{bottom\_cell}] \parallel_{\{p_u,e\}} Q\,[{}^{p_u}/_{pop_u}, {}^{e}/_{signal\_empty}]$$

  where P stands for $Changing$ and Q stands for $Cell_u$ or $Empty$, while any action with subscript u is a shorthand for all such actions.
- Upon $push$ with the external environment, the leftmost cell spawns a new cell to the left for the new top value w of the stack.
- Upon pop with the external environment, the leftmost cell takes on the value of the cell to the right (subtop) and this behavior propagates till the rightmost cell, which becomes empty (no garbage collection).
- *Action prefix, alternative composition, recursion, parallel composition, and relabeling are all necessary to achieve Turing-completeness*.

6.5 Spectrum of Behavioral Equivalences

- Establishing whether two process terms are equivalent amounts to establishing whether the systems they represent behave the same.
- Compositional reasoning: substituting equals for equals in behaviors.
- Abstraction capabilities: behaving the same up to certain details.

6.5 Spectrum of Behavioral Equivalences

- Establishing whether two process terms are equivalent amounts to establishing whether the systems they represent behave the same.
- Compositional reasoning: substituting equals for equals in behaviors.
- Abstraction capabilities: behaving the same up to certain details.
- Useful for theoretical and applicative purposes:
  - Comparing process terms that are syntactically different on the basis of the behavior they exhibit.
  - Relating process algebraic descriptions of the same system at different abstraction levels (top-down modeling).
  - Manipulating process algebraic descriptions in a way that preserves certain properties (state space reduction before model checking).

- Features of a good behavioral equivalence:
  - Being a congruence with respect to all behavioral operators, so as to support compositional reasoning.

- Features of a good behavioral equivalence:
  - Being a congruence with respect to all behavioral operators, so as to support compositional reasoning.
  - Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).

- Features of a good behavioral equivalence:
  - Being a congruence with respect to all behavioral operators, so as to support compositional reasoning.
  - Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).
  - Having a logical characterization, which shows the behavioral properties preserved by the equivalence (diagnostics for inequivalence).

- Features of a good behavioral equivalence:
  - Being a congruence with respect to all behavioral operators, so as to support compositional reasoning.
  - Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).
  - Having a logical characterization, which shows the behavioral properties preserved by the equivalence (diagnostics for inequivalence).
  - Being equipped with an efficient verification algorithm, which runs in polynomial time in the worst case (finite-state systems – undecidable otherwise).

- Features of a good behavioral equivalence:
  - Being a congruence with respect to all behavioral operators, so as to support compositional reasoning.
  - Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).
  - Having a logical characterization, which shows the behavioral properties preserved by the equivalence (diagnostics for inequivalence).
  - Being equipped with an efficient verification algorithm, which runs in polynomial time in the worst case (finite-state systems – undecidable otherwise).
  - Being able to abstract from invisible actions.

- Features of a good behavioral equivalence:
  - Being a congruence with respect to all behavioral operators, so as to support compositional reasoning.
  - Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).
  - Having a logical characterization, which shows the behavioral properties preserved by the equivalence (diagnostics for inequivalence).
  - Being equipped with an efficient verification algorithm, which runs in polynomial time in the worst case (finite-state systems – undecidable otherwise).
  - Being able to abstract from invisible actions.
- Three fundamental approaches: *trace*, *bisimulation*, *testing* (1980's).

- Trace approach (Hoare et al.): two processes are equivalent iff they are able to execute the same sequences of visible actions ($\approx_{\mathrm{Tr}}$).
- Abstraction from branching points leads to deadlock insensitivity: $\mathrm{rec}\, X : (a \,.\, X + a \,.\, \underline{0}) \approx_{\mathrm{Tr}} \mathrm{rec}\, X : a \,.\, X$ but the first one can deadlock.

- Trace approach (Hoare et al.): two processes are equivalent iff they are able to execute the same sequences of visible actions ($\approx_{\mathrm{Tr}}$).
- Abstraction from branching points leads to deadlock insensitivity: $\mathrm{rec}\, X : (a \,.\, X + a \,.\, \underline{0}) \approx_{\mathrm{Tr}} \mathrm{rec}\, X : a \,.\, X$ but the first one can deadlock.
- Deadlock-sensitive (hence finer) variants of trace equivalence:
  - Completed-trace equivalence: compares process terms also with respect to traces that lead to deadlock ($\approx_{\mathrm{Tr,c}}$).
  - Failure equivalence: takes into account the set of visible actions that can be refused after executing a trace ($\approx_{\mathrm{F}}$).
  - Readiness equivalence: takes into account the set of visible actions that are enabled after executing a trace ($\approx_{\mathrm{R}}$).
  - Failure-trace equivalence: takes into account the sets of visible actions that can be refused after the individual steps of a trace ($\approx_{\mathrm{FTr}}$).
  - Ready-trace equivalence: takes into account the sets of visible actions that are enabled after the individual steps of a trace ($\approx_{\mathrm{RTr}}$).

- Trace approach (Hoare et al.): two processes are equivalent iff they are able to execute the same sequences of visible actions ($\approx_{\mathrm{Tr}}$).
- Abstraction from branching points leads to deadlock insensitivity: $\mathrm{rec}\, X : (a \,.\, X + a \,.\, \underline{0}) \approx_{\mathrm{Tr}} \mathrm{rec}\, X : a \,.\, X$ but the first one can deadlock.
- Deadlock-sensitive (hence finer) variants of trace equivalence:
  - Completed-trace equivalence: compares process terms also with respect to traces that lead to deadlock ($\approx_{\mathrm{Tr,c}}$).
  - Failure equivalence: takes into account the set of visible actions that can be refused after executing a trace ($\approx_{\mathrm{F}}$).
  - Readiness equivalence: takes into account the set of visible actions that are enabled after executing a trace ($\approx_{\mathrm{R}}$).
  - Failure-trace equivalence: takes into account the sets of visible actions that can be refused after the individual steps of a trace ($\approx_{\mathrm{FTr}}$).
  - Ready-trace equivalence: takes into account the sets of visible actions that are enabled after the individual steps of a trace ($\approx_{\mathrm{RTr}}$).
- Exponential time for trace equivalence checking in the worst case.

- Bisimulation approach (Park; Milner): two processes are equivalent iff they are able to mimic each other's behavior stepwise ($\sim_{\mathrm{B}}$).
- Faithful account of branching points leads to overdiscrimination: $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})$ is hardly justifiable.

- Bisimulation approach (Park; Milner): two processes are equivalent iff they are able to mimic each other's behavior stepwise ($\sim_{\mathrm{B}}$).
- Faithful account of branching points leads to overdiscrimination: $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})$ is hardly justifiable.
- Coarser variants of bisimulation equivalence:
  - Simulation equivalence: it is the intersection of two preorders, each considering the capability of stepwise behavior mimicking in one single direction ($\sim_{\mathrm{S}}$).
  - Failure-simulation equivalence: same as simulation equivalence, with each of the two preorders additionally checking for the equality of the sets of actions that can be stepwise refused ($\sim_{\mathrm{FS}}$).
  - Ready-simulation equivalence: same as simulation equivalence, with each of the two preorders additionally checking for the equality of the sets of actions that are stepwise enabled ($\sim_{\mathrm{RS}}$).

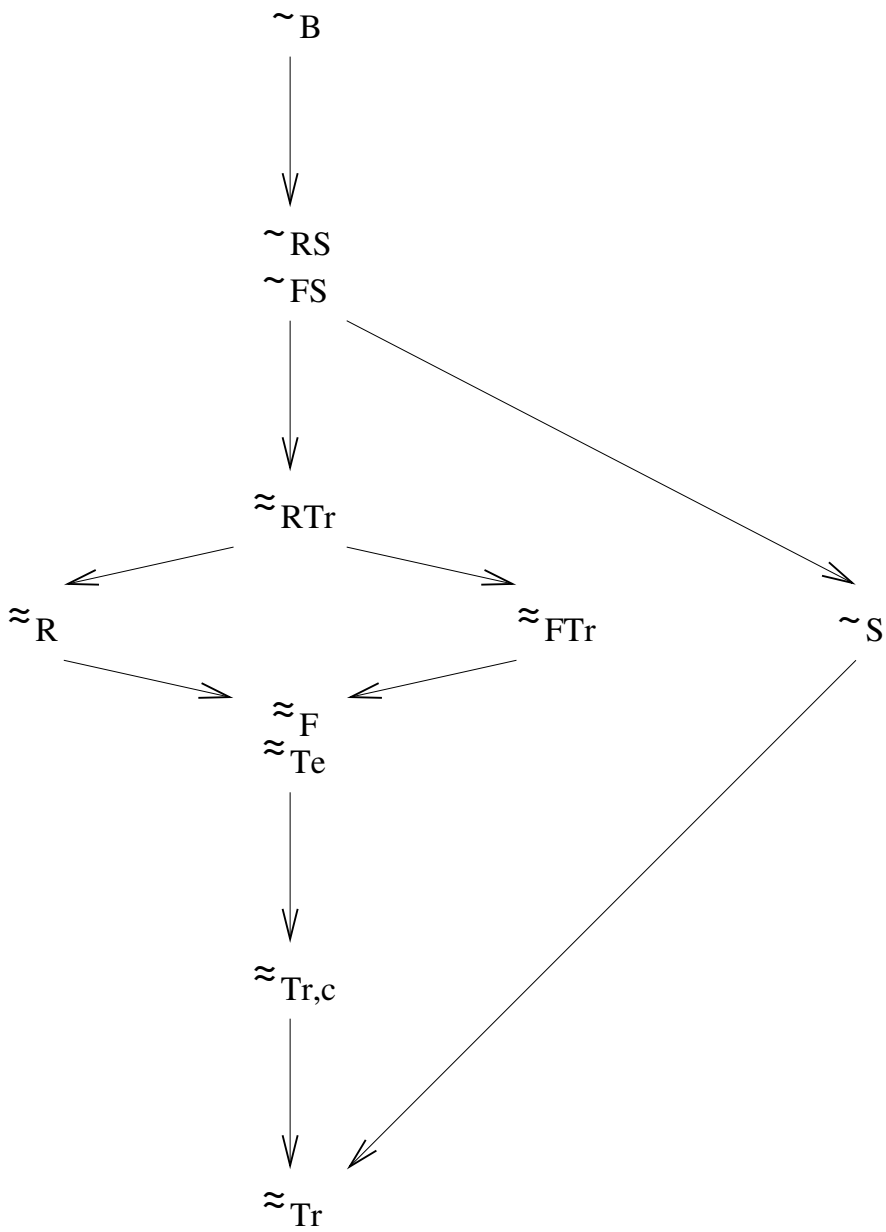
- Bisimulation approach (Park; Milner): two processes are equivalent iff they are able to mimic each other's behavior stepwise ($\sim_{\mathrm{B}}$).
- Faithful account of branching points leads to overdiscrimination: $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})$ is hardly justifiable.
- Coarser variants of bisimulation equivalence:
  - Simulation equivalence: it is the intersection of two preorders, each considering the capability of stepwise behavior mimicking in one single direction ($\sim_{\mathrm{S}}$).
  - Failure-simulation equivalence: same as simulation equivalence, with each of the two preorders additionally checking for the equality of the sets of actions that can be stepwise refused ($\sim_{\mathrm{FS}}$).
  - Ready-simulation equivalence: same as simulation equivalence, with each of the two preorders additionally checking for the equality of the sets of actions that are stepwise enabled ($\sim_{\mathrm{RS}}$).
- Abstraction capabilities later on (Milner; Van Glabbeek & Weijland).
- Decidable in polynomial time.

- Testing approach (De Nicola & Hennessy): two processes are equivalent iff their reaction to tests is the same ($\approx_{\mathrm{Te}}$).
- Tests formalized as processes extended with success action/state ω.
- Interaction between a process and a test formalized through their parallel composition with synchronization on any visible action.
- $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \approx_{\mathrm{Te}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})$ now holds.

- Testing approach (De Nicola & Hennessy): two processes are equivalent iff their reaction to tests is the same ($\approx_{\mathrm{Te}}$).
- Tests formalized as processes extended with success action/state ω.
- Interaction between a process and a test formalized through their parallel composition with synchronization on any visible action.
- $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \approx_{\mathrm{Te}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})$ now holds.
- Intersection of may-testing equivalence (at least one computation leads to success) and must-testing equivalence (all computations lead to success).
- May-testing equivalence coincides with trace equivalence.
- Testing equivalence coincides with failure equivalence in the case of nondiverging (no τ-loops), finitely-branching processes.
- The diverging process $\mathrm{rec}\,X : (\tau\,.\,X + a\,.\,\underline{0})$ is not must-testing equivalent to $a\,.\,\underline{0}$ because it can fail test $a\,.\,\omega$ (they are failure equivalent instead).

- Testing approach (De Nicola & Hennessy): two processes are equivalent iff their reaction to tests is the same ($\approx_{\mathrm{Te}}$).
- Tests formalized as processes extended with success action/state ω.
- Interaction between a process and a test formalized through their parallel composition with synchronization on any visible action.
- $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \approx_{\mathrm{Te}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})$ now holds.
- Intersection of may-testing equivalence (at least one computation leads to success) and must-testing equivalence (all computations lead to success).
- May-testing equivalence coincides with trace equivalence.
- Testing equivalence coincides with failure equivalence in the case of nondiverging (no τ-loops), finitely-branching processes.
- The diverging process $\mathrm{rec}\,X : (\tau\,.\,X + a\,.\,\underline{0})$ is not must-testing equivalent to $a\,.\,\underline{0}$ because it can fail test $a\,.\,\omega$ (they are failure equivalent instead).
- Checking whether every test may/must be passed is expensive.

- Linear-time/branching-time spectrum for finitely-branching processes with no τ-actions and hence no divergence (De Nicola; Van Glabbeek):



6.6 Strong Bisimilarity and Its Properties

- Two players: a challenger (action) and a defender (reaction).
- *Simulation game*: whenever the challenger performs a given action, then the defender has to respond with the same action and the two derivative states must be able to repeat this game.
- *Bisimulation game*: a simulation game in both directions.

6.6 Strong Bisimilarity and Its Properties

- Two players: a challenger (action) and a defender (reaction).
- *Simulation game*: whenever the challenger performs a given action, then the defender has to respond with the same action and the two derivative states must be able to repeat this game.
- *Bisimulation game*: a simulation game in both directions.
- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a strong bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for all actions $a \in \mathcal{A}$:
  - For each $P_1 \xrightarrow{a} P_1'$ there exists $P_2 \xrightarrow{a} P_2'$ such that $(P_1', P_2') \in \mathcal{B}$.
  - For each $P_2 \xrightarrow{a} P_2'$ there exists $P_1 \xrightarrow{a} P_1'$ such that $(P_1', P_2') \in \mathcal{B}$.
- Strong bisimulation equivalence or strong bisimilarity $\sim_{\mathrm{B}}$ is defined as the union of all strong bisimulations.
- Strong means not abstracting from τ-actions.

- Properties of strong bisimulations:
  - The identity relation over $\mathbb{P}$ is a strong bisimulation.
  - The inverse of a strong bisimulation is a strong bisimulation.
  - The composition of two strong bisimulations is a strong bisimulation.
  - The denumerable union of strong bisimulations is a strong bisimulation.
- Properties of strong bisimilarity: reflexivity, symmetry, transitivity.

- Properties of strong bisimulations:
  - The identity relation over $\mathbb{P}$ is a strong bisimulation.
  - The inverse of a strong bisimulation is a strong bisimulation.
  - The composition of two strong bisimulations is a strong bisimulation.
  - The denumerable union of strong bisimulations is a strong bisimulation.
- Properties of strong bisimilarity: reflexivity, symmetry, transitivity.
- *Coinductive definition*: strong bisimilarity is the *maximum fixed point* of the higher-order relation $\mathcal{F}$ such that, for all binary relations $\mathcal{R}$ and $P_1, P_2 \in \mathbb{P}$, $(P_1, P_2) \in \mathcal{F}(\mathcal{R})$ iff for all actions $a \in \mathcal{A}$:
  - For each $P_1 \xrightarrow{a} P_1'$ there exists $P_2 \xrightarrow{a} P_2'$ such that $(P_1', P_2') \in \mathcal{R}$.
  - For each $P_2 \xrightarrow{a} P_2'$ there exists $P_1 \xrightarrow{a} P_1'$ such that $(P_1', P_2') \in \mathcal{R}$.
- $\mathcal{R}$ is a strong bisimulation iff $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$.

- In order for $P_1 \sim_{\mathrm{B}} P_2$ it is necessary they enable the same actions, i.e., for all actions $a \in \mathcal{A}$ there exist $P_1', P_2' \in \mathbb{P}$ such that:

$$P_1 \xrightarrow{a} P_1' \iff P_2 \xrightarrow{a} P_2'$$

- In order for $P_1 \sim_{\mathrm{B}} P_2$ it is necessary they enable the same actions, i.e., for all actions $a \in \mathcal{A}$ there exist $P_1', P_2' \in \mathbb{P}$ such that:

$$P_1 \xrightarrow{a} P_1' \iff P_2 \xrightarrow{a} P_2'$$

- Strong bisimilarity is proven by finding a strong bisimulation.
- Focus on important pairs of processes in the strong bisimulation.
- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a strong bisimulation up to $\sim_{\mathrm{B}}$ iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for all actions $a \in \mathcal{A}$:
  - For each $P_1 \xrightarrow{a} P_1'$ there is $P_2 \xrightarrow{a} P_2'$ s.t. $P_1' \sim_{\mathrm{B}} Q_1 \mathcal{B} Q_2 \sim_{\mathrm{B}} P_2'$.
  - For each $P_2 \xrightarrow{a} P_2'$ there is $P_1 \xrightarrow{a} P_1'$ s.t. $P_1' \sim_{\mathrm{B}} Q_1 \mathcal{B} Q_2 \sim_{\mathrm{B}} P_2'$.
- In order for $P_1 \sim_{\mathrm{B}} P_2$ it is sufficient to find a strong bisimulation up to $\sim_{\mathrm{B}}$ that contains (P_1, P_2).

- Examples:
  - $a\,.\,\underline{0} + a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:

$$\mathcal{B} = \{(a\,.\,\underline{0} + a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$

- Examples:
  - $a\,.\,\underline{0} + a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
$$\mathcal{B} = \{(a\,.\,\underline{0} + a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$ as witnessed by the symm. closure of:
$$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ b\,.\,\underline{0}),\ (a\,.\,\underline{0} \parallel_{\emptyset} \underline{0},\ a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} \underline{0},\ \underline{0})\}$$

- Examples:
  - $a\,.\,\underline{0} + a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
$$\mathcal{B} = \{(a\,.\,\underline{0} + a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$ as witnessed by the symm. closure of:
$$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ b\,.\,\underline{0}),\ (a\,.\,\underline{0} \parallel_{\emptyset} \underline{0},\ a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} \underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\{a\}} a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
$$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\{a\}} a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$

- Examples:
  - $a\,.\,\underline{0} + a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
    $$\mathcal{B} = \{(a\,.\,\underline{0} + a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$ as witnessed by the symm. closure of:
    $$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ b\,.\,\underline{0}),\ (a\,.\,\underline{0} \parallel_{\emptyset} \underline{0},\ a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} \underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\{a\}} a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
    $$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\{a\}} a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$
  - $\mathrm{rec}\,X : (a\,.\,X + a\,.\,\underline{0}) \not\sim_{\mathrm{B}} \mathrm{rec}\,X : a\,.\,X\ [\approx_{\mathrm{Tr}}]$
    $\Longrightarrow \sim_{\mathrm{B}}$ is sensitive to deadlock.

- Examples:
  - $a\,.\,\underline{0} + a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
    $$\mathcal{B} = \{(a\,.\,\underline{0} + a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$ as witnessed by the symm. closure of:
    $$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} b\,.\,\underline{0},\ b\,.\,\underline{0}),\ (a\,.\,\underline{0} \parallel_{\emptyset} \underline{0},\ a\,.\,\underline{0}),\ (\underline{0} \parallel_{\emptyset} \underline{0},\ \underline{0})\}$$
  - $a\,.\,\underline{0} \parallel_{\{a\}} a\,.\,\underline{0} \sim_{\mathrm{B}} a\,.\,\underline{0}$ as witnessed by the symmetric closure of:
    $$\mathcal{B} = \{(a\,.\,\underline{0} \parallel_{\{a\}} a\,.\,\underline{0},\ a\,.\,\underline{0}),\ (\underline{0},\ \underline{0})\}$$
  - $\mathrm{rec}\,X : (a\,.\,X + a\,.\,\underline{0}) \not\sim_{\mathrm{B}} \mathrm{rec}\,X : a\,.\,X\ [\approx_{\mathrm{Tr}}]$
    $\Longrightarrow \sim_{\mathrm{B}}$ is sensitive to deadlock.
  - $a\,.\,b\,.\,\underline{0} + a\,.\,c\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,(b\,.\,\underline{0} + c\,.\,\underline{0})\ [\approx_{\mathrm{Tr}}]$ as well as
    $a\,.\,b\,.\,c\,.\,\underline{0} + a\,.\,b\,.\,d\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,(b\,.\,c\,.\,\underline{0} + b\,.\,d\,.\,\underline{0})\ [\approx_{\mathrm{Te}}, \approx_{\mathrm{Tr}}]$
    $\Longrightarrow \sim_{\mathrm{B}}$ is very sensitive to branching points.

- $\sim_{\mathrm{B}}$ is a congruence with respect to all dynamic and static operators as well as recursion.
- Substituting equals for equals does not alter the overall behavior in any process context (compositional manipulation).

- $\sim_{\mathrm{B}}$ is a congruence with respect to all dynamic and static operators as well as recursion.
- Substituting equals for equals does not alter the overall behavior in any process context (compositional manipulation).
- Let $P_1, P_2 \in \mathbb{P}$. If $P_1 \sim_{\mathrm{B}} P_2$ then:
  - $a \,.\, P_1 \sim_{\mathrm{B}} a \,.\, P_2$ for all $a \in \mathcal{A}$.
  - $P_1 + P \sim_{\mathrm{B}} P_2 + P$, $P + P_1 \sim_{\mathrm{B}} P + P_2$ for all $P \in \mathbb{P}$.
  - $P_1 \parallel_S P \sim_{\mathrm{B}} P_2 \parallel_S P$, $P \parallel_S P_1 \sim_{\mathrm{B}} P \parallel_S P_2$ for all $P \in \mathbb{P}, S \subseteq \mathcal{A}_{\mathrm{v}}$.
  - $P_1 / H \sim_{\mathrm{B}} P_2 / H$ for all $H \subseteq \mathcal{A}_{\mathrm{v}}$.
  - $P_1 \setminus L \sim_{\mathrm{B}} P_2 \setminus L$ for all $L \subseteq \mathcal{A}_{\mathrm{v}}$.
  - $P_1[\varphi] \sim_{\mathrm{B}} P_2[\varphi]$ for all $\varphi \in \mathit{Relab}$.

- Recursion: extend $\sim_{\mathrm{B}}$ to *open* process terms by replacing all variables freely occurring outside rec binders with every closed process term.

- Recursion: extend $\sim_{\mathrm{B}}$ to *open* process terms by replacing all variables freely occurring outside rec binders with every closed process term.
- Let P_1, P_2 be guarded process terms having free occurrences of at most $k \in \mathbb{N}$ process variables $X_1, \ldots, X_k \in Var$.
- Then we define $P_1 \sim_{\mathrm{B}} P_2$ iff for all $Q_1, \ldots, Q_k \in \mathbb{P}$:

$$P_1\{Q_i \hookrightarrow X_i \mid 1 \leq i \leq k\} \sim_{\mathrm{B}} P_2\{Q_i \hookrightarrow X_i \mid 1 \leq i \leq k\}$$

- Recursion: extend $\sim_{\mathrm{B}}$ to *open* process terms by replacing all variables freely occurring outside rec binders with every closed process term.
- Let P_1, P_2 be guarded process terms having free occurrences of at most $k \in \mathbb{N}$ process variables $X_1, \ldots, X_k \in Var$.
- Then we define $P_1 \sim_{\mathrm{B}} P_2$ iff for all $Q_1, \ldots, Q_k \in \mathbb{P}$:

$$P_1\{Q_i \hookrightarrow X_i \mid 1 \leq i \leq k\} \sim_{\mathrm{B}} P_2\{Q_i \hookrightarrow X_i \mid 1 \leq i \leq k\}$$

- If $P_1 \sim_{\mathrm{B}} P_2$ then $\mathrm{rec}\, X : P_1 \sim_{\mathrm{B}} \mathrm{rec}\, X : P_2$.

- $\sim_{\mathrm{B}}$ has a sound and complete axiomatization over the set $\mathbb{P}_{\mathrm{nrec}}$ of nonrecursive process terms of $\mathbb{P}$.
- Equational laws usable as rewriting rules.

- $\sim_{\mathrm{B}}$ has a sound and complete axiomatization over the set $\mathbb{P}_{\mathrm{nrec}}$ of nonrecursive process terms of $\mathbb{P}$.
- Equational laws usable as rewriting rules.
- Basic laws (associativity, commutativity, and neutral element of $+$):

$$\begin{array}{lrcl}
(\mathsf{A}_{\mathrm{B},1}) & (P_1 + P_2) + P_3 & = & P_1 + (P_2 + P_3) \\
(\mathsf{A}_{\mathrm{B},2}) & P_1 + P_2 & = & P_2 + P_1 \\
(\mathsf{A}_{\mathrm{B},3}) & P + \underline{0} & = & P
\end{array}$$

- Characterizing law (idempotency of $+$):

$$\begin{array}{lrcl}
(\mathsf{A}_{\mathrm{B},4}) & P + P & = & P
\end{array}$$

- $\sim_{\mathrm{B}}$ has a sound and complete axiomatization over the set $\mathbb{P}_{\mathrm{nrec}}$ of nonrecursive process terms of $\mathbb{P}$.
- Equational laws usable as rewriting rules.
- Basic laws (associativity, commutativity, and neutral element of $+$):

$$\begin{array}{lrcl} (\mathsf{A}_{\mathrm{B},1}) & (P_1 + P_2) + P_3 & = & P_1 + (P_2 + P_3) \\ (\mathsf{A}_{\mathrm{B},2}) & P_1 + P_2 & = & P_2 + P_1 \\ (\mathsf{A}_{\mathrm{B},3}) & P + \underline{0} & = & P \end{array}$$

- Characterizing law (idempotency of $+$):

$$\begin{array}{lrcl} (\mathsf{A}_{\mathrm{B},4}) & P + P & = & P \end{array}$$

- We can reduce any nonrecursive process in *sum normal form*, which is either $\underline{0}$ or $\sum_{i \in I} a_i \,.\, P_i$ with every P_i in sum normal form.
- How does $a \,.\, \underline{0} \,\|_{\emptyset}\, b \,.\, \underline{0}$ reduce to $a \,.\, b \,.\, \underline{0} + b \,.\, a \,.\, \underline{0}$?

- Expansion law (interleaving view of concurrency; I and J nonempty and finite):

$$(\mathrm{A}_{\mathrm{B},5}) \quad \sum_{i\in I} a_i \,.\, P_i \parallel_S \sum_{j\in J} b_j \,.\, Q_j = \sum_{k\in I, a_k\notin S} a_k \,.\, \left(P_k \parallel_S \sum_{j\in J} b_j \,.\, Q_j\right) + \sum_{h\in J, b_h\notin S} b_h \,.\, \left(\sum_{i\in I} a_i \,.\, P_i \parallel_S Q_h\right) + \sum_{k\in I, a_k\in S} \sum_{h\in J, b_h = a_k} a_k \,.\, (P_k \parallel_S Q_h)$$

$$(\mathrm{A}_{\mathrm{B},6}) \quad \sum_{i\in I} a_i \,.\, P_i \parallel_S \underline{0} = \sum_{k\in I, a_k\notin S} a_k \,.\, P_k$$

$$(\mathrm{A}_{\mathrm{B},7}) \quad \underline{0} \parallel_S \sum_{j\in J} b_j \,.\, Q_j = \sum_{h\in J, b_h\notin S} b_h \,.\, Q_h$$

$$(\mathrm{A}_{\mathrm{B},8}) \quad \underline{0} \parallel_S \underline{0} = \underline{0}$$

- Distribution laws (for unary static operators):

$$
\begin{array}{lrcll}
(\mathsf{A}_{\mathrm{B},9}) & \underline{0} \,/\, H & = & \underline{0} & \\
(\mathsf{A}_{\mathrm{B},10}) & (a \,.\, P) \,/\, H & = & \tau \,.\, (P \,/\, H) & \text{if } a \in H \\
(\mathsf{A}_{\mathrm{B},11}) & (a \,.\, P) \,/\, H & = & a \,.\, (P \,/\, H) & \text{if } a \notin H \\
(\mathsf{A}_{\mathrm{B},12}) & (P_1 + P_2) \,/\, H & = & P_1 \,/\, H + P_2 \,/\, H & \\
 & & & & \\
(\mathsf{A}_{\mathrm{B},13}) & \underline{0} \setminus L & = & \underline{0} & \\
(\mathsf{A}_{\mathrm{B},14}) & (a \,.\, P) \setminus L & = & \underline{0} & \text{if } a \in L \\
(\mathsf{A}_{\mathrm{B},15}) & (a \,.\, P) \setminus L & = & a \,.\, (P \setminus L) & \text{if } a \notin L \\
(\mathsf{A}_{\mathrm{B},16}) & (P_1 + P_2) \setminus L & = & P_1 \setminus L + P_2 \setminus L & \\
 & & & & \\
(\mathsf{A}_{\mathrm{B},17}) & \underline{0} \,[\varphi] & = & \underline{0} & \\
(\mathsf{A}_{\mathrm{B},18}) & (a \,.\, P) \,[\varphi] & = & \varphi(a) \,.\, (P \,[\varphi]) & \\
(\mathsf{A}_{\mathrm{B},19}) & (P_1 + P_2) \,[\varphi] & = & P_1 \,[\varphi] + P_2 \,[\varphi] &
\end{array}
$$

- $Ded(\mathsf{A}_{\mathrm{B}})$ is a deduction system based on all previous axioms plus:
  - Reflexivity: $\mathsf{A}_{\mathrm{B}} \vdash P = P$.
  - Symmetry: $\mathsf{A}_{\mathrm{B}} \vdash P_1 = P_2 \implies \mathsf{A}_{\mathrm{B}} \vdash P_2 = P_1$.
  - Transitivity: $\mathsf{A}_{\mathrm{B}} \vdash P_1 = P_2 \wedge \mathsf{A}_{\mathrm{B}} \vdash P_2 = P_3 \implies \mathsf{A}_{\mathrm{B}} \vdash P_1 = P_3$.
  - Substitutivity: $\mathsf{A}_{\mathrm{B}} \vdash P_1 = P_2 \implies \mathsf{A}_{\mathrm{B}} \vdash a \,.\, P_1 = a \,.\, P_2 \wedge \ldots$
- Remember that $\sim_{\mathrm{B}}$ is an equivalence relation and a congruence.

- $Ded(\mathsf{A}_\mathrm{B})$ is a deduction system based on all previous axioms plus:
  - Reflexivity: $\mathsf{A}_\mathrm{B} \vdash P = P$.
  - Symmetry: $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \implies \mathsf{A}_\mathrm{B} \vdash P_2 = P_1$.
  - Transitivity: $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \wedge \mathsf{A}_\mathrm{B} \vdash P_2 = P_3 \implies \mathsf{A}_\mathrm{B} \vdash P_1 = P_3$.
  - Substitutivity: $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \implies \mathsf{A}_\mathrm{B} \vdash a \,.\, P_1 = a \,.\, P_2 \wedge \ldots$
- Remember that $\sim_\mathrm{B}$ is an equivalence relation and a congruence.
- $Ded(\mathsf{A}_\mathrm{B})$ is sound and complete for $\sim_\mathrm{B}$ over $\mathbb{P}_\mathrm{nrec}$, i.e., $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \iff P_1 \sim_\mathrm{B} P_2$ for all $P_1, P_2 \in \mathbb{P}_\mathrm{nrec}$.

- $Ded(\mathsf{A}_\mathrm{B})$ is a deduction system based on all previous axioms plus:
  - Reflexivity: $\mathsf{A}_\mathrm{B} \vdash P = P$.
  - Symmetry: $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \implies \mathsf{A}_\mathrm{B} \vdash P_2 = P_1$.
  - Transitivity: $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \wedge \mathsf{A}_\mathrm{B} \vdash P_2 = P_3 \implies \mathsf{A}_\mathrm{B} \vdash P_1 = P_3$.
  - Substitutivity: $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \implies \mathsf{A}_\mathrm{B} \vdash a \,.\, P_1 = a \,.\, P_2 \wedge \ldots$
- Remember that $\sim_\mathrm{B}$ is an equivalence relation and a congruence.
- $Ded(\mathsf{A}_\mathrm{B})$ is sound and complete for $\sim_\mathrm{B}$ over $\mathbb{P}_\mathrm{nrec}$, i.e., $\mathsf{A}_\mathrm{B} \vdash P_1 = P_2 \iff P_1 \sim_\mathrm{B} P_2$ for all $P_1, P_2 \in \mathbb{P}_\mathrm{nrec}$.
- There is another deduction system that is sound and complete over the set of possibly recursive, sequential process terms of $\mathbb{P}$ (no static operators).

- $\sim_{\mathrm{B}}$ has a modal logic characterization based on a logic known as HML – Hennessy-Milner logic.
- Basic truth values and logical connectives, plus modal operators expressing how to behave after executing certain actions.

- $\sim_{\mathrm{B}}$ has a modal logic characterization based on a logic known as HML – Hennessy-Milner logic.
- Basic truth values and logical connectives, plus modal operators expressing how to behave after executing certain actions.
- Syntax of HML $(a \in \mathcal{A})$:

$$\begin{array}{rcll} \phi & ::= & \mathrm{true} & \text{basic truth value} \\ & | & \neg\phi & \text{negation} \\ & | & \phi \wedge \phi & \text{conjunction} \\ & | & \langle a \rangle \phi & \text{possibility} \end{array}$$

- $\sim_{\mathrm{B}}$ has a modal logic characterization based on a logic known as HML – Hennessy-Milner logic.
- Basic truth values and logical connectives, plus modal operators expressing how to behave after executing certain actions.
- Syntax of HML $(a \in \mathcal{A})$:

$$\begin{array}{rcll} \phi & ::= & \mathrm{true} & \text{basic truth value} \\ & | & \neg\phi & \text{negation} \\ & | & \phi \wedge \phi & \text{conjunction} \\ & | & \langle a \rangle \phi & \text{possibility} \end{array}$$

plus derived logical operators:

$$\begin{array}{rcll} \mathrm{false} & \equiv & \neg\mathrm{true} & \text{basic truth value} \\ \phi_1 \vee \phi_2 & \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) & \text{disjunction} \\ [a]\phi & \equiv & \neg\langle a \rangle \neg\phi & \text{necessity} \end{array}$$

- Interpretation of HML over $\mathbb{P}$:

$$
\begin{array}{lll}
P & \models & \text{true} \\
P & \models & \neg\phi \qquad \text{if } P \not\models \phi \\
P & \models & \phi_1 \wedge \phi_2 \qquad \text{if } P \models \phi_1 \text{ and } P \models \phi_2 \\
P & \models & \langle a \rangle \phi \qquad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \xrightarrow{a} P' \text{ and } P' \models \phi
\end{array}
$$

- Interpretation of HML over $\mathbb{P}$:

$$
\begin{array}{lll}
P & \models & \text{true} \\
P & \models & \neg\phi \qquad \text{if } P \not\models \phi \\
P & \models & \phi_1 \wedge \phi_2 \qquad \text{if } P \models \phi_1 \text{ and } P \models \phi_2 \\
P & \models & \langle a \rangle \phi \qquad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \xrightarrow{a} P' \text{ and } P' \models \phi
\end{array}
$$

plus derived logical operators:

$$
\begin{array}{lll}
P & \not\models & \text{false} \\
P & \models & \phi_1 \vee \phi_2 \qquad \text{if } P \models \phi_1 \text{ or } P \models \phi_2 \\
P & \models & [a]\phi \qquad \text{if for all } P' \in \mathbb{P}, \text{ whenever } P \xrightarrow{a} P', \text{ then } P' \models \phi
\end{array}
$$

- Interpretation of HML over $\mathbb{P}$:

$$
\begin{array}{lll}
P & \models & \text{true} \\
P & \models & \neg\phi \qquad \text{if } P \not\models \phi \\
P & \models & \phi_1 \wedge \phi_2 \qquad \text{if } P \models \phi_1 \text{ and } P \models \phi_2 \\
P & \models & \langle a \rangle \phi \qquad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \xrightarrow{a} P' \text{ and } P' \models \phi
\end{array}
$$

plus derived logical operators:

$$
\begin{array}{lll}
P & \not\models & \text{false} \\
P & \models & \phi_1 \vee \phi_2 \qquad \text{if } P \models \phi_1 \text{ or } P \models \phi_2 \\
P & \models & [a]\phi \qquad \text{if for all } P' \in \mathbb{P}, \text{ whenever } P \xrightarrow{a} P', \text{ then } P' \models \phi
\end{array}
$$

- $P_1 \sim_{\mathrm{B}} P_2 \iff (\forall \phi \in \mathrm{HML}.\, P_1 \models \phi \iff P_2 \models \phi)$ for all $P_1, P_2 \in \mathbb{P}$.

- Interpretation of HML over $\mathbb{P}$:

$$
\begin{array}{lll}
P & \models & \text{true} \\
P & \models & \neg\phi \qquad \text{if } P \not\models \phi \\
P & \models & \phi_1 \wedge \phi_2 \qquad \text{if } P \models \phi_1 \text{ and } P \models \phi_2 \\
P & \models & \langle a \rangle \phi \qquad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \xrightarrow{a} P' \text{ and } P' \models \phi
\end{array}
$$

plus derived logical operators:

$$
\begin{array}{lll}
P & \not\models & \text{false} \\
P & \models & \phi_1 \vee \phi_2 \qquad \text{if } P \models \phi_1 \text{ or } P \models \phi_2 \\
P & \models & [a]\phi \qquad \text{if for all } P' \in \mathbb{P}, \text{ whenever } P \xrightarrow{a} P', \text{ then } P' \models \phi
\end{array}
$$

- $P_1 \sim_{\mathrm{B}} P_2 \iff (\forall \phi \in \mathrm{HML}.\, P_1 \models \phi \iff P_2 \models \phi)$ for all $P_1, P_2 \in \mathbb{P}$.
- All HML-expressible properties are preserved by bisimilarity.
- If two processes are not bisimilar then they can be distinguished by at least one HML formula (explaining inequivalence).

- $\sim_{\mathrm{B}}$ has a temporal logic characterization based on CTL* (Browne, Clarke, Grümberg).
- *State formulae* with atomic propositions and logical connectives.
- *Path formulae* including temporal operators about the future.
- A path π is a sequence of states $s_0 s_1 s_2 \ldots$ such that $s_i \longrightarrow s_{i+1}$.

- $\sim_{\mathrm{B}}$ has a temporal logic characterization based on CTL* (Browne, Clarke, Grümberg).
- *State formulae* with atomic propositions and logical connectives.
- *Path formulae* including temporal operators about the future.
- A path π is a sequence of states $s_0 s_1 s_2 \ldots$ such that $s_i \longrightarrow s_{i+1}$.
- CTL* is interpreted over *Kripke structures*, hence we redefine $\sim_{\mathrm{B}}$ (propositions labeling states instead of actions labeling transitions).
- A binary relation $\mathcal{B}$ over a Kripke structure $(S, \mathcal{L}, \longrightarrow)$ is a strong bisimulation iff, whenever $(s_1, s_2) \in \mathcal{B}$, then $\mathcal{L}(s_1) = \mathcal{L}(s_2)$ and:
  - For each $s_1 \longrightarrow s_1'$ there exists $s_2 \longrightarrow s_2'$ such that $(s_1', s_2') \in \mathcal{B}$.
  - For each $s_2 \longrightarrow s_2'$ there exists $s_1 \longrightarrow s_1'$ such that $(s_1', s_2') \in \mathcal{B}$.
- Strong bisimilarity $\sim_{\mathrm{B}}$ is the union of all strong bisimulations.

- Syntax of CTL*:

$$
\begin{array}{rcll}
\phi & ::= & p & \text{atomic proposition} \\
 & | & \neg\phi & \text{state formula negation} \\
 & | & \phi \wedge \phi & \text{state formula conjunction} \\
 & | & \mathbf{E}\,\psi & \text{existential path quantifier} \\
 & & & \\
\psi & ::= & \phi & \text{state formula} \\
 & | & \neg\psi & \text{path formula negation} \\
 & | & \psi \wedge \psi & \text{path formula conjunction} \\
 & | & \mathbf{X}\,\psi & \text{next operator} \\
 & | & \psi\,\mathbf{U}\,\psi & \text{until operator}
\end{array}
$$

- Syntax of CTL*:

$$
\begin{array}{rcll}
\phi & ::= & p & \text{atomic proposition} \\
 & | & \neg\phi & \text{state formula negation} \\
 & | & \phi \wedge \phi & \text{state formula conjunction} \\
 & | & \mathbf{E}\,\psi & \text{existential path quantifier} \\
\psi & ::= & \phi & \text{state formula} \\
 & | & \neg\psi & \text{path formula negation} \\
 & | & \psi \wedge \psi & \text{path formula conjunction} \\
 & | & \mathbf{X}\,\psi & \text{next operator} \\
 & | & \psi\,\mathbf{U}\,\psi & \text{until operator}
\end{array}
$$

plus derived logical operators:

$$
\begin{array}{rcll}
\mathbf{A}\,\psi & \equiv & \neg\mathbf{E}\,\neg\psi & \text{universal path quantifier} \\
\mathbf{F}\,\psi & \equiv & \text{true}\,\mathbf{U}\,\psi & \text{eventually operator} \\
\mathbf{G}\,\psi & \equiv & \neg\mathbf{F}\,\neg\psi & \text{globally operator}
\end{array}
$$

- Given a path $\pi = s_0 s_1 s_2 \ldots$, we denote by $\pi[0]$ its initial state s_0 while π^k is its subpath starting at state s_k.
- Interpretation of CTL* over Kripke structure $(S, \mathcal{L}, \longrightarrow)$:

$$
\begin{array}{lcll}
s & \models & p & \text{if } p \in \mathcal{L}(s) \\
s & \models & \neg\phi & \text{if } s \not\models \phi \\
s & \models & \phi_1 \wedge \phi_2 & \text{if } s \models \phi_1 \text{ and } s \models \phi_2 \\
s & \models & \mathbf{E}\,\psi & \text{if there exists a path } \pi \text{ starting from } s \text{ such that } \pi \models \psi \\
\\
\pi & \models & \phi & \text{if } \pi[0] \models \phi \\
\pi & \models & \neg\psi & \text{if } \pi \not\models \psi \\
\pi & \models & \psi_1 \wedge \psi_2 & \text{if } \pi \models \psi_1 \text{ and } \pi \models \psi_2 \\
\pi & \models & \mathbf{X}\,\psi & \text{if } \pi^1 \models \psi \\
\pi & \models & \psi_1\,\mathbf{U}\,\psi_2 & \text{if } \pi^k \models \psi_2 \text{ for some } k \geq 0 \text{ and } \pi^i \models \psi_1 \text{ for all } 0 \leq i < k
\end{array}
$$

- Given a path $\pi = s_0 s_1 s_2 \ldots$, we denote by $\pi[0]$ its initial state s_0 while π^k is its subpath starting at state s_k.
- Interpretation of CTL* over Kripke structure $(S, \mathcal{L}, \longrightarrow)$:

$$
\begin{array}{lcll}
s & \models & p & \text{if } p \in \mathcal{L}(s) \\
s & \models & \neg\phi & \text{if } s \not\models \phi \\
s & \models & \phi_1 \wedge \phi_2 & \text{if } s \models \phi_1 \text{ and } s \models \phi_2 \\
s & \models & \mathbf{E}\,\psi & \text{if there exists a path } \pi \text{ starting from } s \text{ such that } \pi \models \psi \\
\\
\pi & \models & \phi & \text{if } \pi[0] \models \phi \\
\pi & \models & \neg\psi & \text{if } \pi \not\models \psi \\
\pi & \models & \psi_1 \wedge \psi_2 & \text{if } \pi \models \psi_1 \text{ and } \pi \models \psi_2 \\
\pi & \models & \mathbf{X}\,\psi & \text{if } \pi^1 \models \psi \\
\pi & \models & \psi_1\,\mathbf{U}\,\psi_2 & \text{if } \pi^k \models \psi_2 \text{ for some } k \geq 0 \text{ and } \pi^i \models \psi_1 \text{ for all } 0 \leq i < k
\end{array}
$$

- $s_1 \sim_{\mathrm{B}} s_2 \iff (\forall \phi \in \mathrm{CTL}^*.\, s_1 \models \phi \iff s_2 \models \phi)$ for all $s_1, s_2 \in S$.

- $\sim_{\mathrm{B}}$ is decidable in polynomial time over the set $\mathbb{P}_{\mathrm{fin}}$ of finite-state terms of $\mathbb{P}$ through Kanellakis-Smolka partition refinement algorithm.
- It is based on the fact that $\sim_{\mathrm{B}}$ can be characterized as the limit of a sequence of successively finer equivalence relations:

$$\sim_{\mathrm{B}} = \bigcap_{i \in \mathbb{N}} \sim_{\mathrm{B},i}$$

- $\sim_{\mathrm{B}}$ is decidable in polynomial time over the set $\mathbb{P}_{\mathrm{fin}}$ of finite-state terms of $\mathbb{P}$ through Kanellakis-Smolka partition refinement algorithm.
- It is based on the fact that $\sim_{\mathrm{B}}$ can be characterized as the limit of a sequence of successively finer equivalence relations:

$$\sim_{\mathrm{B}} = \bigcap_{i \in \mathbb{N}} \sim_{\mathrm{B},i}$$

- $\sim_{\mathrm{B},0} = \mathbb{P} \times \mathbb{P}$ thus inducing the trivial partition $\{\mathbb{P}\}$.
- $\sim_{\mathrm{B},1}$ refines $\{\mathbb{P}\}$ by creating an equivalence class for each set of process terms that satisfy the necessary condition for $\sim_{\mathrm{B}}$.
- Whenever $P_1 \sim_{\mathrm{B},i} P_2$, $i \in \mathbb{N}_{\geq 1}$, then for all actions $a \in \mathcal{A}$:
  - For each $P_1 \xrightarrow{a} P_1'$ there exists $P_2 \xrightarrow{a} P_2'$ such that $P_1' \sim_{\mathrm{B},i-1} P_2'$.
  - For each $P_2 \xrightarrow{a} P_2'$ there exists $P_1 \xrightarrow{a} P_1'$ such that $P_1' \sim_{\mathrm{B},i-1} P_2'$.

- Steps of the algorithm for checking whether $P_1 \sim_{\mathrm{B}} P_2$ $(P_1, P_2 \in \mathbb{P}_{\mathrm{fin}})$:
  1. Build an initial partition with a single class including all the states of $[\![P_1]\!]$ and all the states of $[\![P_2]\!]$.
  2. Initialize a list of splitters with the above class as its only element.
  3. While the list of splitters is not empty, select a splitter and remove it from the list after refining the current partition for all $a \in \mathcal{A}_{P_1,P_2}$:
     a. Split each class of the current partition by comparing its states when executing actions of name a that lead to the selected splitter.
     b. For each class that has been split, insert its smallest subclass into the list of splitters ("process the smaller half").
  4. Return yes/no depending on whether the initial states of $[\![P_1]\!]$ and $[\![P_2]\!]$ belong to the same class of the final partition or not.

- Steps of the algorithm for checking whether $P_1 \sim_{\mathrm{B}} P_2$ $(P_1, P_2 \in \mathbb{P}_{\mathrm{fin}})$:
  1. Build an initial partition with a single class including all the states of $[\![P_1]\!]$ and all the states of $[\![P_2]\!]$.
  2. Initialize a list of splitters with the above class as its only element.
  3. While the list of splitters is not empty, select a splitter and remove it from the list after refining the current partition for all $a \in \mathcal{A}_{P_1,P_2}$:
     a. Split each class of the current partition by comparing its states when executing actions of name a that lead to the selected splitter.
     b. For each class that has been split, insert its smallest subclass into the list of splitters ("process the smaller half").
  4. Return yes/no depending on whether the initial states of $[\![P_1]\!]$ and $[\![P_2]\!]$ belong to the same class of the final partition or not.
- The time complexity is $O(m \cdot \log n)$, where n is the number of states and m is the number of transitions of $[\![P_1]\!]$ and $[\![P_2]\!]$ as a whole.
- Also useful for state space minimization before model checking.

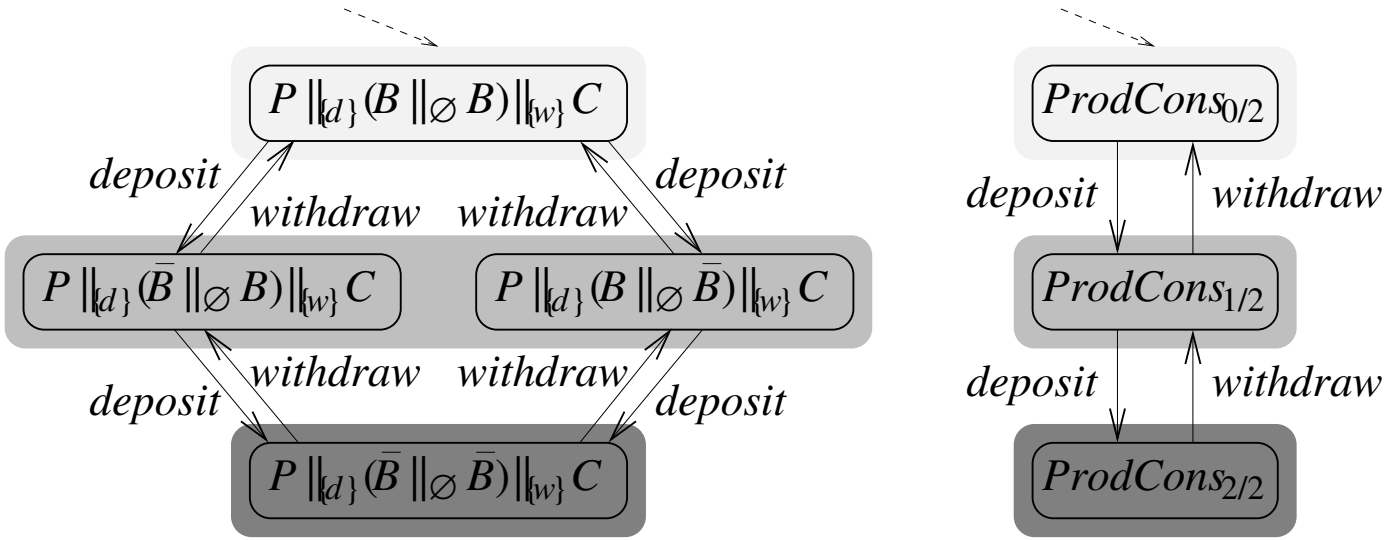
- **Running example** (strong bisimilarity):
  - Concurrent implementation with two independent one-position buffers:

$$
\begin{aligned}
PC_{\mathrm{conc},2} &\triangleq Prod \parallel_{\{deposit\}} (Buff \parallel_{\emptyset} Buff) \parallel_{\{withdraw\}} Cons \\
Prod &\triangleq deposit \,.\, Prod \\
Buff &\triangleq deposit \,.\, withdraw \,.\, Buff \\
Cons &\triangleq withdraw \,.\, Cons
\end{aligned}
$$

- **Running example** (strong bisimilarity):
  - Concurrent implementation with two independent one-position buffers:

$$
\begin{aligned}
PC_{\mathrm{conc},2} &\triangleq Prod \parallel_{\{deposit\}} (Buff \parallel_{\emptyset} Buff) \parallel_{\{withdraw\}} Cons \\
Prod &\triangleq deposit \,.\, Prod \\
Buff &\triangleq deposit \,.\, withdraw \,.\, Buff \\
Cons &\triangleq withdraw \,.\, Cons
\end{aligned}
$$

  - Strong bisimulation proving $PC_{\mathrm{conc},2} \sim_{\mathrm{B}} ProdCons_{0/2}$:



6.7 Weak Bisimilarities and Their Properties

- $\sim_{\mathrm{B}}$ does not abstract from invisible actions: $a\,.\,b\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,\tau\,.\,b\,.\,\underline{0}$.
- Two processes should be deemed equivalent in the bisimulation game when they are able to mimic each other's visible behavior stepwise.
- Need to extend the transition relation $\longrightarrow$ to action sequences.

6.7 Weak Bisimilarities and Their Properties

- $\sim_{\mathrm{B}}$ does not abstract from invisible actions: $a\,.\,b\,.\,\underline{0} \not\sim_{\mathrm{B}} a\,.\,\tau\,.\,b\,.\,\underline{0}$.
- Two processes should be deemed equivalent in the bisimulation game when they are able to mimic each other's visible behavior stepwise.
- Need to extend the transition relation $\longrightarrow$ to action sequences.
- $P \stackrel{a_1 \ldots a_n}{\Longrightarrow} P'$ iff:
  - either $n = 0$ and $P = P'$, meaning that P stays idle;
  - or $n \geq 1$ and there exist $P_0, P_1, \ldots, P_n \in \mathbb{P}$ such that:
    - $P = P_0$;
    - $P_{i-1} \stackrel{a_i}{\longrightarrow} P_i$ for all $1 \leq i \leq n$;
    - $P_n = P'$.
- τ^* denotes a possibly empty, finite sequence of τ-actions.
- $\tau^* a\, \tau^*$ denotes an a-action possibly preceded and followed by finitely many τ-actions.

- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a weak bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then:
  - For each $P_1 \xrightarrow{\tau} P_1'$ there exists $P_2 \stackrel{\tau^*}{\Longrightarrow} P_2'$ such that $(P_1', P_2') \in \mathcal{B}$.
  - For each $P_2 \xrightarrow{\tau} P_2'$ there exists $P_1 \stackrel{\tau^*}{\Longrightarrow} P_1'$ such that $(P_1', P_2') \in \mathcal{B}$.
  - For all visible actions $a \in \mathcal{A}_{\mathrm{v}}$:
    - For each $P_1 \xrightarrow{a} P_1'$ there exists $P_2 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_2'$ such that $(P_1', P_2') \in \mathcal{B}$.
    - For each $P_2 \xrightarrow{a} P_2'$ there exists $P_1 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_1'$ such that $(P_1', P_2') \in \mathcal{B}$.
- Weak bisimulation equivalence or weak bisimilarity $\approx_{\mathrm{B}}$ is the union of all weak bisimulations.

- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a weak bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then:
  - For each $P_1 \xrightarrow{\tau} P_1'$ there exists $P_2 \stackrel{\tau^*}{\Longrightarrow} P_2'$ such that $(P_1', P_2') \in \mathcal{B}$.
  - For each $P_2 \xrightarrow{\tau} P_2'$ there exists $P_1 \stackrel{\tau^*}{\Longrightarrow} P_1'$ such that $(P_1', P_2') \in \mathcal{B}$.
  - For all visible actions $a \in \mathcal{A}_{\mathrm{v}}$:
    - For each $P_1 \xrightarrow{a} P_1'$ there exists $P_2 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_2'$ such that $(P_1', P_2') \in \mathcal{B}$.
    - For each $P_2 \xrightarrow{a} P_2'$ there exists $P_1 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_1'$ such that $(P_1', P_2') \in \mathcal{B}$.
- Weak bisimulation equivalence or weak bisimilarity $\approx_{\mathrm{B}}$ is the union of all weak bisimulations.
- If $P_1 \sim_{\mathrm{B}} P_2$ then $P_1 \approx_{\mathrm{B}} P_2$ (the vice versa does not hold in general).
- The necessary condition for $\sim_{\mathrm{B}}$ is too restrictive for $\approx_{\mathrm{B}}$.
- The sufficient condition for $\sim_{\mathrm{B}}$ generalizes to $\approx_{\mathrm{B}}$ provided that weak bisimulation up to $\approx_{\mathrm{B}}$ uses $\stackrel{\tau^* a \tau^*}{\Longrightarrow}$ on the challenger side too.

- Examples:
  - $a\,.\,b\,.\,\underline{0} \approx_{\mathrm{B}} a\,.\,\tau\,.\,b\,.\,\underline{0}$ as witnessed by the symmetric closure of:

$$\begin{aligned}\mathcal{B} \;=\; \{&(a\,.\,b\,.\,\underline{0},\, a\,.\,\tau\,.\,b\,.\,\underline{0}),\\ &(b\,.\,\underline{0},\, \tau\,.\,b\,.\,\underline{0}),\\ &(b\,.\,\underline{0},\, b\,.\,\underline{0}),\\ &(\underline{0},\, \underline{0})\}\end{aligned}$$

    Note that:

    - The response to $b\,.\,\underline{0} \xrightarrow{b} \underline{0}$ is $\tau\,.\,b\,.\,\underline{0} \xrightarrow{\tau} \xrightarrow{b} \underline{0}$.
    - If $\tau\,.\,b\,.\,\underline{0} \xrightarrow{\tau} b\,.\,\underline{0}$ then $b\,.\,\underline{0}$ responds by staying idle.

- Examples:
  - $a\,.\,b\,.\,\underline{0} \approx_{\mathrm{B}} a\,.\,\tau\,.\,b\,.\,\underline{0}$ as witnessed by the symmetric closure of:

$$\begin{aligned}\mathcal{B} \;=\; \{&(a\,.\,b\,.\,\underline{0},\, a\,.\,\tau\,.\,b\,.\,\underline{0}),\\ &(b\,.\,\underline{0},\, \tau\,.\,b\,.\,\underline{0}),\\ &(b\,.\,\underline{0},\, b\,.\,\underline{0}),\\ &(\underline{0},\, \underline{0})\}\end{aligned}$$

    Note that:
    - The response to $b\,.\,\underline{0} \xrightarrow{b} \underline{0}$ is $\tau\,.\,b\,.\,\underline{0} \xrightarrow{\tau} \xrightarrow{b} \underline{0}$.
    - If $\tau\,.\,b\,.\,\underline{0} \xrightarrow{\tau} b\,.\,\underline{0}$ then $b\,.\,\underline{0}$ responds by staying idle.
  - $\tau\,.\,a\,.\,\underline{0} + b\,.\,\underline{0} \not\approx_{\mathrm{B}} a\,.\,\underline{0} + b\,.\,\underline{0}$ when $a \neq b$ because if $\tau\,.\,a\,.\,\underline{0} + b\,.\,\underline{0}$ performs τ thereby evolving into $a\,.\,\underline{0}$, which enables only a, then $a\,.\,\underline{0} + b\,.\,\underline{0}$ can only respond by staying idle but it enables both a and b hence at that point the weak bisimulation game cannot proceed further.

- $\approx_{\mathrm{B}}$ is a congruence with respect to all behavioral operators except for alternative composition (not a problem in practice).
- Additional τ-laws highlighting its abstraction capabilities:

$$\begin{aligned} \tau \,.\, P &= P \\ a \,.\, \tau \,.\, P &= a \,.\, P \\ P + \tau \,.\, P &= \tau \,.\, P \\ a \,.\, (P_1 + \tau \,.\, P_2) + a \,.\, P_2 &= a \,.\, (P_1 + \tau \,.\, P_2) \end{aligned}$$

- $\approx_{\mathrm{B}}$ is a congruence with respect to all behavioral operators except for alternative composition (not a problem in practice).
- Additional τ-laws highlighting its abstraction capabilities:

$$\begin{aligned} \tau \,.\, P &= P \\ a \,.\, \tau \,.\, P &= a \,.\, P \\ P + \tau \,.\, P &= \tau \,.\, P \\ a \,.\, (P_1 + \tau \,.\, P_2) + a \,.\, P_2 &= a \,.\, (P_1 + \tau \,.\, P_2) \end{aligned}$$

- Weak modal operators replacing those of HML ($a \in \mathcal{A}_{\mathrm{v}}$):

$$\begin{aligned} P &\models \langle\langle \tau \rangle\rangle \phi \quad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \stackrel{\tau^*}{\Longrightarrow} P' \text{ and } P' \models \phi \\ P &\models \langle\langle a \rangle\rangle \phi \quad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \stackrel{\tau^* a \tau^*}{\Longrightarrow} P' \text{ and } P' \models \phi \end{aligned}$$

- Temporal logic characterization based on CTL* without $\mathbf{X}$.

- $P_1 \approx_{\mathrm{B}} P_2$ can be decided in $O(n^2 \cdot m \cdot \log n)$ time with the verification algorithm for $\sim_{\mathrm{B}}$ preceded by the following preprocessing step:
  0. Build the reflexive and transitive closure of $\xrightarrow{\tau}$ in $[\![P_i]\!]$ for $i = 1, 2$:
     a. Add a looping τ-transition to each state.
     b. Add a τ-transition between the initial state and the final state of any sequence of at least two τ-transitions, if the two states are distinct and all the transitions in the sequence are distinct and nonlooping.
     c. Add an a-transition between the initial state and the final state of any sequence of at least two transitions in which one is labeled with $a \in \mathcal{A}_{\mathrm{v}}$, if all the other transitions in the sequence are labeled with τ, distinct, and nonlooping.

- The fact that $\approx_{\mathrm{B}}$ is not a congruence with respect to the alternative composition operator stems from $\tau \,.\, P = P$ (abstraction from initial τ-actions).
- This τ-law cannot be freely used when P does not enable τ-actions: $\tau \,.\, a \,.\, \underline{0} \approx_{\mathrm{B}} a \,.\, \underline{0}$ but $\tau \,.\, a \,.\, \underline{0} + b \,.\, \underline{0} \not\approx_{\mathrm{B}} a \,.\, \underline{0} + b \,.\, \underline{0}$ if $a \neq b$.
- Congruence w.r.t. the alternative composition operator can be restored by enforcing a matching on *initial* τ-actions in the game.

- The fact that $\approx_{\mathrm{B}}$ is not a congruence with respect to the alternative composition operator stems from $\tau \,.\, P = P$ (abstraction from initial τ-actions).
- This τ-law cannot be freely used when P does not enable τ-actions: $\tau \,.\, a \,.\, \underline{0} \approx_{\mathrm{B}} a \,.\, \underline{0}$ but $\tau \,.\, a \,.\, \underline{0} + b \,.\, \underline{0} \not\approx_{\mathrm{B}} a \,.\, \underline{0} + b \,.\, \underline{0}$ if $a \neq b$.
- Congruence w.r.t. the alternative composition operator can be restored by enforcing a matching on *initial* τ-actions in the game.
- $P_1 \in \mathbb{P}$ is weakly bisimulation congruent to $P_2 \in \mathbb{P}$, written $P_1 \approx_{\mathrm{B}}^{\mathrm{c}} P_2$, iff for all actions $a \in \mathcal{A}$ (hence including τ):
  - For each $P_1 \xrightarrow{a} P_1'$ there exists $P_2 \xRightarrow{\tau^* a \tau^*} P_2'$ such that $P_1' \approx_{\mathrm{B}} P_2'$.
  - For each $P_2 \xrightarrow{a} P_2'$ there exists $P_1 \xRightarrow{\tau^* a \tau^*} P_1'$ such that $P_1' \approx_{\mathrm{B}} P_2'$.
- $\approx_{\mathrm{B}}^{\mathrm{c}}$ is strictly finer than $\approx_{\mathrm{B}}$: $\tau \,.\, a \,.\, \underline{0} \not\approx_{\mathrm{B}}^{\mathrm{c}} a \,.\, \underline{0}$.
- $\approx_{\mathrm{B}}^{\mathrm{c}}$ is the largest congruence with respect to $+$ contained in $\approx_{\mathrm{B}}$.

- $\approx_{\mathrm{B}}$ and $\approx_{\mathrm{B}}^{\mathrm{c}}$ *may not fully respect the branching structure* of processes when abstracting from τ-actions.
- Given $P_1 \approx_{\mathrm{B}} P_2$, for each $P_1 \xrightarrow{a} P_1'$ with $a \in \mathcal{A}_{\mathrm{v}}$ there must exist $P_2 \xRightarrow{\tau^*} \bar{P}_2 \xrightarrow{a} \bar{P}_2' \xRightarrow{\tau^*} P_2'$ with $P_1' \approx_{\mathrm{B}} P_2'$, but we do not know whether any relation exists between P_1 and $\bar{P}_2$ as well as P_1' and $\bar{P}_2'$.

- $\approx_{\mathrm{B}}$ and $\approx_{\mathrm{B}}^{\mathrm{c}}$ *may not fully respect the branching structure* of processes when abstracting from τ-actions.
- Given $P_1 \approx_{\mathrm{B}} P_2$, for each $P_1 \xrightarrow{a} P_1'$ with $a \in \mathcal{A}_{\mathrm{v}}$ there must exist $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2 \xrightarrow{a} \bar{P}_2' \stackrel{\tau^*}{\Longrightarrow} P_2'$ with $P_1' \approx_{\mathrm{B}} P_2'$, but we do not know whether any relation exists between P_1 and $\bar{P}_2$ as well as P_1' and $\bar{P}_2'$.
- A symmetric binary relation $\mathcal{B}$ over $\mathbb{P}$ is a branching bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for each $P_1 \xrightarrow{a} P_1'$:
  - either $a = \tau$ and $(P_1', P_2) \in \mathcal{B}$;
  - or there is $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2 \xrightarrow{a} P_2'$ s.t. $(P_1, \bar{P}_2) \in \mathcal{B}$ and $(P_1', P_2') \in \mathcal{B}$.
- Branching bisimulation equivalence or branching bisimilarity $\approx_{\mathrm{B,b}}$ is the union of all branching bisimulations (Van Glabbeek & Weijland).

- $\approx_{\mathrm{B}}$ and $\approx_{\mathrm{B}}^{\mathrm{c}}$ *may not fully respect the branching structure* of processes when abstracting from τ-actions.
- Given $P_1 \approx_{\mathrm{B}} P_2$, for each $P_1 \xrightarrow{a} P_1'$ with $a \in \mathcal{A}_{\mathrm{v}}$ there must exist $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2 \xrightarrow{a} \bar{P}_2' \stackrel{\tau^*}{\Longrightarrow} P_2'$ with $P_1' \approx_{\mathrm{B}} P_2'$, but we do not know whether any relation exists between P_1 and $\bar{P}_2$ as well as P_1' and $\bar{P}_2'$.
- A symmetric binary relation $\mathcal{B}$ over $\mathbb{P}$ is a branching bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for each $P_1 \xrightarrow{a} P_1'$:
  - either $a = \tau$ and $(P_1', P_2) \in \mathcal{B}$;
  - or there is $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2 \xrightarrow{a} P_2'$ s.t. $(P_1, \bar{P}_2) \in \mathcal{B}$ and $(P_1', P_2') \in \mathcal{B}$.
- Branching bisimulation equivalence or branching bisimilarity $\approx_{\mathrm{B,b}}$ is the union of all branching bisimulations (Van Glabbeek & Weijland).
- Requiring $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2 \xrightarrow{a} \bar{P}_2' \stackrel{\tau^*}{\Longrightarrow} P_2'$ s.t. $(P_1, \bar{P}_2) \in \mathcal{B}$ and $(P_1', \bar{P}_2') \in \mathcal{B}$ would not change the distinguishing power of $\approx_{\mathrm{B,b}}$.

- Stuttering property: the branching bisimulation equivalence class does not change while performing τ-actions, i.e., all processes along $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2$ are branching bisimilar to each other.

- Stuttering property: the branching bisimulation equivalence class does not change while performing τ-actions, i.e., all processes along $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2$ are branching bisimilar to each other.
- $\approx_{\mathrm{B,b}}$ is strictly finer than $\approx_{\mathrm{B}}$: $\tau \,.\, a \,.\, \underline{0} + b \,.\, \underline{0}$ and $\tau \,.\, a \,.\, \underline{0} + a \,.\, \underline{0} + b \,.\, \underline{0}$ are identified by $\approx_{\mathrm{B}}$ and told apart by $\approx_{\mathrm{B,b}}$.
- $\approx_{\mathrm{B,b}}$ coincides with $\approx_{\mathrm{B}}$ on any pair of weakly bisimilar processes such that at most one of them reaches a process enabling τ-actions.

- Stuttering property: the branching bisimulation equivalence class does not change while performing τ-actions, i.e., all processes along $P_2 \stackrel{\tau^*}{\Longrightarrow} \bar{P}_2$ are branching bisimilar to each other.
- $\approx_{\mathrm{B,b}}$ is strictly finer than $\approx_{\mathrm{B}}$: $\tau \,.\, a \,.\, \underline{0} + b \,.\, \underline{0}$ and $\tau \,.\, a \,.\, \underline{0} + a \,.\, \underline{0} + b \,.\, \underline{0}$ are identified by $\approx_{\mathrm{B}}$ and told apart by $\approx_{\mathrm{B,b}}$.
- $\approx_{\mathrm{B,b}}$ coincides with $\approx_{\mathrm{B}}$ on any pair of weakly bisimilar processes such that at most one of them reaches a process enabling τ-actions.
- Same compositionality issue with respect to $+$.
- A single τ-law: $a \,.\, (\tau \,.\, (P_1 + P_2) + P_1) = a \,.\, (P_1 + P_2)$.
- A single weak modality: $\phi_1 \, \langle\langle a \rangle\rangle \, \phi_2$ is satisfied by P iff either $a = \tau$ with P satisfying ϕ_2, or $P \stackrel{\tau^*}{\Longrightarrow} \bar{P} \stackrel{a}{\longrightarrow} P'$ with every process along $P \stackrel{\tau^*}{\Longrightarrow} \bar{P}$ satisfying ϕ_1 and P' satisfying ϕ_2 (reminiscent of until).
- Can be decided more efficiently: $O(m \cdot \log n)$ time (Groote et al).

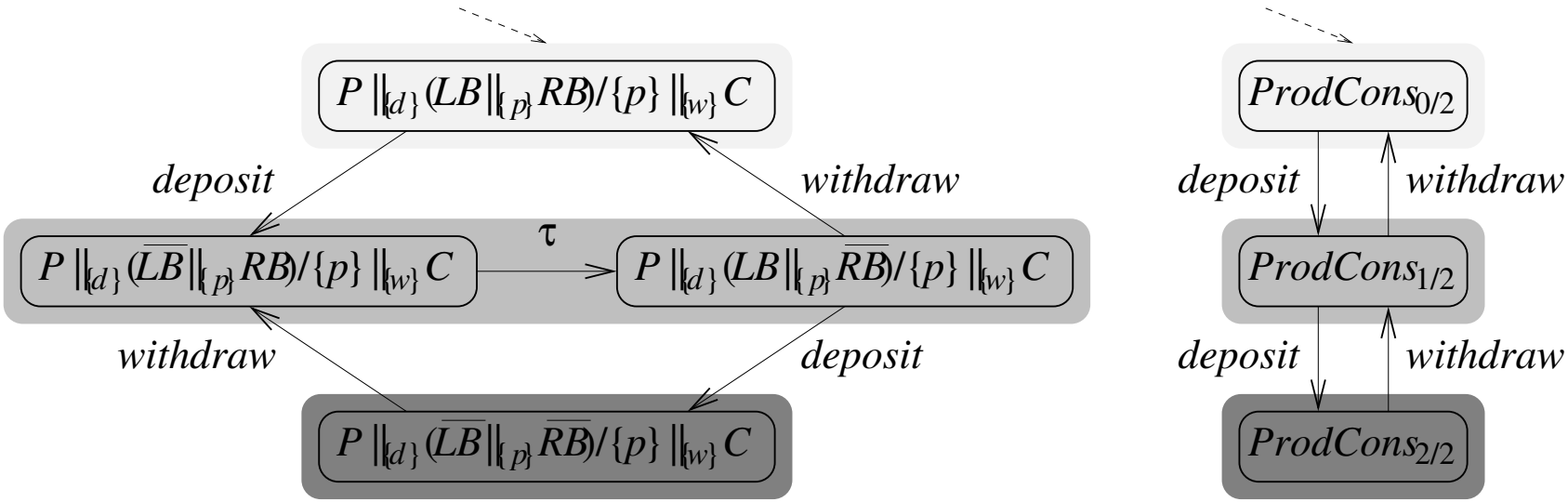
- **Running example** (weak/branching bisimilarity):
  - Pipeline implementation with two communicating one-position buffers:

$$
\begin{aligned}
PC_{\mathrm{pipe},2} &\triangleq Prod \parallel_{\{deposit\}} (LBuff \parallel_{\{pass\}} RBuff)/\{pass\} \parallel_{\{withdraw\}} Cons \\
Prod &\triangleq deposit \,.\, Prod \\
LBuff &\triangleq deposit \,.\, pass \,.\, LBuff \\
RBuff &\triangleq pass \,.\, withdraw \,.\, RBuff \\
Cons &\triangleq withdraw \,.\, Cons
\end{aligned}
$$

- **Running example** (weak/branching bisimilarity):
  - Pipeline implementation with two communicating one-position buffers:

$$
\begin{aligned}
PC_{\mathrm{pipe},2} &\triangleq Prod \parallel_{\{deposit\}} (LBuff \parallel_{\{pass\}} RBuff)/\{pass\} \parallel_{\{withdraw\}} Cons \\
Prod &\triangleq deposit \,.\, Prod \\
LBuff &\triangleq deposit \,.\, pass \,.\, LBuff \\
RBuff &\triangleq pass \,.\, withdraw \,.\, RBuff \\
Cons &\triangleq withdraw \,.\, Cons
\end{aligned}
$$

  - Weak/branching bisimulation identifying $PC_{\mathrm{pipe},2}$ and $ProdCons_{0/2}$:



6.8 Truly Concurrent Semantics via Petri Nets

- Petri nets (1962) are bipartite graphs yielding truly concurrent models whose vertices are respectively called places and transitions.
- A labeled Petri net is a tuple $N = (Pl, \mathcal{A}, Tr, M_0)$ where:
  - Pl is a set of places.
  - $Tr \subseteq \mathcal{M}u_{\text{fin}}(Pl) \times \mathcal{A} \times \mathcal{M}u_{\text{fin}}(Pl)$ is a set of labeled transitions.
  - $M_0 \in \mathcal{M}u_{\text{fin}}(Pl)$ is the initial marking.

6.8 Truly Concurrent Semantics via Petri Nets

- Petri nets (1962) are bipartite graphs yielding truly concurrent models whose vertices are respectively called places and transitions.
- A labeled Petri net is a tuple $N = (Pl, \mathcal{A}, Tr, M_0)$ where:
  - Pl is a set of places.
  - $Tr \subseteq \mathcal{M}u_{\text{fin}}(Pl) \times \mathcal{A} \times \mathcal{M}u_{\text{fin}}(Pl)$ is a set of labeled transitions.
  - $M_0 \in \mathcal{M}u_{\text{fin}}(Pl)$ is the initial marking.
- The notion of state is distributed among places marked with tokens, while transitions correspond to activities or events.
- A place marking is a function $M : Pl \to \mathbb{N}$ (token multiplicity), which belongs to $\mathcal{M}u_{\text{fin}}(Pl)$ iff $\{p \in Pl \mid M(p) > 0\}$ is finite.

6.8 Truly Concurrent Semantics via Petri Nets

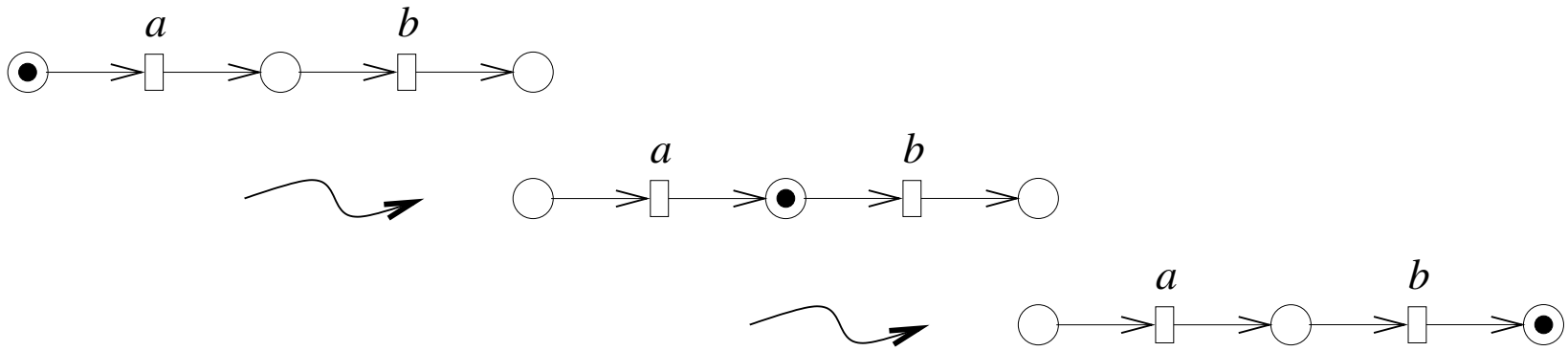
- Petri nets (1962) are bipartite graphs yielding truly concurrent models whose vertices are respectively called places and transitions.
- A labeled Petri net is a tuple $N = (Pl, \mathcal{A}, Tr, M_0)$ where:
  - Pl is a set of places.
  - $Tr \subseteq \mathcal{M}u_{\text{fin}}(Pl) \times \mathcal{A} \times \mathcal{M}u_{\text{fin}}(Pl)$ is a set of labeled transitions.
  - $M_0 \in \mathcal{M}u_{\text{fin}}(Pl)$ is the initial marking.
- The notion of state is distributed among places marked with tokens, while transitions correspond to activities or events.
- A place marking is a function $M : Pl \to \mathbb{N}$ (token multiplicity), which belongs to $\mathcal{M}u_{\text{fin}}(Pl)$ iff $\{p \in Pl \mid M(p) > 0\}$ is finite.
- Places are drawn as circles, transitions are drawn as boxes.
- $M(p)$ black dots are drawn inside $p \in Pl$ if M is the current marking.

- Each transition $t \in Tr$ can be written as ${}^\bullet t \xrightarrow{a} t^\bullet$ where:
  - ${}^\bullet t$ is the weighted preset of t (no. tokens consumed in each place).
  - $t^\bullet$ is the weighted postset of t (no. tokens produced in each place).
- An arrow-headed arc is drawn from every place in ${}^\bullet t$ to t as well as from t to every place in $t^\bullet$, each labeled with its token multiplicity.

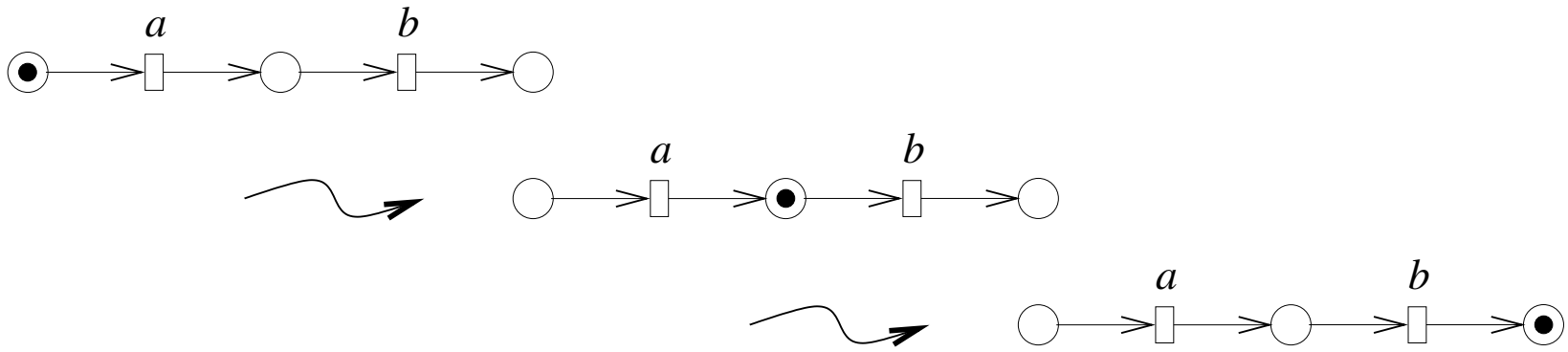
- Each transition $t \in Tr$ can be written as ${}^\bullet t \xrightarrow{a} t^\bullet$ where:
  - ${}^\bullet t$ is the weighted preset of t (no. tokens consumed in each place).
  - $t^\bullet$ is the weighted postset of t (no. tokens produced in each place).
- An arrow-headed arc is drawn from every place in ${}^\bullet t$ to t as well as from t to every place in $t^\bullet$, each labeled with its token multiplicity.
- Transition t is enabled at marking $M \in \mathcal{M}u_{\text{fin}}(Pl)$ iff ${}^\bullet t \subseteq M$.
- The firing of t enabled at M produces marking $M' = (M \setminus {}^\bullet t) \cup t^\bullet$, written $M\,[a\rangle\,M'$ if t is labeled with a.

- Each transition $t \in Tr$ can be written as ${}^\bullet t \xrightarrow{a} t^\bullet$ where:
  - ${}^\bullet t$ is the weighted preset of t (no. tokens consumed in each place).
  - $t^\bullet$ is the weighted postset of t (no. tokens produced in each place).
- An arrow-headed arc is drawn from every place in ${}^\bullet t$ to t as well as from t to every place in $t^\bullet$, each labeled with its token multiplicity.
- Transition t is enabled at marking $M \in \mathcal{M}u_{\text{fin}}(Pl)$ iff ${}^\bullet t \subseteq M$.
- The firing of t enabled at M produces marking $M' = (M \setminus {}^\bullet t) \cup t^\bullet$, written $M\,[a\rangle\,M'$ if t is labeled with a.
- The reachability set $RS(M)$ of marking $M \in \mathcal{M}u_{\text{fin}}(Pl)$ is the smallest subset of $\mathcal{M}u_{\text{fin}}(Pl)$ such that:
  - $M \in RS(M)$.
  - If $M_1 \in RS(M)$ and $M_1\,[a\rangle\,M_2$, then $M_2 \in RS(M)$.
- The reachability graph (or interleaving marking graph) of N is the LTS $\mathcal{RG}[\![N]\!] = (RS(M_0), \mathcal{A}, [\rangle, M_0)$.

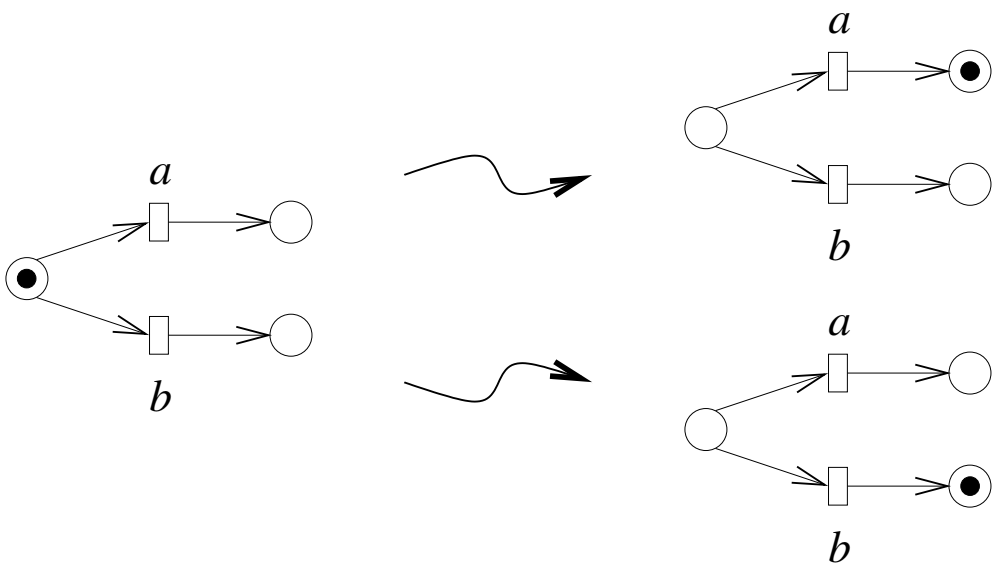
- Examples:
  - Sequentiality/causality:



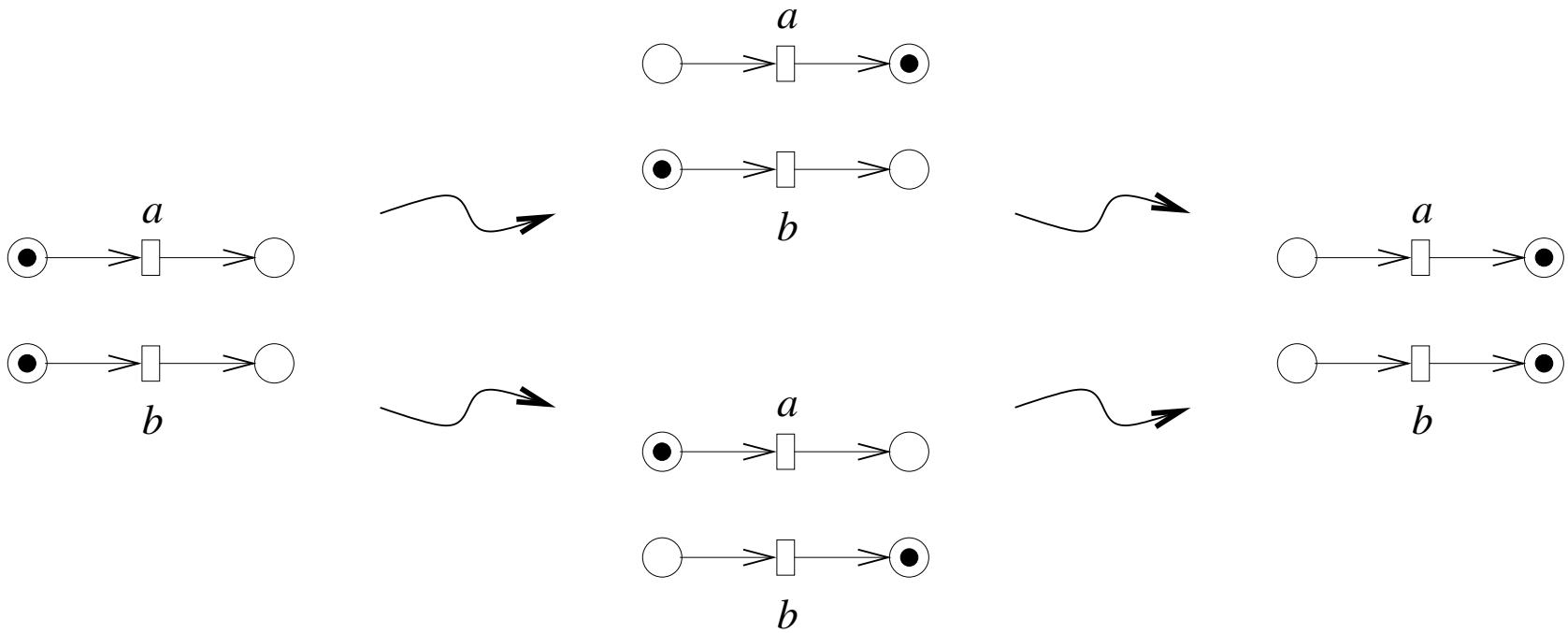
- Examples:
  - Sequentiality/causality:



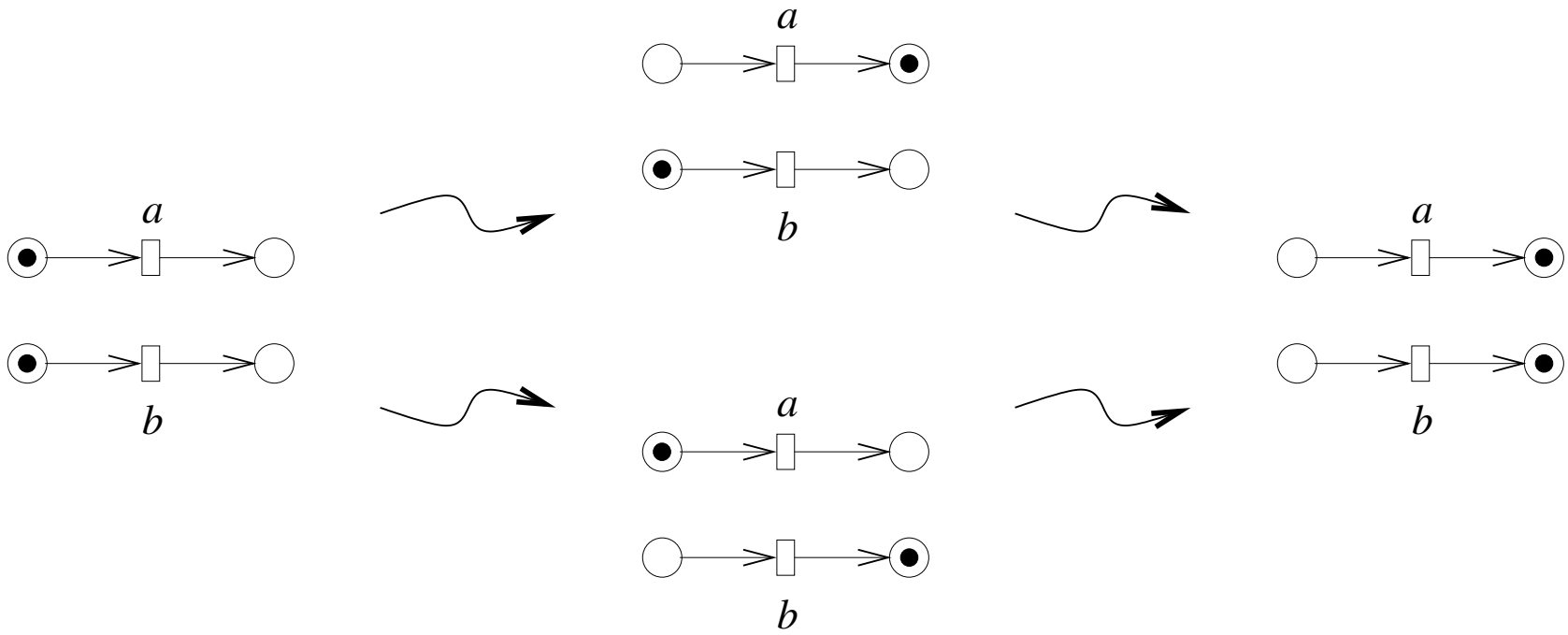
  - Choice/conflict:



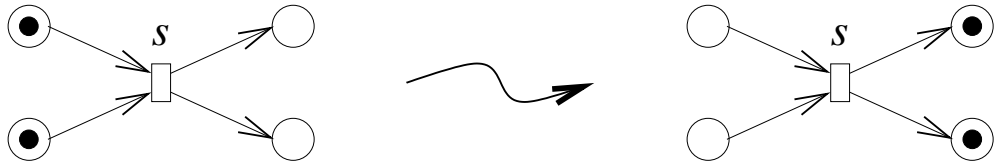
- Examples:
  - Concurrency:



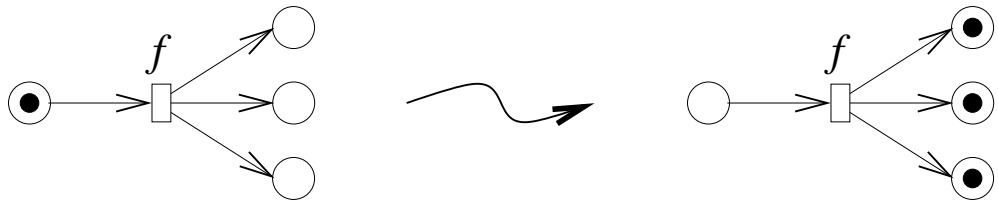
- Examples:
  - Concurrency:



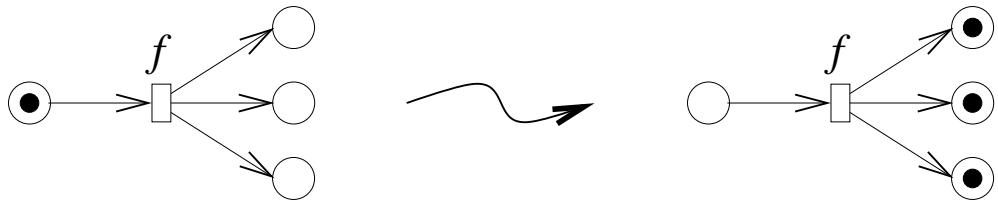
  - Synchronization:



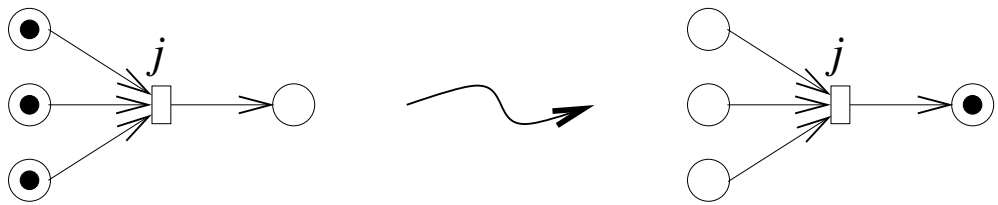
- Examples:
  - Fork:



- Examples:
  - Fork:



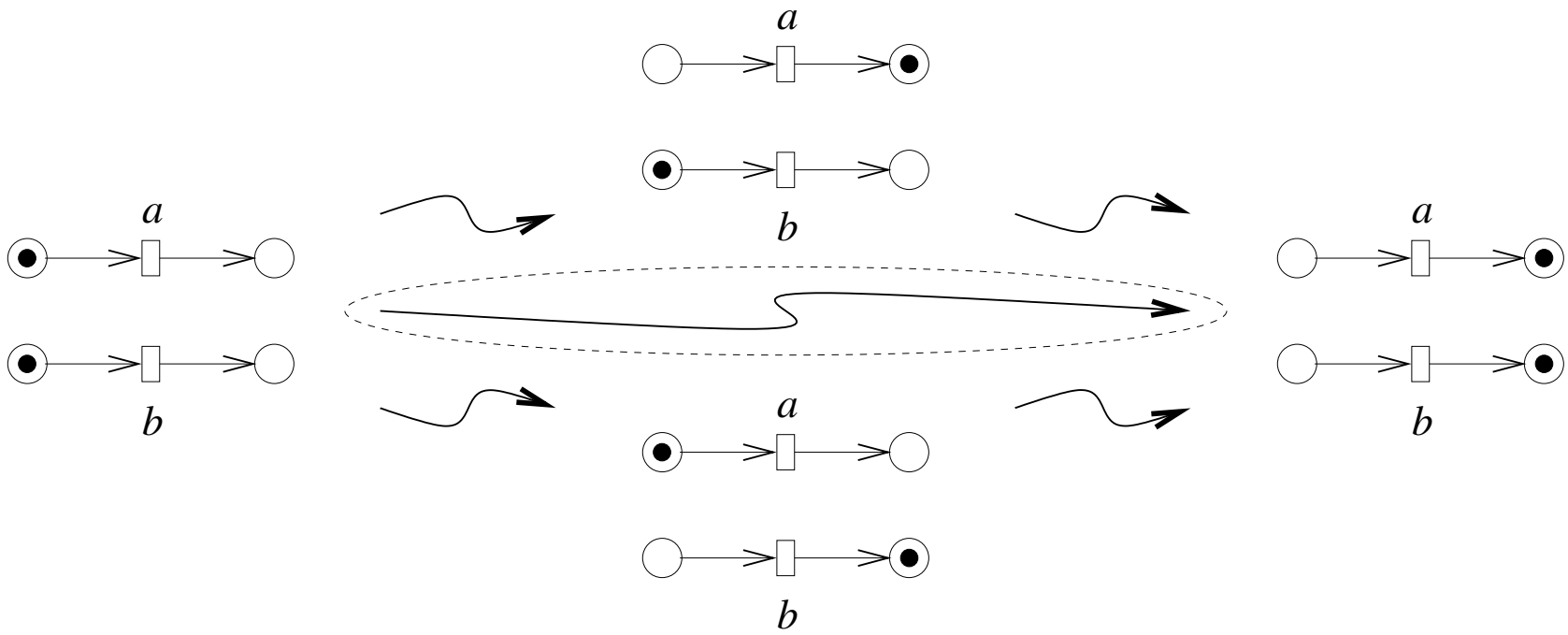
  - Join:



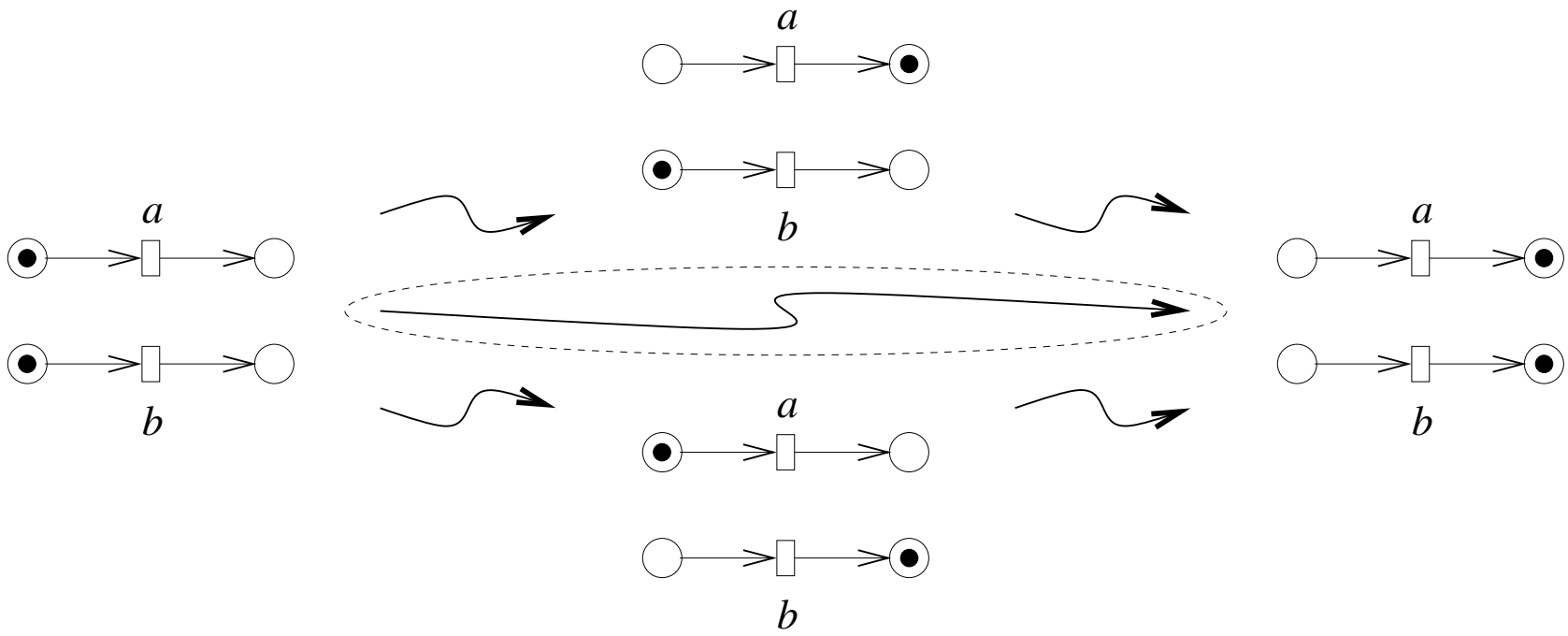
- Classification of Petri nets based on tokens:
  - Condition/event nets: every place can contain at most one token (occurrence nets).
  - Place/transition nets: every place can contain several tokens that are indistinguishable.
  - Predicate/event nets: every place can contain several tokens of different kinds (colored Petri nets).
- Petri nets have the same computational power as Turing machines if inhibitor arcs are admitted (inhibit transitions when tokens are *present* in their presets).

- Classification of Petri nets based on tokens:
  - Condition/event nets: every place can contain at most one token (occurrence nets).
  - Place/transition nets: every place can contain several tokens that are indistinguishable.
  - Predicate/event nets: every place can contain several tokens of different kinds (colored Petri nets).
- Petri nets have the same computational power as Turing machines if inhibitor arcs are admitted (inhibit transitions when tokens are *present* in their presets).
- Concurrent variants of the reachability graph are also possible.
- Step semantics: several transitions can fire *simultaneously* in one step as long as their presets and their postsets are pairwise disjoint.
- ST semantics: the beginning of each transition firing (t^+) is separate from the end of the corresponding transition firing (t^-), so as to gain real-time consistency (Van Glabbeek & Vaandrager 1987).

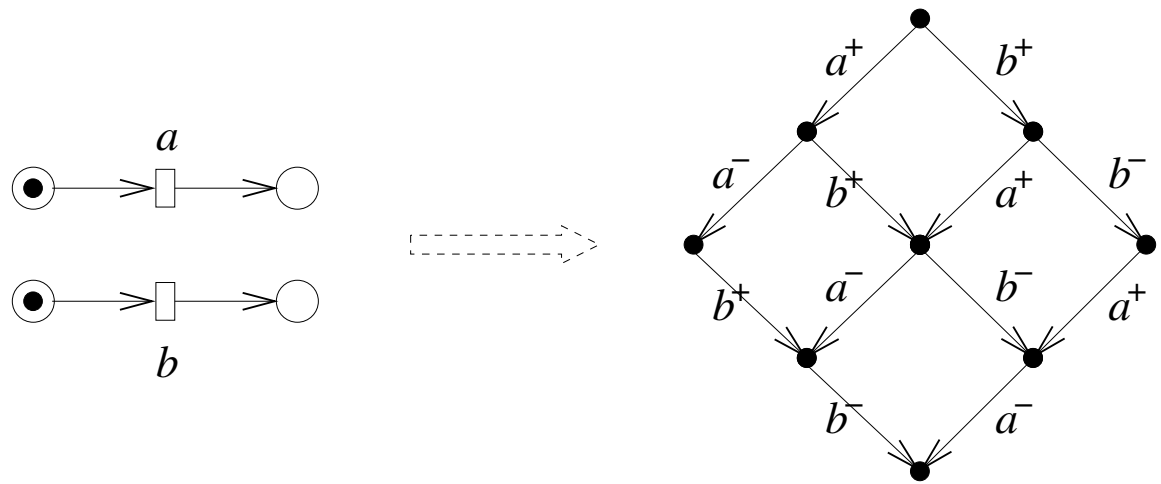
- Examples:
  - Step semantics:



- Examples:
  - Step semantics:



  - ST semantics:



- The Petri net semantics $\mathcal{N}$ (Degano, De Nicola, Montanari 1988) associates a place with any state of a *sequential subprocess* of $P \in \mathbb{P}$.
- The syntax of places $\mathbb{V}$ is the same as the one of process terms, but binary parallel composition is replaced by unary $\_ \parallel_S \mathrm{id}$ and $\mathrm{id} \parallel_S \_$.

- The Petri net semantics $\mathcal{N}$ (Degano, De Nicola, Montanari 1988) associates a place with any state of a *sequential subprocess* of $P \in \mathbb{P}$.
- The syntax of places $\mathbb{V}$ is the same as the one of process terms, but binary parallel composition is replaced by unary $\_ \parallel_S \mathrm{id}$ and $\mathrm{id} \parallel_S \_$.
- Decomposition $\mathit{dec} : \mathbb{P} \to \mathcal{M}u_{\mathrm{fin}}(\mathbb{V})$ of process terms into places:

$$
\begin{aligned}
\mathit{dec}(\underline{0}) &= \{\!|\, \underline{0} \,|\!\} \\
\mathit{dec}(a \,.\, P) &= \{\!|\, a \,.\, P \,|\!\} \\
\mathit{dec}(P_1 + P_2) &= \{\!|\, V_1 + V_2 \mid V_1 \in \mathit{dec}(P_1), V_2 \in \mathit{dec}(P_2) \,|\!\} \\
\mathit{dec}(P_1 \parallel_S P_2) &= \{\!|\, V \parallel_S \mathrm{id} \mid V \in \mathit{dec}(P_1) \,|\!\} \cup \{\!|\, \mathrm{id} \parallel_S V \mid V \in \mathit{dec}(P_2) \,|\!\} \\
\mathit{dec}(P \,/\, H) &= \{\!|\, V \,/\, H \mid V \in \mathit{dec}(P) \,|\!\} \\
\mathit{dec}(P \setminus L) &= \{\!|\, V \setminus L \mid V \in \mathit{dec}(P) \,|\!\} \\
\mathit{dec}(P\,[\varphi]) &= \{\!|\, V\,[\varphi] \mid V \in \mathit{dec}(P) \,|\!\} \\
\mathit{dec}(B) &= \mathit{dec}(P) \quad \text{if } B \stackrel{\Delta}{=} P
\end{aligned}
$$

- The Petri net semantics $\mathcal{N}$ (Degano, De Nicola, Montanari 1988) associates a place with any state of a *sequential subprocess* of $P \in \mathbb{P}$.
- The syntax of places $\mathbb{V}$ is the same as the one of process terms, but binary parallel composition is replaced by unary $\_ \parallel_S \mathrm{id}$ and $\mathrm{id} \parallel_S \_$.
- Decomposition $\mathit{dec} : \mathbb{P} \to \mathcal{M}u_{\mathrm{fin}}(\mathbb{V})$ of process terms into places:

$$\begin{aligned}
\mathit{dec}(\underline{0}) &= \{\!|\, \underline{0} \,|\!\} \\
\mathit{dec}(a \,.\, P) &= \{\!|\, a \,.\, P \,|\!\} \\
\mathit{dec}(P_1 + P_2) &= \{\!|\, V_1 + V_2 \mid V_1 \in \mathit{dec}(P_1), V_2 \in \mathit{dec}(P_2) \,|\!\} \\
\mathit{dec}(P_1 \parallel_S P_2) &= \{\!|\, V \parallel_S \mathrm{id} \mid V \in \mathit{dec}(P_1) \,|\!\} \cup \{\!|\, \mathrm{id} \parallel_S V \mid V \in \mathit{dec}(P_2) \,|\!\} \\
\mathit{dec}(P \,/\, H) &= \{\!|\, V \,/\, H \mid V \in \mathit{dec}(P) \,|\!\} \\
\mathit{dec}(P \setminus L) &= \{\!|\, V \setminus L \mid V \in \mathit{dec}(P) \,|\!\} \\
\mathit{dec}(P\,[\varphi]) &= \{\!|\, V\,[\varphi] \mid V \in \mathit{dec}(P) \,|\!\} \\
\mathit{dec}(B) &= \mathit{dec}(P) \quad \text{if } B \stackrel{\Delta}{=} P
\end{aligned}$$

- Transitions $\mathbb{T}$ stem from operational semantic rules like those for $\mathbb{P}$.
- Retrievability: $\mathcal{RG}[\![\mathcal{N}[\![P]\!]]\!]$ with initial marking $\mathit{dec}(P)$ is isomorphic to $[\![P]\!]$.

$$\{\!|\, a \,.\, P \,|\!\} \xrightarrow{a} \mathit{dec}(P)$$

$$\frac{\mathcal{V}_1 \cup \mathcal{V}_2 \xrightarrow{a} \mathcal{V}' \quad \mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset \quad \mathcal{V} \text{ full}}{\mathcal{V}_1 \cup (\mathcal{V}_2 + \mathcal{V}) \xrightarrow{a} \mathcal{V}'} \qquad \frac{\mathcal{V}_1 \cup \mathcal{V}_2 \xrightarrow{a} \mathcal{V}' \quad \mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset \quad \mathcal{V} \text{ full}}{\mathcal{V}_1 \cup (\mathcal{V} + \mathcal{V}_2) \xrightarrow{a} \mathcal{V}'}$$

$$\frac{\mathcal{V} \xrightarrow{a} \mathcal{V}' \quad a \notin S}{\mathcal{V} \parallel_S \mathrm{id} \xrightarrow{a} \mathcal{V}' \parallel_S \mathrm{id}} \qquad \frac{\mathcal{V} \xrightarrow{a} \mathcal{V}' \quad a \notin S}{\mathrm{id} \parallel_S \mathcal{V} \xrightarrow{a} \mathrm{id} \parallel_S \mathcal{V}'}$$

$$\frac{\mathcal{V}_1 \xrightarrow{a} \mathcal{V}_1' \quad \mathcal{V}_2 \xrightarrow{a} \mathcal{V}_2' \quad a \in S}{\mathcal{V}_1 \parallel_S \mathrm{id} \cup \mathrm{id} \parallel_S \mathcal{V}_2 \xrightarrow{a} \mathcal{V}_1' \parallel_S \mathrm{id} \cup \mathrm{id} \parallel_S \mathcal{V}_2'}$$

$$\frac{\mathcal{V} \xrightarrow{a} \mathcal{V}' \quad a \in H}{\mathcal{V} \,/\, H \xrightarrow{\tau} \mathcal{V}' \,/\, H} \qquad \frac{\mathcal{V} \xrightarrow{a} \mathcal{V}' \quad a \notin H}{\mathcal{V} \,/\, H \xrightarrow{a} \mathcal{V}' \,/\, H}$$

$$\frac{\mathcal{V} \xrightarrow{a} \mathcal{V}' \quad a \notin L}{\mathcal{V} \setminus L \xrightarrow{a} \mathcal{V}' \setminus L}$$

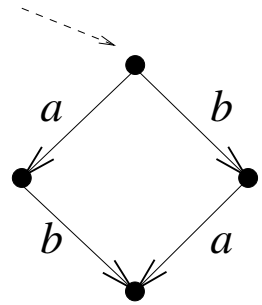
$$\frac{\mathcal{V} \xrightarrow{a} \mathcal{V}'}{\mathcal{V}\,[\varphi] \xrightarrow{\varphi(a)} \mathcal{V}'\,[\varphi]}$$

"$\mathcal{V}$ full" means $\mathcal{V} = \mathit{dec}(P)$ for some $P \in \mathbb{P}$.

- Consider again the two process terms:

$$a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

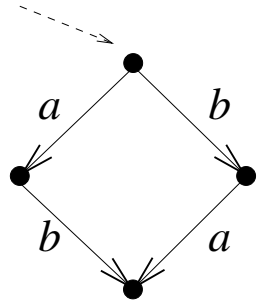
- They are indistinguishable according to their interleaving semantics:



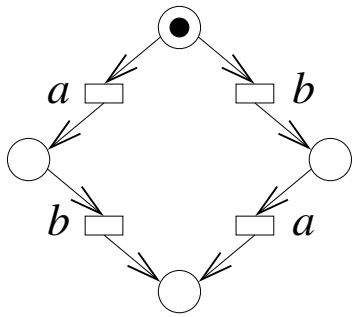
- Consider again the two process terms:

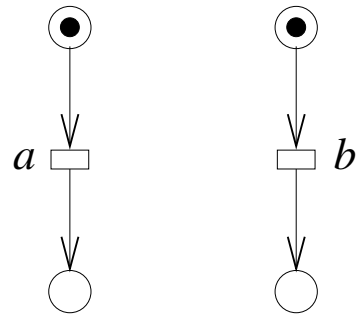
$$a\,.\,b\,.\,\underline{0} \;+\; b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

- They are indistinguishable according to their interleaving semantics:



- Their Petri net semantics show that they are structurally different:





- The two underlying reachability graphs are isomorphic to the LTS.

6.9 Truly Concurrent Semantics via Event Structures

- Winskel event structures (1980) are models in which to represent *causality*, *conflict*, *concurrency* more abstractly than in Petri nets.
- Relationships with domain theory and category theory.

6.9 Truly Concurrent Semantics via Event Structures

- Winskel event structures (1980) are models in which to represent *causality*, *conflict*, *concurrency* more abstractly than in Petri nets.
- Relationships with domain theory and category theory.
- The basic entities of these models are called events.
- Events describe occurrences of actions and are labeled with them.
- States are called configurations.

6.9 Truly Concurrent Semantics via Event Structures

- Winskel event structures (1980) are models in which to represent *causality*, *conflict*, *concurrency* more abstractly than in Petri nets.
- Relationships with domain theory and category theory.
- The basic entities of these models are called events.
- Events describe occurrences of actions and are labeled with them.
- States are called configurations.
- Configurations are defined as conflict-free sets of events containing all the events causing the ones in the sets (only causality and concurrency).
- Configurations specify the occurrences of events in system runs and determine the remaining behavior in terms of events not yet occurred and not excluded because of conflicts with already occurred events.

- A labeled prime event structure is a tuple $\mathcal{E} = (E, <, \#, \ell)$ where:
  - E is a set of events.
  - $< \subseteq E \times E$ is the causality relation, which satisfies:
    - Irreflexivity, antisymmetry, transitivity.
    - Principle of finite causes: $\{d \in E \mid d < e\}$ is finite for all $e \in E$.
  - $\# \subseteq E \times E$ is the conflict relation, which satisfies:
    - Irreflexivity and symmetry.
    - $\# \cap < = \emptyset$.
    - Principle of conflict heredity: $d < e \wedge d \# f \implies e \# f$ for all d, e, f.
  - $\ell : E \to \mathcal{A}$ is the labeling function.

- A labeled prime event structure is a tuple $\mathcal{E} = (E, <, \#, \ell)$ where:
  - E is a set of events.
  - $< \subseteq E \times E$ is the causality relation, which satisfies:
    - Irreflexivity, antisymmetry, transitivity.
    - Principle of finite causes: $\{d \in E \mid d < e\}$ is finite for all $e \in E$.
  - $\# \subseteq E \times E$ is the conflict relation, which satisfies:
    - Irreflexivity and symmetry.
    - $\# \cap < = \emptyset$.
    - Principle of conflict heredity: $d < e \wedge d \# f \implies e \# f$ for all d, e, f.
  - $\ell : E \to \mathcal{A}$ is the labeling function.
- The concurrency relation is derived as $co = (E \times E) \setminus (\leq \cup \geq \cup \#)$.

- A labeled prime event structure is a tuple $\mathcal{E} = (E, <, \#, \ell)$ where:
  - E is a set of events.
  - $< \subseteq E \times E$ is the causality relation, which satisfies:
    - Irreflexivity, antisymmetry, transitivity.
    - Principle of finite causes: $\{d \in E \mid d < e\}$ is finite for all $e \in E$.
  - $\# \subseteq E \times E$ is the conflict relation, which satisfies:
    - Irreflexivity and symmetry.
    - $\# \cap < = \emptyset$.
    - Principle of conflict heredity: $d < e \wedge d \# f \implies e \# f$ for all d, e, f.
  - $\ell : E \to \mathcal{A}$ is the labeling function.
- The concurrency relation is derived as $co = (E \times E) \setminus (\leq \cup \geq \cup \#)$.
- $X \subseteq E$ is a configuration of $\mathcal{E}$ iff it is:
  - Finite.
  - Left closed: $e \in X \wedge d < e \implies d \in X$ for all $d, e \in E$.
  - Conflict free: $\# \cap (X \times X) = \emptyset$.

- The event structure semantics $\mathcal{E}$ is defined *denotationally*:
  - $\mathcal{E}[\![\underline{0}]\!] = (\emptyset, \emptyset, \emptyset, \emptyset)$.
  - $\mathcal{E}[\![a\,.\,P]\!] = (E \cup \{e_a\}, < \cup \{(e_a, e) \mid e \in E\}, \#, \ell \cup \{(e_a, a)\})$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$ and $e_a \notin E$.
  - $\mathcal{E}[\![P_1 + P_2]\!] = (E_1 \cup E_2, <_1 \cup <_2, \#_1 \cup \#_2 \cup (E_i \times E_j), \ell_1 \cup \ell_2)$
    where $\mathcal{E}[\![P_k]\!] = (E_k, <_k, \#_k, \ell_k)$ for $k \in \{1, 2\}$ and $E_1 \cap E_2 = \emptyset$.
  - $\mathcal{E}[\![P \,/\, H]\!] = (E, <, \#, \ell \setminus \{(e, \ell(e)) \mid \ell(e) \in H\} \cup \{(e, \tau) \mid \ell(e) \in H\})$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$.
  - $\mathcal{E}[\![P \setminus L]\!] = (E', < \cap (E' \times E'), \# \cap (E' \times E'), \ell \cap (E' \times E'))$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$
    and $E' = \{e \in E \mid \ell(e) \notin L \wedge \nexists d \in E.\, d < e \wedge \ell(d) \in L\}$.
  - $\mathcal{E}[\![P\,[\varphi]]\!] = (E, <, \#, \{(e, \varphi(a)) \mid (e, a) \in \ell\})$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$.

- The event structure semantics $\mathcal{E}$ is defined *denotationally*:
  - $\mathcal{E}[\![\underline{0}]\!] = (\emptyset, \emptyset, \emptyset, \emptyset)$.
  - $\mathcal{E}[\![a\,.\,P]\!] = (E \cup \{e_a\}, < \cup \{(e_a, e) \mid e \in E\}, \#, \ell \cup \{(e_a, a)\})$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$ and $e_a \notin E$.
  - $\mathcal{E}[\![P_1 + P_2]\!] = (E_1 \cup E_2, <_1 \cup <_2, \#_1 \cup \#_2 \cup (E_i \times E_j), \ell_1 \cup \ell_2)$
    where $\mathcal{E}[\![P_k]\!] = (E_k, <_k, \#_k, \ell_k)$ for $k \in \{1, 2\}$ and $E_1 \cap E_2 = \emptyset$.
  - $\mathcal{E}[\![P \,/\, H]\!] = (E, <, \#, \ell \setminus \{(e, \ell(e)) \mid \ell(e) \in H\} \cup \{(e, \tau) \mid \ell(e) \in H\})$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$.
  - $\mathcal{E}[\![P \setminus L]\!] = (E', < \cap (E' \times E'), \# \cap (E' \times E'), \ell \cap (E' \times E'))$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$
    and $E' = \{e \in E \mid \ell(e) \notin L \wedge \nexists d \in E.\, d < e \wedge \ell(d) \in L\}$.
  - $\mathcal{E}[\![P\,[\varphi]]\!] = (E, <, \#, \{(e, \varphi(a)) \mid (e, a) \in \ell\})$
    where $\mathcal{E}[\![P]\!] = (E, <, \#, \ell)$.
- Parallel composition over prime event structures may not simply be the cartesian product of the events that have to synchronize and the union of the other events (unless some events are duplicated).

- More liberal family of event structures (Boudol & Castellani 1988) in which only direct causes are formalized and cycles are expressible.
- A labeled flow event structure is a tuple $\mathcal{E} = (E, \prec, \#, \ell)$ where:
  - E is a set of events.
  - $\prec \subseteq E \times E$ is the flow relation, which is irreflexive.
  - $\# \subseteq E \times E$ is the conflict relation, which is symmetric.
  - $\ell : E \to \mathcal{A}$ is the labeling function.

- More liberal family of event structures (Boudol & Castellani 1988) in which only direct causes are formalized and cycles are expressible.
- A labeled flow event structure is a tuple $\mathcal{E} = (E, \prec, \#, \ell)$ where:
  - E is a set of events.
  - $\prec \subseteq E \times E$ is the flow relation, which is irreflexive.
  - $\# \subseteq E \times E$ is the conflict relation, which is symmetric.
  - $\ell : E \to \mathcal{A}$ is the labeling function.
- The concurrency relation is $co = (E \times E) \setminus (\preceq^+ \cup \succeq^+ \cup \#)$ where $^+$ is the transitive closure operator over relations.

- More liberal family of event structures (Boudol & Castellani 1988) in which only direct causes are formalized and cycles are expressible.
- A labeled flow event structure is a tuple $\mathcal{E} = (E, \prec, \#, \ell)$ where:
  - E is a set of events.
  - $\prec \subseteq E \times E$ is the flow relation, which is irreflexive.
  - $\# \subseteq E \times E$ is the conflict relation, which is symmetric.
  - $\ell : E \to \mathcal{A}$ is the labeling function.
- The concurrency relation is $co = (E \times E) \setminus (\preceq^+ \cup \succeq^+ \cup \#)$ where $^+$ is the transitive closure operator over relations.
- Some constraints are recovered in the notion of configuration.
- $X \subseteq E$ is a configuration of $\mathcal{E}$ iff it is:
  - Finite.
  - Left closed up to conflicts: for all $d, e \in E$ $e \in X \wedge d \prec e \wedge d \notin X \implies \exists f \in X.\, f \prec e \wedge d \# f$.
  - Causality cycle free: $(\prec \cap (X \times X))^+$ is irreflexive.
  - Conflict free: $\# \cap (X \times X) = \emptyset$.

- $\mathcal{E}[\![P_1 \parallel_S P_2]\!] = (E, \prec, \#, \ell)$ where:
  - Let $\mathcal{E}[\![P_k]\!] = (E_k, \prec_k, \#_k, \ell_k)$ for $k \in \{1, 2\}$.
  - $E = \{(e, *) \mid e \in E_1 \wedge \ell_1(e) \notin S\} \cup$
    $\{(*, e) \mid e \in E_2 \wedge \ell_2(e) \notin S\} \cup$
    $\{(e_1, e_2) \in E_1 \times E_2 \mid \ell_1(e_1) = \ell_2(e_2) \in S\}$.
  - $e \prec_1 d \implies (e, e') \prec (d, d')$ where $e', d' \in E_2 \cup \{*\}$,
    $e \prec_2 d \implies (e', e) \prec (d', d)$ where $e', d' \in E_1 \cup \{*\}$.
  - $e \#_1 d \implies (e, e') \# (d, d')$ where $e', d' \in E_2 \cup \{*\}$,
    $e \#_2 d \implies (e', e) \# (d', d)$ where $e', d' \in E_1 \cup \{*\}$.
  - $\ell = \{((e, *), \ell_1(e)) \mid e \in E_1 \wedge \ell_1(e) \notin S\} \cup$
    $\{((*, e), \ell_2(e)) \mid e \in E_2 \wedge \ell_2(e) \notin S\} \cup$
    $\{((e_1, e_2), \ell_1(e_1)) \mid (e_1, e_2) \in E_1 \times E_2 \wedge \ell_1(e_1) = \ell_2(e_2) \in S\}$.
  - Every configuration of $\mathcal{E}[\![P_1 \parallel_S P_2]\!]$ must be such that its projection on E_k is a configuration of $\mathcal{E}[\![P_k]\!]$ for $k \in \{1, 2\}$.

- $\mathcal{E}[\![P_1 \parallel_S P_2]\!] = (E, \prec, \#, \ell)$ where:
  - Let $\mathcal{E}[\![P_k]\!] = (E_k, \prec_k, \#_k, \ell_k)$ for $k \in \{1, 2\}$.
  - $E = \{(e, *) \mid e \in E_1 \wedge \ell_1(e) \notin S\} \cup$
    $\{(*, e) \mid e \in E_2 \wedge \ell_2(e) \notin S\} \cup$
    $\{(e_1, e_2) \in E_1 \times E_2 \mid \ell_1(e_1) = \ell_2(e_2) \in S\}$.
  - $e \prec_1 d \implies (e, e') \prec (d, d')$ where $e', d' \in E_2 \cup \{*\}$,
    $e \prec_2 d \implies (e', e) \prec (d', d)$ where $e', d' \in E_1 \cup \{*\}$.
  - $e \#_1 d \implies (e, e') \# (d, d')$ where $e', d' \in E_2 \cup \{*\}$,
    $e \#_2 d \implies (e', e) \# (d', d)$ where $e', d' \in E_1 \cup \{*\}$.
  - $\ell = \{((e, *), \ell_1(e)) \mid e \in E_1 \wedge \ell_1(e) \notin S\} \cup$
    $\{((*, e), \ell_2(e)) \mid e \in E_2 \wedge \ell_2(e) \notin S\} \cup$
    $\{((e_1, e_2), \ell_1(e_1)) \mid (e_1, e_2) \in E_1 \times E_2 \wedge \ell_1(e_1) = \ell_2(e_2) \in S\}$.
  - Every configuration of $\mathcal{E}[\![P_1 \parallel_S P_2]\!]$ must be such that its projection on E_k is a configuration of $\mathcal{E}[\![P_k]\!]$ for $k \in \{1, 2\}$.
- A notion of transition between configurations is missing.

- The relevant behavior of an event structure is determined by its set of configurations and their transitions (Van Glabbeek & Goltz 2001).
- A labeled configuration structure is a tuple $C = (E, \mathcal{C}, \ell)$ where:
  - E is a set of events.
  - $\mathcal{C} \subseteq \mathcal{P}_{\text{fin}}(E)$ is a set of configurations.
  - $\ell : \bigcup_{X \in \mathcal{C}} X \to \mathcal{A}$ is the labeling function.

- The relevant behavior of an event structure is determined by its set of configurations and their transitions (Van Glabbeek & Goltz 2001).
- A labeled configuration structure is a tuple $C = (E, \mathcal{C}, \ell)$ where:
  - E is a set of events.
  - $\mathcal{C} \subseteq \mathcal{P}_{\text{fin}}(E)$ is a set of configurations.
  - $\ell : \bigcup_{X \in \mathcal{C}} X \to \mathcal{A}$ is the labeling function.
- A configuration structure C is stable iff it is:
  - Rooted: $\emptyset \in \mathcal{C}$.
  - Connected: $\forall X \in \mathcal{C} \setminus \{\emptyset\}. \exists e \in X. X \setminus \{e\} \in \mathcal{C}$.
  - Closed under bounded unions and intersections: $\forall X, Y, Z \in \mathcal{C}. X \cup Y \subseteq Z \implies X \cup Y \in \mathcal{C} \wedge X \cap Y \in \mathcal{C}$.

- The relevant behavior of an event structure is determined by its set of configurations and their transitions (Van Glabbeek & Goltz 2001).
- A labeled configuration structure is a tuple $C = (E, \mathcal{C}, \ell)$ where:
  - E is a set of events.
  - $\mathcal{C} \subseteq \mathcal{P}_{\text{fin}}(E)$ is a set of configurations.
  - $\ell : \bigcup_{X \in \mathcal{C}} X \to \mathcal{A}$ is the labeling function.
- A configuration structure C is stable iff it is:
  - Rooted: $\emptyset \in \mathcal{C}$.
  - Connected: $\forall X \in \mathcal{C} \setminus \{\emptyset\}. \exists e \in X. X \setminus \{e\} \in \mathcal{C}$.
  - Closed under bounded unions and intersections: $\forall X, Y, Z \in \mathcal{C}. X \cup Y \subseteq Z \implies X \cup Y \in \mathcal{C} \wedge X \cap Y \in \mathcal{C}$.
- The causality relation over $X \in \mathcal{C}$ is defined by letting $e_1 <_X e_2$ for $e_1, e_2 \in X$ s.t. $e_1 \neq e_2$ iff $\forall Y \in \mathcal{C}. Y \subseteq X \wedge e_2 \in Y \implies e_1 \in Y$.
- The concurrency relation over X is $co_X = (X \times X) \setminus (\leq_X \cup \geq_X)$.

- The relevant behavior of an event structure is determined by its set of configurations and their transitions (Van Glabbeek & Goltz 2001).
- A labeled configuration structure is a tuple $C = (E, \mathcal{C}, \ell)$ where:
  - E is a set of events.
  - $\mathcal{C} \subseteq \mathcal{P}_{\text{fin}}(E)$ is a set of configurations.
  - $\ell : \bigcup_{X \in \mathcal{C}} X \to \mathcal{A}$ is the labeling function.
- A configuration structure C is stable iff it is:
  - Rooted: $\emptyset \in \mathcal{C}$.
  - Connected: $\forall X \in \mathcal{C} \setminus \{\emptyset\}. \exists e \in X. X \setminus \{e\} \in \mathcal{C}$.
  - Closed under bounded unions and intersections: $\forall X, Y, Z \in \mathcal{C}. X \cup Y \subseteq Z \implies X \cup Y \in \mathcal{C} \wedge X \cap Y \in \mathcal{C}$.
- The causality relation over $X \in \mathcal{C}$ is defined by letting $e_1 <_X e_2$ for $e_1, e_2 \in X$ s.t. $e_1 \neq e_2$ iff $\forall Y \in \mathcal{C}. Y \subseteq X \wedge e_2 \in Y \implies e_1 \in Y$.
- The concurrency relation over X is $co_X = (X \times X) \setminus (\leq_X \cup \geq_X)$.
- $X \xrightarrow{a} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = \{e\} \wedge \ell(e) = a$.

- Consider once more the two process terms:

$$a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

- Consider once more the two process terms:

$$a\,.\,b\,.\,\underline{0} \;+\; b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

- $\mathcal{E}[\![a\,.\,b\,.\,\underline{0} \;+\; b\,.\,a\,.\,\underline{0}]\!] = (E_1, \prec_1, \#_1, \ell_1)$ where:
  - $E_1 = \{e_{a,\mathrm{l}}, e_{b,\mathrm{l}}, e_{b,\mathrm{r}}, e_{a,\mathrm{r}}\}$.
  - $\ell_1 = \{(e_{a,\mathrm{l}}, a), (e_{b,\mathrm{l}}, b), (e_{b,\mathrm{r}}, b), (e_{a,\mathrm{r}}, a)\}$.
  - $\prec_1 = \{(e_{a,\mathrm{l}}, e_{b,\mathrm{l}}), (e_{b,\mathrm{r}}, e_{a,\mathrm{r}})\}$.
  - $\#_1 = \{(e_{a,\mathrm{l}}, e_{b,\mathrm{r}}), (e_{a,\mathrm{l}}, e_{a,\mathrm{r}}), (e_{b,\mathrm{l}}, e_{b,\mathrm{r}}), (e_{b,\mathrm{l}}, e_{a,\mathrm{r}})\}$ up to symmetry.
  - $co_1 = \emptyset$.
  - $\mathcal{C}_1 = \{\emptyset, \{e_{a,\mathrm{l}}\}, \{e_{a,\mathrm{l}}, e_{b,\mathrm{l}}\}, \{e_{b,\mathrm{r}}\}, \{e_{b,\mathrm{r}}, e_{a,\mathrm{r}}\}\}$.

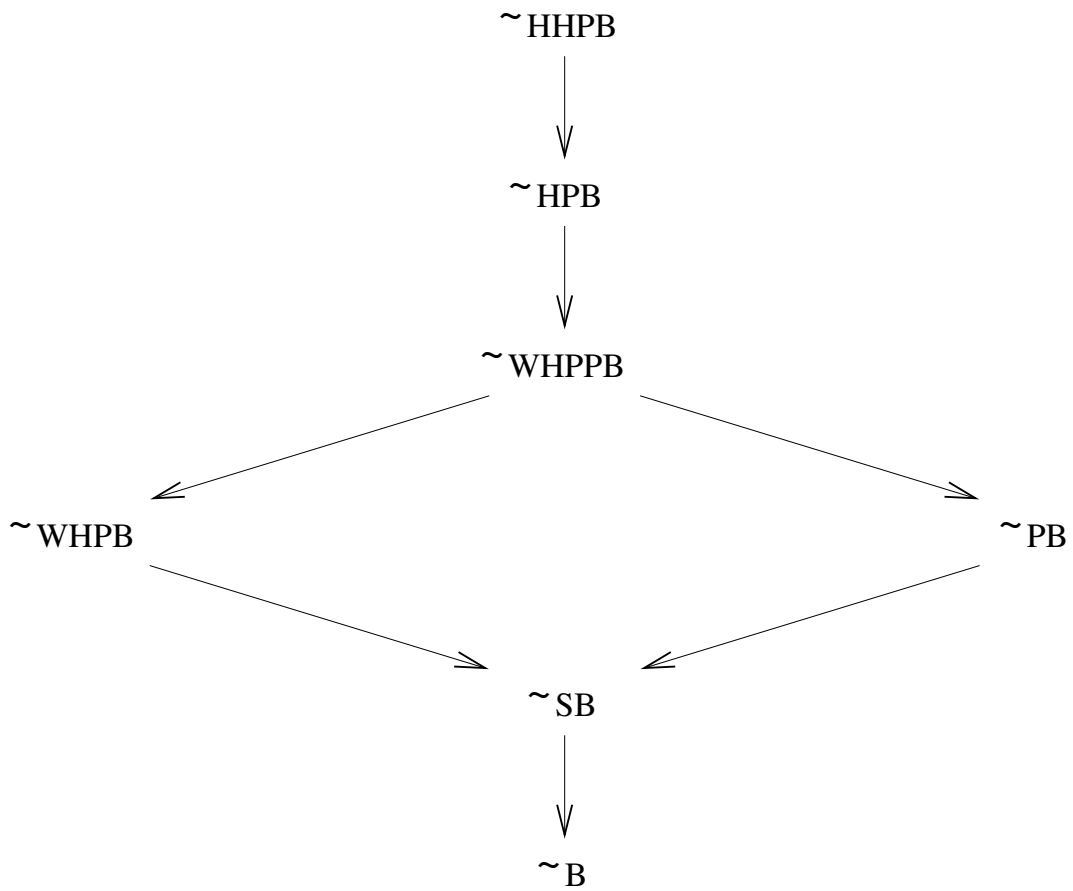
- Consider once more the two process terms:

$$a\,.\,b\,.\,\underline{0} \;+\; b\,.\,a\,.\,\underline{0}$$
$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

- $\mathcal{E}[\![a\,.\,b\,.\,\underline{0} \;+\; b\,.\,a\,.\,\underline{0}]\!] = (E_1, \prec_1, \#_1, \ell_1)$ where:
  - $E_1 = \{e_{a,\mathrm{l}}, e_{b,\mathrm{l}}, e_{b,\mathrm{r}}, e_{a,\mathrm{r}}\}$.
  - $\ell_1 = \{(e_{a,\mathrm{l}}, a), (e_{b,\mathrm{l}}, b), (e_{b,\mathrm{r}}, b), (e_{a,\mathrm{r}}, a)\}$.
  - $\prec_1 = \{(e_{a,\mathrm{l}}, e_{b,\mathrm{l}}), (e_{b,\mathrm{r}}, e_{a,\mathrm{r}})\}$.
  - $\#_1 = \{(e_{a,\mathrm{l}}, e_{b,\mathrm{r}}), (e_{a,\mathrm{l}}, e_{a,\mathrm{r}}), (e_{b,\mathrm{l}}, e_{b,\mathrm{r}}), (e_{b,\mathrm{l}}, e_{a,\mathrm{r}})\}$ up to symmetry.
  - $co_1 = \emptyset$.
  - $\mathcal{C}_1 = \{\emptyset, \{e_{a,\mathrm{l}}\}, \{e_{a,\mathrm{l}}, e_{b,\mathrm{l}}\}, \{e_{b,\mathrm{r}}\}, \{e_{b,\mathrm{r}}, e_{a,\mathrm{r}}\}\}$.
- $\mathcal{E}[\![a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}]\!] = (E_2, \prec_2, \#_2, \ell_2)$ where:
  - $E_2 = \{e_a, e_b\}$.
  - $\ell_2 = \{(e_a, a), (e_b, b)\}$.
  - $\prec_2 = \emptyset$.
  - $\#_2 = \emptyset$.
  - $co_2 = \{(e_a, e_b)\}$.
  - $\mathcal{C}_2 = \{\emptyset, \{e_a\}, \{e_b\}, \{e_a, e_b\}\}$.
- The two configuration structures are not isomorphic.

6.10 Truly Concurrent Bisimilarities

- Truly concurrent bisimilarity spectrum for finitely-branching processes with no τ-actions (Van Glabbeek & Goltz 2001; Fecher 2004):



- All defined over stable configuration structures.

- Step semantics allows multiple concurrent events, possibly labeled with the same action, to take place in one step (Pomello 1985).
- $X \xrightarrow{A} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = G \wedge \ell(G) = A \wedge A \in \mathbb{N}^{\mathcal{A}} \wedge \forall e, d \in G.\, e \; co_{X'} \; d$.

- Step semantics allows multiple concurrent events, possibly labeled with the same action, to take place in one step (Pomello 1985).
- $X \xrightarrow{A} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = G \wedge \ell(G) = A \wedge A \in \mathbb{N}^{\mathcal{A}} \wedge \forall e, d \in G.\, e \; co_{X'} \; d$.
- Two stable configuration structures $C_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are step bisimilar, written $C_1 \sim_{\mathrm{SB}} C_2$,
  iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then for all action multisets $A \in \mathbb{N}^{\mathcal{A}}$:
    - For each $X_1 \xrightarrow{A}_{C_1} X_1'$ there exists $X_2 \xrightarrow{A}_{C_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.

- Step semantics allows multiple concurrent events, possibly labeled with the same action, to take place in one step (Pomello 1985).
- $X \xrightarrow{A} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = G \wedge \ell(G) = A \wedge A \in \mathbb{N}^{\mathcal{A}} \wedge \forall e, d \in G.\, e\ co_{X'}\ d$.
- Two stable configuration structures $C_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are step bisimilar, written $C_1 \sim_{\mathrm{SB}} C_2$,
  iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then for all action multisets $A \in \mathbb{N}^{\mathcal{A}}$:
    - For each $X_1 \xrightarrow{A}_{C_1} X_1'$ there exists $X_2 \xrightarrow{A}_{C_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.
- $\sim_{\mathrm{SB}}$ reduces to $\sim_{\mathrm{B}}$ when considering only individual actions.
- $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$ and $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$ are $\sim_{\mathrm{B}}$-equivalent but $\sim_{\mathrm{SB}}$-inequiv (likewise $a\,.\,\underline{0} \parallel_{\emptyset} a\,.\,\underline{0}$ and $a\,.\,a\,.\,\underline{0}$ are $\sim_{\mathrm{B}}$-equivalent but $\sim_{\mathrm{SB}}$-inequiv).

- Generalization to partially ordered multisets of events that are concurrent or causally related (Boudol & Castellani 1988).
- $X \xrightarrow{U} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = G \wedge \ell(G) = U \wedge U = [(G, <_{X'} \cap (G \times G), \ell \cap (G \times \mathcal{A})]_{\mathrm{iso}}$.

- Generalization to partially ordered multisets of events that are concurrent or causally related (Boudol & Castellani 1988).
- $X \xrightarrow{U} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = G \wedge \ell(G) = U \wedge U = [(G, <_{X'} \cap (G \times G), \ell \cap (G \times \mathcal{A})]_{\mathrm{iso}}$.
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are pomset bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{PB}} \mathcal{C}_2$,
  iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then for all pomsets U over $\mathcal{A}$:
    - For each $X_1 \xrightarrow{U}_{\mathcal{C}_1} X_1'$ there exists $X_2 \xrightarrow{U}_{\mathcal{C}_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.

- Generalization to partially ordered multisets of events that are concurrent or causally related (Boudol & Castellani 1988).
- $X \xrightarrow{U} X'$ for $X, X' \in \mathcal{C}$ iff $X \subseteq X' \wedge X' \setminus X = G \wedge \ell(G) = U \wedge U = [(G, <_{X'} \cap (G \times G), \ell \cap (G \times \mathcal{A})]_{\mathrm{iso}}$.
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are pomset bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{PB}} \mathcal{C}_2$,
  iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then for all pomsets U over $\mathcal{A}$:
    - For each $X_1 \xrightarrow{U}_{\mathcal{C}_1} X_1'$ there exists $X_2 \xrightarrow{U}_{\mathcal{C}_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.
- $a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}$ and $(a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}) + a \,.\, b \,.\, \underline{0}$ are identified by $\sim_{\mathrm{SB}}$ but told apart by $\sim_{\mathrm{PB}}$.

- Variant going back to actions that imposes configuration isomorphism (Degano, De Nicola, Montanari 1986).

- Variant going back to actions that imposes configuration isomorphism (Degano, De Nicola, Montanari 1986).
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are weak history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{WHPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then there exists a bijection from X_1 to X_2 that preserves labeling and causality and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exists $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.

- Variant going back to actions that imposes configuration isomorphism (Degano, De Nicola, Montanari 1986).
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are weak history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{WHPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then there exists a bijection from X_1 to X_2 that preserves labeling and causality and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exists $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.
- $a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}$ and $(a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}) + a \,.\, b \,.\, \underline{0}$ are identified by $\sim_{\mathrm{SB}}$ but told apart by $\sim_{\mathrm{WHPB}}$.

- Variant going back to actions that imposes configuration isomorphism (Degano, De Nicola, Montanari 1986).
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are weak history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{WHPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then there exists a bijection from X_1 to X_2 that preserves labeling and causality and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exists $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.
- $a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}$ and $(a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}) + a \,.\, b \,.\, \underline{0}$ are identified by $\sim_{\mathrm{SB}}$ but told apart by $\sim_{\mathrm{WHPB}}$.
- $a.(b.\underline{0} + c.\underline{0}) + (a.\underline{0} \parallel_\emptyset b.\underline{0}) + a.b.\underline{0}$ and $a.(b.\underline{0} + c.\underline{0}) + (a.\underline{0} \parallel_\emptyset b.\underline{0})$ are identified by $\sim_{\mathrm{PB}}$ but told apart by $\sim_{\mathrm{WHPB}}$.
- It is finer than $\sim_{\mathrm{PB}}$ in the absence of autoconcurrency, i.e., when no occurrences of the same action are concurrent to each other.

- Variant combining pomsets and configuration isomorphism (Van Glabbeek, Goltz 2001).

- Variant combining pomsets and configuration isomorphism (Van Glabbeek, Goltz 2001).
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are weak history-preserving pomset bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{WHPPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then there exists a bijection from X_1 to X_2 that preserves labeling and causality and for all pomsets U over $\mathcal{A}$:
    - For each $X_1 \xrightarrow{U}_{\mathcal{C}_1} X_1'$ there exists $X_2 \xrightarrow{U}_{\mathcal{C}_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.

- Variant combining pomsets and configuration isomorphism (Van Glabbeek, Goltz 2001).
- Two stable configuration structures $C_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are weak history-preserving pomset bisimilar, written $C_1 \sim_{\mathrm{WHPPB}} C_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ such that:
  - $(\emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2) \in \mathcal{B}$, then there exists a bijection from X_1 to X_2 that preserves labeling and causality and for all pomsets U over $\mathcal{A}$:
    - For each $X_1 \xrightarrow{U}_{C_1} X_1'$ there exists $X_2 \xrightarrow{U}_{C_2} X_2'$ such that $(X_1', X_2') \in \mathcal{B}$, and vice versa.
- $\sim_{\mathrm{WHPPB}}$ is finer than both $\sim_{\mathrm{PB}}$ and $\sim_{\mathrm{WHPB}}$.

- Finer variant imposing that configuration isomorphism is incremental (Rabinovich & Trakhtenbrot 1988).
- Its distinguishing power does not change if instead of actions we consider action multisets or pomsets as transition labels.

- Finer variant imposing that configuration isomorphism is incremental (Rabinovich & Trakhtenbrot 1988).
- Its distinguishing power does not change if instead of actions we consider action multisets or pomsets as transition labels.
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{HPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{P}(E_1 \times E_2)$ such that:
  - $(\emptyset, \emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2, f) \in \mathcal{B}$, then f is a bijection from X_1 to X_2 that preserves causality and labeling and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exist $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ and f' such that $(X_1', X_2', f') \in \mathcal{B}$ and $f' \cap (X_1 \times X_2) = f$, and vice versa.

- Finer variant imposing that configuration isomorphism is incremental (Rabinovich & Trakhtenbrot 1988).
- Its distinguishing power does not change if instead of actions we consider action multisets or pomsets as transition labels.
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{HPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{P}(E_1 \times E_2)$ such that:
  - $(\emptyset, \emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2, f) \in \mathcal{B}$, then f is a bijection from X_1 to X_2 that preserves causality and labeling and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exist $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ and f' such that $(X_1', X_2', f') \in \mathcal{B}$ and $f' \cap (X_1 \times X_2) = f$, and vice versa.
- $(a \,.\, \underline{0} \parallel_\emptyset (b \,.\, \underline{0} + c \,.\, \underline{0})) + (a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}) + ((a \,.\, \underline{0} + c \,.\, \underline{0}) \parallel_\emptyset b \,.\, \underline{0})$ and $(a \,.\, \underline{0} \parallel_\emptyset (b \,.\, \underline{0} + c \,.\, \underline{0})) + ((a \,.\, \underline{0} + c \,.\, \underline{0}) \parallel_\emptyset b \,.\, \underline{0})$ is a non-trivial example of processes identified by $\sim_{\mathrm{HPB}}$ (absorption law).

- Even finer variant considering not only outgoing transitions but also incoming transitions (Bednarczyk 1991).
- Its distinguishing power does not change if instead of actions we consider action multisets or pomsets as transition labels.

- Even finer variant considering not only outgoing transitions but also incoming transitions (Bednarczyk 1991).
- Its distinguishing power does not change if instead of actions we consider action multisets or pomsets as transition labels.
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are hereditary history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{HHPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{P}(E_1 \times E_2)$ such that:
  - $(\emptyset, \emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2, f) \in \mathcal{B}$, then f is a bijection from X_1 to X_2 that preserves causality and labeling and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exist $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ and f' such that $(X_1', X_2', f') \in \mathcal{B}$ and $f' \cap (X_1 \times X_2) = f$, and vice versa.
    - For each $X_1' \xrightarrow{a}_{\mathcal{C}_1} X_1$ there exist $X_2' \xrightarrow{a}_{\mathcal{C}_2} X_2$ and f' such that $(X_1', X_2', f') \in \mathcal{B}$ and $f \cap (X_1' \times X_2') = f'$, and vice versa.

- Even finer variant considering not only outgoing transitions but also incoming transitions (Bednarczyk 1991).
- Its distinguishing power does not change if instead of actions we consider action multisets or pomsets as transition labels.
- Two stable configuration structures $\mathcal{C}_k = (E_k, \mathcal{C}_k, \ell_k)$ for $k \in \{1, 2\}$ are hereditary history-preserving bisimilar, written $\mathcal{C}_1 \sim_{\mathrm{HHPB}} \mathcal{C}_2$, iff there exists a relation $\mathcal{B} \subseteq \mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{P}(E_1 \times E_2)$ such that:
  - $(\emptyset, \emptyset, \emptyset) \in \mathcal{B}$.
  - Whenever $(X_1, X_2, f) \in \mathcal{B}$, then f is a bijection from X_1 to X_2 that preserves causality and labeling and for all actions $a \in \mathcal{A}$:
    - For each $X_1 \xrightarrow{a}_{\mathcal{C}_1} X_1'$ there exist $X_2 \xrightarrow{a}_{\mathcal{C}_2} X_2'$ and f' such that $(X_1', X_2', f') \in \mathcal{B}$ and $f' \cap (X_1 \times X_2) = f$, and vice versa.
    - For each $X_1' \xrightarrow{a}_{\mathcal{C}_1} X_1$ there exist $X_2' \xrightarrow{a}_{\mathcal{C}_2} X_2$ and f' such that $(X_1', X_2', f') \in \mathcal{B}$ and $f \cap (X_1' \times X_2') = f'$, and vice versa.
- $(a \,.\, \underline{0} \parallel_\emptyset (b \,.\, \underline{0} + c \,.\, \underline{0})) + (a \,.\, \underline{0} \parallel_\emptyset b \,.\, \underline{0}) + ((a \,.\, \underline{0} + c \,.\, \underline{0}) \parallel_\emptyset b \,.\, \underline{0})$ and $(a \,.\, \underline{0} \parallel_\emptyset (b \,.\, \underline{0} + c \,.\, \underline{0})) + ((a \,.\, \underline{0} + c \,.\, \underline{0}) \parallel_\emptyset b \,.\, \underline{0})$ are not $\sim_{\mathrm{HHPB}}$-equiv.

- In the true concurrency spectrum of bisimilarity, $\sim_{\mathrm{HPB}}$ and $\sim_{\mathrm{HHPB}}$ respectively are the *coarsest* and the *finest* behavioral equivalences:
  - Capable of respecting causality, branching, and their interplay while abstracting from choices between identical alternatives ($a + a = a$).
  - Preserved under action refinement (substituting processes for actions).
- Similar properties are valid for ST-bisimilarity, which does not rely on a partial order semantics (Van Glabbeek & Vaandrager 1987; Aceto & Hennessy 1993).

- In the true concurrency spectrum of bisimilarity, $\sim_{\mathrm{HPB}}$ and $\sim_{\mathrm{HHPB}}$ respectively are the *coarsest* and the *finest* behavioral equivalences:
  - Capable of respecting causality, branching, and their interplay while abstracting from choices between identical alternatives ($a + a = a$).
  - Preserved under action refinement (substituting processes for actions).
- Similar properties are valid for ST-bisimilarity, which does not rely on a partial order semantics (Van Glabbeek & Vaandrager 1987; Aceto & Hennessy 1993).
- $\sim_{\mathrm{HHPB}}$ can be obtained as a special case of a categorical definition of bisimilarity on concurrency models (Joyal, Nielsen, Winskel 1996).
- $\sim_{\mathrm{HHPB}}$ is akin to other bisimilarities defined over reversible systems (De Nicola, Montanari, Vaandrager 1990; Danos & Krivine 2004; Phillips & Ulidowski 2007, 2012; Aubert & Cristescu 2020).
- $\sim_{\mathrm{HHPB}}$ coincides with forward-reverse bisimilarity enriched with backward ready multiset equality (Bernardo et al 2025).

- Modal logics by Baldan & Crafa and Phillips & Ulidowski (2014).
- What about axiomatizations of truly concurrent bisimilarities?
- In the interleaving semantics the expansion law is used to identify processes such as $a \,.\, \underline{0} \parallel_{\emptyset} b \,.\, \underline{0}$ and $a \,.\, b \,.\, \underline{0} + b \,.\, a \,.\, \underline{0}$.

- Modal logics by Baldan & Crafa and Phillips & Ulidowski (2014).
- What about axiomatizations of truly concurrent bisimilarities?
- In the interleaving semantics the expansion law is used to identify processes such as $a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$ and $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$.
- In the truly concurrent semantics it is used instead to distinguish processes like the aforementioned two.
- This requires an extension of the syntax of action prefix with suitable discriminating information.

- Modal logics by Baldan & Crafa and Phillips & Ulidowski (2014).
- What about axiomatizations of truly concurrent bisimilarities?
- In the interleaving semantics the expansion law is used to identify processes such as $a \,.\, \underline{0} \,\|_{\emptyset}\, b \,.\, \underline{0}$ and $a \,.\, b \,.\, \underline{0} + b \,.\, a \,.\, \underline{0}$.
- In the truly concurrent semantics it is used instead to distinguish processes like the aforementioned two.
- This requires an extension of the syntax of action prefix with suitable discriminating information.
- Pomset bisimilarity: instead of a single action, a prefix may contain the combination of concurrent or causally related actions, so that the former process expands to $a \,.\, b \,.\, \underline{0} + b \,.\, a \,.\, \underline{0} + (a \,\|\, b) \,.\, \underline{0}$.

- Modal logics by Baldan & Crafa and Phillips & Ulidowski (2014).
- What about axiomatizations of truly concurrent bisimilarities?
- In the interleaving semantics the expansion law is used to identify processes such as $a\,.\,\underline{0} \parallel_\emptyset b\,.\,\underline{0}$ and $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$.
- In the truly concurrent semantics it is used instead to distinguish processes like the aforementioned two.
- This requires an extension of the syntax of action prefix with suitable discriminating information.
- Pomset bisimilarity: instead of a single action, a prefix may contain the combination of concurrent or causally related actions, so that the former process expands to $a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0} + (a \parallel b)\,.\,\underline{0}$.
- FR and HHP bisimilarities: every action is enriched with the backward ready set or multiset of the process reached by executing that action, so we obtain $<a, \{a\}>\,.\,<b, \{a, b\}>\,.\,\underline{0} + <b, \{b\}>\,.\,<a, \{b, a\}>\,.\,\underline{0}$ as opposed to $<a, \{a\}>\,.\,<b, \{b\}>\,.\,\underline{0} + <b, \{b\}>\,.\,<a, \{a\}>\,.\,\underline{0}$.

- Causal bisimilarity corresponds to history-preserving bisimilarity (Darondeau & Degano 1990).
- Every action is enriched with the set of its causing actions, each expressed as a backward pointer.
- The former expands to $<a,\emptyset>.<b,\emptyset>.\underline{0} + <b,\emptyset>.<a,\emptyset>.\underline{0}$ while the latter gets $<a,\emptyset>.<b,\{1\}>.\underline{0} + <b,\emptyset>.<a,\{1\}>.\underline{0}$.

- Causal bisimilarity corresponds to history-preserving bisimilarity (Darondeau & Degano 1990).
- Every action is enriched with the set of its causing actions, each expressed as a backward pointer.
- The former expands to $<a, \emptyset> . <b, \emptyset> . \underline{0} + <b, \emptyset> . <a, \emptyset> . \underline{0}$ while the latter gets $<a, \emptyset> . <b, \{1\}> . \underline{0} + <b, \emptyset> . <a, \{1\}> . \underline{0}$.
- Location bisimilarity corresponds to local history-preserving bisim. (Boudol, Castellani, Hennessy, Kiehn 1994; Castellani 1995) and distributed bisimilarity in the absence of synchronization (Castellani 1988; Castellani & Hennessy 1989).
- Every action is enriched with the name of the location in which it is executed.
- The former expands to $<a, l_a> . <b, l_b> . \underline{0} + <b, l_b> . <a, l_a> . \underline{0}$ while the latter gets $<a, l_a> . <b, l_a l_b> . \underline{0} + <b, l_b> . <a, l_b l_a> . \underline{0}$.