Verification of Functional Programs

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2026-1

Pacto Pedagógico

Pacto Pedagógico

Como miembros de la Universidad EAFIT, nos comprometemos a actuar de manera íntegra siguiendo los más altos estándares éticos y morales.

- Respeto
- Tolerancia
- Honradez
- Compromiso

Pacto Pedagógico

Página web del curso

`https://asr.github.io/courses/verification-of-functional-programs/2026-1`

Pacto Pedagógico

Página web del curso

`https://asr.github.io/courses/verification-of-functional-programs/2026-1`

Conducto regular, fechas y porcentajes de las evaluaciones

La información está en la página web del curso.

Pacto Pedagógico

Página web del curso

`https://asr.github.io/courses/verification-of-functional-programs/2026-1`

Conducto regular, fechas y porcentajes de las evaluaciones

La información está en la página web del curso.

Responsabilidad compartida

- Profesor
- Estudiantes

Pacto Pedagógico

Asistencia a clase

Reglamento académico de los programas de posgrado, Capítulo VI, Artículo 62, Parágrafo 2.

> *"El estudiante de posgrado cuyas faltas de asistencia lleguen al treinta por ciento (30%) del total de las horas de clase programadas para el curso o para una parte de éste, cuando se desarrolle con más de un profesor, en secciones temáticas denominadas 'módulo', pierde con calificación de cero punto cero (0.0) del seminario o curso correspondiente y esta nota afecta el promedio crédito acumulado."*

Pacto Pedagógico

Orientaciones para el curso

- Se recomienda cuatro horas de trabajo por semana (dos horas por cada hora de clase).
- Las clases son presenciales.
- La evaluación a la docencia es obligatoria.
- Se recomienda revisar periódicamente los canales de comunicación institucionales (EAFIT Interactiva y el correo institucional).
- El estudiante podrá entrar a clase a más tardar 20 minutos después de la hora asignada para su inicio.

Interactive Proof Assistants

Interactive Proof Assistants

Description

> *"Proof assistants are computer systems that allow a user to do mathematics on a computer, but not so much the computing (numerical or symbolical) aspect of mathematics but the aspects of proving and defining. So a user can set up a mathematical theory, define properties and do logical reasoning with them."*
> *(Geuvers 2009, p. 3.)*

Interactive Proof Assistants

Description

> *"Proof assistants are computer systems that allow a user to do mathematics on a computer, but not so much the computing (numerical or symbolical) aspect of mathematics but the aspects of proving and defining. So a user can set up a mathematical theory, define properties and do logical reasoning with them."*
> *(Geuvers 2009, p. 3.)*

Examples

- Based on set theory: `Isabelle/ZFC`, `Metamath` and `Mizar`
- Based on higher-order logic: `HOL4`, `HOL Light` and `Isabelle/HOL`
- Based on type theories: `Agda`, `Rocq` and `Lean`.

Isabelle

- University of Cambridge (England) and Technical University of Munich (German)
- Based on higher-order logic
- Tactic-based
- Extraction of programs to `Haskell`, `OCaml`, `Scala` and `SML`
- Written in `SML`
- Integration with ATPs and SMT solvers

Induction

Sets Defined by Induction

Example

We inductively define the set of natural numbers Nat.

Usual definition.

(i) $\mathrm{zero} : \mathrm{Nat}$.

(ii) If $n \in \mathrm{Nat}$ then $\mathrm{succ}\ n : \mathrm{Nat}$.

Definition using inference rules.

$$\frac{}{\mathrm{zero} : \mathrm{Nat}} \qquad \frac{n : \mathrm{Nat}}{\mathrm{succ}\ n : \mathrm{Nat}}$$

Sets Defined by Induction

Example

We inductively define the set of natural numbers Nat.

Usual definition.

(i) $\mathrm{zero} : \mathrm{Nat}$.

(ii) If $n \in \mathrm{Nat}$ then $\mathrm{succ}\ n : \mathrm{Nat}$.

Definition using inference rules.

$$\frac{}{\mathrm{zero} : \mathrm{Nat}} \qquad \frac{n : \mathrm{Nat}}{\mathrm{succ}\ n : \mathrm{Nat}}$$

Remark: In both definitions the fact that Nat is the smallest set generated by the clauses/rules is not stated explicitly.

Structural Induction

Example

Let P be a unary property on Nat. To prove that $P\ n$ for all $n : \mathrm{Nat}$, we can use **structural induction** for P.

$$\dfrac{\begin{array}{c}[P\ n] \\ \vdots \\ P\ \mathrm{zero} \qquad P\ (\mathrm{succ}\ n)\end{array}}{P\ n}\ \text{(structural induction)}$$

Structural Induction

Example

Let A be a set. We inductively define the lists of elements of A, denoted $\mathrm{List}\ A$, by the following inference rules:

$$\frac{}{\mathrm{nil} : \mathrm{List}\ A} \qquad \frac{x : A \qquad xs : \mathrm{List} A}{\mathrm{cons}\ x\ xs : \mathrm{List} A}$$

Structural Induction

Example

Let A be a set. We inductively define the lists of elements of A, denoted $\text{List } A$, by the following inference rules:

$$\frac{}{\text{nil} : \text{List } A} \qquad \frac{x : A \qquad xs : \text{List} A}{\text{cons } x \; xs : \text{List} A}$$

Let A be a set and let P be a unary property on $\text{List } A$. To prove that $P \; xs$ for all $xs : \text{List } A$, we can use **structural induction** for P and $\text{List } A$.

$$\begin{array}{c} [P \; xs] \\ \vdots \\ \dfrac{P \text{ nil} \qquad P \; (\text{cons } x \; xs)}{P \; xs} \end{array} \text{ (structural induction)}$$

Computation Induction

Example

We recursively define the division by 2 (rounded downwards) function.

$$
\begin{aligned}
&\lfloor\,\mathrm{div2}\,\rfloor : \mathrm{Nat} \to \mathrm{Nat} \\
&\lfloor\,\mathrm{div2}\,\rfloor\ \mathrm{zero} &&= \mathrm{zero} \\
&\lfloor\,\mathrm{div2}\,\rfloor\ (\mathrm{succ}\ \mathrm{zero}) &&= \mathrm{zero} \\
&\lfloor\,\mathrm{div2}\,\rfloor\ (\mathrm{succ}\ (\mathrm{succ}\ n)) &&= \mathrm{succ}\ (\lfloor\,\mathrm{div2}\,\rfloor\ n)
\end{aligned}
$$

Computation Induction

Example

We recursively define the division by 2 (rounded downwards) function.

$$\begin{aligned}
&\lfloor\,\mathrm{div2}\,\rfloor : \mathrm{Nat} \to \mathrm{Nat} \\
&\lfloor\,\mathrm{div2}\,\rfloor\ \mathrm{zero} &&= \mathrm{zero} \\
&\lfloor\,\mathrm{div2}\,\rfloor\ (\mathrm{succ}\ \mathrm{zero}) &&= \mathrm{zero} \\
&\lfloor\,\mathrm{div2}\,\rfloor\ (\mathrm{succ}\ (\mathrm{succ}\ n)) &&= \mathrm{succ}\ (\lfloor\,\mathrm{div2}\,\rfloor\ n)
\end{aligned}$$

Let P be a unary property on Nat and $\lfloor\,\mathrm{div2}\,\rfloor$. To prove that $P\ n$ for all $n : \mathrm{Nat}$, we can use **computation induction** for P and $\lfloor\,\mathrm{div2}\,\rfloor$.

$$\frac{\begin{array}{c}[P\ n] \\ \vdots \\ P\ \mathrm{zero} \qquad P\ (\mathrm{succ}\ \mathrm{zero}) \qquad P\ (\mathrm{succ}\ (\mathrm{succ}\ n))\end{array}}{P\ n}\ \text{(computation induction)}$$

Rule Induction

Example

We inductively define the inductive relation Even on natural numbers by the following rules:

$$\frac{}{\mathrm{zero} : \mathrm{Even}} \qquad \frac{n : \mathrm{Even}}{\mathrm{succ}\ (\mathrm{succ}\ n) : \mathrm{Even}}$$

Rule Induction

Example

We inductively define the inductive relation Even on natural numbers by the following rules:

$$\frac{}{\mathrm{zero} : \mathrm{Even}} \qquad \frac{n : \mathrm{Even}}{\mathrm{succ}\ (\mathrm{succ}\ n) : \mathrm{Even}}$$

Let P be a unary property on Nat. To prove that $\mathrm{Even}\ n \Rightarrow P\ n$, we can use **rule induction** for P and Even.

$$\frac{\mathrm{Even}\ n \qquad P\ \mathrm{zero} \qquad P\ (\mathrm{succ}\ (\mathrm{succ}\ n)) \qquad \begin{array}{c}[\mathrm{Even}\ n, P\ n] \\ \vdots \end{array}}{P\ n}\ \text{(rule induction)}$$

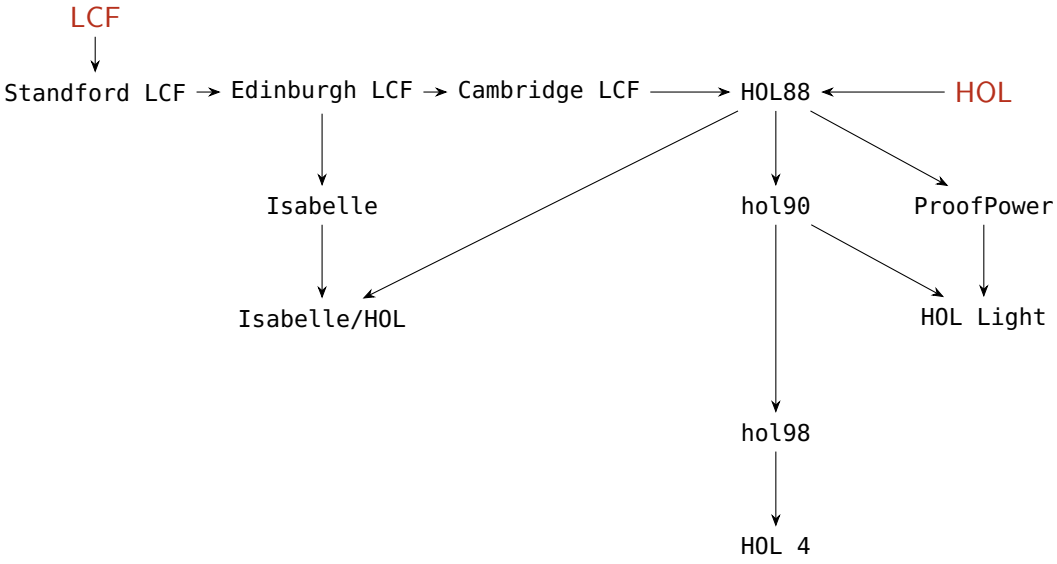
Rule Inversion

Example

Rule inversion goes in the opposite direction to rule induction. For example, given $\text{Even}\ n$ we can use rule inversion for asking which rules could have been used to derive $\text{Even}\ n$ and what constraints these rules impose on n.

Isabelle/HOL

LCF and HOL Proof Checkers and Proof Assistants



Generalisation of first, second, third, . . . -order logics

"It seems very natural to extend the system F of first-order logic by permitting quantification on predicate, propositional, and function variables as well as individual variables. One thus obtains the wffs of a system of second-order logic. One might then introduce predicate and function variables of higher type to denote relations and functions whose arguments may be relations and functions of individuals as well as individuals. Thus one would be led to a system of third-order logic, and if one permitted quantification with respect to these new variables, one would obtain a system of fourth-order logic. This process can be continued indefinitely to obtain logics of arbitrarily high order. Of course, after a while one runs out of different types of letters to use for different types of variables, but this problem can be solved by introducing type symbols to indicate the types of variables, and using a letter with type symbol α as subscript for a variable of type α." (Andrews [1986] 2002, p. 201)

HOL (Higher-Order Logic)

Paper: Alonzo Church (1940). "A Formulation of the Simple Theory of Types".

Also called "simple type theory" or "Church's type theory".

Types: Base types (ι of individuals and o of truth values), variables have fixed types, function types.

Terms: Variables, constants, λ-abstractions and function application.

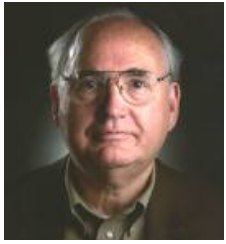
First-order logic with equality: Formulae are of type o.

Sets: Sets of individuals with type $\iota \to o$, sets of sets of individuals with type $(\iota \to o) \to o$, etc.

"Church's type theory has been extensively studied by two of Church's students, L. Henkin and P. Andrews... Henkin also showed in [30] that Church's type theory could be reformulated using only four primitive notions: function application, function abstraction, equality, and definite description... Andrews formulated a version of Church's type theory called Q_0 *that employs the ideas developed by Church, Henkin, and himself." (Farmer 2008, p. 269)*

LCF (Logic for Computable Functions)

Paper: Dana S. Scott ([1969] 1993). "A Type-Theoretical Alternative to ISWIM, CUCH, OWHY".



Typed combinatory logic for reasoning about computable (continuos) functions including partial and non-terminating ones.

Types: Interpreted via domain theory.

Recursive functions: Interpreted via least-fixed points.

Stanford LCF

Paper: Robin Milner (1972). “Logic for Computable Functions. Description of a Machine Implementation”.

Proof checker for LCF.

Edinburgh LCF

Book: Michael J. Gordon, Robin Milner and Christopher P. Wadsworth (1979). "Edinburgh LCF. A Mechanised Logic of Computation".

Edinburgh LCF

Book: Michael J. Gordon, Robin Milner and Christopher P. Wadsworth (1979). "Edinburgh LCF. A Mechanised Logic of Computation".

A problem with `Stanford LCF`: The size of proofs was limited by available memory.

Edinburgh LCF

Book: Michael J. Gordon, Robin Milner and Christopher P. Wadsworth (1979). "Edinburgh LCF. A Mechanised Logic of Computation".

A problem with `Stanford LCF`: The size of proofs was limited by available memory.

A solution:

> *"He [Milner] had the idea that instead of saving whole proofs, the system should just remember the results of proofs, namely theorems... To ensure that theorems could only be created by proof, Milner had the brilliant idea of using an abstract data type whose predefined values were instances of axioms and whose operations were inference rules. Strict typechecking then ensured that the only values that could be created were those that could be obtained from axioms by applying a sequence of inference rules—namely theorems." (M. J. C. Gordon 2000, p. 170)*

Edinburgh LCF

Book: Michael J. Gordon, Robin Milner and Christopher P. Wadsworth (1979). "Edinburgh LCF. A Mechanised Logic of Computation".

A problem with `Stanford LCF`: The size of proofs was limited by available memory.

A solution:

> *"He [Milner] had the idea that instead of saving whole proofs, the system should just remember the results of proofs, namely theorems... To ensure that theorems could only be created by proof, Milner had the brilliant idea of using an abstract data type whose predefined values were instances of axioms and whose operations were inference rules. Strict typechecking then ensured that the only values that could be created were those that could be obtained from axioms by applying a sequence of inference rules—namely theorems." (M. J. C. Gordon 2000, p. 170)*

Introduction: The functional programming language `ML` (Meta Language) and the concepts of "tactics" and "tacticals".

Cambridge LCF

Book: Lawrence C. Paulson ([1987] 2003). “Logic and Computation. Interactive Proof with Cambridge LCF”.

Cambridge LCF

Book: Lawrence C. Paulson ([1987] 2003). "Logic and Computation. Interactive Proof with Cambridge LCF".

Many theoretical and technical improvements in relation to `Cambridge LCF`.

HOL

Paper: Mike Gordon (1985). "HOL. A Machine Oriented Formulation of Higher Order Logic".

HOL

Paper: Mike Gordon (1985). "HOL. A Machine Oriented Formulation of Higher Order Logic".

"The design of HOL was largely taken 'off the shelf', the theory being classical higher order logic and the implementation being LCF." (M. J. C. Gordon 2000, p. 174)

HOL

Paper: Mike Gordon (1985). "HOL. A Machine Oriented Formulation of Higher Order Logic".

"The design of HOL was largely taken 'off the shelf', the theory being classical higher order logic and the implementation being LCF." (M. J. C. Gordon 2000, p. 174)

"In this paper we describe a formal language intended as a basis for hardware specification and verification. This language is not new; the only originality in what follows lies in the presentation of details." (M. J. C. Gordon 1985, p. 2)

HOL

Paper: Mike Gordon (1985). "HOL. A Machine Oriented Formulation of Higher Order Logic".

"The design of HOL was largely taken 'off the shelf', the theory being classical higher order logic and the implementation being LCF." (M. J. C. Gordon 2000, p. 174)

"In this paper we describe a formal language intended as a basis for hardware specification and verification. This language is not new; the only originality in what follows lies in the presentation of details." (M. J. C. Gordon 1985, p. 2)

Logic: Church's HOL with polymorphism and the Axiom of Choice is built-in via Hilbert's ϵ-operator.

(continued on next slide)

Primitive definitional principles

> *"The HOL system, unlike LCF, emphasises definition rather than axiom postulation as the primary method of developing theories. Higher order logic makes possible a purely definitional development of many mathematical objects (numbers, lists, trees, etc.) and this is supported and encouraged." (M. J. C. Gordon 2000, p. 174)*

(continued on next slide)

Derived definitional principles (high-level principles programmed in ML)

Derived definitional principles (high-level principles programmed in ML)

- Melham (1989) *"implemented a derived type-definition principle that converts descriptions of recursive datatypes into primitive definitions and then automatically derives the natural induction and primitive recursion principles for the datatype." (M. J. C. Gordon 2000, p. 174)*

Derived definitional principles (high-level principles programmed in ML)

- Melham (1989) *"implemented a derived type-definition principle that converts descriptions of recursive datatypes into primitive definitions and then automatically derives the natural induction and primitive recursion principles for the datatype." (M. J. C. Gordon 2000, p. 174)*
- Melham (1991) also implemented a derived definitional principle that *"allows inductively defined relations to be specified by a transition system, and then a rule-induction tactic to be automatically generated" (M. J. C. Gordon 2000, p. 174)*

Derived definitional principles (high-level principles programmed in ML)

- Melham (1989) *"implemented a derived type-definition principle that converts descriptions of recursive datatypes into primitive definitions and then automatically derives the natural induction and primitive recursion principles for the datatype." (M. J. C. Gordon 2000, p. 174)*
- Melham (1991) also implemented a derived definitional principle that *"allows inductively defined relations to be specified by a transition system, and then a rule-induction tactic to be automatically generated" (M. J. C. Gordon 2000, p. 174)*
- Slind (1996) implemented a derived definitional principle for handling general recursive definitions of functions.

Isabelle

Paper: Lawrence C. Paulson (1986). “Natural Deduction as Higher-Order Resolution”.
Paper: Lawrence C. Paulson (1989). “The Foundation of a Generic Theorem Prover”.

Isabelle

Paper: Lawrence C. Paulson (1986). “Natural Deduction as Higher-Order Resolution”.
Paper: Lawrence C. Paulson (1989). “The Foundation of a Generic Theorem Prover”.

Isabelle is a generic interactive proof assistant for a broad class of logics.

Isabelle

Paper: Lawrence C. Paulson (1986). "Natural Deduction as Higher-Order Resolution".
Paper: Lawrence C. Paulson (1989). "The Foundation of a Generic Theorem Prover".

Isabelle is a generic interactive proof assistant for a broad class of logics.

Meta-logic (logic framework): Intuitionistic higher-order logic.

Isabelle

Paper: Lawrence C. Paulson (1986). "Natural Deduction as Higher-Order Resolution".
Paper: Lawrence C. Paulson (1989). "The Foundation of a Generic Theorem Prover".

Isabelle is a generic interactive proof assistant for a broad class of logics.

Meta-logic (logic framework): Intuitionistic higher-order logic.

Isabelle
- Isabelle/CTT (extensional constructive type theory)
- Isabelle/FOL (first-order logic)
- Isabelle/ZF (Zermelo-Fraenkel set theory)
- Isabelle/HOL (higher-order logic)

Isabelle/HOL

Book: Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel (2002). "Isabelle/HOL. A Proof Assistant for Higher-Order Logic".

Features: HOL + polymorphism + inductive data types + inductive predicates + recursion

Analysis of Algorithms

Analysis of Algorithms

Description

The **computational complexity** of an algorithm/problem is the amount of resources (e.g. time, space, energy) required to execute/solve it.

Description

The **analysis of algorithms**—term coined by Donald Knuth—is the study of the computational complexity of algorithms. A classical book is (Cormen, Leiserson, Rivest and Stein [1990] 2022).

Description

The **computational complexity theory** is the study of the computacional complexity of problems. A classical book is (Garey and Johnson 1979).

Analysis of Algorithms

Relation between analysis of algorithms and computational complexity theory

> *"The [computational] complexity of an algorithm is always an upper bound on the [computational] complexity of the problem solved by this algorithm."*
> *(Wikipedia, Computational complexity, 2026-03-16).*

Analysis of Algorithms

Two abstractions

For the analysis of algorithms we required two abstractions:

Analysis of Algorithms

Two abstractions

For the analysis of algorithms we required two abstractions:

(i) Where do the algorithms run? In a theoretical computer, i.e., we are interested in machine-independent algorithms.

Analysis of Algorithms

Two abstractions

For the analysis of algorithms we required two abstractions:

(i) Where do the algorithms run? In a theoretical computer, i.e., we are interested in machine-independent algorithms.

(ii) Which complexity are we interested? We are interested in **asymptotic complexity**, i.e., we are interested in the behaviour of the algorithm for large values of the input.

Analysis of Algorithms

The running time function

If the running time of an algorithm depends of the input then it usually means it depends of the size of the input.

We shall use a function

$$T(n) : \mathbb{N} \to \mathbb{R}^{\geq 0}$$

which will denote the running time of an algorithm on inputs of size n.

Analysis of Algorithms

Complexity functions

Given an input of size n we can think in three complexity functions: **best-case complexity** (Ω), **worst-case complexity** (O) and **average-case complexity** (Θ).

Asymptotic Notation: Big O

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big O of $g(n)$**, denoted $O(g(n))$, by

$$O(g(n)) \coloneqq \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c \in \mathbb{R}^{+} \text{ and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0 \,\}.$$

Asymptotic Notation: Big O

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big O of $g(n)$**, denoted $O(g(n))$, by

$$O(g(n)) \coloneqq \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c \in \mathbb{R}^{+} \text{ and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0 \,\}.$$

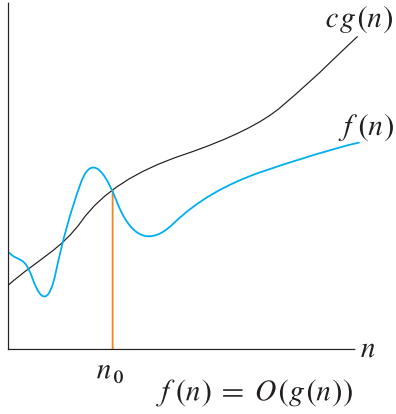
Notation

That $f(n) \in O(g(n))$ is sometimes denoted by $f(n) = O(g(n))$.

Asymptotic Notation: Big O

Remark

If $f(n) \in O(g(n))$ then function $g(n)$ is an upper bound on the growth rate of the function $f(n)$.†



†Figure source: (Cormen, Leiserson, Rivest and Stein [1990] 2022, Fig. 3.1a).

Asymptotic Notation: Big Ω

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big Ω of** $g(n)$, denoted $\Omega(g(n))$, by

$$\Omega(g(n)) = \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c \in \mathbb{R}^{+} \text{ and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0 \,\}.$$

Asymptotic Notation: Big Ω

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big Ω of** $g(n)$, denoted $\Omega(g(n))$, by

$$\Omega(g(n)) = \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0 \,\}.$$

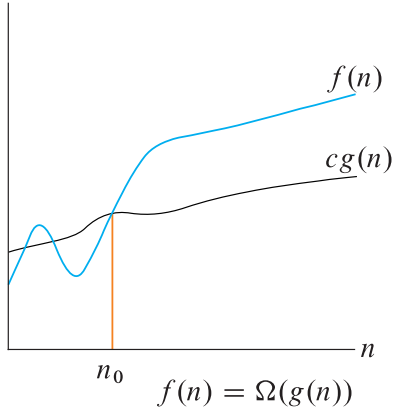
Notation

That $f(n) \in \Omega(g(n))$ is sometimes denoted by $f(n) = \Omega(g(n))$.

Asymptotic Notation: Big Ω

Remark

If $f(n) \in \Omega(g(n))$ then function $g(n)$ is a lower bound on the growth rate of the function $f(n)$.[†]



[†]Figure source: (Cormen, Leiserson, Rivest and Stein [1990] 2022, Fig. 3.1b).

Asymptotic Notation: Big Θ

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big Θ of** $g(n)$, denoted $\Theta(g(n))$, by

$$\Theta(g(n)) = \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c_1, c_2 \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \,\}.$$

Asymptotic Notation: Big Θ

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big Θ of** $g(n)$, denoted $\Theta(g(n))$, by

$$\Theta(g(n)) = \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c_1, c_2 \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \,\}.$$

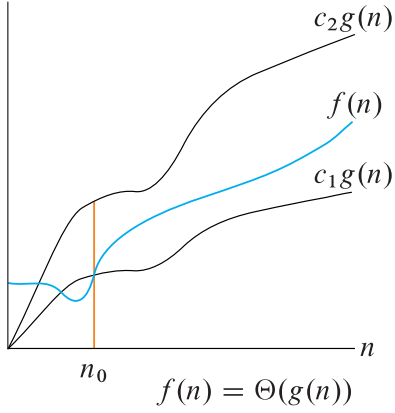
Notation

That $f(n) \in \Theta(g(n))$ is sometimes denoted by $f(n) = \Theta(g(n))$.

Asymptotic Notation: Big Θ

Remark

If $f(n) \in \Theta(g(n))$ then function $g(n)$ is a lower bound and an upper bound on the growth rate of the function $f(n)$.[†]



[†]Figure source: (Cormen, Leiserson, Rivest and Stein [1990] 2022, Fig. 3.1c).

The Tyranny of Growth Rate

Growing rates of some functions

Each operation takes one nanosecond (10^9 seconds).[†]

$n/T(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 yrs
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 yrs		

[†] Table adapted from (Skiena [1997] 2020, Fig.2.2).

The Tyranny of Growth Rate

Supercomputers

Machines from: www.top500.org (last updated: March 2026)
PetaFLOP (PFLOP): 10^{15} floating-point operations per second

Date	Machine	PFLOPs
2026-11	El Capitan	1,809.00
2020-06	Fugaku	415.53
2019-06	Summit	148.60
2018-11	Summit	143.50
2018-06	Summit	122.30
2016-06	Sunway TaihuLight	93.01
2013-06	Tianhe-2	33.86
2012-06	Blue Gene/Q	16.32
2011-06	K computer	8.16

The Tyranny of Growth Rate

Example (3-SAT problem)

A **literal** is an atomic formula (propositional variable) or the negation of an atomic formula.

The Tyranny of Growth Rate

Example (3-SAT problem)

A **literal** is an atomic formula (propositional variable) or the negation of an atomic formula.

A (propositional logic) formula F is in **conjunctive normal form** iff

$$F \text{ has the form } F_1 \wedge \cdots \wedge F_n,$$

where each $F_1, \ldots, F_n$ is a disjunction of literals.

The Tyranny of Growth Rate

Example (3-SAT problem)

A **literal** is an atomic formula (propositional variable) or the negation of an atomic formula.

A (propositional logic) formula F is in **conjunctive normal form** iff

$$F \text{ has the form } F_1 \wedge \cdots \wedge F_n,$$

where each $F_1, \ldots, F_n$ is a disjunction of literals.

3-SAT problem: To determine the satisfiability of a propositional formula in conjunctive normal form where each disjunction of literals is limited to at most three literals.

The Tyranny of Growth Rate

Example (3-SAT problem)

A **literal** is an atomic formula (propositional variable) or the negation of an atomic formula.

A (propositional logic) formula F is in **conjunctive normal form** iff

$$F \text{ has the form } F_1 \wedge \cdots \wedge F_n,$$

where each $F_1, \ldots, F_n$ is a disjunction of literals.

3-SAT problem: To determine the satisfiability of a propositional formula in conjunctive normal form where each disjunction of literals is limited to at most three literals.

The problem was proposed in Karp's 21 NP-complete problems (Karp 1972).

The Tyranny of Growth Rate

Example (Improvements on the time complexity of 3-SAT deterministic algorithmic[†])

$O(1.32793^n)$	Liu (2018)
$O(1.3303^n)$	Makino, Tamaki and Yamamoto (2011) and Makino, Tamaki and Yamamoto (2013)
$O(1.3334^n)$	Moser and Scheder (2011)
$O(1.439^n)$	Kutzkov and Scheder (2010)
$O(1.465^n)$	Scheder (2008)
$O(1.473^n)$	Brueggemann and Kern (2004)
$O(1.481^n)$	Dantsin, Goerdt, Hirsch, Kannan, Kleinberg, Papadimitriou, Raghavan and Schöning (2002)

(continued on next slide)

[†]Main sources: Hertli (2011) and Hertli (2015). Last updated: March 2026.

The Tyranny of Growth Rate

Example (continuation)

$O(1.497^n)$	Schiermeyer (1996)
$O(1.505^n)$	Kullmann (1999)
$O(1.6181^n)$	Monien and Speckenmeyer (1979) and Monien and Speckenmeyer (1985)
$O(2^n)$	Brute-force search

The Tyranny of Growth Rate

Example (3-SAT simulation)

Running 3-SAT times on different supercomputers using the faster deterministic algorithm, i.e. $T(n) = 1.32793^n$.

Date	Machine	PFLOPs	$n = 150$	$n = 200$	$n = 400$
2025-11	El Capitan	1,809.00	1.7 sec	27.6 days	3.2×10^{23} yrs
2020-06	Fugaku	415.53	7.2 sec	120.2 days	1.4×10^{24} yrs
2019-06	Summit	148.60	20.1 sec	336.1 days	4.0×10^{24} yrs
2018-11	Summit	143.50	20.8 sec	348.1 days	4.1×10^{24} yrs
2018-06	Summit	122.30	24.5 sec	1.1 yrs	4.8×10^{24} yrs
2016-06	Sunway TaihuLight	93.01	32.2 sec	1.5 yrs	6.4×10^{24} yrs
2013-06	Tianhe-2	33.86	1.5 min	4.1 yrs	1.7×10^{25} yrs
2012-06	Blue Gene/Q	16.32	3.1 min	8.4 yrs	3.6×10^{25} yrs
2011-06	K computer	8.16	6.1 min	16.8 yrs	7.3×10^{25} yrs

The Tyranny of Growth Rate

Example (3-SAT simulation)

Running 3-SAT times for different deterministic algorithms using the faster supercomputer, i.e. 1,809.00 PFLOPs.

$T(n)$	$n = 150$	$n = 200$	$n = 400$
1.32793^n	1.7 sec	27.6 days	3.2×10^{23} yrs
1.3303^n	2.1 sec	39.4 days	6.7×10^{23} yrs
1.3334^n	3.1 sec	62.8 days	1.69×10^{24} yrs
1.439^n	3.3 days	7.2×10^5 yrs	2.9×10^{37} yrs
1.465^n	48.0 days	2.6×10^7 yrs	3.8×10^{40} yrs
2^n	2.5×10^{19} yrs	2.8×10^{34} yrs	4.5×10^{94} yrs

References

References

Peter B. Andrews [1986] (2002). An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. 2nd ed. Vol. 27. Applied Logic Series. Kluwer (cit. on p. 26).

Tobias Brueggemann and Walter Kern (2004). An Improved Deterministic Local Search Algorithm for 3-SAT. Theoretical Computer Science 329.1–3, pp. 303–313. DOI: 10.1016/j.tcs.2004.08.002 (cit. on p. 73).

Alonzo Church (1940). A Formulation of the Simple Theory of Types. The Journal of Symbolic Logic 5.2, pp. 55–68. DOI: 10.2307/2266170 (cit. on p. 27).

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein [1990] (2022). Introduction to Algorithms. 4th ed. MIT Press (cit. on pp. 51, 60, 63, 66).

Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan and Uwe Schöning (2002). A Deterministic $(2 - 2/(k + 1))^n$ Algorithm for k-SAT Based on Local Search. Theoretical Computer Science 289.1, pp. 69–83. DOI: 10.1016/S0304-3975(01)00174-8 (cit. on p. 73).

Willam M. Farmer (2008). The Seven Virtues of Simple Type Theory. Journal of Applied Logic 6.3, pp. 267–286. DOI: 10.1016/j.jal.2007.11.001 (cit. on p. 27).

Michael R. Garey and David S. Johnson (1979). Computers and Intractability. A Guide to the Theory of NP-completeness. W. H. Freeman and Company (cit. on p. 51).

References

H. Geuvers (2009). Proof Assistants: History, Ideas and Future. Sadhana 34.1, pp. 3–25. DOI: 10.1007/s12046-009-0001-5 (cit. on pp. 10, 11).

Michael J. Gordon, Robin Milner and Christopher P. Wadsworth (1979). Edinburgh LCF. A Mechanised Logic of Computation. Vol. 78. Lecture Notes in Computer Science. Springer. DOI: 10.1007/3-540-09724-4 (cit. on pp. 30–33).

Mike Gordon (July 1985). HOL. A Machine Oriented Formulation of Higher Order Logic. Tech. rep. 68. Computer Laboratory. University of Cambridge. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-68.pdf (cit. on pp. 36–39).

Mike Gordon (2000). From LCF to HOL: A Short History. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner. Ed. by Gordon Plotkin, Colin Stirling and Mads Tofte. Foundations of Computing Series. MIT Press. Chap. 6. DOI: 10.7551/mitpress/5641.003.0012 (cit. on pp. 30–33, 36–44).

Timon Hertli (2011). 3-SAT Faster and Simpler - Unique-SAT Bounds for PPSZ Hold in General. In: Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011). IEEE, pp. 277–284. DOI: 10.1109/FOCS.2011.22 (cit. on p. 73).

Timon Hertli (2015). Improved Exponential Algorithms for SAT and CISP. PhD thesis. ETH Zurich. DOI: 10.3929/ethz-a-010512781 (cit. on p. 73).

References

Richard M. Karp (1972). Reducibility Among Combinatorial Problems. In: Complexity of Computer Computations. Ed. by Raymond E. Miller and James W. Thatcher. Plenum Press, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9 (cit. on pp. 69–72).

O. Kullmann (1999). New Methods for 3-SAT Decision and Worst-Case Analysis. Theoretical Computer Science 223.1–2, pp. 1–72. DOI: 10.1016/S0304-3975(98)00017-6 (cit. on p. 74).

Konstantin Kutzkov and Dominik Scheder (2010). Using CSP to Improve Deterministic 3-SAT. CoRR abs/1007.1166. URL: https://arxiv.org/abs/1007.1166 (cit. on p. 73).

Sixue Liu (2018). Chain, Generalization of Covering Code, and Deterministic Algorithm for k-SAT. In: 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018). Ed. by Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx and Donald Sannella. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs), 88:1–88:13. DOI: 10.4230/LIPIcs.ICALP.2018.88 (cit. on p. 73).

Kazuhisa Makino, Suguru Tamaki and Masaki Yamamoto (2011). Derandomizing HSSW Algorithm for 3-SAT. In: Computing and Combinatorics (COCOON 2011). Ed. by Bin Fu and Ding-Zhu Du. Vol. 6842. Lecture Notes in Computer Science. Springer, pp. 1–12. DOI: 10.1007/978-3-642-22685-4_1 (cit. on p. 73).

References

Kazuhisa Makino, Suguru Tamaki and Masaki Yamamoto (2013). Derandomizing HSSW Algorithm for 3-SAT. Algorithmica 67.2, pp. 112–124. DOI: 10.1007/s00453-012-9741-4 (cit. on p. 73).

T. F. Melham (1991). A Package For Inductive Relation Definitions In HOL. In: 1991 International Workshop on the HOL Theorem Proving System and Its Applications. Ed. by Myla Archer, Jeffrey J. Joyce, Karl N. Levitt and Phillip J. Windley, pp. 350–357. DOI: 10.1109/HOL.1991.596299 (cit. on pp. 41–44).

Thomas F. Melham (1989). Automating Recursive Type Definitions in Higher Order Logic. In: Current Trends in Hardware Verification and Automated Theorem Proving. Ed. by Graham Birtwistle and P. A. Subrahmanyam. Springer. Chap. 9. DOI: 10.1007/978-1-4612-3658-0_9 (cit. on pp. 41–44).

Robin Milner (1972). Logic for Computable Functions. Description of a Machine Implementation. Tech. rep. Stanford Artificial Intelligence Project, Memo AIM-169. STAN-CS-72-288. Stanford University (cit. on p. 29).

B. Monien and E. Speckenmeyer (1979). 3-Satisfiability is Testable in $O(1.62^r)$ Steps. Tech. rep. 3/1979. Reihe Theoretische Informatik, Universität Gesamthochschule Paderborn (cit. on p. 74).

References

B. Monien and E. Speckenmeyer (1985). Solving Satisfiability in less than 2^n Steps. Discrete Applied Mathematics 10.3, pp. 287–295. DOI: 10.1016/0166-218X(85)90050-2 (cit. on p. 74).

Robin A. Moser and Dominik Scheder (2011). A Full Derandomization of Schöning's k-SAT Algorithm. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing (STOC 2011), pp. 245–252. DOI: 10.1145/1993636.1993670 (cit. on p. 73).

Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel (2002). Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Vol. 2283. Lecture Notes in Computer Science. Springer. DOI: 10.1007/3-540-45949-9 (cit. on p. 49).

Lawrence C. Paulson (1986). Natural Deduction as Higher-Order Resolution. The Journal of Logic Programming 3.3, pp. 237–258. DOI: 10.1016/0743-1066(86)90015-4 (cit. on pp. 45–48).

Lawrence C. Paulson (1989). The Foundation of a Generic Theorem Prover. Journal of Automated Reasoning 5.3, pp. 363–397. DOI: 10.1007/BF00248324 (cit. on pp. 45–48).

Lawrence C. Paulson [1987] (2003). Logic and Computation. Interactive Proof with Cambridge LCF. Digitally printed version. Vol. 2. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (cit. on pp. 34, 35).

References

Dominik Scheder (2008). Guided Search and a Faster Deterministic Algorithm for 3-SAT. In: Proc. of the 8th Latin American Symposium on Theoretical Informatic (LATIN 2008). Ed. by Eduardo Sany Laber, Claudson Bornstein, Tito Loana Nogueira and Luerbio Faria. Vol. 4957. Lecture Notes in Computer Science. Springer, pp. 60–71. DOI: 10.1007/978-3-540-78773-0_6 (cit. on p. 73).

Ingo Schiermeyer (1996). Pure Literal Look Ahead: An $O(1.497^n)$ 3-Satisfability Algorithm (Extended Abstract). Workshop on the Satisfability Problem, Siena 1996. URL: http://gauss.ececs.uc.edu/franco_files/SAT96/sat-workshop-abstracts.html (cit. on p. 74).

Dana S. Scott [1969] (1993). A Type-Theoretical Alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science 121.1–2, pp. 441–440. DOI: https://doi.org/10.1016/0304-3975(93)90095-B (cit. on p. 28).

Steven S. Skiena [1997] (2020). The Algorithm Design Manual. 3rd ed. Springer. DOI: 10.1007/978-3-030-54256-6 (cit. on p. 67).

References

Konrad Slind (1996). Function Definition in Higher-Order. In: Theorem Proving in Higher Order Logics (TPHOLs'96). Ed. by Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Joakim Wright, Jim Grundy and John Harrison. Vol. 1125. Lecture Notes in Computer Science. Springer, pp. 381–397. DOI: 10.1007/BFb0105417 (cit. on pp. 41–44).