Continue

This article possibly contains original research. Please improve it by verifying the claims made and adding inline citations. Statements consisting only of original research should be removed. (August 2009) (Learn how and when to remove this message) Programming languages can be grouped by the number and types of paradigms supported. A concise reference for the programming paradigms listed in this article. Concurrent programming – have language constructs for concurrency, these may involve multi-threading, support for distributed computing, message passing, shared resources (including shared memory), or futures Actor programming – concurrent computation with actors that make local decisions in response to the environment (capable of selfish or competitive behaviour) Constraint programming – relations between variables are expressed as constraints (or constraint networks), directing allowable solutions (uses constraint satisfaction or simplex algorithm) Dataflow programming – forced recalculation of formulas when data values change (e.g. spreadsheets) Declarative programming – describes what computation should perform, without specifying detailed state changes cf. imperative programming (functional and logic programming are major subgroups of declarative programming) Distributed programming – have support for multiple autonomous computers that communicate via computer networks Functional programming – uses evaluation of mathematical functions and avoids state and mutable data Generic programming – uses algorithms written in terms of to-be-specified-later types that are then instantiated as needed for specific types provided as parameters Imperative programming – explicit statements that change a program state Logic programming – uses explicit mathematical logic for programming Metaprogramming – writing programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime Template metaprogramming – metaprogramming methods in which a compiler uses templates to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled Reflective programming – metaprogramming methods in which a program modifies or extends itself Object-oriented programming – uses data structures consisting of data fields and methods together with their interactions (objects) to design programs Class-based – object-oriented programming in which inheritance is achieved by defining classes of objects, versus the objects themselves Prototype-based – object-oriented programming that avoids classes and implements inheritance via cloning of instances Pipeline programming – a simple syntax change to add syntax to nest function calls to language originally designed with none Rule-based programming – a network of rules of thumb that comprise a knowledge base and can be used for expert systems and problem deduction & resolution Visual programming – manipulating program elements graphically rather than by specifying them textually (e.g. Simulink); also termed diagrammatic programming[1] List of multi-paradigm programming languages Language Paradigm count Concurrent Constraints Dataflow Declarative Distributed Functional Metaprogramming Generic Imperative Logic Reflection Object-oriented Pipelines Visual Rule-based Other Ada[2][3][4][5][6] 5 Yes[a 1] — — — Yes — — Yes Yes — Yes[a 2] — — — — ALF 2 — — — — Yes — — — Yes — — — — — — AmigaE[citation needed] 2 — — — — — — — — Yes — — Yes[a 2] — — — — APL 3 — — — — — Yes — — Yes — — — — — — Array (multi-dimensional) BETA[citation needed] 3 — — — — — — Yes — — Yes — — Yes[a 2] — — — — C++ 7 (15) Yes[7][8][9] Library[10] Library[11][12] Library[13][14] Library[15][16] Yes Yes[17] Yes[a 3] Yes Library[18][19] Library[20] Yes[a 2] Library[21] — Library[22] Array (multi-dimensional; using STL) C# 6 (7) Yes — Library[a 4] — — Yes[a 5] — Yes Yes — Yes Yes[a 2] — — — Reactive[a 6] ChucK[citation needed] 3 Yes — — — — — — — Yes — — Yes[a 2] — — — — Claire 2 — — — — — Yes — — — — — — Yes[a 2] — — — — Clojure 5 Yes[23][24] — — Yes — Yes[25] Yes[26] — — Library[27] — — Yes[28] — — Multiple dispatch,[29] Agents[30] Common Lisp 7 (14) Library[31] Library[32] Library[33] Yes[34] Library[35] Yes Yes[36] Yes Library[37] Yes Yes[a 7][a 2][38] Library[39] Library[40] Library[41] Multiple dispatch, meta-OOP system,[42] Language is extensible via metaprogramming. Curl 5 — — — — — Yes — Yes[a 3] Yes — Yes Yes[a 2] — — — — Curry 4 Yes Yes — — — Yes — — — — Yes — — — — — — D (version 2.0)[43][44] 7 Yes[a 8] — — — — Yes Yes[45][a 3] Yes[a 3] Yes — Yes Yes[a 2] — — — — Delphi 3 — — — — — — — Yes[a 3] Yes — — Yes[a 2] — — — — Dylan[citation needed] 3 — — — — — Yes — — — — Yes Yes[a 2] — — — — E 3 Yes — — — Yes — — — — — — Yes[a 2] — — — — ECMAScript[46][47] (ActionScript, E4X, JavaScript, JScript) 4 (5) Partial[a 9][a 10] — — Library[48][49] — Yes — — Yes — Yes Yes[a 11] Library[50][51] — — Reactive,[a 12][52] event driven[a 13][a 14] Erlang 3 Yes — — Yes Yes Yes — — — — — — Yes — — — — Elixir 4 Yes — — — Yes Yes Yes — — — — — — Yes — — — — Elm 6 Yes — Yes Yes — Yes (pure)[a 15] — Yes — — — — Yes — — — Reactive F# 7 (8) Yes[a 8] — Library[a 4] Yes — Yes — Yes Yes — Yes Yes[a 2] — — — Reactive[a 6] Fortran 4 (5) Yes — — — Yes[a 15] — Yes[a 16] — — — Yes[a 2] — — — Array (multi-dimensional) Go 4 Yes — — — — — — — — Yes — Yes — — Yes — — — Haskell 8 (15) Yes Library[53] Library[54] Yes Library[55] Yes (lazy) (pure)[a 15] Yes[56] Yes Yes Library[57] — Partial[a 17] Yes Yes Library[58] Literate, reactive, dependent types (partial) Io 4 Yes[a 8] — — — — Yes — — — Yes — — Yes[a 11] — — — — J[citation needed] 3 — — — — — Yes — — Yes — — Yes[a 2] — — — — Java 6 Yes Library[59] Library[60] — — Yes — Yes Yes — Yes Yes[a 2] — — — — Julia 9 (17) Yes Library[61] Library[62][63] Library[64] Yes Yes (eager) Yes Yes Yes Library[65] Yes Partial[a 18] Yes — Library[66][67] Multiple dispatch,Array (multi-dimensional); optionally lazy[68] and reactive (with libraries) Kotlin 8 Yes — — — — Yes Yes Yes Yes — Yes Yes Yes — — — LabVIEW 4 Yes — — — — — — — — — — Yes — — Yes — — — Lava 2 — — — — — — — — — — — — Yes[a 2] — Yes — — LispWorks (version 6.0 with support for symmetric multi-processing, rules, logic (Prolog), CORBA) 9 Yes — — — Yes Yes Yes — Yes Yes Yes Yes[a 2] — — Yes — Lua[citation needed] 3 — — — — — Yes — — Yes — — Yes[a 11] — — — — MATLAB 6 (10) Toolbox[69] Toolbox[70] Yes[71] — Toolbox[72] — Yes[73] Yes[74] — — Yes[75] Yes[76] — Yes[77] — Array (multi-dimensional) Nemerle 7 Yes — — — — Yes Yes Yes Yes — Yes Yes[a 2] — — — — Object Pascal 4 Yes — — — — Yes — — Yes — — Yes[a 2] — — — — OCaml 4 — — — — — Yes — Yes Yes — — Yes[a 2] — — — — Oz 11 Yes Yes Yes Yes Yes Yes — — Yes Yes — Yes[a 2] Yes — Yes — Perl[citation needed] 8 (9) Yes[78] — Yes[79] — — Yes Yes — Yes Yes — Yes[a 2] Yes[a 2] Yes — — — PHP[80][81][82] 4 — — — — — Yes — — Yes — Yes Yes[a 2] — — — — Poplog 3 — — — — — Yes — — Yes Yes — — — — — — — Prograph 3 — — Yes — — — — — — — — Yes[a 2] — Yes — — Python 5 (10) Library[83][84] Library[85] — — Library[86] Yes Yes[87][88] Yes[89][90] Yes Library[91] Yes Yes[a 2] — — — Structured R 4 (6) Library[92] — — — Library[93] Yes — — Yes — Yes Yes Yes[94] — — Array (multi-dimensional) Racket 10 Yes[95] Yes[96] Yes[97] — Yes[98] Yes Yes — Yes Yes Yes Yes — — — Lazy[99] Raku 10 Yes[100] Library[101] Yes[102] — Library[103] Yes Yes[104] Yes[105] Yes — Yes[106] Yes[107] Yes — — Multiple dispatch, lazy lists, reactive. ROOP 3 — — — — — — — — Yes Yes — — — — Yes — — Yes — Ruby 5 — — — — — Yes Yes — Yes — Yes — Yes Yes[a 2] — — — — Rust (version 1.0.0-alpha) 6 Yes[a 8] — — — — Yes Yes[108][109] Yes[110] Yes — — Yes — — — Linear, affline, and ownership types Sather[citation needed] 2 — — — — — Yes — — — — — Yes[a 2] — — — — Scala[111][112] 9 Yes[a 8] — Yes[a 19] Yes — Yes Yes Yes Yes — Yes Yes[a 2] — — — — Simula[citation needed] 2 — — — — — — — — Yes — — Yes[a 2] — — — — SISAL 3 Yes — Yes — — Yes — — — — — — — — — — Spreadsheets 2 — — — — — — Yes — — — — — — — — Yes — — Swift 7 Yes — — — — Yes Yes Yes Yes — Yes Yes[a 2] — — — Block-structured Tcl with Snit extension[citation needed] 3 — — — — — Yes — — Yes[113] — — Yes — — Yes[a 11][114] — — — — Visual Basic .NET 6 (7) Yes — Library[a 4] — — Yes — Yes — Yes Yes[a 2] — — — Reactive[a 6] Windows PowerShell 6 — — — — — Yes — Yes Yes — Yes Yes[a 2] Yes — — — Wolfram Language & Mathematica 13[115] (14) Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes[116] — Yes Knowledge Based Programming paradigm List of programming languages by type Domain-specific language Domain-specific multimodeling ^ rendezvous and monitor-like based ^ a b c d e f g h i j k l m n o p q r s t u v w x y z aa ab ac ad ae af ag ah ai class-based ^ a b c d e template metaprogramming ^ a b using TPL Dataflow ^ only lambda support (lazy functional programming) ^ a b c using Reactive Extensions (Rx) ^ multiple dispatch, method combinations ^ a b c d e actor programming ^ promises, native extensions ^ using Node.js' cluster module or child_process.fork method, web workers in the browser, etc. ^ a b c d Prototype-based ^ using Reactive Extensions (RxJS) ^ in Node.js via their events module ^ in browsers via their native EventTarget API ^ a b c purely functional ^ parameterized classes ^ immutable ^ Uses structs with function polymorphism and multiple dispatch ^ Akka Archived 2013-01-19 at the Wayback Machine ^ Bragg, S.D.; Driskill, C.G. (20–22 September 1994). "Diagrammatic-graphical programming languages and DoD-STD-2167A". Proceedings of AUTOTESTCON '94 (IEEEXplore). Institute of Electrical and Electronics Engineers (IEEE). pp. 211–220. doi:10.1109/AUTEST.1994.381508. ISBN 978-0-7803-1910-3. S2CID 62509261. ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, Section 9: Tasks and Synchronization ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3 Annex E: Distributed Systems ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, Section 12: Generic Units ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, Section 6: Subprograms ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, 3.9 Tagged Types and Type Extensions ^ Thread support ^ Atomics support ^ Memory model ^ Gecode ^ SystemC ^ Boost.Iostreams ^ Boolinq ^ "AraRat" (PDF). Archived from the original (PDF) on 2019-08-19. Retrieved 2019-09-15. ^ OpenMPI ^ Boost.MPI ^ Boost.MPL ^ LC++ ^ Castor Archived 2013-01-25 at the Wayback Machine ^ Reflect Library ^ N3534 ^ Boost.Spirit ^ Clojure - Concurrent Programming ^ Clojure - core.async ^ Clojure - Functional Programming ^ Clojure - Macros ^ Clojure - core.logic ^ Clojure - Threading Macros Guide ^ Multimethods and Hierarchies ^ Agents and Asynchronous Actions ^ "concurrency". CLiki. ^ [1] constraint programming inside CL through extensions ^ [2] dataflow extension ^ [3] by creating DSLs using the built-in metaprogramming; also see note on functional, constraint and logic paradigms, which are part of declarative ^ [4] MPI, etc via language extensions ^ template metaprogramming using macros (see C++) ^ [5] [6] [7] Prolog implemented as a language extension ^ Common Lisp Object System see Wikipedia article on CLOS, the Common Lisp Object System. ^ implemented by the user via a short macro, example of implementation ^ - Visual programming tool based on Common Lisp ^ [8] rule-based programming extension ^ [9] Archived 2018-04-26 at the Wayback Machine through the Meta Object Protocol ^ D Language Feature Table ^ Phobos std.algorithm ^ D language String Mixins ^ The Little JavaScripter demonstrates fundamental commonality with Scheme, a functional language. ^ Object-Oriented Programming in JavaScript Archived 2019-02-10 at the Wayback Machine gives an overview of object-oriented programming techniques in JavaScript. ^ "React – A JavaScript library for building user interfaces". 2019-04-08. ^ "TNG-Hooks". GitHub. 2019-04-08. ^ "Lodash documentation". 2019-04-08. ^ "mori". 2019-04-08. ^ "TNG-Hooks". GitHub. 2019-04-08. ^ "Prolog embedding". Haskell.org. ^ "Functional Reactive Programming". HaskellWiki. ^ Cloud Haskell ^ "Template Haskell". HaskellWiki. ^ "Logict: A backtracking logic-programming monad". Haskell.org. ^ Kollmansberger, Steve; Erwig, Martin (30 May 2006). "Haskell Rules: Embedding Rule Systems in Haskell" (PDF). Oregon State University. ^ JSR 331: Constraint Programming API ^ Google Cloud Platform Dataflow SDK ^ "JuliaOpt/JuMP.jl". GitHub. JuliaOpt. 11 February 2020. Retrieved 12 February 2020. ^ "GitHub - MikeInnes/DataFlow.jl". GitHub. 2019-01-15. ^ "GitHub - JuliaGizmos/Reactive.jl: Reactive programming primitives for Julia". GitHub. 2018-12-28. ^ Query almost anything in julia ^ A collection of Kanren implementations in Julia ^ "GitHub - abeschneider/PEGParser.jl: PEG Parser for Julia". GitHub. 2018-12-03. ^ "GitHub - gitfoxi/Parsimonious.jl: A PEG parser generator for Julia". GitHub. 2017-08-03. ^ Lazy ^ "Execute loop iterations in parallel". mathworks.com. Retrieved 21 October 2016. ^ "Write Constraints". mathworks.com. Retrieved 21 October 2016. ^ "Getting Started with SimEvents". mathworks.com. Retrieved 21 October 2016. ^ "Execute loop iterations in parallel". mathworks.com. Retrieved 21 October 2016. ^ "Execute MATLAB expression in text - MATLAB eval". mathworks.com. Retrieved 21 October 2016. ^ "Determine class of object". mathworks.com. Retrieved 21 October 2016. ^ "Class Metadata". mathworks.com. Retrieved 21 October 2016. ^ "Object-Oriented Programming". mathworks.com. Retrieved 21 October 2016. ^ "Simulink". mathworks.com. Retrieved 21 October 2016. ^ interpreter based threads ^ Higher Order Perl ^ PHP Manual, Chapter 17. Functions ^ PHP Manual, Chapter 19. Classes and Objects (PHP 5) ^ PHP Manual, Anonymous functions ^ "Parallel Processing and Multiprocessing in Python". Python Wiki. Retrieved 21 October 2016. ^ "threading — Higher-level threading interface". docs.python.org. Retrieved 21 October 2016. ^ "python-constraint". pypi.python.org. Retrieved 21 October 2016. ^ "DistributedProgramming". Python Wiki. Retrieved 21 October 2016. ^ "Chapter 9. Metaprogramming". chimera.labs.oreilly.com. Archived from the original on 23 October 2016. Retrieved 22 October 2016. ^ "Metaprogramming". readthedocs.io. Retrieved 22 October 2016. ^ "PEP 443 – Single-dispatch generic functions". python.org. Retrieved 22 October 2016. ^ "PEP 484 – Type Hints". python.org. Retrieved 22 October 2016. ^ "PyDatalog". Retrieved 22 October 2016. ^ "Futureverse". ^ "future batchtools". ^ "Magrittr: A Forward Pipe Operator for R". cran.r-project.org\access-date=13 July 2017. 17 November 2020. ^ Racket Guide: Concurrency and Synchronization ^ The Rosette Guide ^ FrTime: A Language for Reactive Programs ^ Racket Guide: Distributed Places ^ Lazy Racket ^ Channels and other mechanisms ^ "Problem Solver module". ^ Feed operator ^ Cro module ^ "Meta-programming: What, why and how". 2011-12-14. ^ Parametrized Roles ^ "Meta-object protocol (MOP)". ^ Classes and Roles ^ "The Rust macros guide". Rust. Retrieved 19 January 2015. ^ "The Rust compiler plugins guide". Rust. Retrieved 19 January 2015. ^ The Rust Reference §6.1.3.1 ^ An Overview of the Scala Programming Language ^ Scala Language Specification ^ "Tcl Programming/Introduction". en.wikibooks.org. Retrieved 22 October 2016. ^ "TCLLIB - Tcl Standard Library: snitfaq". sourceforge.net. Retrieved 22 October 2016. ^ Notes for Programming Language Experts, Wolfram Language Documentation. ^ External Programs, Wolfram Language Documentation. Jim Coplien, Multiparadigm Design for C++, Addison-Wesley Professional, 1998. Retrieved from "

Programming language type systems Type systems General concepts Type safety Strong vs. weak typing Major categories Static vs. dynamic Manifest vs. inferred Nominal vs. structural Duck typing Minor categories Abstract Dependent Flow-sensitive Gradual Intersection Latent Refinement Substructural Unique Session vte In computer programming, one of the many ways that programming languages are colloquially classified is whether the language's type system makes it strongly typed or weakly typed (loosely typed). However, there is no precise technical definition of what the terms mean and different authors disagree about the implied meaning of the terms and the relative rankings of the "strength" of the type systems of mainstream programming languages.[1] For this reason, writers who wish to write unambiguously about type systems often eschew the terms "strong typing" and "weak typing" in favor of specific expressions such as "type safety". Generally, a strongly typed language has stricter typing rules at compile time, which implies that errors are more likely to happen during compilation. Most of these rules affect variable assignment, function return values, procedure arguments and function calling. Dynamically typed languages (where type checking happens at run time) can also be strongly typed. In dynamically typed languages, values, rather than variables, have types. A weakly typed language has looser typing rules and may produce unpredictable or even erroneous results or may perform implicit type conversion at runtime.[2] A different but related concept is latent typing. In 1974, Barbara Liskov and Stephen Zilles defined a strongly typed language as one in which "whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function."[3] In 1977, K. Jackson wrote, "In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types."[4] A number of different language design decisions have been referred to as evidence of "strong" or "weak" typing. Many of these are more accurately understood as the presence or absence of type safety, memory safety, static type-checking, or dynamic type-checking. "Strong typing" generally refers to use of programming language types in order to both capture invariants of the code, and ensure its correctness, and definitely exclude certain classes of programming errors. Thus there are many "strong typing" disciplines used to achieve these goals. Some programming languages make it easy to use a value of one type as if it were a value of another type. This is sometimes described as "weak typing". For example, Aahz Maruch observes that "Coercion occurs when you have a statically typed language and you use the syntactic features of the language to force the usage of one type as if it were a different type (consider the common use of void* in C). Coercion is usually a symptom of weak typing. Conversion, on the other hand, creates a brand-new object of the appropriate type."[5] As another example, GCC describes this as type-punning and warns that it will break strict aliasing. Thiago Macieira discusses several problems that can arise when type-punning causes the compiler to make inappropriate optimizations.[6] There are many examples of languages that allow implicit type conversions, but in a type-safe manner. For example, both C++ and C# allow programs to define operators to convert a value from one type to another with well-defined semantics. When a C++ compiler encounters such a conversion, it treats the operation just like a function call. In contrast, converting a value to the C type void* is an unsafe operation that is invisible to the compiler. Some programming languages expose pointers as if they were numeric values, and allow users to perform arithmetic on them. These languages are sometimes referred to as "weakly typed", since pointer arithmetic can be used to bypass the language's type system. Some programming languages support untagged unions, which allow a value of one type to be viewed as if it were a value of another type. In Luca Cardelli's article Typeful Programming,[7] a "strong type system" is described as one in which there is no possibility of an unchecked runtime type error. In other writing, the absence of unchecked run-time errors is referred to as safety or type safety; Tony Hoare's early papers call this property security.[8] This section possibly contains original research. Please improve it by verifying the claims made and adding inline citations. Statements consisting only of original research should be removed. (May 2018) (Learn how and when to remove this message) This section needs additional citations for verification. Please help improve this article by adding citations to reliable sources in this section. Unsourced material may be challenged and removed. (May 2020) (Learn how and when to remove this message) Some of these definitions are contradictory, others are merely conceptually independent, and still others are special cases (with additional constraints) of other, more "liberal" (less strong) definitions. Because of the wide divergence among these definitions, it is possible to defend claims about most programming languages that they are either strongly or weakly typed. For instance: Java, Pascal, Ada, and C require variables to have a declared type, and support the use of explicit casts of arithmetic values to other arithmetic types. Java, C#, Ada, and Pascal are sometimes said to be more strongly typed than C, because C supports more kinds of implicit conversions, and allows pointer values to be explicitly cast while Java and Pascal do not. Java may be considered more strongly typed than Pascal as methods of evading the static type system in Java are controlled by the Java virtual machine's type system. C# and VB.NET are similar to Java in that respect, though they allow disabling of dynamic type checking by explicitly putting code segments in an "unsafe context". Pascal's type system has been described as "too strong", because the size of an array or string is part of its type, making some programming tasks very difficult. However, Delphi fixes this issue.[9][10] Smalltalk, Ruby, Python, and Self are all "strongly typed" in the sense that typing errors are prevented at runtime and they do little implicit type conversion, but these languages make no use of static type checking: the compiler does not check or enforce type constraint rules. The term duck typing is now used to describe the dynamic typing paradigm used by the languages in this group. The Lisp family of languages are all "strongly typed" in the sense that typing errors are prevented at runtime. Some Lisp dialects like Common Lisp or Clojure do support various forms of type declarations[11] and some compilers (CMU Common Lisp (CMUCL)[12] and related) use these declarations together with type inference to enable various optimizations and limited forms of compile time type checks. Standard ML, F#, OCaml, Haskell, Go and Rust are statically type-checked, but the compiler automatically infers a precise type for most values. Assembly language and Forth can be characterized as untyped. There is no type checking; it is up to the programmer to ensure that data given to functions is of the appropriate type. Comparison of programming languages Data type includes a more thorough discussion of typing issues Design by contract (strong typing as implicit contract form) Latent typing Memory safety Type safety Type system Strongly typed identifier ^ "What to know before debating type systems | Ovid [blogs.perl.org]". blogs.perl.org. Retrieved 2023-06-27. ^ "CS1130. Transition to OO programming. – Spring 2012 --self-paced version". Cornell University, Department of Computer Science. 2005. Archived from the original on 2015-11-23. Retrieved 2015-11-23.{{cite web}}: CS1 maint: bot: original URL status unknown (link) ^ Liskov, B; Zilles, S (1974). "Programming with abstract data types". ACM SIGPLAN Notices. 9 (4): 50–59. CiteSeerX 10.1.1.136.3043. doi:10.1145/942572.807045. ^ Jackson, K. (1977). "Parallel processing and modular software construction". Design and Implementation of Programming Languages. Lecture Notes in Computer Science. Vol. 54. pp. 436–443. doi:10.1007/BFb0021435. ISBN 3-540-08360-X. ^ Aahz. "Typing: Strong vs. Weak, Static vs. Dynamic". Retrieved 16 August 2015. ^ "Type-punning and strict-aliasing - Qt Blog". Qt Blog. Retrieved 18 February 2020. ^ Luca Cardelli, "Typeful programming" ^ Hoare, C. A. R. 1974. Hints on Programming Language Design. In Computer Systems Reliability, ed. C. Bunyan. Vol. 20 pp. 505–534. ^ InfoWorld. 1983-04-25. Retrieved 16 August 2015. ^ Kernighan, Brian (1981). "Why Pascal is not my favorite programming language". Archived from the original on 2012-04-06. Retrieved 2011-10-22. ^ "CLHS: Chapter 4". Retrieved 16 August 2015. ^ "CMUCL User's Manual: The Compiler". Archived from the original on 8 March 2016. Retrieved 16 August 2015. Retrieved from "

Computer science concept This article is about type systems in computer programming. For the formal study of type systems, see Type theory. This article includes a list of general references, but it lacks sufficient corresponding inline citations. Please help to improve this article by introducing more precise citations. (October 2010) (Learn how and when to remove this message) This article is written like a personal reflection, personal essay, or argumentative essay that states a Wikipedia editor's personal feelings or presents an original argument about a topic. Please help improve it by rewriting it in an encyclopedic style. (July 2016) (Learn how and when to remove this message) Type systems General concepts Type safety Strong vs. weak typing Major categories Static vs. dynamic Manifest vs. inferred Nominal vs. structural Duck typing Minor categories Abstract Dependent Flow-sensitive Gradual Intersection Latent Refinement Substructural Unique Session vte In computer programming, a type system is a logical system comprising a set of rules that assigns a property called a type (for example, integer, floating point, string) to every term (a word, phrase, or other set of symbols). Usually the terms are various language constructs of a computer program, such as variables, expressions, functions, or modules.[1] A type system dictates the operations that can be performed on a term. For variables, the type system determines the allowed values of that term. Type systems formalize and enforce the otherwise implicit categories the programmer uses for algebraic data types, data structures, or other data types, such as "string", "array of float", "function returning boolean". Type systems are often specified as part of programming languages and built into interpreters and compilers, although the type system of a language can be extended by optional tools that perform added checks using the language's original type syntax and grammar. The main purpose of a type system in a programming language is to reduce possibilities for bugs in computer programs due to type errors.[2] The given type system in question determines what constitutes a type error, but in general, the aim is to prevent operations expecting a certain kind of value from being used with values of which that operation does not make sense (validity errors). Type systems allow defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen statically (at compile time), dynamically (at run time), or as a combination of both. Type systems have other purposes as well, such as expressing business rules, enabling certain compiler optimizations, allowing for multiple dispatch, and providing a form of documentation. An example of a simple type system is that of the C language. The portions of a C program are the function definitions. One function is invoked by another function. The interface of a function states the name of the function and a list of parameters that are passed to the function's code. The code of an invoking function states the name of the invoked, along with the names of variables that hold values to pass to it. During a computer program's execution, the values are placed into temporary storage, then execution jumps to the code of the invoked function. The invoked function's code accesses the values and makes use of them. If the instructions inside the function are written with the assumption of receiving an integer value, but the calling code passed a floating-point value, then the wrong result will be computed by the invoked function. The C compiler checks the types of the arguments passed to a function when it is called against the types of the parameters declared in the function's definition. If the types do not match, the compiler throws a compile-time error or warning. A compiler may also use the static type of a value to optimize the storage it needs and the choice of algorithms for operations on the value. In many C compilers the float data type, for example, is represented in 32 bits, in accord with the IEEE specification for single-precision floating point numbers. They will thus use floating-point-specific microprocessor operations on those values (floating-point addition, multiplication, etc.). The depth of type constraints and the manner of their evaluation affect the typing of the language. A programming language may further associate an operation with various resolutions for each type, in the case of type polymorphism. Type theory is the study of type systems. The concrete types of some programming languages, such as integers and strings, depend on practical issues of computer architecture, compiler implementation, and language design. Formally, type theory studies type systems. A programming language must have the opportunity to type check using the type system whether at compile time or runtime, manually annotated or automatically inferred. As Mark Manasse concisely put it:[3] The fundamental problem addressed by a type theory is to ensure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension. Assigning a data type, termed typing, gives meaning to a sequence of bits such as a value in memory or some object such as a variable. The hardware of a general purpose computer is unable to discriminate between for example a memory address and an instruction code, or between a character, an integer, or a floating-point number, because it makes no intrinsic distinction between any of the possible values that a sequence of bits might mean.[note 1] Associating a sequence of bits with a type conveys that meaning to the programmable hardware to form a symbolic system composed of that hardware and some program. A program associates each value with at least one specific type, but it also can occur that one value is associated with many subtypes. Other entities, such as objects, modules, communication channels, and dependencies can become associated with a type. Even a type can become associated with a type. An implementation of a type system could in theory associate identifications called data type (a type of a value), class (a type of an object), and kind (a type of a type, or metatype). These are the abstractions that typing can go through, on a hierarchy of levels contained in a system. When a programming language evolves a more elaborate type system, it gains a more finely grained rule set than basic type checking, but this comes at a price when the type inferences (and other properties) become undecidable, and when more attention must be paid by the programmer to annotate code or to consider computer-related operations and functioning. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in a type safe manner. A programming language compiler can also implement a dependent type or an effect system, which enables even more program specifications to be verified by a type checker. Beyond simple value-type pairs, a virtual "region" of code is associated with an "effect" component describing what is being done with what, and enabling for example to "throw" an error report. Thus the symbolic system may be a type and effect system, which endows it with more safety checking than type checking alone. Whether automated by the compiler or specified by a programmer, a type system renders program behavior illegal if it falls outside the type-system rules. Advantages provided by programmer-specified type systems include: Abstraction (or modularity) – Types enable programmers to think at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can begin to think of a string as a set of character values instead of as an array of bytes. Higher still, types enable programmers to think about and express interfaces between two of any-sized subsystems. This enables more levels of localization so that the definitions required for interoperability of the subsystems remain consistent when those two subsystems communicate. Documentation – In more expressive type systems, types can serve as a form of documentation clarifying the intent of the programmer. For example, if a programmer declares a function as returning a timestamp type, this documents the function when the timestamp type can be explicitly declared deeper in the code to be an integer type. Advantages provided by compiler-specified type systems include: Optimization – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of four bytes, the compiler may be able to use more efficient machine instructions. Safety – A type system enables the compiler to detect meaningless or invalid code. For example, we can identify an expression 3 / "Hello, World" as invalid, when the rules do not specify how to divide an integer by a string. Strong typing offers more safety, but cannot guarantee complete type safety. A type error occurs when an operation receives a different type of data than it expected.[4] For example, a type error would happen if a line of code divides two integers, and is passed a string of letters instead of an integer.[4] It is an unintended condition[note 2] which might manifest in multiple stages of a program's development. Thus a facility for detection of the error is needed in the type system. In some languages, such as Haskell, for which type inference is automated, lint might be available to its compiler to aid in the detection of error. Type safety contributes to program correctness, but might only guarantee correctness at the cost of making the type checking itself an undecidable problem (as in the halting problem). In a type system with automated type checking, a program may prove to run incorrectly yet produce no compiler errors. Division by zero is an unsafe and incorrect operation, but a type checker which only runs at compile time does not scan for division by zero in most languages; that division would surface as a runtime error. To prove the absence of these defects, other kinds of formal methods, collectively known as program analyses, are in common use. Alternatively, a sufficiently expressive type system, such as in dependently typed languages, can prevent these kinds of errors (for example, expressing the type of non-zero numbers). In addition, software testing is an empirical method for finding errors that such a type checker would not detect. The process of verifying and enforcing the constraints of types—type checking—may occur at compile time (a static check) or at run-time (a dynamic check). If a language specification requires its typing rules strongly, more or less allowing only those automatic type conversions that do not lose information, one can refer to the process as strongly typed; if not, as weakly typed. The terms are not usually used in a strict sense. See also: Category:Statically typed programming languages Static type checking is the process of verifying the type safety of a program based on analysis of a program's text (source code). If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs. Static type checking can be considered a limited form of program verification (see type safety), and in a type-safe language, can also be considered an optimization. If a compiler can prove that a program is well-typed, then it does not need to emit dynamic safety checks, allowing the resulting compiled binary to run faster and to be smaller. Static type checking for Turing-complete languages is inherently conservative. That is, if a type system is both sound (meaning that it rejects all incorrect programs) and decidable (meaning that it is possible to write an algorithm that determines whether a program is well-typed), then it must be incomplete (meaning there are correct programs, which are also rejected, even though they do not encounter runtime errors).[7] For example, consider a program containing the code: if then else Even if the expression always evaluates to true at run-time, most type checkers will reject the program as ill-typed, because it is difficult (if not impossible) for a static analyzer to determine that the else branch will not be taken.[8] Consequently, a static type checker will quickly detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors, because the combination of all places where values are created and all places where a certain value is used must be taken into account. A number of useful and common programming language features cannot be checked statically, such as downcasting. Thus, many languages will have both static and dynamic type checking; the static type checker verifies what it can, and dynamic checks verify the rest. Many languages with static type checking provide a way to bypass the type checker. Some languages allow programmers to choose between static and dynamic type safety. For example, historically C# declares variables statically,[9]:77,Section 3.2 but C# 4.0 introduces the dynamic keyword, which is used to declare variables to be checked dynamically at runtime.[9]:117,Section 4.1 Other languages allow writing code that is not type-safe; for example, in C, programmers can freely cast a value between any two types that have the same size, effectively subverting the type concept. See also: Dynamic programming language, Interpreted language, and Category:Dynamically typed programming languages Dynamic type checking is the process of verifying the type safety of a program at runtime. Implementations of dynamically type-checked languages generally associate each runtime object with a type tag (i.e., a reference to a type) containing its type information. This runtime type information (RTTI) can also be used to implement dynamic dispatch, late binding, downcasting, reflective programming (reflection), and similar features. Most type-safe languages include some form of dynamic type checking, even if they also have a static type checker.[10] The reason for this is that many useful features or properties are difficult or impossible to verify statically. For example, suppose that a program defines two types, A and B, where B is a subtype of A. If the program tries to convert a value of type A to type B, which is known as downcasting, then the operation is legal only if the value being converted is actually a value of type B. Thus, a dynamic check is needed to verify that the operation is safe. This requirement is one of the criticisms of downcasting. By definition, dynamic type checking may cause a program to fail at runtime. In some programming languages, it is possible to anticipate and recover from these failures. In others, type-checking errors are considered fatal. Programming languages that include dynamic type checking but not static type checking are often called "dynamically typed programming languages". "Type hinting" redirects here. For hinting of typefaces, see font hinting. Certain languages allow both static and dynamic typing. For example, Java and some other ostensibly statically typed languages support downcasting types to their subtypes, querying an object to discover its dynamic type and other type operations that depend on runtime type information. Another example is C++ RTTI. More generally, most programming languages include mechanisms for dispatching over different 'kinds' of data, such as disjoint unions, runtime polymorphism, and variant types. Even when not interacting with type annotations or type checking, such mechanisms are materially similar to dynamic typing implementations. See programming language for more discussion of the interactions between static and dynamic typing. Objects in object-oriented languages are usually accessed by a reference whose static target type (or manifest type) is equal to either the object's run-time type (its latent type) or a supertype thereof. This is conformant with the Liskov substitution principle, which states that all operations performed on an instance of a given type can also be performed on an instance of a subtype. This concept is also known as subsumption or subtype polymorphism. In some languages subtypes may also possess covariant or contravariant return types and argument types respectively. Certain languages, for example Clojure, Common Lisp, or Cython are dynamically type checked by default, but allow programs to opt into static type checking by providing optional annotations. One reason to use such hints would be to optimize the performance of critical sections of a program. This is formalized by gradual typing. The programming environment DrRacket, a pedagogic environment based on Lisp, and a precursor of the language Racket is also soft-typed.[11] Conversely, as of version 4.0, the C# language provides a way to indicate that a variable should not be statically type checked. A variable whose type is dynamic will not be subject to static type checking. Instead, the program relies on runtime type information to determine how the variable may be used.[12][9]:113–119 In Rust, the dyn std::any::Any type provides dynamic typing of 'static types.[13] The choice between static and dynamic typing requires certain trade-offs. Static typing can find type errors reliably at compile time, which increases the reliability of the delivered program. However, programmers disagree over how commonly type errors occur, resulting in further disagreements over the proportion of those bugs that are coded that would be caught by appropriately representing the designed types in code.[14][15] Static typing advocates[who?] believe programs are more reliable when they have been well type-checked, whereas dynamic-typing advocates[who?] point to distributed code that has proven reliable and to small bug databases.[citation needed] The value of static typing increases as the strength of the type system is increased. Advocates of dependent typing,[who?] implemented in languages such as Dependent ML and Epigram, have suggested that almost all bugs can be considered type errors, if the types used in a program are properly declared by the programmer or correctly inferred by the compiler.[16] Static typing usually results in compiled code that executes faster. When the compiler knows the exact data types that are in use (which is necessary for static verification, either through declaration or inference) it can produce optimized machine code. Some dynamically typed languages such as Common Lisp allow optional type declarations for optimization for this reason. By contrast, dynamic typing may allow compilers to run faster and interpreters to dynamically load new code, because changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit.[clarification needed] This too may reduce the edit-compile-test-debug cycle. Statically typed languages that lack type inference (such as C and Java prior to version 10) require that programmers declare the types that a method or function must use. This can serve as added program documentation, that is active and dynamic, instead of static. This allows a compiler to prevent it from drifting out of synchrony, and from being ignored by programmers. However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala, OCaml, F#, Swift, and to a lesser extent C# and C++), so explicit type declaration is not a necessary requirement for static typing in all languages. Dynamic typing allows constructs that some (simple) static type checking would reject as illegal. For example, eval functions, which execute arbitrary data as code, become possible. An eval function is possible with static typing, but requires advanced uses of algebraic data types. Further, dynamic typing better accommodates transitional code and prototyping, such as allowing a placeholder data structure (mock object) to be transparently used in place of a full data structure (usually for the purposes of experimentation and testing). Dynamic typing typically allows duck typing (which enables easier code reuse). Many[specify] languages with static typing also feature duck typing or other mechanisms like generic programming that also enable easier code reuse. Dynamic typing typically makes metaprogramming easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code since C++ has stronger rules regarding type definitions (for both functions and variables). This forces a developer to write more boilerplate code for a template than a Python developer would need to. More advanced run-time constructs such as metaclasses and introspection are often harder to use in statically typed languages. In some languages, such features may also be used e.g. to generate new types and behaviors on the fly, based on run-time data. Such advanced constructs are often provided by dynamic programming languages; many of these are dynamically typed, although dynamic typing need not be related to dynamic programming languages. Main article: Strong and weak typing Languages are often colloquially referred to as strongly typed or weakly typed. In fact, there is no universally accepted definition of what these terms mean. In general, there are more precise terms to represent the differences between type systems that lead people to call them "strong" or "weak". Main articles: Type safety and Memory safety A third way of categorizing the type system of a programming language is by the safety of typed operations and conversions. Computer scientists use the term type-safe language to describe languages that do not allow operations or conversions that violate the rules of the type system. Computer scientists use the term memory-safe language (or just safe language) to describe languages that do not allow programs to access memory that has not been assigned for their use. For example, a memory-safe language will check array bounds, or else statically guarantee (i.e., at compile time before execution) that array accesses out of the array boundaries will cause compile-time and perhaps runtime errors. Consider the following program of a language that is both type-safe and memory-safe:[17] var x := 5; var y := "37"; var z := x + y; In this example, the variable z will have the value 42. Although this may not be what the programmer anticipated, it is a well-defined result. If y were a different string, one that could not be converted to a number (e.g. "Hello World"), the result would be well-defined as well. Note that a program can be type-safe or memory-safe and still crash on an invalid operation. This is for languages where the type system is not sufficiently advanced to precisely specify the validity of operations on all possible operands. But if a program encounters an operation that is not type-safe, terminating the program is often the only option. Now consider a similar example in C: int x = 5; char y[] = "37"; char* z = x + y; printf("%c", *z); In this example z will point to a memory address five characters beyond y, equivalent to three characters after the terminating zero character of the string pointed to by y. This is memory that the program is not expected to access. In C terms this is simply undefined behaviour and the program may do anything; with a simple compiler it might actually print whatever byte is stored after the string "37". As this example shows, C is not memory-safe. As arbitrary data was assumed to be a character, it is also not a type-safe language. In general, type-safety and memory-safety go hand in hand. For example, a language that supports pointer arithmetic and number-to-pointer conversions (like C) is neither memory-safe nor type-safe, because it allows arbitrary memory to be accessed as if it were valid memory of any type. Some languages allow different levels of checking to apply to different regions of code. Examples include: The use strict directive in JavaScript[18][19][20] and Perl applies stronger checking. The declare(strict_types=1) in PHP[21] on a per-file basis allows only a variable of exact type of the type declaration will be accepted, or a TypeError will be thrown. The Option Strict On in VB.NET allows the compiler to require a conversion between objects. Additional tools such as lint and IBM Rational Purify can also be used to achieve a higher level of strictness. It has been proposed, chiefly by Gilad Bracha, that the choice of type system be made independent of choice of language; that a type system should be a module that can be plugged into a language as needed. He believes this is advantageous, because what he calls mandatory type systems make languages less expressive and code more fragile.[22] The requirement that the type system does not affect the semantics of the language is difficult to fulfill. Optional typing is related to, but distinct from, gradual typing. While both typing disciplines can be used to perform static analysis of code (static typing), optional type systems do not enforce type safety at runtime (dynamic typing).[22][23] Main article: Polymorphism (computer science) The term polymorphism refers to the ability of code (especially, functions or classes) to act on values of multiple types, or to the ability of different instances of the same data structure to contain elements of different types. Type systems that allow polymorphism generally do so in order to improve the potential for code re-use: in a language with polymorphism, programmers need only implement a data structure such as a list or an associative array once, rather than once for each type of element with which they plan to use it. For this reason computer scientists sometimes call the use of certain forms of polymorphism generic programming. The type-theoretic foundations of polymorphism are closely related to those of abstraction, modularity and (in some cases) subtyping. Many type systems have been created that are specialized for use in certain environments with certain types of data, or for out-of-band static program analysis. Frequently, these are based on ideas from formal type theory and are only available as part of prototype research systems. The following table gives an overview over type theoretic concepts that are used in specialized type systems. The names M, N, O range over terms and the names σ , τ {\displaystyle \sigma ,\tau } range over types. The following notation will be used: M : σ {\displaystyle M:\sigma } means that M {\displaystyle M} has type σ {\displaystyle \sigma } ; M (N) {\displaystyle M(N)} is that application of M {\displaystyle M} on N {\displaystyle N} ; τ [α := σ] {\displaystyle \tau [\alpha :=\sigma]} (resp. τ [x := N] {\displaystyle \tau [x:=N]}) describes the type which results from replacing all occurrences of the type variable α (resp. term variable x) in τ {\displaystyle \tau } by the type σ (resp. term N). Type notion Notation Meaning Function σ → τ {\displaystyle \sigma \to \tau } If M : σ → τ {\displaystyle M:\sigma \to \tau } and N : σ {\displaystyle N:\sigma } , then M (N) : τ {\displaystyle M(N):\tau } . Product σ × τ {\displaystyle \sigma \times \tau } If M : σ × τ {\displaystyle M:\sigma \times \tau } , then M = (N , O) {\displaystyle M=(N,O)} is a pair s.t. N : σ {\displaystyle N:\sigma } and O : τ {\displaystyle O:\tau } . Sum σ + τ {\displaystyle \sigma +\tau } If M : σ + τ {\displaystyle M:\sigma +\tau } , then M = ι 1 (N) {\displaystyle M=\iota _{1}(N)} is the first injection s.t. N : σ {\displaystyle N:\sigma } , or M = ι 2 (N) {\displaystyle M=\iota _{2}(N)} is the second injection s.t. N : τ {\displaystyle N:\tau } . Intersection σ ∩ τ {\displaystyle \sigma \cap \tau } If M : σ ∩ τ {\displaystyle M:\sigma \cap \tau } , then M : σ {\displaystyle M:\sigma } and M : τ {\displaystyle M:\tau } . Union σ ∪ τ {\displaystyle \sigma \cup \tau } If M : σ ∪ τ {\displaystyle M:\sigma \cup \tau } , then M : σ {\displaystyle M:\sigma } or M : τ {\displaystyle M:\tau } . Record ⟨ x : τ ⟩ {\displaystyle \langle x:\tau \rangle } If M : ⟨ x : τ ⟩ {\displaystyle M:\langle x:\tau \rangle } , then M has a member x : τ {\displaystyle x:\tau } . Polymorphic ∀ α . τ {\displaystyle \forall {}\alpha .\tau } If M : ∀ α . τ {\displaystyle M:\forall {}\alpha .\tau } , then M : τ [α := σ] {\displaystyle M:\tau [\alpha :=\sigma]} for any type σ. Existential ∃ α . τ {\displaystyle \exists {}\alpha .\tau } If M : ∃ α . τ {\displaystyle M:\exists {}\alpha .\tau } , then M : τ [α := σ] {\displaystyle M:\tau [\alpha :=\sigma]} for some type σ. Recursive μ α . τ {\displaystyle \mu \alpha .\tau } If M : μ α . τ {\displaystyle M:\mu \alpha .\tau } , then M : τ [α := μ α . τ] {\displaystyle M:\tau [\alpha :=\mu \alpha .\tau]} . Dependent function[a] (x : σ) → τ {\displaystyle (x:\sigma)\to \tau } If M : (x : σ) → τ {\displaystyle M:(x:\sigma)\to \tau } and N : σ {\displaystyle N:\sigma } , then M (N) : τ [x := N] {\displaystyle M(N):\tau [x:=N]} . Dependent pair[b] (x : σ) × τ {\displaystyle (x:\sigma)\times \tau } If M : (x : σ) × τ {\displaystyle M:(x:\sigma)\times \tau } , then M = (N , O) {\displaystyle M=(N,O)} is a pair s.t. N : σ {\displaystyle N:\sigma } and O : τ [x := N] {\displaystyle O:\tau [x:=N]} . Dependent intersection[24] (x : σ) ∩ τ {\displaystyle (x:\sigma)\cap \tau } If M : (x : σ) ∩ τ {\displaystyle M:(x:\sigma)\cap \tau } , then M : σ {\displaystyle M:\sigma } and M : τ [x := M] {\displaystyle M:\tau [x:=M]} . Familial intersection[24] ⋂ x : σ τ {\displaystyle \bigcap _{x:\sigma }\tau } If M : ⋂ x : σ τ {\textstyle M:\bigcap _{x:\sigma }\tau } , then M : τ [x := N] {\displaystyle M:\tau [x:=N]} for any term N : σ {\displaystyle N:\sigma } . Familial union[24] ⋃ x : σ τ {\displaystyle \bigcup _{x:\sigma }\tau } If M : ⋃ x : σ τ {\textstyle M:\bigcup _{x:\sigma }\tau } , then M : τ [x := N] {\displaystyle M:\tau [x:=N]} for some term N : σ {\displaystyle N:\sigma } . ^ Also referred to as dependent product type, since (x : σ) → τ = ∏ x : σ τ {\textstyle (x:\sigma)\to \tau =\prod _{x:\sigma }\tau } . ^ Also referred to as dependent sum type, since (x : σ) × τ = ∑ x : σ τ {\textstyle (x:\sigma)\times \tau =\sum _{x:\sigma }\tau } . Main article: Dependent type Dependent types are based on the idea of using scalars or values to more precisely describe the type of some other value. For example, m a t r i x (3 , 3) {\displaystyle \mathrm {matrix} (3,3)} might be the type of a 3 × 3 {\displaystyle 3\times 3} matrix. We can then define typing rules such as the following rule for matrix multiplication: m a t r i x m u l t i p l y : m a t r i x (k , m) × m a t r i x (m , n) → m a t r i x (k , n) {\displaystyle \mathrm {matrix} _{\mathrm {multiply} }:\mathrm {matrix} (k,m)\times \mathrm {matrix} (m,n)\to \mathrm {matrix} (k,n)} where k, m, n are arbitrary positive integer values. A variant of ML called Dependent ML has been created based on this type system, but because type checking for conventional dependent types is undecidable, not all programs using them can be type-checked without some kind of limits. Dependent ML limits the sort of equality it can decide to Presburger arithmetic. Other languages such as Epigram make the value of all expressions in the language decidable so that type checking can be decidable. However, in general proof of decidability is undecidable, so many programs require hand-written annotations that may be very non-trivial. As this impedes the development process, many language implementations provide an easy way out in the form of an option to disable this condition. This, however, comes at the cost of making the type-checker run in an infinite loop when fed programs that do not type-check, causing the compilation to fail. Main article: Linear type Linear types, based on the theory of linear logic, and closely related to uniqueness types, are types assigned to values having the property that they have one and only one reference to them at all times. These are valuable for describing large immutable values such as files, strings, and so on, because any operation that simultaneously destroys a linear object and creates a similar object (such as str = str + "a") can be optimized "under the hood" into an in-place mutation. Normally this is not possible, as such mutations could cause side effects on parts of the program holding other references to the object, violating referential transparency. They are also used in the prototype operating system Singularity for interprocess communication, statically ensuring that processes cannot share objects in shared memory in order to prevent race conditions. The Clean language (a Haskell-like language) uses this type system in order to gain a lot of speed (compared to performing a deep copy) while remaining safe. Main article: Intersection type Intersection types are types describing values that belong to both of two other given types with overlapping value sets. For example, in most implementations of C the

signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting either signed or unsigned chars, because it is compatible with both types. Intersection types are useful for describing overloaded function types: for example, if "int → int" is the type of functions taking an integer argument and returning an integer, and "float → float" is the type of functions taking a float argument and returning a float, then the intersection of these two types can be used to describe functions that do one or the other, based on what type of input they are given. Such a function could be passed into another function expecting an "int → int" function safely; it simply would not use the "float → float" functionality. In a subclassing hierarchy, the intersection of a type and an ancestor type (such as its parent) is the most derived type. The intersection of sibling types is empty. The Forsythe language includes a general implementation of intersection types. A restricted form is refinement types. Main article: Union type Union types are types describing values that belong to either of two types. For example, in C, the signed char has a -128 to 127 range, and the unsigned char has a 0 to 255 range, so the union of these two types would have an overall "virtual" range of -128 to 255 that may be used partially depending on which union member is accessed. Any function handling this union type would have to deal with integers in this complete range. More generally, the only valid operations on a union type are operations that are valid on both types being unioned. C's "union" concept is similar to union types, but is not typesafe, as it permits operations that are valid on either type, rather than both. Union types are important in program analysis, where they are used to represent symbolic values whose exact nature (e.g., value or type) is not known. In a subclassing hierarchy, the union of a type and an ancestor type (such as its parent) is the ancestor type. The union of sibling types is a subtype of their common ancestor (that is, all operations permitted on their common ancestor are permitted on the union type, but they may also have other valid operations in common). Main article: Existential quantifier Existential types are frequently used in connection with record types to represent modules and abstract data types, due to their ability to separate implementation from interface. For example, the type "T = ∃X { a: X; f: (X → int); }" describes a module interface that has a data member named a of type X and a function named f that takes a parameter of the same type X and returns an integer. This could be implemented in different ways; for example: intT = { a: int; f: (int → int); } floatT = { a: float; f: (float → int); } These types are both subtypes of the more general existential type T and correspond to concrete implementation types, so any value of one of these types is a value of type T. Given a value "t" of type "T", we know that "t.f(t.a)" is well-typed, regardless of what the abstract type X is. This gives flexibility for choosing types suited to a particular implementation, while clients that use only values of the interface type—the existential type—are isolated from these choices. In general it's impossible for the typechecker to infer which existential type a given module belongs to. In the above example intT { a: int; f: (int → int); } could also have the type ∃X { a: X; f: (int → int); }. The simplest solution is to annotate every module with its intended type, e.g.: intT = { a: int; f: (int → int); } as ∃X { a: X; f: (X → int); } Although abstract data types and modules had been implemented in programming languages for quite some time, it wasn't until 1988 that John C. Mitchell and Gordon Plotkin established the formal theory under the slogan: "Abstract [data] types have existential type".[25] The theory is a second-order typed lambda calculus similar to System F, but with existential instead of universal quantification. Main article: Gradual typing In a type system with Gradual typing, variables may be assigned a type either at compile-time (which is static typing), or at run-time (which is dynamic typing).[26] This allows software developers to choose either type paradigm as appropriate, from within a single language.[26] Gradual typing uses a special type named dynamic to represent statically unknown types; gradual typing replaces the notion of type equality with a new relation called consistency that relates the dynamic type to every other type. The consistency relation is symmetric but not transitive.[27] Further information: Type inference Many static type systems, such as those of C and Java, require type declarations: the programmer must explicitly associate each variable with a specific type. Others, such as Haskell's, perform type inference: the compiler draws conclusions about the types of variables based on how programmers use those variables. For example, given a function f(x, y) that adds x and y together, the compiler can infer that x and y must be numbers—since addition is only defined for numbers. Thus, any call to f elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error. Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression 3.14 might imply a type of floating-point, while [1, 2, 3] might imply a list of integers—typically an array. Type inference is in general possible, if it is computable in the type system in question. Moreover, even if inference is not computable in general for a given type system, inference is often possible for a large subset of real-world programs. Haskell's type system, a version of Hindley–Milner, is a restriction of System Fω to so-called rank-1 polymorphic types, in which type inference is computable. Most Haskell compilers allow arbitrary-rank polymorphism as an extension, but this makes type inference not computable. (Type checking is decidable, however, and rank-1 programs still have type inference; higher rank polymorphic programs are rejected unless given explicit type annotations.) Main article: Type theory § Decision problems A type system that assigns types to terms in type environments using typing rules is naturally associated with the decision problems of type checking, typability, and type inhabitation.[28] Given a type environment Γ , a term e , and a type τ , decide whether the term e is assigned the type τ in the type environment. Given a term e , decide whether there exists a type environment Γ and a type τ such that the term e is assigned the type τ in the type environment Γ . Given a type environment Γ and a type τ , decide whether there exists a term e that can be assigned the type τ in the type environment. Some languages like C# or Scala have a unified type system.[29] This means that all C# types including primitive types inherit from a single root object. Every type in C# inherits from the Object class. Some languages, like Java and Raku, have a root type but also have primitive types that are not objects.[30] Java provides wrapper object types that exist together with the primitive types so developers can use either the wrapper object types or the simpler non-object primitive types. Raku automatically converts primitive types to objects when their methods are accessed.[31] A type checker for a statically typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. For example, in an assignment statement of the form x := e, the inferred type of the expression e must be consistent with the declared or inferred type of the variable x. This notion of consistency, called compatibility, is specific to each programming language. If the type of e and the type of x are the same, and assignment is allowed for that type, then this is a valid expression. Thus, in the simplest type systems, the question of whether two types are compatible reduces to that of whether they are equal (or equivalent). Different languages, however, have different criteria for when two type expressions are understood to denote the same type. These different equational theories of types vary widely, two extreme cases being structural type systems, in which any two types that describe values with the same structure are equivalent, and nominative type systems, in which no two syntactically distinct type expressions denote the same type (i.e., types must have the same "name" in order to be equal). In languages with subtyping, the compatibility relation is more complex: If B is a subtype of A, then a value of type B can be used in a context where one of type A is expected (covariant), even if the reverse is not true. Like equivalence, the subtype relation is defined differently for each programming language, with many variations possible. The presence of parametric or ad hoc polymorphism in a language may also have implications for type compatibility. Computer programming portal Comparison of type systems Covariance and contravariance (computer science) Polymorphism in object-oriented programming Type signature Type theory ^ The Burroughs ALGOL computer line determined a memory location's contents by its flag bits. Flag bits specify the contents of a memory location. Instruction, data type, and functions are specified by a 3 bit code in addition to its 48 bit contents. Only the MCP (Master Control Program) could write to the flag code bits. ^ For example, a leaky abstraction might surface during development, which may show that more type development is needed. —"The evaluation of a well-typed program always terminates."—B. Nordström, K. Petersson, and J. M. Smith[5] A systematic change in variables to avoid capture of a free variable can introduce error, in a functional programming language where functions are first class citizens.[6] —From the lambda calculus article. ^ Pierce 2002, p. 1: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." ^ Cardelli 2004, p. 1: "The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program." ^ Pierce 2002, p. 208. ^ a b Sethi, R. (1996). Programming languages: Concepts and constructs (2nd ed.). Addison-Wesley. p. 142. ISBN 978-0-201-59065-4. OCLC 604732680. ^ Nordström, B.; Petersson, K.; Smith, J.M. (2001). "Martin-Löf's Type Theory". Algebraic and Logical Structures. Handbook of Logic in Computer Science. Vol. 5. Oxford University Press. p. 2. ISBN 978-0-19-154627-3. ^ Turner, D.A. (12 June 2012). "Some History of Functional Programming Languages" (PDF). invited lecture at TFP12, at St Andrews University. See the section on Algol 60. ^ "... any sound, decidable type system must be incomplete" —D. Remy (2017). p. 29, Remy, Didier. "Type systems for programming languages" (PDF). Archived from the original (PDF) on 14 November 2017. Retrieved 26 May 2013. ^ Pierce 2002. ^ a b c Skeet, Jon (2019). C# in Depth (4 ed.). Manning. ISBN 978-1617294532. ^ Miglani, Gaurav (2018). "Dynamic Method Dispatch or Runtime Polymorphism in Java". Archived from the original on 2020-12-07. Retrieved 2021-03-28. ^ Wright, Andrew K. (1995). Practical Soft Typing (PhD). Rice University. hdl:1911/16900. ^ "dynamic (C# Reference)". MSDN Library. Microsoft. Retrieved 14 January 2014. ^ "std::any — Rust". doc.rust-lang.org. Retrieved 2021-07-07. ^ Meijer, Erik; Drayton, Peter. "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages" (PDF). Microsoft Corporation. ^ Laucher, Amanda; Snively, Paul (2012). "Types vs Tests". InfoQ. ^ Xi, Hongwei (1998). Dependent Types in Practical Programming (PhD). Department of Mathematical Sciences, Carnegie Mellon University. CiteSeerX 10.1.1.41.548.Xi, Hongwei; Pfenning, Frank (1999). "Dependent Types in Practical Programming". Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM. pp. 214–227. CiteSeerX 10.1.1.69.2042. doi:10.1145/292540.292560. ISBN 1581130953. S2CID 245490. ^ Visual Basic is an example of a language that is both type-safe and memory-safe. ^ "4.2.2 The Strict Variant of ECMAScript". ECMAScript® 2020 Language Specification (11th ed.). ECMA. June 2020. ECMA-262. ^ "Strict mode – JavaScript". MDN. Developer.mozilla.org. 2013-07-03. Retrieved 2013-07-17. ^ "Strict Mode (JavaScript)". MSDN. Microsoft. Retrieved 2013-07-17. ^ "Strict typing". PHP Manual: Language Reference: Functions. ^ a b Bracha, G. "Pluggable Types" (PDF). ^ "Sure. It's called "gradual typing", and I would qualify it as trendy. ..." Is there a language that allows both static and dynamic typing?. stackoverflow. 2012. ^ a b c Kopylov, Alexei (2003). "Dependent intersection: A new way of defining records in type theory". 18th IEEE Symposium on Logic in Computer Science. LICS 2003. IEEE Computer Society. pp. 86–95. CiteSeerX 10.1.1.89.4223. doi:10.1109/LICS.2003.1210048. ^ Mitchell, John C.; Plotkin, Gordon D. (July 1988). "Abstract Types Have Existential Type" (PDF). ACM Trans. Program. Lang. Syst. 10 (3): 470–502. doi:10.1145/44501.45065. S2CID 1222153. ^ a b Siek, Jeremy (24 March 2014). "What is gradual typing?". ^ Siek, Jeremy; Taha, Walid (September 2006). Gradual Typing for Functional Languages (PDF). Scheme and Functional Programming 2006. University of Chicago. pp. 81–92. ^ Barendregt, Henk; Dekkers, Wil; Statman, Richard (20 June 2013). Lambda Calculus with Types. Cambridge University Press. p. 66. ISBN 978-0-521-76614-2. ^ "8.2.4 Type system unification". C# Language Specification (5th ed.). ECMA. December 2017. ECMA-334. ^ "Native Types". Perl 6 Documentation. ^ "Numerics, § Auto-boxing". Perl 6 Documentation. Cardelli, Luca; Wegner, Peter (December 1985). "On Understanding Types, Data Abstraction, and Polymorphism" (PDF). ACM Computing Surveys. 17 (4): 471–523. CiteSeerX 10.1.1.117.695. doi:10.1145/6041.6042. S2CID 2921816. Pierce, Benjamin C. (2002). Types and Programming Languages. MIT Press. ISBN 978-0-262-16209-8. Cardelli, Luca (2004). "Type systems" (PDF). In Allen B. Tucker (ed.). CRC Handbook of Computer Science and Engineering (2nd ed.). CRC Press. ISBN 978-1584883609. Tratt, Laurence (July 2009). "5. Dynamically Typed Languages". Advances in Computers. Vol. 77. Elsevier. pp. 149–184. doi:10.1016/S0065-2458(09)01205-4. ISBN 978-0-12-374812-6. The Wikibook Ada Programming has a page on the topic of: Types The Wikibook Haskell has a page on the topic of: Class declarations Media related to Type systems at Wikimedia Commons Smith, Chris (2011). "What to Know Before Debating Type Systems". Retrieved from "