Lectures Notes on Secure and Dependable Systems

Florian Rabe

2017

Contents

Part I

Introduction

Chapter 1

Meta-Remarks

Important stuff that you should read carefully!

State of these notes I constantly work on my lecture notes. Therefore, keep in mind that:

- I am developing these notes in parallel with the lecture—they can grow or change throughout the semester.
- These notes are neither a subset nor a superset of the material discussed in the lecture.
- Unless mentioned otherwise, all material in these notes is exam-relevant (in addition to all material discussed in the lectures).

Collaboration on these notes I am writing these notes using LaTeX and storing them in a git repository on GitHub at https://github.com/florian-rabe/Teaching. Familiarity with LaTeX as well as Git and GitHub is not part of this lecture. But it is essential skill for you. Ask in the lecture if you have difficulty figuring it out on your own.

As an experiment in teaching, I am inviting all of you to collaborate on these lecture notes with me.

By forking and by submitting pull requests for this repository, you can suggest changes to these notes. For example, you are encouraged to:

- Fix typos and other errors.
- Add examples and diagrams that I develop on the board during lectures.
- Add solutions for the homeworks if I did not provide any (of course, I will only integrate solutions after the deadline).
- Add additional examples, exercises, or explanations that you came up or found in other sources. If you use material from other sources (e.g., by copying an diagram from some website), make sure that you have the license to use it and that you acknowledge sources appropriately!

The TAs and I will review and approve or reject the changes. If you make substantial contributions, I will list you as a contributor (i.e., something you can put in your CV).

Any improvement you make will not only help your fellow students, it will also increase your own understanding of the material. Therefore, I can give you up to 10% bonus credit for such contributions. (Make sure your git commits carry a user name that I can connect to you.) Because this is an experiment, I will have to figure out the details along the way.

Other Advice I maintain a list of useful advice for students at https://svn.kwarc.info/repos/frabe/Teaching/general/advice_for_students.pdf. It is mostly targeted at older students who work in individual projects with me (e.g., students who work on their BSc thesis). But much of it is useful for you already now or will become useful soon. So have a look.

Chapter 2

Concepts

This section mostly follows [ALR01], including some tables, which is available online.

Computer systems can be evaluated according to

- functionality
- usability
- performance
- cost
- dependability

Dependability means to deliver service to a user that can justifiably be trusted. The user is another system (physical or human) that interacts with the former at the service interface. The **function** of a system is what the system is intended to do as described by its functional specification. Correct service is delivered when the service implements the specification.

A systematic exposition of the concepts of dependability consists of three parts: the attributes of dependability, their threats, and means to achieve dependability. An overview is given in Fig. 2.1.

2.1 Attributes

2.1.1 Definitions

Availability means the readiness for correct service. This includes two subaspects: the functionality has to **exist** and be realized **correctly**, i.e., according to its specification.

Reliability means the continuity of correct service. This is similar to availability but emphasizes that the service not only exists but exist without downtime or intermittent failures.

Safety means the absence of catastrophic consequences on the user(s) and the environment. Often safety involves interpreting signals received from and sending signals to external devices that operate in the real world, e.g., the cameras and the engine of the car. This introduces additional uncertainty (not to mention the other cars and pedestrians) that can be difficult to anticipate in the specification.

Security means the availability for authorized users only. This includes protection against any malicious influenced from the outside, i.e., any kind of attack or hacking. This includes all defenses against hacking.[1]

Confidentiality means the absence of unauthorized disclosure of information. This includes the security of all private data including any intermediate results of computation such as passwords or keys.

Maintainability means the ability to undergo repairs and modifications. This includes all necessary changes needed during long-term deployment.

2.1.2 Incomplete Specifications

Often the attributes are not or only implicitly part of the specification of a system.

[1] [ALR01] calls a related property *integrity*, and then defines security as the combination of integrity and confidentiality.

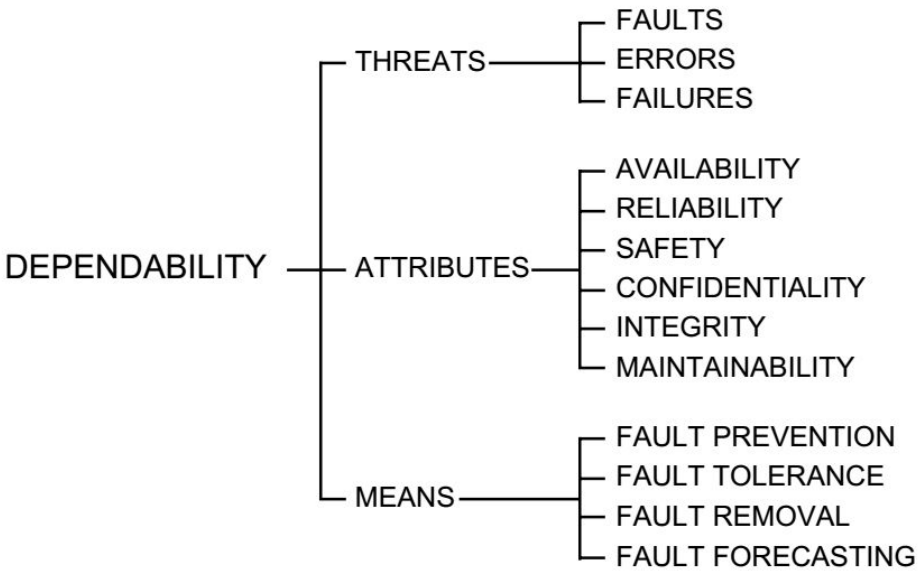


Figure 2.1: Concepts Related to Dependability

Availability is usually implicitly required, but details may be omitted. For example, the acceptable variation of response times may be unspecified.

Reliability is also usually implicit required, but details may be omitted. For example, the acceptable downtime may be unspecified.

Safety is usually specified well if a potential safety danger is realized. But it can be easy to foresee all necessary safety requirements.

Security is often forgotten completely or partially. It is usually difficult to translate the abstract requirement of security into concrete, testable properties. After all, the first mistake of security is to assume to know what the attacker might do.

Confidentiality is often considered even less then security.

Maintainability is often ignored completely. That is a typical pitfall for large projects, where realizing any requirement at deploy time may be completely different from realizing it after a year or later. This is because intermediate changes have messed up the system so much that, e.g., security flaws are not noticed anymore.

2.1.3 Imperfect Implementations

All attributes are very difficult to realize perfectly in implementations.

Availability and reliability often fail. In addition to plain design or implementation errors, there may be failures in hardware, networks, operating system, external components that were not foreseen by the developer.

Together with correctness, safety is the only property that is at least in principle accessible to a formal definition. But the resulting problem is undecidable. So in practice, we have to use extensive testing.

Security is very complex to prove because any proof must make assumptions about what kind of attacks there are. Attacking a system often requires intentionally violating the specification and supply unanticipated input.

Perfect confidentiality is impossible to realize because all computation leaks some information other than the output: This reaches from runtime and resource use to obscure effects like the development of heat due to CPU activity.

Maintainability is hard to realize because especially inexperienced developers or unskilled managers cannot assess whether a particular design is maintainable.

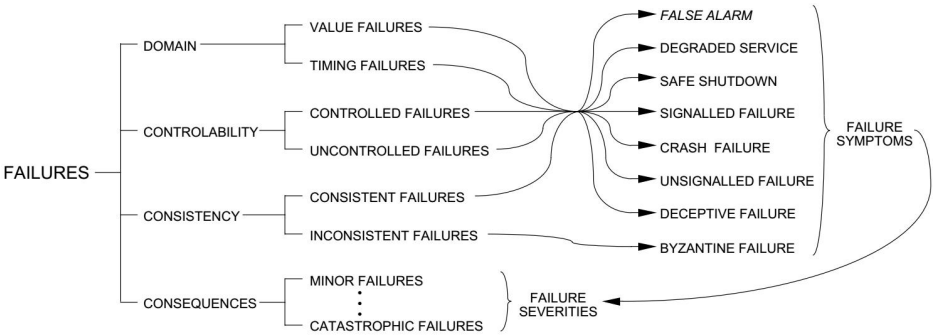


Figure 2.2: Failure Modes

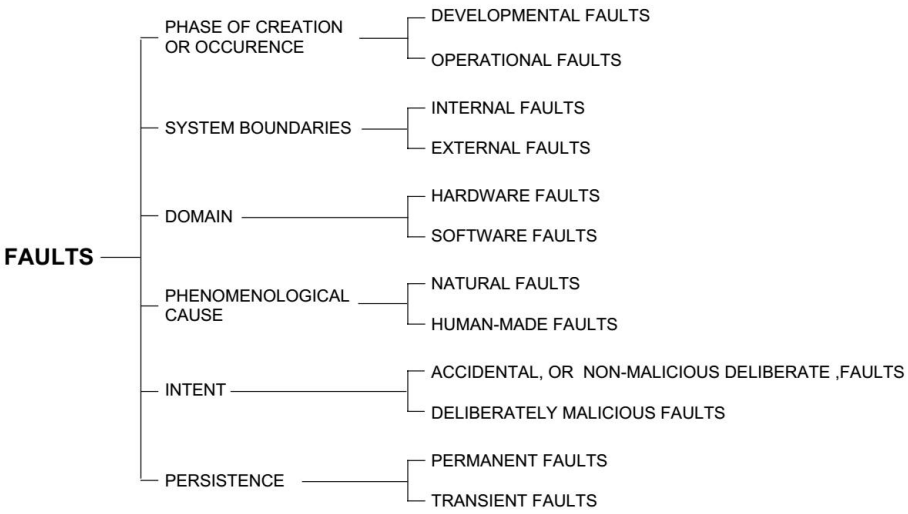


Figure 2.3: Fault Classes

2.2 Threats

2.2.1 Failures

A system **failure** is an event that occurs when the delivered service deviates from correct service. See Fig. 2.2 for an overview of related concepts.

A **fault** is the adjudged or hypothesized cause of an error. A fault is **active** when it produces an error; otherwise it is **dormant**. See Fig. 2.3 for an overview of related concepts.

An **error** is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service.

2.3 Means

2.3.1 Fault Prevention

Fault prevention requires quality control techniques employed during the design and manufacturing of the system. That includes, e.g., structured programming, information hiding, or modularization.

Shielding, radiation hardening, etc., are needed to prevent physical faults. Training and maintenance procedures aim at preventing interaction faults. Firewalls and similar defenses intend to prevent malicious faults.

2.3.2 Fault Tolerance

Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. This may refer to accidental or malicious faults.

Error detection An error that is present but not detected is a **latent** error. Concurrent error detection takes place during service delivery. Preemptive error detection takes place while service delivery is suspended; it checks the system for latent errors and dormant faults.

Recovery Recovery consists of error handling and fault handling.

Error handling eliminates errors from the system state by

- rollback: the state is transformed to a previously saved state
- compensation: the erroneous state is redundant enough to eliminate the error,
- rollforward: the state is transformed to a new state without the detected error.

Fault handling prevents located faults from being activated again by

- fault diagnosis: identify location and type of the cause of an error
- fault isolation: exclude the fault from future service delivery, i.e., make the fault dormant
- system reconfiguration: switch to non-failed components
- system reinitialization: update to a new configuration of the system

The choice of error detection, error handling, and fault handling techniques, and of their implementation, is directly related to the underlying fault assumption.

Fault tolerance is recursive: Its implementation must be protected against faults as well.

2.3.3 Fault Removal

Development Phase Fault removal during the development phase of a system consists of three steps:

- Verification checks whether the system satisfies required properties.
- If verification fails, diagnosis identifies the faults that prevented verification.
- After diagnosis, correction is carried, and verification repeated.

Multiple different properties can be verified:

- Verifying the specification is usually referred to as validation. Static verification does not exercise the systems and uses static analysis (e.g., inspections or walk-through), model-checking, or theorem proving. Dynamic verification exercises the systems and uses testing.
- Verifying the fault tolerance mechanism. This can employ formal static verification or fault injection, i.e., testing where intentional faults or errors are part of the test.
- Verifying that the system cannot do more than specified is especially relevant for safety and security.

Design for verifiability means to design a system in such a way that verification becomes easy.

Operation Phase Fault removal during the life time of a system employs two methods:

- Corrective maintenance removes faults that have produced errors that were detected and reported.
- Preventive maintenance uncovers faults before they cause errors. This may also include design faults that have caused errors in similar systems.

Fault removal during operation often first isolates the fault (e.g., by a workaround or patch) before the actual removal is carried out.

2.3.4 Fault Forecasting

Fault forecasting evaluates a system with respect to fault occurrence or activation. It has two aspects:

- Qualitative evaluation identifies, classifies, and ranks the failure modes, or the event combinations that would lead to system failures.
- Quantitative evaluation determines the probabilities to which some of the attributes of dependability are satisfied, which are then viewed as measures of dependability. This may use, e.g., Markov chains or Petri nets.

Chapter 3

Challenges

This chapter lists examples of disasters and failures that serve as examples of what secure and dependable systems should avoid.

The lists are not complete and may be biased by whether

- I became aware of it and found it interesting enough
- the cause could be determined and was made public

Feel free to edit these notes by adding important examples that I forgot when I compiled the lists.

All damage estimates are relative to the time of the event and not adjusted to inflation.

Note that for security problems, the size of the damage is naturally unknown because attacks will typically remain secret. Only the cost of updating the systems can be estimated, which may or may not be indicative of the severity of the security problem.

3.1 General Aspects

State-of-the-art software and hardware systems simply are not safe, secure, and dependable. Moreover, we do not understand very well yet how to make them so.

This is different from many other areas such as mechanical or chemical engineering. While these occasionally cause disasters, these can usually be traced back to human error, foul play, or negligent or intentional violation of regulations. Such disasters usually result in criminal proceedings, civil litigation, or revision or extension of regulations.

The situation is very different for computer systems. There is no general methodology for designing and operating computer systems well that can be easily described, taught, or codified.

The situation will hopefully improve over the course of the 21st century. The problem has been recognized decades ago, and many companies and researchers are working on it. They approach from very different directions with different goals and different methodologies.

This has resulted in a wide and diverse variety of not coherently connected methods with varying degrees of depth, maturity, cost, benefit, and practical adoption.

A typical effect is a trade-off along a spectrum of methods:

- cheap but weak methods on one end
- strong but expensive methods on the other end.

Therefore, it is often necessary to choose a degree of safety assurance rather than actually guarantee safety. This spectrum is so extreme that

- the majority of practical software development does not systematically ensure any kind of safety,
- the majority of theoretical solutions are neither ready nor affordable for practical use.

Incidentally, this means that this course's subject matter is much less well-defined than that of other courses.[1] That makes it particular difficult to design a syllabus for. It will give an overview of the most important state-of-the-art

[1] For example, the other two courses in this module almost design themselves because the subject matter is very well understood and standardized.

methods.

3.2 Major Disasters Caused by Programming Errors

Space Exploration There have been a number of failures in space exploration due to minor programming errors. These include

- 1962, Mariner 1 rocket lost: misread specification (overlooked bar over a variable) was implemented (damage around $20 million)
- 1982, Viking I lost: software update written to wrong memory area overriding vital parameters for antenna
- 1988, Phobos 1 lost: one character missing in software update led to accidentally executing a testing routing at the wrong time
- 1996, Mars Global Surveyor lost: data written to wrong memory addresses
- 1999, Mars Polar Lander lost: presumably software not accounting for false positive when detecting shutdown even though the possibility was known
- 2004, Mars Rover Spirit lost for 16 days: delay in deleting obsolete files led to lack of available flash memory, which triggered a reboot, which led to a reboot cycle

Therac-25 Between 1985 and 1987, the Therac-25 machine for medical radiation therapy caused death and/or serious injury in at least 6 cases. Patients received a radiation overdose because the high intensity energy beam was administered while using the protection meant for the low intensity beam.

The cause was that the hardware protection was discontinued, relying exclusively on software to prevent a mismatch of beam and protection configuration. But the software had always been buggy due to a systemic failures in the software engineering process including complex systems (code written in assembly, machine had its own OS), lack of software review, insufficient testing (overall system could not be tested), bad documentation (error codes were not documented), and bad user interface (critical safety errors could be manually overridden, thus effectively being warnings).

Details: https://en.wikipedia.org/wiki/Therac-25

Patriot Rounding Error In 1991 during the Gulf war, a US Patriot anti-missile battery failed to track an incoming Iraqi Scud missile resulting the death of 28 people.

The cause was a rounding error in the floating point computation used for analyzing the missile's path. The software had to divide a large integer (number of $0.1s$ clock cycles since boot 100 hours ago) by 10 to obtain the time. This was done using a floating-point multiplication by 0.1—but 0.1 is off by around 0.000000095 when chopped to a 24-bits binary float. The resulting time was off by 0.3 seconds, which combined with the high speed of Scud missile led to a serious miscalculation of the flight path.

Details: http://www-users.math.umn.edu/~arnold/disasters/patriot.html

Ariane 5 In 1996, the first launch of an Ariane 5 rocket (at a cost of over $300 million for rocket and payload) failed, and the rocket had to be destructed after launch. Both the primary and the backup system had shut down, each trying to transfer control to the other, after encountering the same behavior, which they interpreted as a hardware error.

The cause was an overflow exception in the alignment system caused by converting a 64-bit float to a 16-bit integer, which was not caught and resulted in the display of diagnostic data that the autopilot could not interpret. The programmers were aware of the problem but had falsely concluded that no conversion check was needed (and therefore omitted the check to speed up processing). Their conclusion had been made based on Ariane 4 flight data that turned out to be inappropriate for Ariane 5.

The faulty component was not even needed for flight and was only kept active for a brief time after launch for convenience and in order to avoid changing a running system.

Details: http://www-users.math.umn.edu/~arnold/disasters/ariane5rep.html

Intel Pentium Bug In 1994, it was discovered that the Intel Pentium processor (at the time widely used in desktop computers) wrongly computed certain floating point divisions. The cost of replacing the CPUs was estimated at about $400 million.

The error occurred in about 1 in 9 billion divisions. For example, 4195835.0/3145727.0 yielded 1.333739068902037589 instead of 1.333820449136241000.

The cause was a bug in the design of the floating point unit's circuit.

Kerberos Random Number Generator From 1988 to 1996, the network authentification protocol Kerberos used a mis-designed random number generation algorithm. The resulting keys were so predictable that brute force attacks became trivial although it is unclear if the bug was ever exploited.

The cause was the lack of a truly random seed value for the algorithm. Moreover, the error persisted across attempted fixes because of process failures (code hard to read, programmers had moved on to next version).

Detail: http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2331&context=cstech

USS Yorktown In 1997, critical navigation and weapons hardware on the USS Yorktown was paralyzed at sea for 3 hours while rebooting machines.

The cause was a blank field in a database that was interpreted as 0 leading to a division-by-zero. Special floating point values such as infinity or NaN were not used, thus resulting in an exception. The exception was handled by neither the software nor the operating systems (Windows NT) thus crashing both.

Details: http://www.cs.berkeley.edu/~wkahan/Boulder.pdf

Mars Climate Orbiter In 1998 the Mars Climate Orbiter was lost causing damage of around $300 million after software had calculated a false trajectory when updating the position of the spacecraft.

The cause was that two components by different manufacturers exchanged physical quantities as plain numbers (i.e., without units). One component assumed customary units (pound seconds) whereas the other assumed SI units (Newton seconds). The first component was in violation of the specification of the interface.

Year 2000 and 2038 Problems Leading to the year 2000, about $300 billion were spent worldwide to update outdated software that was unable to handle dates with a year of 2000 or higher.

The cause was that much software was used far beyond the originally envisioned lifetime. At programming time, especially at times when memory was still scarce, it made sense to use only two digits for the year in a date. That assumption became flawed when dates over 2000 had to be handled.

A related problem is expected in the year 2038. At that point the number of seconds since 1970-01-01, which is the dominant way of storing time on Unix, will exceed the capacity of a 32-bit integer. While application software is expected to be updated by then anyway, modern embedded systems may or may not still be in use.

Los Angeles Airport Network Outage In 2007, LA airport was partially blocked for 10 hours due to a network outage that prevented passenger processing. About 17,000 passengers were affected.

The cause was a single network card malfunction that flooded the network and propagated through the local area network.

Details: https://www.oig.dhs.gov/assets/Mgmt/OIGr_08-58_May08.pdf

Debian OpenSSL Random Number Generator From 2006 to 2008 Debian's variant of OpenSSL used a flawed random number generator. This made the generated keys easily predictable and thus compromised. It is unclear whether this was exploited.

The cause was that two values were used to obtain random input: the process ID and an uninitialized memory field. Uninitialized memory should never be used but is sometimes used as a convenient way to cheaply obtain a random number in a low-level programming language like C. The respective line of code had no immediately obvious purpose because it was not commented. Therefore, it was removed by one contributor after code analysis tools had detected the use of uninitialized memory and flagged it as a potential bug.

Detail: https://github.com/g0tmi1k/debian-ssh

Knight Capital Trading Software In 2012, high-frequency trading company Knight Capital lost about $10 million per minute for 45 minutes trading on the New York Stock Exchange.

The cause was an undisclosed bug in their automatic trading software.

Heartbleed From 2012 to 2014, the OpenSSL library was susceptible to an attack that allowed remotely reading out sections of raw physical memory. The affected sections were random but repeated attacks could piece together large parts of the memory. The compromised memory sections could include arbitrary critical data such as passwords or encryption keys. OpenSSL was used not only by many desktop and server applications but also in portable and embedded devices running Linux. The upgrade costs are very hard to estimate but were put at multiple $100 millions by some experts.

The cause was a bug in the Heartbeat component, which allowed sending a message to the server, which the server echoed back to test if the connection is alive. The server code did not check whether the given message length l was actually the length of the message m. Instead, it always returned l bytes starting from the memory address of m even if l was larger than the length of m. This was possible because the used low-level programming language (C) let the programmers store m in a memory buffer and then over-read from that buffer. Moreover, their C code is so hard to read that it is impossible to notice such minor errors on a cursory inspection.

Details: http://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Shellshock From 1998 to 2014, it was possible for any user to gain root access in the bash shell on Unix-based systems. The upgrade cost is unknown but was generally small because updates were rolled out within 1 week of publication. Moreover, in certain server applications that passed data to bash, clients could execute arbitrary code on the server.

The cause was the use of unvalidated strings to represent complex data. Bash allowed storing function definitions as environment variables in order to share function definitions across multiple instances. The content of these environment variables was trusted because function definitions are meant to be side-effect-free. However, users could append $;C$ to the value of an environment variable defining a function. When executing this function definition, bash also executed C.

Independently, many server applications (including the widely used cgi-bin) pass input provided by remote users to bash through environment variables. This resulted in input provided by remote clients being passed to the bash parser, which was against the assumptions of the parser. Indeed, several bugs in the bash parser caused remotely exploitable vulnerabilities.

Details: https://fedoramagazine.org/shellshock-how-does-it-actually-work/

Apple 'goto fail' Bug From 2012 to 2014, Apple's iOS SSL/TLS library falsely accepted faulty certificates. This left most iOS applications susceptible to impersonation or man-in-the-middle attacks. Because Apple updated the software after detecting the bug, its cost is unclear.

The immediately cause was a falsely-duplicated line of code, which ended the verification of the certificate instead of moving on to the next check. But a number of insufficiencies in the code and the software engineering process exacerbated the effect of the small bug.

The code was as follows:

```
static OSStatus SSLVerifySignedServerKeyExchange(
  SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
  uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
```

```
    SSLFreeBuffer(&hashCtx);
    return err;
```

In a better programming language that emphasizes the use of high-level data structures, the bug would likely not have happened or be caught easily. But even using C, it could have been caught by a variety of measures including unreachable code analysis, indentation style analysis, code coverage analysis, unit testing, or coding styles that enforce braces around one-command blocks.

Details: https://www.imperialviolet.org/2014/02/22/applebug.html

3.3 Other Interesting Failures

Odyssey Court Software In an ongoing crisis since 2016, US county court and California and other states have been having difficulties using the new Odyssey software for recording and disseminating court decisions. This has caused dozens of human rights violations due to erroneous arrests or imprisonment. This includes cases where people spent 20 days in jail based on warrants that had already been dismissed.

The cause is a tight staffing situation combined with the switch to a new, more modern software system for recording court decisions. The new software expects uses more high-level data types (e.g., reference to a law instead of string) in many places. This has led to the erroneous recording of decisions and a backlog of converting old decisions into the new database (including decisions that invalidate decisions that are already in the database).

Details: https://arstechnica.com/tech-policy/2016/12/court-software-glitches-result-in-erroneous-arrests-de

Other Failures Caused By System Updates This is a selection of failures that did not cause direct damage but led to availability failures on important infrastructure.

In 1990, all AT&T phone switching centers shut down for 9 hours due to a bug in a software update. An estimated 75 million phone calls were missed.

In 1999, a faulty software update in the British passport office delayed procedures. About half a million passports were issued late.

In 2004, the UK's child support agency EDS introduced a software update while restructuring the personnel. This led to several million people receiving too much or too little money and hundreds of thousands of back-logged cases.

In 2015, the New York Stock Exchange had to pause for 3 hours for a reboot after a software problem. 700,000 trades had to be canceled.

In 2015, hundreds of flights in the North Eastern US had to be canceled or delayed for several hours. The cause was a problem with new and behind-schedule computer system installed in air traffic control centers.

FBI Virtual Case File Project In 2005 the Virtual Case File project of the FBI, which had been developed since 2000, was scrapped. The software was never deployed, but the project resulted in the loss of $170 million of development cost.

The cause was systemic failures in the software engineering process including:

- poor specification, which caused bad design decisions
- repeated specification changes
- repeated change in management
- micromanagement of software developers
- inclusion of many personnel with little training in computer science in key positions

These problems were exacerbated by the planned flash deployment instead of a gradual phasing-in of the new system—a decision that does have advantages but made the systems difficult to test and made it easier for design flaws to creep in. The above had two negative effects on the code base

- increasing code size due to changing specifications
- increasing scope due to continually added features

which exacerbated the management and programming problems.

Fooling Neural Networks Deep learning neural networks have become very popular recently for pattern recognition (pictures and speech in particular). They are already used in, e.g., in self-driving cars or the AlphaGo system.

Despite their economic importance, they can err spectacularly in ways that are entirely different from the ways human pattern recognition errs. This is unavoidable because it is a consequence of their mathematical foundations.

Researchers were able to demonstrate that

- minor changes to an image that would be imperceptible to a human can make the network identity as something entirely different (e.g., a lion as a library)
- images that are unrecognizable to humans can make the network see an object with absolute certainty (e.g., labeling white noise as a lion)

This has major implications for the safety and security of systems based on these networks.

Details: http://www.evolvingai.org/fooling

Excel Gene Names In 2016, researchers found that about 20% of papers in genomics journals contain errors in supplementary spreadsheets.

The cause is that Microsoft Excel by default guesses the type of cell data that is entered as a string and converts the string into that type. This affects gene names like "SEPT2" (Septin 2, converted to the date September 02) or REKIN identifiers like "2310009E13" (converted to the floating point number $2.31E+13$).

Details: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-1044-7

Failures in Involving Computer-Related Manufacturing This is a selection of other notable failures that involve hardware manufacturing.

In 2006, two Airbus plants used incompatible version of CAD software. This resulted in cables being produced too short to connect.

In 2006, Sony batteries mostly used in Dell notebooks had to be recalled. The resulting cost was about $100 million.

In 2016, Samsung Galaxy phones had to be recalled due to faulty batteries.

3.4 Major Vulnerabilities due to Weak Security

3.4.1 Software and Internet

Operating Systems Vulnerabilities in operating systems are dangerous because only a few systems are used worldwide, therefore any problem is shared by many users. Moreover, the operating system usually has full access to the computer and its network, which allows any attack to do great damage.

Moreover, operating systems are usually bundled with standard applications (e.g., web browser, email viewer). These are tightly integrated with the OS (e.g., by using the same libraries for encryption or accessing files). Thus, a vulnerability often badly affects the majority of users who use these standard applications.

In 2000, the ILOVEYOU worm exploited weaknesses in the Windows OS and Outlook mail service, by infecting a significant share of all internet-connected computers within a few days. Its damage was estimated at over $5 billion and the removal costs at over $410 billion.

In recent years operating system companies have reacted to these problems. They have become more sensitive to security issues and allow for coordinated disclosure of vulnerabilities together with swift updates. Most noticeable for end users is the urged tendency to frequently install updates. For example,

- Microsoft Windows 10 automatically downloads and installs updates in a way that users cannot prevent.
- Google's Android now reserves the right to download minor updates immediately, even via mobile data.

This has greatly reduced the frequency of major problems.

An additional problem is that attacks are often conducted by state governments for purposes of terrorism, oppresion, espionage, sabotage, or law enforcement.

In 2010, the stuxnet worm was used by presumably the US and/or Israel to sabotage Iran's nuclear program. It was the most sophisticated attack to become public, involving multiple zero-day exploits and including attacks on programmable logic controllers.

In 2013, Edward Snowden revealed a massive secret surveillance program run by the US government. It used many ways to intercept data sent by users either at transmission nodes or at company data centers. This included connection metadata and any unencrypted or decryptable content. The stolen data remains mostly secret so that it prevents clarity on the information compromised and its usage. In response, many software companies introduced

end-to-end encryption that precludes even themselves to access their users data.

In 2016, Citizen Lab discovered an attack that used previously unknown vulnerabilities in Apple's Safari on iOS. It allowed, for the first time, an attacker to remotely take full control of the iPhone, triggered as soon as Safari was pointed to the attack URL. It is suspected that the exploit was produced commercially by the Israeli company NSO Group and used (at least) by the United Arab Emirates to spy on dissidents.

Details: http://www.vanityfair.com/news/2016/11/how-bill-marczak-spyware-can-control-the-iphone

Cloud Services Consumers are more and more using internet services for their processing needs. These include

- file storage, e.g., via Dropbox
- email and calendar services, e.g., via gmail
- office applications, e.g., directly via Google's office web site or indirectly via Microsoft's office suite
- social networking, e.g., via Facebook

Most modern operating systems and their bundled applications store large amounts of user data on the company's web servers, including, e.g., message archive, photographs, or location history. This creates unprecented risks for privacy, with legal regulation mostly lagging behind. (Most legislation were designed to limit the government from violating privacy. Corporations were barely restricted, in fact they used to have less power.)

Because most users do not understand the technical issues and blindly accept terms of service, thus generously granting access rights to applications, more and more user data becomes available to the free market. This is used for both legitimate (e.g., advertising-financed free services) or questionable purposes (e.g., manipulating voter preferences through personalized messages).

In 2014, The Fappening was an attack that combined phishing and password-guessing to gain access to many user accounts on Apple's iCloud. These acounts included, among other things, backups of all photographs taken with iPhones. Among the private data stolen and published were hundreds of nude pictures of celebrities.

Large Institutions In 2014, Sony Pictures suffered a major break-in (possibly by North Korea to blackmail or punish Sony in relation to the movie *The Interview*) mostly facilitated by unprecedented negligence. Problems included

- unencrypted storage of sensitive information
- password stored in plain text files (sometimes even called "passwords" or placed in the same directory as encrypted files)
- easily guessable passwords
- large number of unmonitored devices
- lack of accountability and responsibility for security, ignorance towards recommendations and audits
- lack of systematic lesson-learning from previous failures (which included 2011 hacks of Sony PlayStation Network and Sony Pictures that stole account information including unsalted or plain text passwords)
- weak IT and information security teams

Stolen data included employee data (including financial data), internal emails, and movies.

In 2016, the US democratic party's headquarters suffered a break-in (possibly by Russia to manipulate or discredit that year's presidential election). The stolen data included in particular internal emails and personal data of donors. Especially, the former hurt the public perception of the party's campaign to an unknown degree that may or may not have been decisive.

User Account Data Many organizations holding user data employ insufficient security against digital break-ins and insufficient (if any) encryption of user data. They get hacked or otherwise compromised so routinely that a strong market for stolen identities has developed, often pricing bulk datasets at a few dollars per identity. Overviews can be found at https://haveibeenpwned.com/ or https://en.wikipedia.org/wiki/SQL_injection.

This development is exacerbated by two human problems:

- System administrators are not sufficiently educated about password hashing and often falsely believe default hash configurations to be secure. Thus, hacks often allow inverting the hash function thus exposing passwords in addition to the possibly sensitive user data.
- Users are not sufficiently educated about systematically using different passwords on every site. Thus, any breach also compromises accounts on any other sites that use the same user name or email address and password.

Many websites now offer and nudge users to use two-factor authentication to protect accounts from identity theft. A second factor (e.g., via email or text message) may be required for every login, for every login from a new location, or for every sensitive action like changing the password. Security questions, which are particularly vulnerable, are phased out by leading companies. But both websites and users are slow to get used to this.

This problem is exacerbated by wide-ranging technically legal access by government agencies to private data. Companies usually comply with subpoenas by the country they operate in (both democracies and others) even if that compromises private user data. For example, in 2017, a US court decided that US companies (i.e., the majority of companies holding sensitive user data) must, if subpoenaed, hand over to US government also all user data stored on servers abroad. That makes it very difficult for other countries to enforce stricter data protection laws.

These constraints interact with market forces and foreign and economic policy. For example, China uses a protectionist policy to strengthen Chinese alternatives to US web services like Google or Facebook. But they also allow foreign companies under the condition that they provide the Chinese government with strong access to private data. Complying with these rules opens Western web service providers up to accusations that they support human rights violations. But dismissing these market opportunities opens them up to competition and shareholder pressure.

The following describes a few high-profile cases of stolen user data:

- In a 2005 hack of the US credit card payment processing company CardSystems, over 40 million accounts were compromised. The stolen data included name and credit card number. The reason was an SQL injection attack.
- In a 2005/2006 hack (reported in 2007) of the US retailer TJX, about 45 million accounts were compromised with an estimated damage of $1 billion. The stolen data included name and credit card number. The cause was the use of the obsolete WEP security standard for communication between pricing devices in one store. This allowed hackers to access the central database, where data was stored unencrypted that should have been deleted.
- In a (estimated) 2008 (only reported in 2016) of myspace, about 360 million accounts were compromised. The stolen data included user name, email address, and badly hashed passwords (unsalted SHA1).
- In a 2012 hack of linkedin, 160 million accounts were compromised. The stolen data included user name, email address, and badly hashed passwords (unsalted SHA1).
- In 2014, data for over 1 billion accounts from $420,000$ websites were discovered by the security company Hold Security. It is claimed that many of these break-ins stems from bot carrying out SQL injection attacks. The details are unclear as some claims by Hold Security are disputed.
- In a 2015 hack of Ashley Madison, about 30 million accounts were compromised. The stolen records included name, email address, hashed password, physical description, and sexual preferences. Most passwords were hashed securely (using bcrypt for salting and stretching), but about 10 million passwords were hashed insecurely (using a single MD5 application). This led to multiple extortion attempts and possibly suicides.
- In a 2016 hack of the Friend Finder network, about 400 million accounts were compromised. The stolen records included name, email address, registration date, and unhashed or badly hashed passwords.
- In two separate hacks in 2013 (only reported in 2016) and 2014 of Yahoo, over 1 billion user accounts were compromised by presumably state-sponsored actors. The stolen records included name, email address, phone number, date of birth, and hashed passwords, and in some cases security questions and answers.

3.4.2 Dedicated Systems

Many domains are increasingly using computer technology. Often this is done by engineers with little training in computer science and even less training in security aspects. In many cases, the resulting systems are highly susceptible to attacks, spared only by the priorities of potential hackers and terrorists.

Embedded Systems Embedded systems are increasingly running high-level operating systems, typically variants of Linux or Windows, and software. They are particularly vulnerable due to a number of systemic flaws:

- Software often cannot be updated at all or not conveniently. Thus, they collect many security vulnerabilities over time.
- Affected devices may be in use for years or decades, thus accumulating many vulnerabilities.
- It is hard or impossible for users to interact with the software in a way that would allow them to understand or patch its vulnerabilities.
- Access is often not secured or not secured well. Often master passwords (possibly the same on every instance of the system and possibly hard-coded) are used to allow access for technicians.

- It is often difficult to list all access rights (e.g., of remote control access via smartphone) or to revoke some of them (e.g., of previous owners).

Cars The upcoming wave of self-driving cars requires the heavy use of experienced software developers and a thorough regulation process. It is therefore reasonable to hope that security will play a major role in the design and legal regulation.

But even today's traditional cars are susceptible to attacks including remote takeover of locks, wheels, or engine. The causes are

- not or not properly protected physical interfaces for diagnostics and repair,
- permanent internet connections, which are useful for navigation and entertainment, that are not strictly separated from engine controls.

One of the more high-profile benevolent attack demonstrations was described in https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

Medical Systems Hospitals and manufacturers of medical devices are notoriously easy to hack.

Weaknesses include unchangeable master passwords, unencrypted communication between devices, outdated and non-updateable software running in devices, and outdated or non-existent protection against attackers. Systemic causes include a highly-regulated release process that precludes fast patching of software and a slow update cycle.

Details: http://cacm.acm.org/magazines/2015/4/184691-security-challenges-for-medical-devices/fulltext

See also the Symantec 2016 Healthcare Internet Security Threat Report available at https://www.symantec.com/solutions/healthcare

Part II

Systematic Software Development

Chapter 4

Implementation

4.1 Process Aspects

This section collects general methods that have been developed by practitioners to get a better handle on managing the software engineering process.
They are generally very cheap and easy to deploy. Therefore, it should[1] be considered gross[2] negligence to develop dependency-critical software without any one of these.
However, training lacks behind with many current developers not updated on latest technologies.

4.1.1 Coding Style

There are many aspects of a program that have no semantic relevance. These include

- most whitespace including
  - indentation and vertical alignment
  - tabs vs. spaces
  - placement of opening and closing brackets
  - optional spaces between operators and arguments
- choice of names for any names that are not fixed in the specification of the interface
  - private methods and field of a class
  - local variables of a function
  - classes and similar units that are not part of the specification
- syntactic restrictions on names including
  - length of names (documenting effect vs. readability)
  - capitalization of first character (e.g., lower case for values, upper case for types)
  - capitalization of inner characters (e.g., camel case vs. underscores)
- syntactic restrictions on declarations including
  - length of a method
  - number of declarations in a class
  - order of declarations (e.g., public vs. private, values vs. methods, mutable vs. immutable)
- preferred methods when multiple options are available, e.g.,
  - $l.nonEmpty$ vs. $!l.isEmpty$
  - $a.getX()$ vs. $a.getX$ if the compilers allows both
  - placement of optional semicolons
  - spelling out the argument or return types of a function when the compiler can infer them
- formatting of structured documentation including
  - placement of documentation inside the comment
  - use of markdown syntax inside comments
  - presence and order of keywords (e.g., author, param, return)
- placement and formatting of comments containing verification-relevant information including

[1] It is not, though.
[2] *Gross* negligence is the kind that makes people liable to prosecution or litigation.

  - pre/postconditions of methods
  - class invariants
  - loop invariants
  - termination orderings for loops and recursive functions

By standardizing these across a large project, readability is greatly enhanced. This is particularly important when it happens frequently that

- new programmers join a team and have to be quickly retrained on the entire code base
- programmers move between teams
- different teams work on the same code

Style checkers can be standalone or integrated into an IDE. Either way, they allow customizing a style and enforcing it throughout a project.

Modern IDEs (e.g., IntelliJ) provide a wide variety of coding style configurations whose violation results in special warning.

4.1.2 Documentation

Thorough documentation is used for multiple purposes:

- tie the implementation to the specification (e.g., by referencing the exact page or item of the specification corresponding to a declaration)
- inform other programmers about functionality and important subtleties
- provide examples for how to instantiate a class or call a function
- automatically extract web pages containing API documentation
- attach verification-relevant information that can be automatically extracted by verifiers

Most programming languages come with supporting tools that allow for structured documentation. For example, Javadoc is a structured documentation language for Java. The following example snippet is taken from the Javadoc home page:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

Note how it

- uses /** instead of the usual /* to indicate a structured comment
- can link to other code parts (which requires some compilation knowledge to resolve relative references in the documentation)
- integrates HTML which is carried through to the generated web pages
- uses keywords like @param so that better web pages can be produced

- is connected to the code by repeating the names of the function variables (which can be flagged by the style checker)

4.1.3 Versioning

Sophisticated version management systems like svn and recently git have tremendously improved the development process. Specifically, the use of git for the Linux kernel and the success of github have made a huge impact in the open source community.

They allow

- collaboration across physical distances
- maintaining different version of a software (e.g., a release and a development branch)
- using commit messages to log and communicate the reason and effect of a change
- retroactively determining when and by whom a bug was introduced
- automated building and testing on every commit

The following article about google repository management is particularly interesting: http://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

4.1.4 Code Review

Code review is the process of programmers reviewing each other's code before (or sometimes after) it becomes part of the stable parts of the code base.

Many modern versioning tools simplify the process by

- reifying changes (e.g., as diffs/patches or commits) so that each change can be reviewed and applied individually
- managing the available changes and applying them to branches
- maintain the proposed changes and the feedbacks and decisions by the reviewer

Github's maintenance of git pull requests is a simple example of a systematic code review process.

4.1.5 Automated Building and Testing

Most commit-based software repositories allow for hooks that are executed automatically before or after every commit (called push in git).

This is typically used for testing. A typical commit hook should

- create a fresh checkout of the source code
- build the source
- run a test suite (where the test may be part of the source or provided externally)

For most repository managers, automated build managers exist. They can generate reports for every commit and, e.g., alert users by emails upon new commits that fail the tests. A pre-commit hook can even reject the commit if the test failed.

travis is a typical example. It is well-integrated with, e.g., github.

4.1.6 Issue-Tracking

Issue track is the systematic management of known problems, their discussion, and eventual solution. Usually an issue is maintained as a discussion thread, and all open issues are available in a list that allows for filtering and sorting. They are typically provided via web interfaces, in particular to allow users to easily submit issues.

The typical process is

1. The initial post **opens** the issue.
2. After some discussion, the issue is **assigned** to a programmer or team.
3. After posting and checking the solution, the issue is **closed**.

There is a wide variety of issue tracking systems, usually independent of the programming language. Many are integrated with wikis or versioned repositories. Examples are trac and github.

4.2 Programming Aspects

This section collects mostly independent practices for individual aspects of programming.

4.2.1 Input Validation and Internal Syntax

The following is a fundamental principle that is absolutely necessary when handling user input:

- There is a data structure that represents the input/external syntax. This data structure is called the internal syntax. It should exactly follow the grammar of the input language.[3]
- All processing proceeds in the following steps:
  1. User input is parsed from a string holding external syntax into an object of type of the internal syntax. The parser must be side-effect-free: It does not nothing but parse and return an object of the internal syntax or an error message. Failure results in immediately rejecting the user input.
  2. All processing that ever happens works with the internal syntax. External syntax is never visible to any other function than the parser.
  3. The data structure provides a printer (also called serializer) that turns it into a string. Any output that is to be displayed to the user is generated in this way.

It is desirable that parser and printer are exactly inverse to each other. However, it is common that certain aspects of the internal syntax are lost after parsing and printing (e.g., whitespace). However, no meaningful information should be lost, e.g., the parser should not insert default value for omitted optional arguments—instead, it must record that the argument was omitted.

Conversely, parsing followed by printing must succeed and must result in the original object. Thus, we must have $parse(print(i)) = i$ and ideally also $print(parse(e)) = e$.

4.2.2 Common Bugs

We know that correctness is undecidable. But that is irrelevant for a pragmatic approach: the more can be avoided, the better.

Therefore, much work has gone into training programmers to avoid or building static analysis tools (see Sect. 6) to spot common bugs. The following gives some examples of frequent bugs and they can be systematically avoided. Note that some avoidance strategies are hypothetical that not every programming language supports the respective feature.

Array and List Bounds When iterating over an array or a list, we often use for-loops with an integer variable that runs from 0 to $n - 1$, where n is the length. Getting the index bounds wrong is a common mistake.

A good solution is to avoid the use of loops that only count up index variables. Instead, we can use *map* or *foreach* operations that traverse a list.

For example, to shift all values in x one up, we can use a counter with subtle traps

```
for i from 1 to x.length − 1
    x[i − 1] := x[i]
x[x.length − 1] := 0
```

A better solution is to use language constructs that enforce the correctness such as traversal operators on traversable data structures:

```
x.indices.tail foreach i =>
    x[i − 1] := x[i]
x[−1] := 0
```

Here our knowledge about the methods *indices* and *tail* guarantee that the assingment is only executed for indices of elements that have an element before them—no fiddling with the bounds of the for-loop is needed. Moreover, the last assignment uses modular arithmetic to access the last element without fiddling with the length.

[3] Incidentally, this means that untyped languages are out

Buffer Bounds Buffers can be treated as special arrays. The same techniques apply.

A particularly insane (and useless) property of buffers in certain low level programming languages is that the length of the buffer can be fixed in memory but not fixed in the programming language. Thus, after creating a buffer of size n, we can overread from it, i.e., read $m > n$ values from it. In this case, a C-like language will happily read whatever resides in memory after the buffer.

That can easily be avoided by

- using high-level data structures that hide the memory allocation from users of the data structure,
- or (even better) using high-level programming languages that hide the memory allocation from the programmer entirely.

In the latter case, buffer overreads are at least run-time exceptions.

Null Pointers Null pointers are routinely used by bad programmers, especially in bad programming languages. It is best not to use *null* at all. Indeed, good programmers program as if *null* does not exist.

A better solution is to use language constructs that enforce the correctness: the option type

```
data Option[A] = Some(value : A) | None
```

which provides an explicit value for absent values. Now every access to $x : Option[A]$ has to say what to for each of the two possible cases.

The only exception is when calling an external library that uses *null*. But even in that case, it is best to use back-and-forth translations between possibly-null and option values:

```
fun fromMaybeNull[A](x : A) : Option[A] =
    if x == null
        None
    else
        Some(x)
fun toMaybeNull[A](x : Option[A]) : A =
    x.getOrElse(null)
```

Casting Every type cast x **asInstanceOf** B of a value x to type B must be investigated for its correctness. That includes

- plausibility: is B a subtype of A—if not, the cast is definitely a bug and can be flagged by the compiler
- correctness: is x guaranteed to have type B—that requires a complex argument.

Such casts are typically guarded as in

```
if x isInstanceOf B
    f(x asInstanceOf B)
else
    g(x)
```

A better solution is to use language constructs that enforce the correctness. One way to do this is a case-distinction operator

```
match x
    x : B ↦ f(x)
    _ ↦ g(x)
```

which allows the compiler to spot a missing case if the second case is forgotten.

An even simpler solution is to use a cast operator that returns an options value:

```
fun optCast(x : A) : Option[B] =
    ...
(optCast(x) map f).getOrElse(g(x))
```

Uninitialized Memory Some programming languages allow introducing names without initial values. Those can be variables or (in C-like low-level languages) memory areas.

Uninitialized variables can easily be spotted by analysis tools. They should not be warnings but actual compiler bugs. Allowing them at all is a design flaw of the programming language.

A variant uninitialized variables are variables initialized with *null*. That is equally easy to spot and equally forbidden.

Uninitialized memory areas are occasionally useful when initializing a large memory area is considered too costly. In most cases, this can be avoided entirely by using good data structures.

4.2.3 Safe by Design

This section collects a few implementation principles that help minimize the likely hood of errors.

Safe Defaults Default and initial values should always be chosen in such a way that they lead to the minimal possible behavior.

For example, a security check should be wrapped in an exception handler that treats every exception as failure of the check.

Minimal Access Rights Any component C that needs access to critical shared component D should have only the minimal access rights needed for its correct operation.

For example, if C is a program and D is a database or file system, then D should provide an interface that allows only exactly those write operations that C needs. If C has to write to a specific table T in a specific database, we write add a wrapper around D to the source code base of C. This wrapper includes a specific function f that makes exactly the changes to T that C needs to trigger. C will call only call f and will have no access to T or D as a whole.

Minimal Interfaces All classes D should only make those methods public that are needed by the other classes C using them. Methods that are not needed by any C are made private in D.

Vice versa, a class C should only have access to methods of D if it actually has to call them. Methods of D that are not needed by C are made invisible by introducing an abstract interface I. I declares exactly the methods needed by C, D implements I, and C uses I instead of D.

4.3 Stability

Frequent changes are extremely dangerous to even the best software development process.

Stability is particularly important for

- the specification,
- the design of the data structures and algorithms,
- the project team,
- the coding style,
- the workflows for building, committing, etc.,
- the policies for access, review, and approval of changes.

It is very tempting for managers, marketing department, and customers to request changes because they seem easy to them. Even many programmers often underestimate their dangers.

But every change at a high level introduces lots of increasingly greater and more expensive changes at lower levels. Eventually, the lower levels (especially when resources are tight, which is always the case) have to introduce workarounds, hacks, and special-case treatments to handle the changes.

To the top-level person who requested the change, everything looks fine because the lower levels will usually do a good job of hiding the mess. But below the surface the codebase will become increasingly messy until it is unmanagable.

A good metaphor is to think of a long stick representing the hiearchy. The person at the top points the stick at a slightly different angle, which does not very different from the top. But at the lower end of the stick, the

small change in angle caused a massive shift of the end of the stick. The shift may be much bigger than what the inertia of the stick allows, and the sticks breaks somewhere in the middle. When the stick breaks, the people at the breaking point in the middle move the lower half of the stick so that it points to the right point and hire two new people A and B. A constantly measures the movement of the upper half of the stick and shouts the values to B. B then computes how much the lower half would move if the halves were still connected and moves the lower half accordingly. The person at the top does not notice anything. But over time, the stick has broken into many pieces, and lots of people are in charge of pretending it is still one piece.

4.4 Choice of Programming Language

The choice of programming language can have major implications for dependability. We discuss some evaluation criteria for programming languages.

Learnability A programming language should be easy to learn. Properties that support this are:

- a simple, elegant language constructs, whose meaning is close to human or mathematical intuition,
- a concrete syntax that makes the structure of programs apparent without introducing overhead
- a coherent design that avoids special case treatment as much as possible
- built-in (or automatically imported from the standard library) data structures for common concepts such as strings, lists, options, functions, dictionaries along with simple concrete syntax for them
- a simple hello-world with minimal overhead
- an interactive interpreter that allows experimenting
- easily-installable IDE plugins and command-line compilers/interpreters

Easy-to-learn languages support dependability because

- They guarantee a steady stream of competent programmers.
- They allow programmers to focus on the domain knowledge and the system rather than being distracted by language issues.

Tool Support A programming language should come with a variety of tools including compiler, interpreter, interactive interpreter, IDEs, and dynamic and static analysis tools. Programming languages are in principle independent of the tool support: any language can offer any tool. For example, any language can be compiled or interpreted. However, in practice languages tend to come with a single implementation and at most a small set of third-party tools.

In particular, dynamic and static analysis tools is critical for dependability. Compiled languages are preferable because

- the compiler performs fundamental static analysis, which includes at least in particular scope- and type-checking
- additional static analysis tools can be easily implemented on top of the compiler

Availability of Skilled Programmers Any major software project needs multiple programmers over a long time. Therefore, it is indispensable to have a large supply of competent programmers.

This can be achieved by retraining programmers (see Learnability). Indeed, most programmers can easily learn a new language. But sometimes the choice of language can affect the set of available programmers:

- Existing people may be committed to one language and be unwilling to switch.
- Potential hires might not be interested in job openings due to their personal language preferences.

Availability of Libraries Anticipating and surveying needed libraries is critical. Besides the obvious criterion of dependability, this includes judging libraries based on appropriateness[4], maturity, active support, and license.

Some commonly-used advanced data structures such as for hash tables, heaps, etc. are needed in every project. These should be provided by a mature and well-designed standard library and possibly some auxiliary libraries. Moreover, projects may require libraries for networking, parsing, cryptography, XML etc.

[4] A common mistake is to assume a library will work well without assessing the overhead cost of adapting to it.

Additionally, many projects require the implementation of large amounts of domain-specific knowledge. Often these libraries are written by domain experts and are available for whichever programming language that expert happened to like. Therefore, programming languages may differ widely in the available library support.

If needed libraries are not available, they must be provided in-house. That is a frequent source of faults.

Existing Codebases Most projects are not started in a vacuum but must interact with existing code. The cost of using different programming languages must be assessed in a project-specific way.

Changing to a different language requires not only reimplementing not only programs but also redesigning work flows and relearning tools. This may alienate programmers and render useless institutional memory that has accumulated for years. Moreover, small changes in the reimplementation can introduce subtle faults.

Adding a new programming language to an environment introduces an abrupt interface between components. Some programming languages allow direct compatibility using foreign-function interfaces (often to C) or by compiling to the same virtual machine (often for the JVM). But often the interface requires text-based communication between these components (e.g., using strings or files), which is a common source of faults. Modern data exchange languages like JSON, XML, or protobuffers have improved the situation somewhat but are still much worse than direct compatibility. In any case, unless the separation into components perfectly matches the a logical separation in the design of the software, it leads to design flaws that make the software harder to understand, use, debug, and verify.

In either scenario, the likelihood of faults increases.

Efficiency A programming language should produce efficient programs and allow programmers to optimize their programs. Generally, low-level, hardware-near programming language (like imperative languages, especially the C family) are more efficient than high-level, specification-near ones (like functional languages).

Efficiency contributes little to dependability though. An exception arises when workarounds (e.g., a foreign-function interface to C or a shell-call to a different program) must be introduced to meet efficiency targets that cannot be elegantly met by the otherwise-preferred solution.

Specification-Nearness Most elusively, a programming language should allow for implementations that are as close to the domain knowledge as possible. That allows

- domain experts to program or verify the programs themselves, thus increasing the likelihood of correctness
- understanding and verifying programs more easily

This requires high-level programming languages with sophisticated support for defining data types such as inductive data types from functional or class hierarchies from object-oriented languages.

Ideally, this would be the most important criterion to obtain dependable software. However, it tends to be mutually exclusive with the other criteria: Good high-level languages tend to be younger and therefore have fewer programmers, tool support, libraries and existing codebases. The emphasis on high-level concepts makes them less hardware-near and thus less efficient and more mathematical and thus harder to learn.

Chapter 5

Dynamic Analysis

Dynamic analysis is a collective term for methods that evaluate code by executing it.

There is a variety of different dynamic methods, which should be discussed in a software engineering course. A good overview is given in the corresponding chapters in the Software Engineering Body of Knowledge (SWEBOK) [BF14][1].

The methods can be roughly classified into detection and diagnosis of faults.

5.1 Detect Faults

Testing is the most important dynamic analysis. Testing can be classified using a number of different properties. For achieving dependability, the following combination is most important:

- Unit testing: The tested component is small software unit like a class or a function.
- Conformance testing: The tested component is tested against its specification. This usually involves a set of test cases that are executed:
  - A test case for an implementation f of a function specification F is a pair $(x, F(x))$. The test case runs $f(x)$ and compares the result to $F(x)$.
  - When testing a class C, a simple test case consists of constructor arguments x and a sequence of test cases for methods of C. The test case runs $c :=$ **new** $C(x)$ and then runs each test case on c. Additionally, the test may evaluate a class invariant after each step.
- Automated testing: The test cases are programmed in parallel to the implementation. Thus, they can be build and run automatically.
- Whitebox or blackbox testing: In the simplest case, testing is black-box, i.e., the test does not need to access the source code. However, whitebox testing can be valuable, especially if combined with a coverage check. For example, the test cases can be designed to execute every line of the source code, which can be checked with a coverage analysis tool.

Coverage checking traces program execution to determine which lines of the source code were entered. In combination with an appropriately large set of test cases, this has a good chance of detecting control flow errors.

Memory debugging monitors memory (de)allocation and other dynamic memory operations. This can help detect buffer overflows or memory leaks.

5.2 Diagnose Faults

If errors are confusing, it may be necessary to trace the program behavior step-by-step to locate the fault.

Logging is the simplest diagnostic technique. It prints of variables in specific places to detect exactly which commands computes a faulty value.

[1] Or the respective wikipedia pages, which are often summaries of the SWEBOK or similar sources.

Debugging is a free form of logging. The program is executed step by step interactively, and the programmer can inspect variable values after every step.

Profiling watches the execution of all steps and accumulates which program unit uses how much time and/or memory. This can be used to find efficiency problems caused by design or implementation faults.

Chapter 6

Static Analysis

Static analysis is a collective term for methods that do not execute the program. It is performed by external tools that analyze the source code to detect faults.

Usually static analysis works with the source code. But in principle it can work on the binary as well, e.g., some tools work on Java byte code.

Due to undecidability, detecting faults requires insight and careful case-by-case analysis, which is expensive. Therefore, it has become very successful to focus on classes of faults and then to systematically find their occurrences. We can think of these as individual heuristics, each hunting for a specific common bug, e.g., the ones discussed in Sect. 4.2.2). State-of-the-art tools check for hundreds of such fault classes. Depending on the fault class, the checks may exhibit false-negatives (not all instances are detected) or false-positives (correct code is falsely marked).

6.1 Work Flow

6.1.1 Compilation

The first line of defense against faults is compilation, which is the simplest form of static analysis.

The primary goal of compilation is to translate the source code into another language, usually a binary file in a machine-near language such as assembly or byte code. However, like all good algorithm operating on user input, compilers usually consist of three steps:

1. parse the source files into a value p of an abstract data type L representing the context-free grammar of the source language
2. check p in order to verify the context-sensitive constraints of the source language
3. process p to obtain the compiled program, usually consisting of
   - optimization that transforms $p : L$ to $p' : L$
   - printing p' in the target language

Here the middle step is a static analysis. It usually consists of at least

- scope-checking: ensure that only global or local identifiers are used that
  - have been declared
  - are in scope
  - are accessible (e.g., not private to a class)
- type-checking: ensure that only expressions are formed that are well-typed, i.e., check that
  - arguments of class constructors
  - arguments of function and method invocations
  - assignments to variables
  - parameters of built-in language constructs (e.g., the condition of an if-statement)

  have the expected type

Compilers in particular focus on checks that are necessary to ensure that printing out the compiled program succeeds at all. For example, printing an ill-scoped program might cause a run-time error of the compiler.

But compilers may carry out arbitrarily many additional static analysis checks.

6.1.2 Additional Analysis

Typically scope and type-checking (if successful) are followed by additional static analysis. This can be done by

- the compiler itself
- compiler plugins
- separate tools that use the compiler for parsing and checking and then perform other analysis
- separate tools that use their own parser

Some of these tools can be integrated with the IDEs to report the results in the same way as compiler errors.

6.2 Individual Checks

There is a wide variety of possible checks that can be implemented by static analysis tools. An overview of tools for various programming languages can be found at https://en.wikipedia.org/wiki/Static_code_analysis For example, findbugs is a popular tool working on Java byte code; it checks for over 400 different faults.

In the following, we list some of the most important checks.

Coding Style The simplest static analysis is to check for coding style. This is trivial but technically a special case of static analysis.

It often requires a separate parser because the compiler's parser may ignore the details of whitespace.

Common Error Patterns Many errors are made so often that it is worth performing a static analysis for them.

For example, in some programming languages $m \bmod 2$ (arguably falsely) evaluates to -1 instead of 1 for odd negative numbers. Therefore, a static analysis tool may flag $m \bmod 2 == 1$ as a likely error (because the expressions behaves unexpectedly for negative m).

Non-Exhaustive Match Languages with inductive data types allow performing exhaustiveness checks. This checks whether there is a case for any possible value of the matched expression. However, because the programmer might know from invariants that certain values are impossible, compilers languages often allow not giving cases for all values.

An exhaustiveness check flags these as likely errors.

Uninitialized Variables and Memory Some languages allow declaring variables without assigning an initial value or allocating memory without initializing it. (Especially the latter is often important for efficiency.)

But any use of uninitialized variables or memory is almost certainly an error. (The only flimsy exception arises when random behavior is intended, e.g., in a random number generator, and even then it is bad practice.)

Probable Typing Errors Sometimes the specification of a language makes certain expressions technically well-typed even though they are likely to be errors.

In languages with a universal type U, any equality $a == b$ is well-typed because $a : U$ and $b : U$. However, $a == b$ is usually an error if the smallest common type of a and b is U. For example, if $a : \mathit{int}$ and $b : \mathit{List}[\mathit{int}]$, then $a == b$ is a likely error (because it always returns *false*).

In language with explicit type checks a **isInstanceOf** T and type casts a **asInstanceOf** T, any such check or cast may be well-typed. For example, if the language has *null*, indeed both must be well-typed because they succeed if $a == \mathit{null}$ independent of T. But given $a : A$, such checks and casts are likely errors if T is not a subtype of A. For example, casting $a : \mathit{List}[\mathit{int}]$ into $\mathit{Option}[\mathit{int}]$ is a likely error (because it succeeds only when $a == \mathit{null}$, in which case it is a useless cast).

Null Checks In languages with *null* pointers, it is often possible to analyze whether a pointer is dereferenced that may be null.

Given a value $a : A$, special cases include

- dereferencing a if a is known to be null
- dereferencing a without checking $a == \mathit{null}$

- checking $a == null$ when a has been previously dereferenced

Unreachable Code If no execution can ever execute a certain command, we speack of unreachable code. It is always a bug.

Many instances of unreachable code can be detected by systematic analysis. This includes

- code in a branch of an if-else whose condition always has the same value
- code in a case distinction (switch) statement that come after a default case
- code after a return statement

Part III

Formal Systems

Chapter 7

Type Theory

7.1 Common Structure

Formal systems such as type theories, logics, and programming languages share the same basic structure.

7.1.1 Objects

The objects of a formal system consist of **syntax** and **meta-syntax**[1].

Context-Free Syntax

The syntax consists of

- Concepts. This is a finite set of primitive concepts that defines the different kinds of objects that exist in the formal system.
  Examples are *type*, *term*, *formula*, *program*, or *proof*.
- Constructors. This is a finite set of primitive operations that build objects. Each object is an instance of one of the concepts.
  Examples are
  - *int* builds the type of integers.
  - $\times$ takes two types and returns their product type.
  - == takes two terms and returns a formula.

The concepts and constructors together form a **context-free** grammar by using

- one non-terminal for every concept
- one production for every constructor

Context-Sensitive Meta-Syntax

The meta-syntax consists of

- Judgments. This is a finite set of possible statements we can make about the syntax.
  Judgments are usually written using the $\vdash$ symbol.
  Examples are
  - "term t is well-formed and has type A", which we usually write as $\vdash t : A$.
  - "formula F is well-formed", which we usually write as $\vdash F$ `: form`.
  - "formula F is provable", which we usually write as $\vdash F$.

  A judgment may hold/be true/be derivable or not.
- Rules. This is a finite set of possible ways to derive judgments. Only the judgments that can be derived by using the rules are true.
  Rules are usually written using a horizontal line between assumption/premise/hypothesis judgments and conclusion judgment. Examples are

[1] There is no standard word for this. So I made up *meta-syntax*, which describes it best.

– the modus ponens rule "if $\vdash F \Rightarrow G$ and $\vdash F$, then $\vdash G$", which we usually write as

$$\frac{\vdash F \Rightarrow G \qquad \vdash F}{\vdash G}$$

– the pairing rule "if $\vdash s\ :\ A$ and $\vdash t\ :\ B$, then $\vdash (s,t)\ :\ A \times B$", which we usually write as

$$\frac{\vdash s\ :\ A \qquad \vdash t\ :\ B}{\vdash (s,t)\ :\ A \times B}$$

– the integer axiom "all elements of the regular expression $[-](0|\ldots|9)^*$ are integers", which we might write as

$$\frac{x \in [-](0|\ldots|9)^*}{\vdash x\ :\ \mathit{int}}$$

(This rule uses an assumption that is not a formal judgment. That is allowed if the assumption can be checked in a well-defined way.)

Judgments are rules together form an **inference system**.

The choice of concepts, constructors, judgments, and rules is not prescribed. Every system is free to choose. However, for every object O there must be one distinguished judgment, that checks whether O is a well-formed object. Typically, there is a separate way to determine well-formedness depending on the concept of which O is an instance. Examples are:

- The judgment $\vdash F\ :\ \texttt{form}$ directly captures the well-formed formulas.
- For a term t to be well-formed, we usually require that the judgment $\vdash t\ :\ A$ holds for some type A, i.e., a term is well-formed if it is well-typed.

The syntax and the meta-syntax together form a **context-sensitive** sublanguage of the context-free syntax. This sublanguage consists of all objects of the context-free language for which the well-formedness judgment holds.

7.1.2 Declarations

Introducing Declarations

While objects are always anonymous, the declarations of a formal system introduce **names** for objects. Like objects, the declarations are defined by a grammar and inference system.

Typically, there is a fixed non-terminal *Decl* and a fixed judgment $\vdash D\ \checkmark$ for declarations D. The each declaration-kind is defined by one production and one rule.

Examples are

- Term definitions are the declarations that introduce a name for a term. Those are the variable definitions known from most programming languages. They consist of the production

$$\mathit{Decl} ::= \textbf{val}\ x : A = t$$

for a new variable x of type A with initial value t and the rule

$$\frac{\vdash t\ :\ A}{\vdash \textbf{val}\ x : A = t\ \checkmark}$$

that checks that the initial value has the expected type.

- Type definitions similarly introduce a name for a type. They consist of the production

$$\mathit{Decl} ::= \textbf{type}\ a = A$$

for a new type variable a with value A and the rule

$$\frac{\vdash A\ :\ \texttt{type}}{\vdash \textbf{type}\ a = A\ \checkmark}$$

that checks that the initial value has the expected type.

To make parsing easier, many languages (e.g., SML or Scala) require each declaration-kind to start with a special keyword (like **val** and **typedef** above).

We need to be able to use the names as objects later on. Therefore, every concept comes with a built-in production for names. For example, for terms we expect a production $term ::= x$ where x is the name of a variable.

Now **scope-checking** is the part of the inference system that ensures that a name x may only be used if it has been previously declared. That shows us that we cannot simply use the judgment $\vdash x : A$—there would be no way to look up if x has been declared. Therefore, we have to collect all declarations that are in scope and carry them around. That is the purpose of signatures and contexts.

Assumptions

Often we distinguish two kinds of declarations:

- **Definitions** are declarations a definiens (i.e., an assignment to the new name). Most declarations in user input are of this form.
- **Assumptions** are declarations without a definiens. This should only be allowed in specific circumstances. Examples are the parameters of a function or a constructor.

Collecting Declarations

A **context** Γ is a list of declarations.

Usually all judgments for objects take a context as an additional argument. It is usually placed to the left of the $\vdash$ symbol. For example, we have $\Gamma \vdash t : A$ or $\Gamma \vdash F : \texttt{form}$.

Now the simplest form of scope-checking uses the rule

$$\frac{\textbf{val}\ x : A = \_ \in \Gamma}{\Gamma \vdash x : A}$$

And when checking input consisting of global declarations $D_1, \ldots, D_n$, we check for each $i = 1, \ldots, n$ that

$$D_1, \ldots, D_{i-1} \vdash D_i \checkmark$$

Thus, each declaration is checked in the context of the previous ones.

This simple checking is not always sufficient. There are many situations that require more complex treatment. Examples are

- Each declaration should introduce a fresh name. Otherwise, the later declaration shadows the previous one.
- A list of mutually recursive functions cannot be checked in order because all functions must be in the context already when the first function is checked.
- Some declarations may contain local declarations inside them. For example, a function declaration may contain local variable declarations.
- Some declarations may only be allowed globally (e.g., class definitions in some programming languages) or only locally (e.g., undefined variables to represent the parameters of a function).

Polymorphic Declarations

Very often it is convenient to introduce a new name that takes type parameters. For example, the type $List[A]$ of lists over A should take a parameter for the type A of the elements. Similarly, the function $revert[A] : List[A] \to List[A]$ takes a type parameter.

That can be accomplished by allowing for type assumptions in all declarations that we allow to be polymorphic. Examples are:

- For parametric type definitions (also called **type operators**), we use

  $$Decl ::= \textbf{type}\ a[(\textbf{type}\ a)^*] = A$$

  $$a ::= a[A^*]$$

  where $a[A_1, \ldots, A_n]$ is the type that arises by applying a to the parameters $A_1, \ldots, A_n$.

- For parametric constants (also called **polymorphic constants**), we use

$$Decl ::= \textbf{val}\ x[a^*] : A = t$$

$$t ::= t[A^*]$$

7.1.3 A Basic Formal System

We now introduce a comprehensive example that we will build on later. We first introduce an empty formal system, i.e., a type theory that has all the necessary structure but no non-trivial objects yet. We can then extend it with specific types and terms in Sect. 7.2

The concepts and constructors are given by the following context-free grammar:

contexts

$\Gamma \quad ::= Decl^*$

declarations

$Decl ::= \textbf{type}\ a[a^*] = A^?$		type declaration (with optional definiens)
$\quad \mid \textbf{val}\ x[a^*] : A = t^?$		term declaration (with optional definiens)

types

$A \quad ::= a$		type names

terms

$t \quad ::= x$		term names

The judgments are:

non-terminal	typing judgment	
Γ	$\vdash \Gamma\ \checkmark$	Γ is well-formed
D	$\Gamma \vdash D\ \checkmark$	declaration D is well-formed in context Γ
A	$\Gamma \vdash A\ :\ \texttt{type}$	A is a well-formed type in context Γ
t	$\Gamma \vdash t\ :\ A$	t is a well-formed term of type A type in context Γ

The rules for this empty type type mostly take care of scope-checking:

- Every declaration can use the previous ones (where ε is the empty context):

$$\frac{}{\vdash \varepsilon\ \checkmark} \qquad \frac{\vdash \Gamma\ \checkmark \quad \Gamma \vdash D\ \checkmark}{\vdash \Gamma, D\ \checkmark}$$

- Types and terms can be declared with type parameters and definiens:

$$\frac{\Gamma, \textbf{type}\ a_1, \ldots, \textbf{type}\ a_n \vdash A\ :\ \texttt{type}}{\Gamma \vdash \textbf{type}\ a[a_1, \ldots, a_n] = A\ \checkmark} \qquad \frac{\Gamma, \textbf{type}\ a_1, \ldots, \textbf{type}\ a_n \vdash t\ :\ A}{\Gamma \vdash \textbf{val}\ x[a_1, \ldots, a_n] : A = t\ \checkmark}$$

- Declared names can be used later on with the right number of type parameters:

$$\frac{\textbf{type}\ a[a_1, \ldots, a_n] = A\ \in\ \Gamma \quad \Gamma \vdash A_1\ :\ \texttt{type} \quad \ldots \quad \Gamma \vdash A_n\ :\ \texttt{type}}{\Gamma \vdash a[A_1, \ldots, A_n]\ :\ \texttt{type}} \qquad \frac{\textbf{val}\ x[a_1, \ldots, a_n] : A = \_\ \in\ \Gamma \quad \Gamma \vdash A_1\ :\ \texttt{type} \quad \ldots \quad \Gamma \vdash A_n\ :\ \texttt{type}}{\Gamma \vdash x[A_1, \ldots, A_n]\ :\ A}$$

7.2 Concrete Objects

7.2.1 Base Types

Types

Base types b are just names that are always in scope. They are very easy to add using one production and one rule:

$$A ::= b$$

$$\frac{}{\Gamma \vdash b \ : \ \texttt{type}}$$

Typical base types b that we often add are

- $b = \mathit{void}$: the empty type
- $b = \mathit{unit}$: the unit type with one element
- $b = \mathit{bool}$: booleans
- $b = \mathit{int}$: integers
- $b = \mathit{string}$: strings

Terms of Base Types: Base Terms

Each base type b comes with special base terms, often called literals that are defined by a regular expression R_b.
They are also easy to add using one production and one rule:

$$t ::= R_b$$

$$\frac{l \in R_b}{\Gamma \vdash l \ : \ b}$$

Typical literals are

- $R_{\mathit{void}} = \varnothing$ (no literals for the empty type)
- $R_{\mathit{unit}} = ()$ (the unique term of the unit type)
- $R_{\mathit{bool}} = \mathit{true}|\mathit{false}$
- $R_{\mathit{int}} = [-](0|\ldots|9)^*$
- $R_{\mathit{string}} = (\backslash\backslash|\backslash"|"[\backslash"])^*"$ (quoted string with \ and " escaped)

Terms Formed From Base Terms

Each base type b comes with special operators that allow working with a term $t : b$.
We need no operations for *void* and *unit*.
For booleans, we can use the if-then-else constructor:

$$t ::= \textbf{if } (t) \ \{t\} \ \textbf{else } \{t\}$$

$$\frac{\Gamma \vdash c \ : \ \mathit{bool} \quad \Gamma \vdash s \ : \ A \quad \Gamma \vdash t \ : \ A}{\Gamma \vdash \textbf{if } (c) \ \{s\} \ \textbf{else } \{t\} \ : \ A}$$

For integers, we need the arithmetic operations such as

$$t ::= t + t$$

$$\frac{\Gamma \vdash s \ : \ \mathit{int} \quad \Gamma \vdash t \ : \ \mathit{int}}{\Gamma \vdash s + t \ : \ \mathit{int}}$$

We omit the other operations on integers and strings.

7.2.2 Product Types

Product types add the Cartesian product.
Like for base types, the definition consists of three parts.

Product Type

The first part introduces the new type:

$$A ::= A \times A$$

$$\frac{\Gamma \vdash A \ : \ \texttt{type} \quad \Gamma \vdash B \ : \ \texttt{type}}{\Gamma \vdash A \times B \ : \ \texttt{type}}$$

Terms of Product Type: Pairs

The second part introduces terms of the new type:

$$t ::= (t, t)$$

$$\frac{\Gamma \vdash s \ : \ A \quad \Gamma \vdash t \ : \ B}{\Gamma \vdash (s, t) \ : \ A \times B}$$

Terms Formed From Pairs: Projections

The third part introduces terms formed from a term $t : A \times B$ of the new type:

$$t ::= t.1 \mid t.2$$

$$\frac{\Gamma \vdash t \ : \ A \times B}{\Gamma \vdash t.1 \ : \ A} \qquad \frac{\Gamma \vdash t \ : \ A \times B}{\Gamma \vdash t.2 \ : \ B}$$

7.2.3 Disjoint Union Types

Disjoint union types add the union $A+B$ of two types in such a way that there is no overlap, i.e., $|A+B| = |A|+|B|$. They are dual to the Cartesian product: all aspects mirror product types with the direction of functions reversed. For example, we have two injections into $A + B$ instead of two projections out of $A \times B$.

Union Type

The first part introduces the new type:

$$A ::= A + A$$

$$\frac{\Gamma \vdash A \ : \ \texttt{type} \quad \Gamma \vdash B \ : \ \texttt{type}}{\Gamma \vdash A + B \ : \ \texttt{type}}$$

Terms of Union Type: Injections

The second part introduces terms of the new type:

$$t ::= inj_1(t) \mid inj_2(t)$$

$$\frac{\Gamma \vdash t \ : \ A}{\Gamma \vdash inj_1(t) \ : \ A + B} \qquad \frac{\Gamma \vdash t \ : \ B}{\Gamma \vdash inj_2(t) \ : \ A + B}$$

Terms Formed From Union Terms: Case Distinctions

The third part introduces terms formed from a term $t : A + B$ of the new type:

$$t ::= cases(t, t, t)$$

$$\frac{\Gamma \vdash t \ : \ A + B \quad \Gamma, \mathbf{val}\ x : A \vdash c_1 \ : \ C \quad \Gamma, \mathbf{val}\ x : B \vdash c_2 \ : \ C}{\Gamma \vdash cases(t, c_1, c_2) \ : \ C}$$

A more intuitive notation for $cases(t, c_1(x), c_2(x))$ is $\mathbf{match}\ t\ \{inj_1(x) \mapsto c_1(x) | inj_2(x) \mapsto c_2(x)\}$. Indeed, disjoint union types are rarely used as a primitive feature because they can be easily defined as an inductive type with constructors inj_1 and inj_2.

The formulation here is tweaked to be maximally modular, i.e., independent of other features. Alternatively, c_1 and c_2 be taken to be functions $A \to C$ and $B \to C$.

7.2.4 Function Types

The most important type constructor in computer science is the one for function types—it is critical to implement computations. It can be introduced systematically in the same way as products.

Function Type

The first part introduces the new type:

$$A ::= A \to A$$

$$\frac{\Gamma \vdash A \ : \ \texttt{type} \quad \Gamma \vdash B \ : \ \texttt{type}}{\Gamma \vdash A \to B \ : \ \texttt{type}}$$

Terms of Function Type: Anonymous Functions

The second part introduces terms of the new type:

$$t ::= (x : A) \mapsto t$$

$$\frac{\Gamma \vdash A \ : \ \texttt{type} \quad \Gamma, \mathbf{val}\ x : A \vdash t \ : \ B}{\Gamma \vdash (x : A) \mapsto t \ : \ A \to B}$$

Terms of function type are the λ-abstractions $\lambda x : A.t$. In programming languages we usually write something like $(x : A) \mapsto t$ instead. The meaning is the same.

Our grammar and rule use the special case where all functions take exactly 1 argument. The general case can be defined accordingly (or can be obtained by using unit and product types).

λ-abstractions are more difficult than most other terms because they introduce a local variable. This is no problem at all if we have understood type theory properly: we can simply add the variable to the context as a local assumption while checking the term t.

However, many programming languages and programmers are confused or overwhelmed by this. Therefore, they often do not use anonymous functions.

Instead, they introduce a special declaration for named functions:

$$Decl ::= \mathbf{fun}\ f(x : A) : B \ = \ \{t\}$$

Using anonymous functions, that is just a special case of a definition:

$$\mathbf{fun}\ f(x : A) : B \ = \ \{t\} \quad \text{ is the same as } \quad \mathbf{val}\ f : A \to B = (x : A) \mapsto t$$

However, named functions come up so often and are so convenient that most languages allow function declarations even if they are redundant.

Terms Formed From Functions: Function Applications

The third part introduces terms formed from a term $t : A \to B$ of the new type:

$$t ::= t(t)$$

$$\frac{\Gamma \vdash s \ : \ A \to B \qquad \Gamma \vdash t \ : \ A}{\Gamma \vdash s(t) \ : \ B}$$

7.2.5 Objects with Local Definitions

Local definitions introduce an abbreviation to be used in subsequent expressions.

Depending on the language, they are usually written as **let val** $x : A = s$ **in** $t(x)$ or simply **val** $x : A = s; t(x)$. The latter tends to be nicer because it can be chained more easily.

Languages differ in what kind of definitions may be used locally in what kind of concepts. For example, the above expressions use a local definitions of a *term* inside a *term*.

The following production and rule can be used to allow *any* local definitions in a *term*.

$$t ::= Decl; t$$

$$\frac{\Gamma \vdash D \checkmark \qquad \Gamma, D \vdash t \ : \ A}{\Gamma \vdash D; t \ : \ A}$$

Many programming language write these terms as $\{D; t\}$ with the understanding that chained local definitions can be grouped in a single pair of brackets as in $\{D_1; \ldots; D_n; t\}$.

7.3 Data Types

Software is most dependable if programs represent the domain knowledge as closely as possible. That requires being able to define new types capturing exactly the concepts of the domain. The terms of these new types should capture exactly the needed domain data.

Data types are the most important way to do that. Most typed programming languages allow defining data types. But they may differ substantially in how they do it.

Data types are always named, i.e., they are introduced by declarations not by objects. Like base, product, and function types, *each* data type declaration introduces 3 kinds of objects.

7.3.1 Inductive Data Types

An inductive data type represents a context-free grammar with a single non-terminal symbol. It uses one term constructor for every production of the grammar such that its terms capture exactly the words of the grammar. The operation on its terms is pattern-matching.

We usually want to work with arbitrary context-free grammars. That is possible by using a set of mutually recursive inductive data types—one for each non-terminal.

Type Declaration

The declaration of an inductive data type looks as follows:

$$Decl ::= \textbf{data}\ a \ = \ Con \mid \ \ldots \ \mid Con$$

$$Con ::= c(A, \ldots, A)$$

$$a, c ::= \text{name}$$

Given **data** $a \ = \ Con_1 \mid \ \ldots \ \mid Con_n$, a is the **name** of the data type and each Con_i is a **constructor**. For a constructor $c(A_1, \ldots, A_n)$, c is the name of the constructor and the A_i are its **argument types**.
The rule for checking a data type declaration looks as follows:

$$\frac{\overbrace{\Gamma, \textbf{type}\ a \vdash A_i \ : \ \texttt{type}}^{\text{for } i=1,\ldots,n}}{\Gamma \vdash \textbf{data}\ a \ = \ c_1(A_1) \mid \ \ldots \ \mid c_n(A_n)\ \checkmark}$$

Here, for simplicity, the typing rule only covers the case where every constructor takes exactly 1 argument. The general case works accordingly (or can be obtained by using the unit type or a product type as the argument type).
Note that the data type a is already assumed to exists when checking the types of the constructors. That is critical to allow writing recursive data types such as:

Example 7.1. The standard example of an inductive data type are the natural numbers. We obtain them as

$$\textbf{data}\ nat \ = \ zero() \mid succ(nat)$$

The Type Provided by an Inductive Data Type

Once declared, the name of the data type is a well-formed type:

$$\frac{\textbf{data}\ a \ = \ \ldots \in \Gamma}{\Gamma \vdash a \ : \ \texttt{type}}$$

The correspondence between grammars and inductive data types is as follows:

Grammar	Inductive Data Type
start symbol	not needed
non-terminal N	inductive data type declaration with name N
production $N ::= T_0 N_0 T_1 \ldots T_{n-1} N_{n-1} T_n$	constructor for N with argument types N_i
name for production (must be invented)	name of constructor
name for occurrence N_i of non-terminal symbol in production (must be invented)	selector
terminal symbols T_i in production	ignored

Terms of an Inductive Data Type

The constructors form elements of the type a:

$$t ::= c(t, \ldots, t)$$

$$\frac{\textbf{data}\ a \ = \ c_1(A_1) \mid \ \ldots \mid c_n(A_n) \ \ \in \ \ \Gamma \qquad \Gamma \vdash t \ : \ A_i}{\Gamma \vdash c_i(t) \ : \ a}$$

Again we only use constructors with exactly 1 argument.

Terms Formed From Terms of Inductive Data Type

To operate on a term of an inductive type, we use **pattern-matching**.

$$t ::= \textbf{match}\ t\ \{Cas | \ldots | Cas\}$$

$$Cas ::= c(x_1, \ldots, x_n) \mapsto t$$

In **match** $t\ \{Cas_1 | \ldots | Cas_n\}$, t is the matched term and the Cas_i are the **cases**. In a case $c(x_1, \ldots, x_n) \mapsto t$, we call $c(x_1, \ldots, x_n)$ the **pattern** where c is a constructor name and the x_i are variables. t is called the **body** of the case.

Finally, the typing rule is (again assuming all constructors take exactly 1 argument):

$$\frac{\textbf{data}\ a\ =\ c_1(A_1)\mid\ \ldots\ \mid c_n(A_n)\ \ \in\ \ \Gamma\qquad \Gamma\vdash t\ :\ a\qquad \overbrace{\Gamma, \textbf{val}\ x : A_i \vdash t_i\ :\ B}^{\text{for } i=1,\ldots,n}}{\Gamma\vdash \textbf{match}\ t\ \{c_1(x)\mapsto t_1 \mid \ldots \mid c_n(x)\mapsto t_n\}\ :\ B}$$

7.3.2 Abstract Data Types

Inductive and abstract data types are dual to each other.

An inductive data type is defined bottom-up: the terms are exactly the *syntax trees* of the words over some context-free grammar. That makes it easy to build terms—we just apply one of the constructor. But it makes it difficult to operate on a term—we have to pattern-match for all possible cases.

An abstract data type is defined top-down by its *behavior*: any object that exhibits the required behavior is a term of the type. That makes it easy to operate on a term—we just apply one of the required behaviors. But it makes it difficult to build terms—we have to implement all required behaviors.

Type Declaration

The declaration of an abstract data type looks as follows:

$$Decl ::= \textbf{abstract class}\ a(x : A, \ldots, x : A)\{Field, \ldots, Field\}$$

$$Field ::= f : A$$

$$a, f, x ::= \text{name}$$

Given **abstract class** $a(x_1 : A_1, \ldots, x_n : A_m)\{Field_1, \ldots, Field_n\}$, a is the **name** of the data type and each $Field_i$ is a **field**. The $x_i : A_i$ are called the **constructor arguments**. For a field $f : A$, f is the name of the field and A its type.

The rule for checking the declarations is as follows:

$$\frac{\overbrace{\Gamma, \textbf{type}\ a \vdash A_i\ :\ \texttt{type}}^{\text{for } i=1,\ldots,m}\qquad \overbrace{\Gamma, \textbf{type}\ a \vdash B_i\ :\ \texttt{type}}^{\text{for } i=1,\ldots,n}}{\Gamma\vdash \textbf{abstract class}\ a(x_1 : A_1, \ldots, x_m : A_m)\{f_1 : B_1, \ldots, f_n : B_n\}\ \checkmark}$$

Like for inductive data types, the new type a is already assumed when checking the types of the constructor arguemtns and the fields. That is critical to describe many practical examples:

Example 7.2. Standard examples of abstract data types tend to be polymorphic.

One of the standard examples of abstract data types is an automaton. In this example, we use numbered states and strings as the input and output of a transition:

$$\textbf{abstract class}\ automaton(state : int)\{inFinalState : bool,\ transition : string \to string\}$$

The constructor argument *state* provides the initial state as well as the private variable that tracks the current state. In concrete implementations (see also Rem. 7.3), the values of $inFinalState$ and $transition$ may depend on and change the value of *state*.

Such an automaton may or may not be finite—that depends on which states are reachable from the initial state by applying transitions.

Remark 7.3 (Duality). Intuitively, inductive data types start with nothing and build all objects inductively by applying constructors. (We do not even need a base case—instead, we use constructors that take 0 arguments.) Therefore, there is wide agreement on their intuition, syntax, and semantics (because it is easy to agree on what *nothing* means). In other words, by default we assume objects are not members of an inductive data type; only those whose membership can be proved by applying constructors to build them are allowed.

Abstract data types are dual in the sense that they start with *everything* and discard all objects that show the wrong behavior. In other words, by default we assume all objects are members of an inductive data type; but we only disallow those membership can be disproved by applying fields to obtain an illegal behavior.

But it is much harder to agree on what *everything* is. Therefore, intuition, syntax, and semantics of abstract data types can vary widely across languages. Particular incarnations include coalgebras, coinductive data types, and object-orientation-like classes.

This has an important consequence on the relation between specification and implementation. If a specification includes an inductive data type, every implementation must include the same data type with the same meaning. (If the specification and the implementation are written in the same language, e.g., by using SML signatures and SML structures, respectively, this can even get awkward because the exact declaration must be repeated twice.)

But if a specification includes an abstract data type, the choice of implementation language can substantially affect what objects can be built. We often write the specification in a logic and the implementation in a programming language. Then it is possible that the abstract data type is very weak in the logic: the logic might be able to define no or only trivial behavior. For example, the type theory of this section cannot define many objects of the abstract data type of automate from Ex. 7.2. The same abstract data type in the programming language may be much larger because the programming can express so much more behavior.

That is not a defect. It shows the abstraction at work: objects of an abstract data type are black boxes that we can never inspect, and how the behavior is defined is completely open and irrelevant. No language ever knows what kind of objects an abstract data type has. So even if a logic cannot define any object of an abstract data type, neither can it prove that the type is empty.

The Type Provided by an Abstract Data Type

Like for inductive data types, once declared, the name is a new type:

$$\frac{\textbf{abstract class}\ a(\ldots)\{\ldots\} \quad \in \quad \Gamma}{\Gamma \vdash a\ :\ \texttt{type}}$$

Remark 7.4 (Relation to Object-Orientation). Object-oriented programming languages provide a very rich formalism for defining abstract data types.

Our variant here is a special case that arises if we make the following restrictions about classes:

- There is no inheritance.
- There is exactly one constructor.
- The constructor arguments are exactly the private variables of the class.
- All methods are public and abstract.
- When creating a new instance, all methods must be implemented.

This is obviously too restrictive for programming. But it is enough to explain the theory systematically. It is straightforward to relax these restrictions when defining practical languages.

There is a correspondence between certain kinds of machines and abstract data types. We do not spell out the formal details of the machines here and only sketch the correspondence informally:

Machine	Abstract Data Type
set of possible states	product $A_1 \times \ldots \times A_n$ of constructor argument types
initial state	constructor arguments $(s_1, \ldots, s_n)$
current state	current tuple of private variables $(x_1, \ldots, x_n)$
query about current state (e.g., being final)	apply field of non–function type
f-transition (push button labeled f)	apply field f of function type
possible inputs for transition f	argument types of f
possible outputs of transition f	return type of f

Terms of an Abstract Data Type: New Instance Creation

Terms of an abstract data type a are created using the **new** operator. They are also called **instances** of a.

$$t ::= \textbf{new}\ a(t, \ldots, t)\{Def, \ldots, Def\}$$

$$Def ::= f = t$$

In **new** $a(t_1, \ldots, t_n)\{Def_1, \ldots, Def_n\}$, the t_i are the **constructor arguments**, and the Def_i are the **field definitions**.

Again, we state the typing rule only for a special case: we assume that the constructor takes exactly 1 argument. The general case works accordingly.

$$\frac{\textbf{abstract class}\ a(x : A)\{f_1 : A_1, \ldots, f_n : A_n\} \ \in\ \Gamma \quad \Gamma \vdash t\ :\ A \quad \overbrace{\Gamma \vdash t_i\ :\ A_i}^{\text{for } i=1,\ldots,n}}{\Gamma \vdash \textbf{new}\ a(s)\{f_1 = t_1, \ldots, f_n = t_n\}\ :\ a}$$

Terms Formed From a Term of an Abstract Data Type: Field Access

To operate on a term of an abstract data type, we access its fields:

$$t ::= t.f$$

$$\frac{\textbf{abstract class}\ a(\ldots)\{f_1 : A_1, \ldots, f_n : A_n\} \ \in\ \Gamma \quad \Gamma \vdash t\ :\ a}{\Gamma \vdash t.f_i\ :\ A_i}$$

7.3.3 Polymorphic Data Types

Data types are usually allowed to be polymorphic.

Therefore, the grammar must actually allow all declarations and all references to constructors and fields to take type parameters.

We omit the rules and only list the resulting productions:

data type declarations	
$Decl ::= \textbf{data}\ a[a^*] = Con^*$	inductive
$\mid\ \textbf{abstract class}\ a[a^*]((x : A)^*)\{Field^*\}$	abstract
types (same productions as before)	
$A ::= a[A^*]$	data type applied to type parameters
building and using terms	
$t ::= c[A^*](t^*)$	constructor application
$\mid\ \textbf{match}\ t\ \{Cas^*\}$	pattern-match
$\mid\ \textbf{new}\ a(t^*)\{Def^*\}$	new instance
$\mid\ t.f[A^*]$	field access
auxiliary non-terminals	
$Con ::= c(A^*)$	constructor declaration
$Cas ::= c(x^*) \mapsto t$	case in pattern-match
$Field ::= f : A$	field declaration
$Def ::= f = t$	field definition in new instance

Example 7.5 (Lists and Trees). The standard example of a polymorphic inductive data type are lists over a type a:

$$\textbf{data}\ List[a]\ =\ nil \mid cons(a, List[a])$$

Trees are only slightly more complex, e.g., for complete binary trees whose leafs are labeled with elements of a:

$$\textbf{data}\ Tree[a] = leaf(a) \mid node(Tree[a], Tree[a])$$

Example 7.6 (Iterators and Streams). The standard example of a polymorphic abstract data type are iterators (also called streams) over a type a:

$$\textbf{abstract class}\ Iterator[a](\{hasNext : bool,\ next : a\}$$

Technically, the return type of a should $a^?$ (i.e., options of a), but it is commonly agreed that the behavior of *next* is unspecified if no next element is left.
If next always returns another element, we obtain streams, i.e., infinite sequences:

$$\textbf{abstract class}\ Stream[a](\{next : a\}$$

Like for automata, we have to generalize the language to allow for defining more interesting objects. In particular, new instances should be allowed to have additional private variables, e.g., to track the current position in the iterator/stream.

7.3.4 Inheritance

Inheritance generalizes abstract data types in two ways:

- Fields may already have a definition. Such fields are not implemented anymore when creating a new instance.
- We can form a new data type by extending an existing one, which has the effect of copying over all its fields.

Both generalizations can be seen as a matter of convenience. However, by fixing certain definitions in an class, we gain enormous power to fine-tune the desirable behavior of the instances.

Remark 7.7 (Inheritance for Inductive Data Types). Inheritance works exactly the same way for inductive data types: fields with definition are skipped when pattern-matching, and new types can reuse the set of constructors of an existing type.
However, this is not used in practice.[2]

Inductive Data Types Via Classes

Another advantage of inheritance is that we can mimic inductive data types.
The correspondence is as follows:

Inductive Data Type	Classes
type declaration	abstract class without constructor arguments or fields
constructor	concrete extension
constructor arguments	constructor arguments of concrete extension
constructor application	new instance
pattern-matching	combination of if-then-else and is-instance-of

Thus, we can mimic the inductive type

$$\textbf{data}\ a = \ldots \mid c_i(A_1, \ldots, A_n) \mid \ldots$$

as

$$\textbf{abstract class}\ a()$$

$$\vdots$$
$$\textbf{class}\ c_i(x_1 : A_1, \ldots, x_n : A_n)\ \textbf{extends}\ a$$

⋮

Constructing a term $c_i(t_1, \ldots, t_n)$ corresponds to **new** $c_i(t_1, \ldots, t_n)$. Equality at a must be implemented such that $x == y$ iff x and y are instances of the same class c_i and created with equal constructor arguments.
Pattern-matching must be implemented manually.

7.4 Implementation

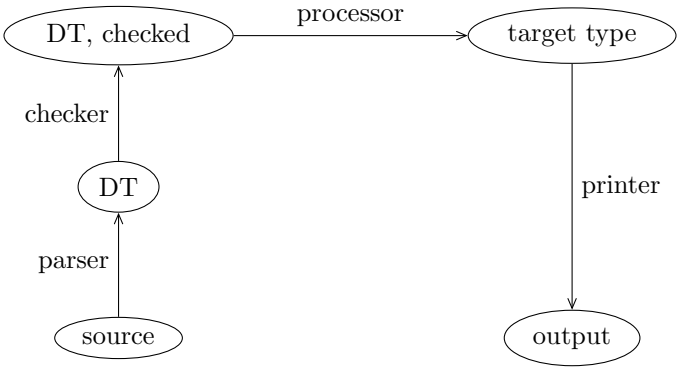
Formal systems can be implemented very systematically in any programming language that supports inductive data types.
The design is as follows:

1. Define inductive types DT that represents the context-free grammar.
2. Define parsing and printing functions that translate between *string* and DT.
3. Define checking functions that check whether DT-objects are well-formed.
4. Define processing functions that take DT-objects. In particular,
   - interpreters or evaluators turn objects into their values,
   - static analysis tool perform additional checks or statistics,
   - compilers transform objects into equivalent objects in a different formal system.

The following table gives an overview:

Component ...	contains for each concept/non-terminal N one ...
data type	inductive data type N
printer	function $printN(o : N) : string$
parser	function $parseN(input : string) : N$
checker	function $checkN(o : N) : bool$

In each component, the respective functions are usually mutually recursive.
The processing pipeline is always



7.5 Evaluation

7.5.1 Overview

The framework from Sect. 7.1 still has to be extended by one aspect: we need an additional judgment

$$\Gamma \vdash t \rightsquigarrow v$$

that describes how a term t is evaluated to a value v. This judgment is called **evaluation**. For example, we expect a rule like $\Gamma \vdash 1 + 1 \rightsquigarrow 2$.

From the perspective of programming languages, we can think of terms as programs and of evaluation as running the program. From the perspective of logic, we can think of the evaluation rules as a set of axioms that describe the equality relation between objects.

Type theories can very quite drastically in how evaluation works. Everything can vary including the abstract shape of the judgment, its notation, and its rules. Often differences between formal systems manifest themselves in the details of this judgment.

In particular, the treatment of function application has received a lot of attention.

7.5.2 Typing vs. Evaluation

Primarily we evaluate *terms* to *values*. Values are a subset of the terms that can no longer be simplified. For example, all the literals of the base types are values.

Because terms occur inside declarations and contexts, we also have to evaluate those, usually by recursing into the terms. Moreover, depending on the specific type theory used, we may also have to evaluate types. For example, if we allow type definitions, we have to expand those evaluation. However, the evaluation of types is very simple. If it becomes too difficult (e.g., non-terminating, undecidable, or with side-effects), the formal system quickly becomes unmanageable.

Both typing and evaluation typically consist of one judgment per non-terminal and one rule per production. Thus, both describe a family of mutually recursive functions on the grammar. This yields the following table:

non-terminal	typing judgment	evaluation judgment	
Γ	$\vdash \Gamma$ ✓	$\vdash \Gamma \rightsquigarrow \Gamma'$	Γ evaluates to Γ'
D	$\Gamma \vdash D$ ✓	$\Gamma \vdash D \rightsquigarrow D'$	declaration D evaluates to D'
A	$\Gamma \vdash A$: `type`	$\Gamma \vdash A \rightsquigarrow A'$	type A evaluates to A'
t	$\Gamma \vdash t : A$	$\Gamma \vdash t \rightsquigarrow t'$	term t evaluates to t'

Because we think of evaluation as running a program, evaluation is the primary *dynamic* operation. Typing is the primary *static* operation. Like all dynamic operations, evaluation always happens *after* type-checking and is only applied to well-formed objects.

Correspondingly, typing errors are called **static** or **compile-time** errors, and evaluation errors are called **dynamic** or **run-time** errors.

As a general rule, the evaluation of well-formed terms should not cause errors, i.e., we want to statically verify that there will be no run-time errors.

In practice, however, some run-time errors must be accepted, e.g., errors that are

- intentionally generated by the programmer (i.e., correct program behavior that happens to take the form of an abort)
- caused by interaction with the environment such as hardware or network failures or missing access rights to files
- caused by missing dependencies (e.g., type-checking against a library that at run-time is missing or present in a different version)

Moreover, as we will see in Sect. 8.1, sufficiently complete formal systems such as programming languages may make the absence of run-time errors undecidable.

7.5.3 Basic Rules

The rules for the empty formal system are

- Contexts are evaluated in order by evaluating every declaration::

$$\frac{}{\vdash \varepsilon \rightsquigarrow \varepsilon} \qquad \frac{\vdash \Gamma \rightsquigarrow \Gamma' \quad \Gamma' \vdash D \rightsquigarrow D'}{\vdash \Gamma, D \rightsquigarrow \Gamma', D'}$$

  Note how every declaration D is evaluated in the context, in which all previous declarations have been evaluated already.

- Declarations are evaluated by evaluating every object in them:

$$\frac{\Gamma \vdash A \rightsquigarrow A'}{\Gamma \vdash \textbf{type}\ a = A \rightsquigarrow \textbf{type}\ a = A'} \qquad \frac{\Gamma \vdash A \rightsquigarrow A' \quad \Gamma \vdash t \rightsquigarrow t'}{\Gamma \vdash \textbf{val}\ a : A = t \rightsquigarrow \textbf{val}\ a : A' = t'}$$

Polymorphic declarations or declarations in which optional parts are absent are evaluated accordingly.

- Declared names evaluate to their definition if they have one and to themselves otherwise:

$$\frac{\textbf{type}\ a = A\ \in\ \Gamma \quad \Gamma \vdash A\ \rightsquigarrow\ A'}{\Gamma \vdash a\ \rightsquigarrow\ A'} \qquad \frac{\textbf{type}\ a\ \in\ \Gamma}{\Gamma \vdash a\ \rightsquigarrow\ a}$$

$$\frac{\textbf{val}\ x : A = t\ \in\ \Gamma \quad \Gamma \vdash t\ \rightsquigarrow\ t'}{\Gamma \vdash x\ \rightsquigarrow\ t'} \qquad \frac{\textbf{val}\ x : A\ \in\ \Gamma}{\Gamma \vdash x\ \rightsquigarrow\ x}$$

Note how the definiens is evaluated recursively here. This is usually redundant because all previous declarations are already evaluated, i.e., the definition will not change anymore. But occasionally, we use formal systems where this step is necessary.

7.5.4 Rules for Individual Language Features

For most language features—such as product and function types—evaluation simply recurses through the expressions. However, in a few key places non-trivial steps happen. Those few places are the reason why evaluation "makes something happen". Most importantly, this includes function application.

In the following, we give evaluation rules for important features. We omit data types—they are similar but more complex.

Types

Evaluation for types just straightforwardly recurses through the type. The expansion of definitions from above is the only non-trivial step:

$$\frac{\text{for base types } b}{\Gamma \vdash b\ \rightsquigarrow\ b}$$

$$\frac{\Gamma \vdash A\ \rightsquigarrow\ A' \quad \Gamma \vdash B\ \rightsquigarrow\ B'}{\Gamma \vdash A \times B\ \rightsquigarrow\ A' \times B'} \qquad \frac{\Gamma \vdash A\ \rightsquigarrow\ A' \quad \Gamma \vdash B\ \rightsquigarrow\ B'}{\Gamma \vdash A \to B\ \rightsquigarrow\ A' \to B'} \qquad \frac{\Gamma \vdash A\ \rightsquigarrow\ A' \quad \Gamma \vdash B\ \rightsquigarrow\ B'}{\Gamma \vdash A + B\ \rightsquigarrow\ A' + B'}$$

Terms Regarding Base Types

The literals trivially evaluate to themselves:

$$\frac{l \in R_b}{\Gamma \vdash l\ \rightsquigarrow\ l}$$

Operator applications evaluate to the result of calling the corresponding function in the underlying system. The underlying system can be another programming language but is ultimately the hardware on which evaluation is executed.

For example,

$$\frac{\Gamma \vdash s\ \rightsquigarrow\ i \quad \Gamma \vdash t\ \rightsquigarrow\ j \quad i, j \in \mathbb{Z}}{\Gamma \vdash s + t\ \rightsquigarrow\ i \oplus j}$$

where s and t evaluate to the integers i and j, and $i \oplus j$ is actual integer addition performed by the underlying system. Note that here $s + t$ is a term, i.e., $+$ is just a symbol, where $i \oplus t$ is the result of an operation.

Operator application is straightforward. However, to allow for generalizations to programming languages (where evaluation may run forever, fail, or have side-effects), we have to be careful about the order in which the arguments are evaluated.

The typical convention is to evaluate the argument left-to-right. For example, s is evaluated before t.

Of particular importance are the rules for the if-operator:

$$\frac{\Gamma \vdash c\ \rightsquigarrow\ \mathit{true} \quad \Gamma \vdash t\ \rightsquigarrow\ t'}{\Gamma \vdash \textbf{if}\ (c)\ \{t\}\ \textbf{else}\ \{e\}\ \rightsquigarrow\ t'} \qquad \frac{\Gamma \vdash c\ \rightsquigarrow\ \mathit{false} \quad \Gamma \vdash e\ \rightsquigarrow\ e'}{\Gamma \vdash \textbf{if}\ (c)\ \{t\}\ \textbf{else}\ \{e\}\ \rightsquigarrow\ e'}$$

These rules only evaluate the branch that is needed—either t or e. The other branch is discarded.
For example, if we define the boolean operators as in $s \wedge t := \textbf{if } (s) \ \{t\} \textbf{ else } \{\mathit{false}\}$, this yields short-circuit evaluation:

$$\frac{\Gamma \vdash s \ \rightsquigarrow \ \mathit{false}}{\Gamma \vdash s \wedge t \ \rightsquigarrow \ \mathit{false}} \qquad \frac{\Gamma \vdash s \ \rightsquigarrow \ \mathit{true} \quad \Gamma \vdash t \ \rightsquigarrow \ t'}{\Gamma \vdash s \wedge t \ \rightsquigarrow \ t'}$$

The equality operator presents a problem though when it is combined with functions: equality of functions is undecidable. There are two approaches to work around this problem:

- Functional languages tend to use typing rules that restrict equality to those types for which equality is decidable.
- Imperative languages tend to allow equality for all types but then implement it using reference equality. In that case, equality of functions will be true less often than it should be.

Terms Regarding Product Types

Pairs are evaluated by a simple recursion:

$$\frac{\Gamma \vdash s \ \rightsquigarrow \ s' \quad \Gamma \vdash t \ \rightsquigarrow \ t'}{\Gamma \vdash (s,t) \ \rightsquigarrow \ (s',t')}$$

Projections are more complicated because something actually happens: we retrieve a component from a pair using the rules

$$\frac{\Gamma \vdash u \ \rightsquigarrow \ (s,t)}{\Gamma \vdash u.1 \ \rightsquigarrow \ s} \qquad \frac{\Gamma \vdash u \ \rightsquigarrow \ (s,t)}{\Gamma \vdash u.2 \ \rightsquigarrow \ t}$$

If u does not evaluate to a pair, we simply recurse.

Terms Regarding Disjoint Union Types

Like pairs, the terms of the new type—the injections—are evaluated by a simple recursion:

$$\frac{\Gamma \vdash t \ \rightsquigarrow \ t'}{\Gamma \vdash \mathit{inj}_1(t) \ \rightsquigarrow \ \mathit{inj}_1(t')} \qquad \frac{\Gamma \vdash t \ \rightsquigarrow \ t'}{\Gamma \vdash \mathit{inj}_2(t) \ \rightsquigarrow \ \mathit{inj}_2(t')}$$

Like for projections, something happens when evaluating the terms that use a term of the new type: a case distinction is contracted if we know which case applies.

$$\frac{\Gamma \vdash u \ \rightsquigarrow \ \mathit{inj}_1(t) \quad \Gamma \vdash c_1(t) \ \rightsquigarrow \ t'}{\Gamma \vdash \mathit{cases}(u, c_1, c_2) \ \rightsquigarrow \ t'} \qquad \frac{\Gamma \vdash u \ \rightsquigarrow \ \mathit{inj}_2(t) \quad \Gamma \vdash c_2(t) \ \rightsquigarrow \ t'}{\Gamma \vdash \mathit{cases}(u, c_1, c_2) \ \rightsquigarrow \ t'}$$

Note that only the case that applies is evaluated—that is important in case there are side effects.
If u does not evaluate to an injection, we simply recurse.

Terms Regarding Function Types

Intuitively, the key evaluation rule relevant to product types is β-reduction: substitute the variable x in $(x : A) \mapsto t$ with s in order to evaluate $((x : A) \mapsto t)\, s$. However, the details present considerable difficulties.
Historically very important was the choice between call-by-name and call-by-value evaluation. **Call-by-value** evaluates the argument of the function first, then substitutes it for x:

$$\frac{\Gamma \vdash f \ \rightsquigarrow \ (x : A) \mapsto t \quad \Gamma \vdash s \ \rightsquigarrow \ s' \quad \Gamma \vdash t[x/s'] \ \rightsquigarrow \ u}{\Gamma \vdash f(s) \ \rightsquigarrow \ u}$$

where $t[x/s]$ is the result of substituting x with s in t. **Call-by-name** first substitutes, then evaluates the result:

$$\frac{\Gamma \vdash f \ \rightsquigarrow \ (x : A) \mapsto t \quad \Gamma \vdash t[x/s] \ \rightsquigarrow \ u}{\Gamma \vdash f(s) \ \rightsquigarrow \ u}$$

Without side-effects, both yield the same result. Call-by-name is usually less efficient because s has to be evaluated multiple times if x occurs multiple times in t. But if side-effects are possible, the order matters; moreover, evaluating s multiple times may be exactly what the programmer wants.

Another problem is the evaluation of anonymous functions. The intuitive rule

$$\frac{\Gamma \vdash A \rightsquigarrow A' \quad \Gamma, x : A' \vdash t \rightsquigarrow t'}{\Gamma \vdash (x : A) \mapsto t \rightsquigarrow (x : A') \mapsto t'}$$

does not work in the presence of side-effects: The side-effects of t should not be triggered in this rule, when the function is *constructed*. Instead, they should occur be triggered later when the function is *called*.

On the other hand, we cannot delay the entire evaluation of an anonymous function until call time. Consider the following program:

$$\textbf{val } v : int = 1 \quad ; \quad \textbf{val } f : int \to int = (x : int) \mapsto x + v \quad ; \quad \textbf{val } v : int = 2 \quad ; \quad \textbf{print } f(1)$$

Here the reference to v must be evaluated before $f(1)$ is applied: otherwise, f would be moved into a different environment in which v refers to a different declaration. Therefore, programming languages have to perform **closures** in which an anonymous function is made independent of its environment.

The following is a simple definition of closure:

Definition 7.8 (Closure)**.** Assume that Γ are fully evaluated. In particular, all declarations in Γ have a definition that does not refer to any previously declared names.

Then given an object X in context Γ, we define $Closure_\Gamma(X)$ as the object that arises from X by replacing every name with its definition.

Then we can use the following rule:

$$\frac{A' = Closure_\Gamma(A) \quad t' = Closure_\Gamma(t)}{\Gamma \vdash (x : A) \mapsto t \rightsquigarrow (x : A') \mapsto t'}$$

This rule resolves all references but does not apply any evaluation rules to t. The resulting function does not refer to any names anymore and can therefore be safely transported into another context.

Many formal systems are simple enough so that the closure of a type is the type itself, i.e., only the terms have to be closed.

Chapter 8

Programming

The framework from Ch. 7 can be applied systematically to build programming languages. Then checking provides the first step of static analysis, and evaluation specifies an interpretation algorithm that runs the programs. In fact, we can systematically read off the implementation of a type checker and an interpreter from the rule systems for typing and evaluation.

8.1 Decidability vs. Soundness vs. Completeness

8.1.1 Definitions

Three properties are desirable for building formal systems:

Decidability states that all well-formedness judgments (i.e., all judgments except possibly evaluation) are decidable. That allows us to tell statically and algorithmically whether an object is well-formed.

Soundness states all well-formed objects are meaningful. The details of this can very across formal systems. But usually it subsumes the requirement that every term t of type A can be semantically interpreted as an element $\overline{t}$ of some set $\overline{A}$.

- In logic, soundness means that all proofs give rise to valid formulas, i.e., theorems.
- In programming, it means that every *function term* $f : A \to B$ should yield an actual function $\overline{t}$.

Completeness states that every meaningful value can be obtained as the interpretation of some term of the formal system. Again the details vary.

- In logic, completeness means that all valid formulas (i.e., all semantically possible theorems) can be obtained as the interpretation of a proof in the formal system, i.e., the formal system can prove every theorem.
- In programming, completeness means that all computable functions (i.e., all algorithmically possible function values) can be obtained as the interpretation of some function term in the formal system, i.e., the formal system can define all functions. That is usualyl called Turing-completeness.

8.1.2 Mutual Exclusivity

Unfortunately, the three properties cannot be realized at once (except in degenerate, trivial cases).

That is easy to prove:

Theorem 8.1. *No formal system that interprets function terms as total functions is decidable, sound, and complete.*

Proof. Assume such a system.

Due to completeness, every computable function can be written as a function term. Due to decidability, we have an algorithm to decide whether such terms are well-formed. Due to soundness, all well-formed terms yield terminating functions. (Otherwise, the functions would be partial.)

Taking together, we could decide the halting problem, in violation of the known result that it is undecidable. □

Therefore, every formal system must sacrifice one of the three properties. This yields a somewhat rough but very intuitive classification of formal systems:

- Type theories sacrifice (Turing-)completeness. Thus, we can check and evaluate all terms but cannot implement all computable functions.
- Programming languages sacrifice soundness. Thus, we can implement and check all functions, but we have no guarantee whether a function terminates.
- Logics sacrifice decidability. We can describe every computable function (as a set of axioms). But we have no algorithm for checking whether such a specification actually defines a function.

Thus, to turn a type theory into a programming language, we have to allow writing well-formed terms whose evaluation does not terminate. The most commonly used features are recursion or while-loops.

8.2 Programs as Terms

The easiest way to extend type theory to programming is to add a new concept for programs with a non-terminal P.
We could then have productions like

new declarations		
$Decl$	$::= \mathbf{var}\ x : A = t$	mutable variables
programs		
P	$::= P; P$	sequencing
	$\mid\ x = t$	assignments to variables
	$\mid\ \mathbf{while}\ t\ \{P\}$	loops
	$\mid\ \mathbf{print}\ P$	user output
	$\mid\ p(t)$	procedure calls
new terms		
t	$::= x$	variable reference
	$\mid\ \mathbf{read}$	user input

Together with data types, this already yields a pretty convenient programming language.
However, the distinction between terms and programs is not always convenient: it quickly leads to a duplication of language feature. For example, we need two productions $t ::= \mathbf{if}\ (t)\ \{t\}\ \mathbf{else}\ \{t\}$ and $P ::= \mathbf{if}\ (t)\ \{P\}\ \mathbf{else}\ \{P\}$. Similarly, procedure calls are essentially the same as function applications except that they do not return a value. Down the line, when we design the rules and implement checking and evaluation, that duplication causes a substantial overhead.

Therefore, it is appealing to treat programs as terms. That means we conflate the non-terminals P and t. But there is one problem: terms must have a type and evaluate to a value. We can solve that problem using the *unit* type: all program terms have type *unit*.
Using the *unit* type in this way is common practice in functional language. In imperative languages, it is (deplorably) not common even though it is very simple and elegant.

8.3 Language Features

Many programming language features can be described as terms given by new productions for terms and new typing and evaluation rules for them.

8.3.1 Sequencing

Grammar

Sequencing is very simple: first do s, then do t, and return the result of the latter t.

$$t ::= t; t$$

If bracketing is necessary to be unambiguous, we usually use $\{s; t\}$. It is also commonly agreed that ; should be associative, i.e., we can write $\{t_1; \ldots; t_n\}$ without further bracketing.
The notation $\{s; t\}$ is very similar to the notation $\{D; t\}$ for local declarations. Indeed, many programs have to alternate between declarations and commands.
Thus, it is convenient to use a production $t ::= (Decl|t); t$. Alternatively, we can use a production $Decl ::= t$ for anonymous declarations.

Typing

Even though the only the last program step determines the type of the overall sequence, we have to check each step to make sure the program is well-formed:

$$\frac{\Gamma \vdash s \ : \ A \quad \Gamma \vdash t \ : \ B}{\Gamma \vdash s; t \ : \ B}$$

Evaluation

Evaluation proceeds in-order in a straightforward fashion:

$$\frac{\Gamma \vdash s \ \rightsquigarrow \ s' \quad \Gamma \vdash t \ \rightsquigarrow \ t'}{\Gamma \vdash s; t \ \rightsquigarrow \ t'}$$

Again it is important to evaluate s first even though s' is discarded. That way we trigger all side-effects of s. Moreover, if s does not terminate, the evaluation of t never starts.

8.3.2 Loops

While-loops are a key ingredient to allow for non-termination.

Grammar

$$t ::= \textbf{while} \ t \ \{t\}$$

Typing

The type of a while-loop is trivially the unit type. So we only have to type-check all subterms and ignore their types:

$$\frac{\Gamma \vdash c \ : \ bool \quad \Gamma \vdash t \ : \ A}{\Gamma \vdash \textbf{while} \ c \ \{t\} \ : \ unit}$$

The type of the body does not matter. But it is usually a programming error if $A \neq unit$. Therefore, we might alternatively enforce $A = unit$.

Evaluation

Evaluation of a while-loop either does not terminate or yields the unit term:

$$\frac{\Gamma \vdash c \ \rightsquigarrow \ true \quad \Gamma \vdash t \ \rightsquigarrow \ t' \quad \Gamma \vdash \textbf{while} \ c \ \{t\} \ \rightsquigarrow \ u}{\Gamma \vdash \textbf{while} \ c \ \{t\} \ \rightsquigarrow \ u} \qquad \frac{\Gamma \vdash c \ \rightsquigarrow \ false}{\Gamma \vdash \textbf{while} \ c \ \{t\} \ \rightsquigarrow \ ()}$$

Here the left rule may look circular. But it is not because the premises must be evaluated left-to-right: the rule performs one iteration of the loop and then revisits the loop.

8.3.3 State and Assignments

Assignments require a whole new primitive concept: state. We must allow for variables whose definition changes during evaluation. There is a large amount of theoretical research on this issue, and we only consider a very simple solution here.

Grammar

Programming languages may choose to unify immutable and mutable variables. Then immutable variables are simply mutable variables that we happen to never assign a new value to. That simplifies the grammar and some implementation aspects.

However, for purposes of static analysis it can be much better to limit the impact of state. Statefulness is notoriously difficult to handle formally and often the source of errors. Moreover, in well-written programs, only very few names actually have to be mutable.

Therefore, we use two different declarations. However, we use the same letter x to represent both kinds of names.

$$\begin{aligned} \mathit{Decl} &::= \mathbf{var}\ x : A = t \\ t &::= x = t \mid x \end{aligned}$$

Typing

The type of an assignment is trivially the unit type.

Variable declarations are well-formed if the initial value has the right type.

$$\frac{\Gamma \vdash t\ :\ A}{\Gamma \vdash \mathbf{var}\ x : A = t\ \checkmark}$$

Imperative programming languages sometimes often allow uninitialized variables. That is a bad idea.

Assignments are well-formed if the assigned term has the right type:

$$\frac{\mathbf{var}\ x : A = \_ \in \Gamma \qquad \Gamma \vdash t\ :\ A}{\Gamma \vdash x = t\ :\ \mathit{unit}}$$

Imperative programming languages sometimes use the type A as the type of the assignment. That makes some programs shorted, but has the huge drawback of encouraging the use of assignments inside terms, which is often the source of errors.

Finally, references to mutable variables are treated in the same way as the immutable case:

$$\frac{\mathbf{var}\ x : A = \_ \in \Gamma}{\Gamma \vdash x\ :\ A}$$

Evaluation

The evaluation rules for variables are difficult because assignments change the value of a variable—something we cannot capture well with context-sensitive inference rules. We have to introduce a whole new formal object: the **environment**.

The environment is a global object that maintains all state throughout program evaluation. Theoretical models of state and environment abound but none is fully convincing in all desirable ways. Therefore, we use a very simple and general definition that captures the main idea:

Definition 8.2 (Environment)**.** A **location** consists of a number $l \in \mathbb{Z}$ and a term t. An **environment** E is a set of differently-numbered locations.

We define the following operations on an environment E:

Operation	Effect	Return value
$add(E, t)$	add a new location to E containing t	the number of the new location
$get(E, l)$	none	the term in location l of E
$update(E, l, t)$	update the term in the location l of E with t	none
$delete(l)$	remove the location l from E	none

We can think of locations l as positions in the tape of a Turing machine or as memory locations in a modern computer. The latter is how the environment is implemented in modern programming languages.

Now we say that during evaluation there is exactly one environment E, and evaluation rules can perform operations on E. Moreover, we allow the numbers of locations to be used as terms:

$$t ::= l \text{ for } l \in \mathbb{Z}$$

It is understood that these terms are used internally during the evaluation. Therefore, we give no typing rule for them—thus, any program containing locations is automatically ill-formed.

Now we can formulate the evaluation rules for mutable variables as follows:

- A variable declaration creates a new location and uses its number as the value of the variable:

$$\frac{\Gamma \vdash A \rightsquigarrow A' \quad \Gamma \vdash t \rightsquigarrow t' \quad l = add(E, t')}{\Gamma \vdash \textbf{var}\ x : A = t \rightsquigarrow \textbf{var}\ x : A' = l}$$

- Assignments change the value in the location and return the unit term:

$$\frac{\Gamma \vdash t \rightsquigarrow t' \quad \textbf{var}\ x : \_ = l \in \Gamma \quad update(E, l, t')}{\Gamma \vdash x = t \rightsquigarrow ()}$$

- Variable references evaluate to the term currently stored in the location:

$$\frac{\textbf{var}\ x : \_ = l \in \Gamma \quad t = get(E, l)}{\Gamma \vdash x \rightsquigarrow t}$$

  We do not have to evaluate t anymore because we only store fully evaluated terms in locations in the first place.

Memory Deallocation

The above evaluation rules use only three of the four operations provided by the environment: we never delete a location. Theoretically, we never have to delete locations because the environment can simply hold on to all locations. We could then discard the environment as a whole after evaluation.

But that is problematic for programs that create many locations, especially programs like servers that are meant to run permanently. Therefore, programming languages have to delete locations from time to time. There are two approaches:

- Explicit **deallocation** uses special commands such as

  $$t ::= dealloc(x)$$

  with the evaluation rule

  $$\frac{\textbf{var}\ x : \_ = l \in \Gamma \quad delete(E, l)}{\Gamma \vdash dealloc(x) \rightsquigarrow ()}$$

  This approach is taken by languages in the C-family. It is problematic because it can be impossible to give typing rules that ensure that a deallocated variable is indeed never referenced again. Moreover, programmer are prone to forger deallocation, leading to what is called *memory leaks*.

- **Garbage collection** is an automated process that runs in the background and automatically deletes inaccessible locations. Here a location l is inaccessible if no declaration is in scope anymore in which l occurs. This approach is taken by Java, where garbage collection is provided by the virtual machine.

Garbage collection is preferable from a dependability perspective because it eliminates a source of errors. Its only caveat is that it typically runs at unpredictable times. This can be problematic for reliability because it becomes harder to give short-term timing guarantees.

8.3.4 Input and Output

Ultimately, a programming languages is practical only it it allows for input and output. Intuitively, input and output are straightforward, but the evaluation rules put additional complexity on the environment.

Grammar

We only consider console I/O and only use two simple commands for input and output:

$$t ::= \textbf{print}\ t \mid \textbf{read}$$

Typing

We only consider I/O of integers in principle any term (especially strings) could be written or read. Then we can use the rules:

$$\frac{\Gamma \vdash t\ :\ \mathit{int}}{\Gamma \vdash \textbf{print}\ t\ :\ \mathit{unit}}$$

$$\frac{}{\Gamma \vdash \textbf{read}\ :\ \mathit{int}}$$

Evaluation

We amend Def. 8.2 as follows:

Definition 8.3. An environment supports I/O if it contains two special locations IN and OUT (usually called *standard input* and *standard output*), each holding a list of integer literals.
We define the following operations:

Operation	Effect	Return value
$in()$	remove the first element from the list in location IN of E	that element
$out(t)$	append t to the list in location OUT of E	none

Then the evaluation rules simply defer to the I/O-supporting environment:

$$\frac{\Gamma \vdash t\ \rightsquigarrow\ i \qquad \mathit{out}(i)}{\Gamma \vdash \textbf{print}\ t\ \rightsquigarrow\ ()} \qquad \frac{i = \mathit{in}()}{\Gamma \vdash \textbf{read}\ \rightsquigarrow\ i}$$

If IN should be empty, we must further specify whether the evaluation of **read** waits until IN is non-empty or yields a special END-OF-INPUT value (e.g., the number 4 when interpreting integers as Unicode characters).

8.4 Recursion

Contrary to the features from the previous section, recursion is a new declaration.

8.4.1 Single Recursion

Grammar

Rules for recursive functions can be obtained relative easily. We introduce a new declaration for **recursive** values, i.e., values whose definition may already use the currently-being-declared name.
Its production looks the same as normal value definitions:

$$\mathit{Decl} ::= \textbf{recval}\ x : A = t$$

Indeed, many programming languages just use one declaration. In other words, they allow every function to be recursive.
It is however good practice to strictly separate them to limit the impact of recursion.

Typing

Recursive declarations are checked as follows:

$$\frac{\Gamma \vdash A \ : \ \texttt{type} \qquad \Gamma, \mathbf{val}\ x : A \vdash t \ : \ A}{\Gamma \vdash \mathbf{recval}\ x : A = t\ \checkmark}$$

Here the existence of an x of type A is already assumed when checking the definition. That allows recursion.
Together with the if-operator, we can already express all computable functions:

Example 8.4. The following definition of the factorial function is a well-formed declaration in the empty context:

$$\vdash \mathbf{recval}\ fact : int \to int = (x : int) \mapsto \mathbf{if}\ (x \leq 0)\ \{1\}\ \mathbf{else}\ \{x \cdot fact(x-1)\}\ \checkmark$$

It clearly terminates, but that is not guaranteed by well-formedness. The following nonsensical definitions are also well-formed:

$$\vdash \mathbf{recval}\ foo : int \to int = (x : int) \mapsto foo(x)\ \checkmark$$

$$\vdash \mathbf{recval}\ bar : int = bar\ \checkmark$$

Evaluation

In principle, the existing evaluation rules still work in the presence of non-termination. However, formal systems may want to make some changes to carefully decide *when* non-termination occurs during the evaluation algorithm.
The simplest evaluation rule for recursive values corresponds to the typing rule:

$$\frac{\Gamma \vdash A \ \rightsquigarrow \ A' \qquad \Gamma, \mathbf{val}\ x : A' \vdash t \ \rightsquigarrow \ t'}{\Gamma \vdash \mathbf{recval}\ x : A = t \ \rightsquigarrow \ \mathbf{recval}\ x : A' = t'}$$

Thus, evaluating a recursive declaration does not cause non-termination.
Later declarations are checked with the full recursive declaration in the context. Thus, every reference to the name triggers a cascade of definition expansion–steps. Every such step corresponds to one iteration of the recursion.

Example 8.5. Consider the program Γ

$$\mathbf{recval}\ fact : int \to int = (x : int) \mapsto F(x),\ \mathbf{val}\ main : int = f(1)$$

where we abbreviate

$$F(x) \qquad := \mathbf{if}\ (x \leq 0)\ \{1\}\ \mathbf{else}\ \{x \cdot fact(n-1)\}$$

The evaluation of Γ first evaluates the first declaration. That terminates easily. Because there are no defined names in the context, the closure of the function is the same as the function.
Evaluating the second declaration leads to

$$\mathbf{recval}\ fact : int \to int = (x : int) \mapsto F(x) \vdash f(1) \ \rightsquigarrow ???$$

At this point evaluation proceeds as follows:

$$f(1) \rightsquigarrow ((x : int) \mapsto F(x))\,1 \rightsquigarrow F(1) \rightsquigarrow 1 \cdot fact(1-1) \rightsquigarrow 1 \cdot fact(0) \rightsquigarrow 1 \cdot F(0) \rightsquigarrow 1 \cdot 1 \rightsquigarrow 1$$

and thus

$$\mathbf{recval}\ fact : int \to int = (x : int) \mapsto F(x) \vdash \mathbf{val}\ main : int = f(1) \ \rightsquigarrow \ \mathbf{val}\ main : int = 1$$

Note how the evaluation rules for **if** and definition expansion work together in such a way that evaluation of $fact(1)$ terminates even though $fact$ is recursive.

8.4.2 Mutual Recursion

Recursion as describe above allows only one declaration to recurse into itself. More generally, we want to allow sets of mutually recursive declarations, i.e., a set of declarations that can refer to each other.
One option is to use groups of declarations like

$$Decl ::= \textbf{mutual}\ \{Decl^*\}$$

This is the approach taken by SML. To type-check **mutual** $\{D_1, \ldots, D_n\}$, we first check all types in all D_i, then we put the types into the context and check all definitions.
In the most general case, all declarations must be allowed to reference any other. This often happens in object-oriented languages, where classes distributed over multiple files must be able to refer to each other.

8.5 Control Flow Operators

The last remaining major programming language feature are control flow operators such as

- **return** t for exiting a function from anywhere inside its body,
- **break** for exiting a while-loop from anywhere inside its body,
- **throw** e for throwing an exception and exiting a try-block at any time during execution of its body.

All of these share the property that they jump to a different place in the programming, discarding the current state and restoring a previous state where execution continues. Contrary to a function call, execution never returns to the place we jumped away from.
It is much harder to give typing and evaluation rules for these operators, and programming language research invests substantial effort into it.

8.5.1 Typing

It is straightforward to give these operators a type: they all have type the empty *void*. That makes sense because they never return and therefore do not have a value from the perspective of the surrounding code.
We must avoid confusion between the following two groups of commands:

- Commands that return nothing like printing, assignment, and while-loop. Those have type *unit*.
- Commands that do not return like breaking, throwing, and returning. Those have type *void*.

However, giving rules that describe the well-formedness or the surrounding constructs (functions, while-loops, and try-blocks) is much harder.

8.5.2 Evaluation

Giving evaluation rules is very hard as well.
However, implementing an interpreter is surprising easy—if we use a programming language that has some similar control flow operator. For example, if we use a programming language that allows throwing exceptions, it is straightforward to implement **return** e, **break**, and **throw** e by using exceptions.

Chapter 9

Logic

9.1 Formulas

Like programming languages, logics add new concepts to type theory: formulas and proofs. Only proofs are fundamentally new and correspond very closely to programs.

9.1.1 Formulas as Terms

Formulas are almost already covered by type theory and part of all programming languages. Indeed, type theory already has the type *bool* with operators for equality and propositional connectives. However, logic goes beyond that by introducing the quantifiers $\forall$ and $\exists$.

Using the type *bool* for formulas is a simple solution in situations where we anyway use type theories and programming languages that have it already. The combination of function types and formulas as terms leads to what is called higher-order logic (HOL). This was the logic originally introduced by Church when developing the λ-calculus [Chu40].

Grammar and Typing

In principle, it is not difficult to add them to type theories as well, and many type theories do that to some extent. This requires just two productions

$$t ::= \forall x{:}A.\ t \mid \exists x : A.t$$

with corresponding typing rules

$$\frac{\Gamma, \mathbf{val}\ x : A \vdash t\ :\ bool}{\Gamma \vdash \forall x{:}A.\ t\ :\ bool} \qquad \frac{\Gamma, \mathbf{val}\ x : A \vdash t\ :\ bool}{\Gamma \vdash \exists x{:}A.\ t\ :\ bool}$$

Evaluation

We cannot extend the typing-evaluation pair of algorithms known from type theory and programming languages to logic: the evaluation of quantified formulas is undecidable.

More precisely, it is undecidable whenever the domain of quantification—the type A above—is infinite. If A is finite, we can (usually inefficiently) evaluate quantified formulas by testing the instances for every possible $x : A$. If A is infinite, testing can only evaluate universally quantified formulas to *false* (by finding some instance that is *false*) or existentially quantified ones to *true* (by finding some instance for which it is true). However, even if A is finite, it is usually too big to make evaluation by testing useful.

Actually, undecidable evaluation in logic is not all that new—we also found it in programming languages. Firstly, they allow for non-termination, which already makes evaluation undecidable. Secondly, programming languages and even many type theories routinely use the equality operator even though its evaluation is also undecidable in certain situations, e.g., for function types. However, while such undecidable behavior is accidental in programming and can be worked around, quantified formulas with undecidable evaluation are essential in logic.

Proving

To handle these formulas, we have to replace evaluation with an entirely new concept: proving.

Proving can be seen as a static variant of evaluation: we try to evaluate terms in the presence of *arbitrary unknown values*. Concretely, we assume that Γ contains undefined names as in **val** $x : A$. In logic, these names are usually called **free variables**. For example, to prove $\forall x{:}A.\ F$, we try to evaluate F in a context that declares **val** $x : A$.

Note that in executable programs, free variables are not allowed: every variable must have a concrete value.

If free variables are allowed, a formula F does not necessarily evaluate to *true* or *false* anymore. Instead, there are three options:

- Free variables can be eliminated, and F evaluates to *true*.
- Free variables can be eliminated, and F evaluates to *false*.
- F evaluates to some term that still contains free variables.

The distinction between the options is undecidable. Proving tries to establish that one of the former holds.

Proving may use evaluation rules but usually a new set of rules and possibly auxiliary non-terminals and judgments have to be invented. Together, these are called the **proof system** or **calculus**.

Implementations of such proof systems are called **theorem provers**. The practical ones are usually extremely sophisticated, often comprising $> 10^5$ lines of code. Still, the problem of finding a proof of a given formula is so complex that *fully automated* provers perform very poorly in practice.

Note that the problem of *finding* a proof has to be distinguished from the problem of *checking* a given proof. The latter is easily decidable as we will see in Sect. 9.3.

9.1.2 Formulas not as Terms

The treatment of formulas has received a great deal of attention, and multiple different approaches have been developed. We will not pursue these in the sequel but list them here for completeness.

Formulas as a Separate Concept

The most obvious alternative is to use a separate concept, i.e., a new non-terminal symbol. This is standard practice in first-order logic (FOL), where terms and formulas are strictly separated.

This is particularly reasonable for untyped FOL—the standard variant of FOL. Here there are no types, i.e., no non-terminal A. Alternatively, we can say that there is exactly one base type, and all terms have the same type.

In typed FOL, we have terms, types, and formulas. Here we usually have a base type *bool*. Thus, equality and propositional connectives must be duplicated as operators on terms and as operators on formulas. An advantage of this design is that the quantifiers can be restricted to the formula-level so that the evaluation terms stays decidable.

Formulas as Types

A surprising but formally appealing variant is to make all formulas special cases of *types*. This is common in constructive type theories like Coq or Agda.

This has the advantage that proofs can be elegantly introduced as terms whose type is a formula. A proof P of F would be represented as a term $\vdash P : F$.

A drawback of this design is that all boolean operators are again duplicated.

An advantage is a striking elegance between type operators and connectives. For example, if formulas are types, product types yield conjunction, and function types yield implication. All logical operators except negation have meaningful analogues as operators on types.

This has made it possible to present theorems as programs. For example, a theorem like $\forall x{:}A.\ \forall y{:}B.\ \exists z{:}C.\ \mathit{true}$ can be represented as a function **fun** $f(x : A, y : B) : C = \{P\}$. Giving the body P of this function becomes equivalent to finding a proof of the theorem.

9.2 Proofs as Terms

As for programs, we have to decide whether proofs are a new non-terminal symbol or a special case of terms. Both work well. But for the same reason as for programs, it makes the language easier to make them terms: it eliminates the need for duplicating productions.
The details of what proof constructors to add and what typing rules to give them goes beyond the scope of this treatment. We only give the necessary features for an empty logic and some examples.

9.2.1 Empty Formal System

We introduce a new type constructor that lifts boolean terms to types:

$$A ::= \mathbf{proof}\ t$$

$$\frac{\Gamma \vdash t\ :\ \mathit{bool}}{\Gamma \vdash \mathbf{proof}\ t\ :\ \mathtt{type}}$$

The basic intuition is that the typing judgment becomes a proving judgment: we say that P is a proof of F using assumptions Γ if

$$\Gamma \vdash P\ :\ \mathbf{proof}\ F$$

9.2.2 Common Logical Features

Logical features are very similar to type theoretical features. In both cases, we usually add three productions and typing rules:

Type theory	Logic
type constructor	formula constructor
term constructor for building ...	
... terms of that type	... proofs of that formula
... new terms from terms of that type	... new proofs using proofs of that formula

Thus, we need three productions and three typing rules each for conjunction, disjunction, negation, implication, universal quantification, and existential quantification.

Implication

We add implication using three productions

$$t ::= t \Rightarrow t \mid \mathit{implIntro}(x : t, t) \mid \mathit{modusPonens}(t, t)$$

and three typing rules

$$\frac{\Gamma \vdash F\ :\ \mathit{bool} \qquad \Gamma \vdash G\ :\ \mathit{bool}}{\Gamma \vdash F \Rightarrow G\ :\ \mathit{bool}}$$

$$\frac{\Gamma, \mathbf{val}\ x : \mathbf{proof}\ F \vdash P\ :\ \mathbf{proof}\ G}{\Gamma \vdash \mathit{implIntro}(x : \mathbf{proof}\ F, P)\ :\ \mathbf{proof}\ (F \Rightarrow G)} \qquad \frac{\Gamma \vdash P\ :\ \mathbf{proof}\ (F \Rightarrow G) \qquad \Gamma \vdash Q\ :\ \mathbf{proof}\ F}{\Gamma \vdash \mathit{modusPonens}(P, Q)\ :\ \mathbf{proof}\ G}$$

In logic textbooks, the typing rules for the proofs are usually written by omitting the proof terms themselves. Then we obtain the more familiar-looking

$$\frac{\Gamma, \mathbf{proof}\ F \vdash \mathbf{proof}\ G}{\Gamma \vdash \mathbf{proof}\ (F \Rightarrow G)} \qquad \frac{\Gamma \vdash \mathbf{proof}\ (F \Rightarrow G) \qquad \Gamma \vdash \mathbf{proof}\ F}{\Gamma \vdash \mathbf{proof}\ G}$$

If we additionally write simply F instead of $\mathbf{proof}\ F$, we obtain the usual notation.

9.2.3 Logics for Reasoning about Systems

Logics like FOL and HOL are sufficient for reasoning about mathematical concepts. (The difficulty here is usually to enrich the type theory in order to allow for more natural representations of mathematical objects.)

But for reasoning about dynamic systems like physical systems and machines, we need more. Specifically, we must be able to represent the *change* of the system over time.

For software systems, we can use **discrete** time, i.e., a representation of change as a sequence of states. This corresponds to representing points in time as natural numbers. For example, to verify a piece of code C, we have to talk about the values of all variables in two different states: *before* and *after* execution of C.

For physical systems, especially those interacting via sensor data, we may have to use **continuous** time: a representation of points in time as real numbers and of all values as functions over the real numbers.

Both present substantial challenges, and a variety of different logics has been developed.

9.3 Axioms and Theorems as Declarations

A major advantage of representing formulas and proofs as terms is that we do not need to introduce new declarations for axioms and theorems. They are normal value declarations whose type happens to be **proof** F for some F.

If there is no definition, this yields an axiom declaration. The declaration **val** a : **proof** F introduces an axiom named a that asserts F. This corresponds to assuming that we have a proof of F without further specifying where it comes from.

If there is a definition, this yields a theorem declaration. The declaration **val** a : **proof** $F = P$ introduces a theorem named a that asserts F and whose proof is P. This corresponds to introducing a as an abbreviation for the proof P.

Type-checking of these declarations proceeds in the same way as before. In particular, to check $\vdash$ **val** a : **proof** $F = P$ ✓, we have to check $\vdash P$: **proof** F, i.e., we have to check that P is indeed a proof of F. Thus, proof-checking becomes a special case of type-checking.

9.4 Discrete-Time Logic

9.4.1 Grammar and Intuitions

The basic idea of discrete time is to use a set of states and transitions between them. All the intuitions and concepts from finite automata (which are sometimes alternatively called *finite state machines*) can be generalized to arbitrary state machines. The set of states is usually infinite or at least very large.

The central is the following:

Definition 9.1. A **state** is a possible value for the set of cells in the environment.
A **transition** is a triple (s, o, s'), where s is a state, o is one of the operations that can be called on the environment, and s' is the state after calling o in state s.

Now the evaluation of every term (including the truth of formulas) depends on the state. Thus, the meaning of a formula is no longer a boolean but a function from states to booleans.

When designing logics, we try to avoid talking about states explicitly—that would just amount to reformulating verification problems in a different formalism. Instead, we try to talk about states indirectly or to introduce names for only a few certain states.

Modal Logic

Historically, the central ideas go back to modal logic, which introduced the following operators in the early 20th century:

Formula	Holds in state s if
$\Box F$	F holds in all possible successor states of s
$\diamond F$	F holds in some possible successor state of s

Dynamic Logic

Dynamic logic expands on modal logic by clarifying what transitions should be taken into consideration:

$$t ::= [t]t \mid \langle t \rangle t$$

Formula	Holds in state s if
$[P]F$	F holds in all successor state of s reachable by executing P
$\langle P \rangle F$	F holds in some successor state of s reachable by executing P

If P is a deterministic program, then for every s there is at most 1 successor state reachable by executing P: if P terminates, there is exactly 1 state; otherwise, there is no state.
In that case, the operators become very easy:

Formula	Holds in state s if
$[P]F$	if P terminates from s, F holds afterwards
$\langle P \rangle F$	P terminates from s and F holds afterwards

Note that all terms considered so far are deterministic with the exception of **read**.

9.4.2 Typing

There are different choices for the typing rules. The following is just a simple example:

$$\frac{\Gamma \vdash P \ : \ A \qquad \Gamma \vdash F \ : \ \mathit{bool}}{\Gamma \vdash [P]F \ : \ \mathit{bool}} \qquad \frac{\Gamma \vdash P \ : \ A \qquad \Gamma \vdash F \ : \ \mathit{bool}}{\Gamma \vdash \langle P \rangle F \ : \ \mathit{bool}}$$

Part IV

Formal Methods

Chapter 10

Program Synthesis

Chapter 11

Model Checking

Chapter 12

Theorem Proving

Part V

Security

Chapter 13

Social Engineering

Chapter 14

Common Criteria

Chapter 15

Cryptography

Acknowledgements This chapter contains contributions by Colin Rothgang.
[1]

15.1 History

15.2 Preliminaries

Definition 15.1 (polynomial-time algorithm)**.** An algorithm A is called polynomial time algorithm iff there exists $k \in \mathbb{N}$ such that the maximum number t of operations A has to perform for input of length n satisfies $t(n) \in O(n^k)$.

Definition 15.2 (A probabilistic polynomial-time algorithm)**.** A probabilistic polynomial-time algorithm (PPT) is a polynomial-time algorithm that might be non-deterministic. If it is deterministic we will just call it polynomial time algorithm.

Definition 15.3. A function $f : \mathbb{N} \to \mathbb{R}$ is called negligible iff

$$\forall k \in \mathbb{N}. \exists N_k \in \mathbb{N}. \forall n \geq N_k : n\,|f(n)| < 1.$$

One important cryptographic primitive is the so called one-way function (OWF). Using these one-way-functions one can build provably secure symmetric encryption algorithms. However, it is currently not known whether there exist any one-way function. In the following we will especially consider one-way permutations (OWP), bijective OWFs with identical input and output length.

Definition 15.4 (One way function)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$, is called a one-way function iff, f is a PPT and for any natural number n and any PPT A, there is a negligible function neg such that:

$$\Pr\left[f(A(f(x), 1^n)) = f(x)\right] \leq neg,$$

where x is chosen uniformly random.

Intuitively, OWFs are simply hard to invert PPTs. Some functions that are commonly believed to be OWFs are modular exponentiation and the multiplication of big prime numbers.
Another important concept is the so called Pseudo-random generator (PRG or PRNG).

[1] This section is under development and should be considered unpublished.

Definition 15.5 (PRGs)**.** A function $H : \{0,1\}^* \to \{0,1\}^*$ is called PRG iff:

- H is a PPT
- There exists a PPT the so called *length extension function* $l : \mathbb{N} \to \mathbb{N}$, s.t. $\forall n \in \mathbb{N}.\, l(n) > n \wedge |H(x)| = l(|x|), \forall x \in \{0,1\}^*$
- There is a negligible function *neg* s.t. for any PPT A, we have:

$$\left|\Pr\left[A(H(U_n)) = 1\right] - \Pr\left[D(U_{l(n)}) = 1\right]\right| < neg(n),$$

  where U_k denotes a uniformly random element of $\{0,1\}^k$.

It can be shown that given any OWF, we can build a PRG.

Definition 15.6 (hard-core bit)**.** Let f be a one-way function. Now the function $b : \{0,1\}^* \to \{0,1\}$ is called a *hard-core bit* for f if b is computable in polynomial time and there exists a negligible function $neg(n)$ such that for any $n \in$
mathbbN and any PPT A:

$$\Pr\left[A(f(x), 1) = b(x)\right] \leq \frac{1}{2} + neg(n),$$

where $x \in \{0,1\}$ uniformly random.

The next step is to find one-way functions (assuming that there are one-way functions at all) for which we can build hard-core bits.

Theorem 15.7. *Let f be a one-way function. Define $g : \{0,1\}^{2n} \to \{0,1\}^*$ as $g(x \circ y) = f(x) \circ y$, where $|x| = |y| = n$. Then g is a one-way function with hard-core bit*

$$b(x,y) = \bigoplus_{i=1}^{n} x_i \wedge y_i = \sum_{i=1}^{n} x_i y_i (\text{mod} 2).$$

If now f was a one-way permutation, we can use the above construction to get an additional pseudo-random bit, so we already have a pseudo-random generator.

15.3 Symmetric Encryption

15.3.1 General Definition

Definition 15.8. An encryption scheme is an ordered triple (G, E, D), where G and E are PPT algorithms and D is a polynomial time algorithm iff for any *security parameter* $n \in \mathbb{N}$:

- The *key generation algorithm* G, takes as input 1 and uses randomness to chose a *key* k from a set of possible keys K_n (the key space)
- The *encryption algorithm* E, takes as input a *message* $m \in P_n$ (P_n is called the *plaintext space* for the security parameter n) and uses k to compute an encrypted message $c \in C_n$ (C_n is called the *ciphertext space* for n). If E uses the key k on the message m and outputs s, we write $E_k(m) = c$.
- The *decryption algorithm* D, takes an encrypted message and uses the key k to decipher it (deterministically). So we have

$$\forall n \in \mathbb{N}, \forall m \in P_n, \forall k \in K_n.\, D_k(E_k(m)) = m.$$

There are various different notions of "security" of encryption. We will discuss the notions of computational indistinguishability (comp. ind.) and security against a chosen Plaintext attack (ind. CPA).

Definition 15.9 (computational indistinguishable)**.** We will call an encryption scheme (E, D, k) guess indistinguishable iff for any PPT A, for any two messages m_1, m_2 of length n and $i \in \{0, 1\}$, uniformly random chosen, there is a negligible function *neg.* s.t.:

$$(\Pr[A(E(m_0)) = 1] - \Pr[A(E(m_1)) = 1]) \leq \frac{1}{2} + neg(n),$$

where $neg(n)$ is a negligible function.

15.3.2 Specific Schemes

Given a PRG, we can iterate it on its own output to get an arbitrarily long pseudo-random output. Now we can construct an encryption scheme by simply taking the key as input to a pseudo random generator and xoring the message with the output of the pseudo-random generator. The decryption can be done by simply encrypting the ciphertext a second time. The resulting encryption scheme can already be shown to be computational indistinguishable, although not necessarily ind. CPA secure. We will now try to improve the security to also reach at least ind. CPA security.

Feistel Ciphers

For this we can use a so called Feistel network.

Definition 15.10 (A Feistel cipher)**.** Let k be any natural number (the number of *rounds*). Let f_{k_i} be a family of functions[2] (f is the so called *round function*) of output length n indexed by the sequence of *round keys* $k_1, k_2, \ldots, k_n$. Then the following encryption algorithm E_k is called Feistel cipher.

- Fix a message $m =: x_1 \circ x_2$, where $|x_1| = |x_2| = n$
- Define the sequences $L_1, L_2, \ldots, L_n$ and $R_1, R_2, \ldots, R_n$ by $L_1 := x_1, R_1 := x_2$ and $L_{n+1} := R_n, R_{n+1} := L_n \oplus f_{k_n}(R_n)$. Finally define $E_k : x_1 \circ x_2 \to L_k \circ R_k$.

Now we can define the corresponding decryption algorithm D_k just like e_k, but with the reversed order of round keys:

- Fix a ciphertext $c =: x_1 \circ x_2$, where $|x_1| = |x_2| = n$
- Define the sequences $L_1, L_2, \ldots, L_n$ and $R_1, R_2, \ldots, R_n$ by $L_1 := x_1, R_1 := x_2$ and $L_{n+1} := R_n, R_{n+1} := L_n \oplus f_{k_{k-n}}(R_n)$. Finally define $D_k : x_1 \circ x_2 \to L_k \circ R_k$.

Feistel ciphers have been shown to fulfill several notions of security assuming that the round function is actually pseudo random. For instance Feistel networks with at least 3 rounds are ind. CPA secure and for more rounds they fulfill even stronger notions of security.
In practice many symmetric encryption schemes are based either on Feistel networks or on substitution-permutation-networks.

Definition 15.11 (Substitution-permutation-network)**.** A substitution-permutation-networks is a series of linked substitutions (*S-Boxes*) and permutations *P-Boxes* of blocks in an encryption algorithm. The corresponding encryption algorithm splits the message into several boxes, which are typically fed into the S-Boxes, then the result is fed into the P-Box (and at some point the round key is used, for instance xored with the result as is AES). The later mentioned AES-encryption algorithm is one example of an encryption algorithm build around a substitution-permutation-network.

Substitution-permutation-network and Feistel networks using S-Boxes are quite similar, but there are also some noticeable differences. Ciphers based on substitution-permutation-networks can be better parallelized, but Feistel

ciphers can use any pseudo random function (for instance any one-way function) and is therefore limited to invertible (P-Boxes).

AES

15.4 Asymmetric Encryption

The idea behind RSA is that if $N = p \cdot q$ for large prime numbers p and q, it is very difficult to compute p and q from n.

15.4.1 RSA

Setup Choose two large primes p and q (typically of roughly equal size). Put $N = p \cdot q$.

Now put $n = (p-1)(q-1)$. (Actually, any common multiple of the two numbers is fine.) Note that $n = \varphi(N)$. Pick $e \in Z_n$ such that there is a $d \in \mathbb{Z}_n$ with $e \cdot d \equiv_n 1$. Such a d exists if $\gcd(e, n) = 1$ and is easy to compute (see Thm. A.16).

The keys are defined as follows:

- public information (encryption key): N and e
- private information (decryption key): n, d, p, and q

Among the private information, only N and d are needed later on. So n, p, and q can be forgotten. But they have to remain private—p (or q) is enough to compute n and d.

Different keys are often compared by their size. That size is the number of bits in N.

Encryption Messages are numbers $x \in F_N$. For example, we can choose the largest k such that $2^k < N$ and use k-bit messages.

Encryption and decryption are functions $\mathbb{Z}_N \to \mathbb{Z}_N$ given by

- encryption: $x \mapsto x^e \bmod N$
- decryption: $x \mapsto x^d \bmod N$

These are indeed inverse to each other:

Theorem 15.12. *For all $x \in Z_N$, we have $(x^d)^e \equiv_N (x^e)^d \equiv_N x$.*

Proof. In general, because $N = p \cdot q$ for prime numbers p and q, we have that $x \equiv_N y$ iff $x \equiv_p y$ and $x \equiv_q y$.

So we have to show that $x^{de} \equiv_p x$. (We also have to show the same result for q, but the proof is the same.) We distinguish two cases:

- $p|x$: Then trivially $x^{de} \equiv_p x \equiv_p 0$.
- Otherwise. Then p and x are coprime.
  By construction of e and d and using Thm. A.16, we have $k \in \mathbb{N}$ such that $e \cdot d + k \cdot n = 1$. Thus, we have to show $x^{de} = x \cdot (x^{p-1})^{k \cdot (q-1)} \equiv_p x$. That follows from $x^{p-1} \equiv_p 1$ as known from Thm. A.28.

□

Attacks To break RSA, d has to be computed. There are 3 natural ways to do that:

- Factor N into p and q. Then compute d easily.
- Compute n using $n = \varphi(N)$ (which may be easier than finding p and q). Then compute d easily.
- Find d such that $e \cdot d \equiv_n 1$ (which may be easier than finding n).

Currently these are believed to be equally hard.

It is believed that there is no algorithm for factoring N that is polynomial in the number of bits of N. That is no proved. There are hypothetical machines (e.g., quantum computers) that can factor N polynomially.

Note that checking if N can be factored (without producing the factors) is polynomial, and practical algorithms exist (in particular, the AKS algorithms). That is important to find the large prime number p and q efficiently.

If there is indeed no polynomial algorithm, factoring relies on brute-force attacks that find all prime numbers $k < \sqrt{N}$ and test $k|N$. Therefore, larger keys are harder than break to smaller ones. Because of improving hardware, the key size that is considered secure grows over time.

Keys of size 1024 are considered secure today, but because security is a relative term, keys of size 2048 are often recommended. Larger keys are especially important if data is needed to remain secure far into the future, when faster hardware will be available.

15.5 Authentication

15.6 Hashing

15.6.1 MDx

15.6.2 SHA-x

15.7 Key Generation and Distribution

Chapter 16

Privacy

Part VI

Appendix

Appendix A

Mathematical Preliminaries

A.1 Binary Relations

A binary relation on a set A is a subset $\# \subseteq A \times A$. We usually write $(x, y) \in \#$ as $x\#y$.

A.1.1 Classification

Definition A.1 (Properties of Binary Relations)**.** We say that $\#$ is ... if the following holds:

- reflexive: for all x, $x\#x$
- irreflexive: for no x, $x\#x$
- transitive: for all x, y, z, if $x\#y$ and $y\#z$, then $x\#z$
- a strict order: irreflexive and transitive
- a preorder: reflexive and transitive
- anti-symmetric: for all x, y, if $x\#y$ and $y\#x$, then $x = y$
- symmetric: for all x, y, if $x\#y$, then $y\#x$
- an order[1]: preorder and anti-symmetric
- an equivalence: preorder and symmetric
- a total order: order and for all x, y, $x\#y$ or $y\#x$

An element $a \in A$ is called ... of $\#$ if the following holds:

- least element: for all x, $a\#x$
- greatest element: for all x, $x\#a$
- least upper bound for x, y: $x\#a$ and $y\#a$ and for all z, if $x\#z$ and $y\#z$, then $a\#z$
- greatest lower bound for x, y: $a\#x$ and $a\#y$ and for all z, if $z\#x$ and $z\#y$, then $z\#a$

Definition A.2 (Dual Relation)**.** For every relation $\#$, the relation $\#^{-1}$ is defined by $x \#^{-1} y$ iff $y \# x$. $\#^{-1}$ is called the **dual** of $\#$.

Theorem A.3 (Dual Relation)**.** *If a relation is reflexive/irreflexive/transitive/symmetric/antisymmetric/total, then so is its dual.*

A.1.2 Equivalence Relations

Equivalence relations are usually written using infix symbols whose shape is reminiscent of horizontal lines, such as $=$, $\sim$, or $\equiv$. Often vertically symmetric symbols are used to emphasize the symmetry property.

Definition A.4 (Quotient)**.** Consider a relation $\equiv$ on A. Then

- For $x \in A$, the set $\{y \in A \mid x \equiv y\}$ is called the (equivalence) **class** of x. It is often written as $[x]_{\equiv}$.
- $A/\equiv$ is the set of all classes. It is called the **quotient** of A by $\equiv$.

[1] Orders are also called *partial order*, *poset* (for partially ordered set), or *ordering*.

Theorem A.5. *For a relation $\equiv$ on A, the following are equivalent[2]:*

- *$\equiv$ is an equivalence.*
- *There is a set B and a function $f : A \to B$ such that $x \equiv y$ iff $f(x) = f(y)$.*
- *Every element of A is in exactly one class in $A/\equiv$.*

In particular, the elements of $A/\equiv$

- *are pairwise disjoint,*
- *have A as their overall union.*

A.1.3 Orders

Theorem A.6 (Strict Order vs. Order)**.** *For every strict order $<$ on A, the relation "$x < y$ or $x = y$" is an order.*
For every order $\leq$ on A, the relation "$x \leq y$ and $x \neq y$" is a strict order.

Thus, strict orders and orders come in pairs that carry the same information.
Strict orders are usually written using infix symbols whose shape is reminiscent of a semi-circle that is open to the right, such as $<$, $\subset$, or $\prec$. This emphasizes the anti-symmetry ($x < y$ is very different from $y < x$.) and the transitivity ($< \ldots <$ is still $<$.) The corresponding order is written with an additional horizontal bar at the bottom, i.e., $\leq$, $\subseteq$, or $\preceq$. In both cases, the mirrored symbol is used for the dual relation, i.e., $>$, $\supset$, or $\succ$, and $\geq$, $\supseteq$, and $\succeq$.

Theorem A.7. *If $\leq$ is an order, then least element, greatest element, least upper bound of x, y, and greatest lower bound of x, y are unique whenever they exist.*

Theorem A.8 (Preorder vs. Order)**.** *For every preorder $\leq$ on A, the relation "$x \leq y$ and $y \leq x$" is an equivalence.*
For equivalence classes X and Y of the resulting quotient, $x \leq y$ holds for either all pairs or no pairs $(x, y) \in X \times Y$.
If it holds for all pairs, we write $X \leq Y$.
The relation $\leq$ on the quotient is an order.

Remark A.9 (Totality). If $\leq$ is a preorder, then for all elements x, y, there are four mutually exclusive options:

	$x \leq y$	$x \geq y$	$x = y$
x strictly smaller than y, i.e., $x > y$	true	false	false
x strictly greater than y, i.e., $x < y$	false	true	false
x and y incomparable	false	false	false
x and y similar	true	true	maybe

Now anti-symmetry excludes the option of similarity (except when $x = y$ in which case trivially $x \leq y$ and $x \geq y$). And totality excludes the option of incomparability.
Combining the two exclusions, a total order only allows for $x > y$, $y < x$, and $x = y$.

A.2 Binary Functions

A binary function on A is a function $\circ : A \times A \to A$. We usually write $\circ(x, y)$ as $x \circ y$.

Definition A.10 (Properties of Binary Functions)**.** We say that $\circ$ is ... if the following holds:

- associative: for all x, y, z, $x \circ (y \circ z) = (x \circ y) \circ z$
- commutative: for all x, y, $x \circ y = y \circ x$
- idempotent: for all x, $x \circ x = x$

An element $a \in A$ is called a ... element of $\circ$ if the following holds:

- left-neutral: for all x, $a \circ x = x$

- right-neutral: for all x, and $x \circ a = x$
- neutral: left-neutral and right-neutral
- left-absorbing: for all x, $a \circ x = a$
- right-absorbing: for all x, $x \circ a = a$
- absorbing: left-absorbing and right-absorbing
- if e is a neutral element:
  - left-inverse of x: $a \circ x = e$
  - right-inverse of x: $x \circ a = e$
  - inverse of x: left-neutral and right-neutral of x

Moreover, we say that $\circ$ is a ... if it is/has:

- semigroup: associative
- monoid: associative and neutral element
- group: monoid and inverse elements for all x
- semilattice: associative, commutative, and idempotent
- bounded semilattice: semilattice and neutral element

Terminology A.11. The terminology for *absorbing* is not well-standardized. *Attractive* is an alternative word sometimes used instead.

Theorem A.12. *Neutral and absorbing element of $\circ$ are unique whenever they exist.*
If $\circ$ is a monoid, then the inverse of x is unique whenever it exists.

A.3 The Integer Numbers

A.3.1 Divisibility

Definition A.13 (Divisibility)**.** For $x, y \in \mathbb{Z}$, we write $x|y$ iff there is a $k \in \mathbb{Z}$ such that $x * k = y$.
We say that y is divisible by x or that x divides y.

Remark A.14 (Divisible by 0 and 1). Even though division by 0 is forbidden, the case $x = 0$ is perfectly fine. But it is boring: $0|x$ iff $x = 0$.
Similarly, the case $x = 1$ is trivial: $1|x$ for all x.

Theorem A.15 (Divisibility)**.** *Divisibility has the following properties for all $x, y, z \in Z$*

- *reflexive: $x|x$*
- *transitive: if $x|y$ and $y|z$ then $x|z$*
- *anti-symmetric for natural numbers $x, y \in \mathbb{N}$: if $x|y$ and $y|x$, then $x = y$*
- *1 is a least element: $1|x$*
- *0 is a greatest element: $x|0$*
- *$\gcd(x, y)$ is a greatest lower bound of x, y*
- *$\mathrm{lcm}(x, y)$ is a least upper bound of x, y*

Thus, $|$ is a preorder on $\mathbb{Z}$ and an order on $\mathbb{N}$.

Divisibility is preserved by arithmetic operations: If $x|m$ and $y|m$, then

- *preserved by addition: $x + y|m$*
- *preserved by subtraction: $x - y|m$*
- *preserved by multiplication: $x * y|m$*
- *preserved by division if $x/y \in Z$: $x/y|m$*
- *preserved by negation of any argument: $-x|m$ and $x| - m$*

gcd *has the following properties for all $x, y \in \mathbb{N}$:*

- *associative:* $\gcd(\gcd(x, y), z) = \gcd(x, \gcd(y, z))$

- *commutative:* $\gcd(x, y) = \gcd(y, x)$
- *idempotent:* $\gcd(x, x) = x$
- 0 *is a neutral element:* $\gcd(0, x) = x$
- 1 *is an absorbing element:* $\gcd(1, x) = 1$

lcm *has the same properties as* gcd *except that* 1 *is neutral and* 0 *is absorbing.*

Theorem A.16. *For all $x, y \in \mathbb{Z}$, there are numbers $a, b \in \mathbb{Z}$ such that $ax + by = \gcd(x, y)$.*
a and b can be computed using the extended Euclidean algorithm.

Definition A.17. If $\gcd(x, y) = 1$, we call x and y **coprime**.
For $x \in \mathbb{N}$, the number of coprime $y \in \{0, \dots, x-1\}$ is called $\varphi(x)$. φ is called Euler's **totient function**.

Example A.18. We have $\varphi(0) = 0$, $\varphi(1) = \varphi(2) = 1$, $\varphi(3) = 2$, $\varphi(4) = 1$, and so on. Because $\gcd(x, 0) = x$, we have $\varphi(x) \leq x - 1$. x is prime iff $\varphi(x) = x - 1$.

A.3.2 Equivalence Modulo

Definition A.19 (Equivalence Modulo)**.** For $x, y, m \in \mathbb{Z}$, we write $x \equiv_m y$ iff $m | x - y$.

Theorem A.20 (Relationship between Divisibility and Modulo)**.** *The following are equivalent:*

- $m | n$
- $\equiv_m \supseteq \equiv_n$ *(i.e., for all x, y we have that $x \equiv_n y$ implies $x \equiv_m y$)*
- $n \equiv_m 0$

Remark A.21 (Modulo 0 and 1). In particular, the cases $m = 0$ and $m = 1$ are trivial again:

- $x \equiv_0 y$ iff $x = y$,
- $x \equiv_1 y$ always

Thus, just like 0 and 1 are greatest and least element for $|$, we have that $\equiv_0$ and $\equiv_1$ are the smallest and the largest equivalence relation on $\mathbb{Z}$.

Theorem A.22 (Modulo)**.** *The relation $\equiv_m$ has the following properties*

- *reflexive: $x \equiv_m x$*
- *transitive: if $x \equiv_m y$ and $y \equiv_m z$ then $x \equiv_m z$*
- *symmetric: if $x \equiv_m y$ then $y \equiv_m x$*

Thus, it is an equivalence relation.

It is also preserved by arithmetic operations: If $x \equiv_m x'$ and $y \equiv_m y'$, then

- *preserved by addition: $x + y \equiv_m x' + y'$*
- *preserved by subtraction: $x - y \equiv_m x' - y'$*
- *preserved by multiplication: $x \cdot y \equiv_m x' \cdot y'$*
- *preserved by division if $x/y \in \mathbb{Z}$ and $x'/y' \in \mathbb{Z}$: $x/y \equiv_m x'/y'$*
- *preserved by negation of both arguments: $-x \equiv_m -x'$*

A.3.3 Arithmetic Modulo

Definition A.23 (Modulus)**.** We write $x \bmod m$ for the smallest $y \in \mathbb{N}$ such that $x \equiv_m y$.
We also write $modulus_m$ for the function $x \mapsto x \bmod m$. We write $\mathbb{Z}_m$ for the image of $modulus_m$.

Remark A.24 (Modulo 0 and 1). The cases $m = 0$ and $m = 1$ are trivial again:
- $x \bmod 0 = x$ and $\mathbb{Z}_0 = \mathbb{Z}$
- $x \bmod 1 = 0$ and $\mathbb{Z}_1 = \{0\}$

Remark A.25 (Possible Values). For $m \neq 0$, we have $x \bmod m \in \{0, \ldots, m-1\}$. In particular, there are m possible values for $x \bmod m$.
For example, we have $x \bmod 1 \in \{0\}$. And we have $x \bmod 2 = 0$ if x is even and $x \bmod 2 = 1$ if x is odd.

Definition A.26 (Arithmetic Modulo m)**.** For $x, y \in \mathbb{Z}$, we define arithmetic operations modulo m by

$$x \circ_m y = (x \circ y) \bmod m \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, if there is a unique $q \in \mathbb{Z}_m$ such that $q \cdot x \equiv_m y$, we define $x/_m y = q$.

Note that the condition $y|x$ is neither necessary nor sufficient for $x/_m y$ to de defined. For example, $2/_4 2$ is undefined because $1 \cdot 2 \equiv_4 3 \cdot 2 \equiv_4 2$. Conversely, $2/_4 3$ is defined, namely 2.

Theorem A.27 (Arithmetic Modulo m)**.** *For $x, y \in \mathbb{Z}$,* mod *commutes with arithmetic operations in the sense that*

$$(x \circ y) \bmod m = (x \bmod m) \circ_m (y \bmod m) \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, $x/_m y$ is defined iff $\gcd(y, m) = 1$ *and*

$$(x/y) \bmod m = (x \bmod m)/_m (y \bmod m) \qquad \text{if} \qquad y|x$$

$$x/_m y = x \cdot_m a \qquad \text{if } ay + bm = 1 \text{ as in Thm. A.16}$$

Theorem A.28 (Fermat's Little Theorem)**.** *For all prime numbers p and $x \in \mathbb{Z}$, we have that $x^p \equiv_p x$.*
If x and p are coprime, that is equivalent to $x^{p-1} \equiv_p 1$.

A.3.4 Digit-Base Representations

Fix $m \in \mathbb{N} \setminus \{0\}$, which we call the base.

Theorem A.29 (Div-Mod Representation)**.** *Every $x \in \mathbb{Z}$ can be uniquely represented as $a \cdot m + b$ for $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_m$.*
Moreover, $b = x \bmod m$. We write $b \operatorname{div} m$ for a.

Definition A.30 (Base-m-Notation)**.** For $d_i \in \mathbb{Z}_m$, we define $(d_k \ldots d_0)_m = d_k \cdot m^k + \ldots + d_1 \cdot k + d_0$.
The d_i are called **digits**.

Theorem A.31 (Base-m Representation)**.** *Every $x \in \mathbb{N}$ can be uniquely represented as $(0)_m$ or $(d_k \ldots d_0)_m$ such that $d_k \neq 0$.*
Moreover, we have $k = \lfloor \log_m x \rfloor$ and $d_0 = x \bmod m$, $d_1 = (x \operatorname{div} m) \bmod m$, $d_2 = ((x \operatorname{div} m) \operatorname{div} m) \bmod m$ and so on.

Example A.32 (Important Bases). We call $(d_k \ldots d_0)_m$ the binary/octal/decimal/hexadecimal representation if $m = 2, 8, 10, 16$, respectively.
In case $m = 16$, we write the elements of $\mathbb{Z}_m$ as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

A.3.5 Finite Fields

In this section, let $m = p$ be prime.

Construction Because p is prime, $x/_p y$ is defined for all $x, y \in \mathbb{Z}_p$ with $y \neq 0$. Moreover, $\mathbb{Z}_p$ is a field.

Up to isomorphism, all finite fields are obtained as n-dimensional vector spaces $\mathbb{Z}_p^n$ for some prime p and $n \geq 1$. This field is usually called F_{p^n} because it has p^n elements. From now on, let $q = p^n$.

The elements of F_q are vectors $(a_0, \ldots, a_{n-1})$ for $a_i \in \mathbb{Z}_p$. Addition and subtraction are component-wise, the 0-element is $(0, \ldots, 0)$, the 1-elements is $(1, 0, \ldots, 0)$.

However, multiplication in F_q is tricky if $n > 1$. To multiply two elements, we think of the vectors $(a_0, \ldots, a_{n-1})$ as polynomials $a_{n-1}X^{n-1} + \ldots + a_1 X + a_0$ and multiply the polynomials. This can introduce powers X^n and higher, which we eliminate using $X^n = k_{n-1}X^{n-1} + \ldots + k_1 X + k_0$ for certain k_i. The resulting polynomial has degree at most $n-1$, and its coefficients (modulo p) yield the result.

The values k_i always exists but are non-trivial to find. They must be such that the polynomial $X^n - k_{n-1}X^{n-1} - \ldots - k_1 X - k_0$ has no roots in $\mathbb{Z}_p$. There may be multiple such polynomials, which may lead to different multiplication operations. However, all of them yield isomorphic fields.

Binary Fields The operations become particularly easy if $p = 2$. The elements of F_{2^n} are just the bit vectors of length n. Addition and subtraction are the same operation and can be computed by component-wise XOR. Multiplication is a bit more complex but can be obtained as a sequence of bit-shifts and XORs.

Exponentiation and Logarithm Because F_q has multiplication, we can define natural powers in the usual way:

Definition A.33. For $x \in F^q$ and $l \in \mathbb{N}$, we define $x^l \in F_q$ by $x^0 = 1$ and $x^{l+1} = x \cdot x^l$.
If l is the smallest number such that $x^l = y$, we write $l = \log_x y$ and call n the **discrete q-logarithm** of y with base x.

The powers $1, x, x^2, \ldots \in F_q$ of x can take only $q-1$ different values because F_q has only q elements and x^l can never be 0 (unless $x = 0$). Therefore, they must be periodic:

Theorem A.34. *For every $x \in F_q$, we have $x^q = x$. If $x \neq 0$, that is equivalently to $x^{q-1} = 1$.*

For some x, the period is indeed $q-1$, i.e., we have $\{1, x, x^2, \ldots, x^{q-1}\} = F_q \setminus \{0\}$. Such an x is called a **primitive element** of F_q. But the period may be smaller. For example, the powers of 1 are $1, \ldots, 1$, i.e., 1 has period 1. For a non-trivial example consider $p = 5$, $n = 1$, (i.e., $q = 5$): The powers of 4 are $4^0 = 1$, $4^1 = 4$, $4^2 = 16 \bmod 5 = 1$, and $4^3 = 4$.

If the period is smaller than $q-1$, x^l does not take all possible values in F_q. In that case $\log_x y$ is not defined for all $y \in F_q$.

Computing x^l is straightforward and can be done efficiently. (If $n > 1$, we first have to find the values k_i needed to do the multiplication, but we can precompute them once and for all.)

Determining whether $\log_x y$ is defined and computing its value is also straightforward: We can enumerate all powers $x, x^2, \ldots$ until $x^l = 1$ (in which case the logarithm is undefined) or $x^l = y$ (in which case the logarithm is l). However, no efficient algorithm is known.

A.3.6 Infinity

Occasionally, it is useful to compute also with infinity ∞ or $-\infty$. When adding infinity, some but not all arithmetic operations still behave nicely.

Positive Infinity We write $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$.

The order $\leq$ works as usual. ∞ is the greatest element.

Addition works as usual. ∞ is an attractive element.

Subtraction is introduced as usual, i.e., $a - b = x$ whenever x is the unique value such that $a = x + b$. Thus, $\infty - n = \infty$ for $n \in N$. $x - \infty$ is undefined. The law $x - x = 0$ does not hold anymore.
Multiplication becomes partial because $\infty \cdot 0$ is undefined. For $x \neq 0$, we put $\infty \cdot x = \infty$.
Divisibility $|$ is defined as usual. Thus, we have $x|\infty$ for all $x \neq 0$, and $\infty|x$ iff $x = \infty$. There is no greatest element anymore because: 0 and ∞ are both greater than every other element except for each other.

Negative Infinity We write $\mathbb{Z}^\infty = \mathbb{Z} \cup \{\infty, -\infty\}$.
The order $\leq$ works as usual. $-\infty$ is the least and ∞ the greatest element.
Addition becomes partial because $-\infty + \infty$ is undefined. We put $-\infty + z = -\infty$ for $z \neq \infty$.
Subtraction is introduced as usual. Thus, $z - \infty = -\infty - z = -\infty$ for $z \in \mathbb{Z}$. $\infty - \infty$ is undefined.
Multiplication works similarly to $\mathbb{N}^\infty$. $-\infty \cdot 0$ is undefined. And for $x \neq 0$, we define $\infty \cdot x$ and $-\infty \cdot x$ as ∞ or $-\infty$ depending on the signs.

A.4 Size of Sets

The size $|S|$ of a set S is a very complex topic of mathematics because there are different degrees of infinity. Specifically, we have that $|\mathcal{P}(S)| > |S|$, i.e., we have infinitely many degrees of infinity.
In computer science, we are only interested in countable sets. We use a very simple definition that writes C for countable and merges all greater sizes into uncountable sets, whose size we write as U.

Definition A.35 (Size of sets)**.** The size $|S| \in \mathbb{N} \cup \{C, U\}$ of a set S is defined by:

- if S is finite: $|S|$ is the number of elements of S
- if S is infinite and bijective to $\mathbb{N}$: $|S| = C$, and we say that S is countable
- if S is infinite and not bijective to $\mathbb{N}$: $|S| = U$, and we say that S is uncountable

We can compute with set sizes as follows:

Definition A.36 (Computing with Sizes)**.** For two sizes $s, t \in \mathbb{N} \cup \{C, U\}$, we define addition, multiplication, and exponentiation by the following tables:

	$s+t$	t: $n \in \mathbb{N}$	C	U
	$m \in \mathbb{N}$	$m+n$	C	U
s	C	C	C	U
	U	U	U	U

	$s*t$	t: $n \in \mathbb{N}$	C	U
	$m \in \mathbb{N}$	$m*n$	C	U
s	C	C	C	U
	U	U	U	U

	s^t	t: 0	1	$n \in \mathbb{N} \setminus \{0\}$	C	U
	0	1	0	0	0	0
	1	1	1	1	1	1
s	$m \in \mathbb{N} \setminus \{0\}$	1	m	m^n	U	U
	C	1	C	C	U	U
	U	1	U	U	U	U

Because exponentiation s^t is not commutative, the order matters: s is given by the row and t by the column.

The intuition behind these rules is given by the following:

Theorem A.37. *For all sets S, T, we have for the size of the*

- *disjoint union:*

$$|S \uplus T| = |S| + |T|$$

- *Cartesian product:*
$$|S \times T| = |S| * |T|$$
- *set of functions from T to S:*
$$|S^T| = |S|^{|T|}$$

Thus, we can understand the rules for exponentiation as follows. Let us first consider the 4 cases where one of the arguments has size 0 or 1: For every set A

1. there is exactly one function from the empty set (namely the empty function): $|A^{\varnothing}| = 1$,
2. there are as many functions from a singleton set as there are elements of A: $|A^{\{x\}}| = |A|$,
3. there are no functions to the empty set (unless A is empty): $|\varnothing^A| = 0$ if $A \neq \varnothing$,
4. there is exactly one function into a singleton set (namely the constant function): $|\{x\}^A| = 1$,

Now we need only one more rule: The set of functions from a non-empty finite set to a finite/countable/uncountable set is again finite/countable/uncountable. In all other cases, the set of functions is uncountable.

A.5 Important Sets and Functions

The meaning and purpose of a data structure is to describe a set in the sense of mathematics. Similarly, the meaning and purpose of an algorithm is to describe a function between two sets.

Thus, it is helpful to collect some sets and functions as examples. These are typically among the first data structures and algorithms implemented in any programming language and they serve as test cases for evaluating our languages.

A.5.1 Base Sets

When building sets, we have to start somewhere with some sets that are assumed to exist. These are called the *bases sets* or the *primitive sets*.

The following table gives an overview, where we also list the size of each set according to Def. A.35:

set	description/definition	size
typical base sets of mathematics[3]		
$\varnothing$	empty set	0
$\mathbb{N}$	natural numbers	C
$\mathbb{Z}$	integers	C
$\mathbb{Z}_m$ for $m > 0$	integers modulo m, $\{0, \ldots, m-1\}$ [4]	m
$\mathbb{Q}$	rational numbers	C
$\mathbb{R}$	real numbers	U
additional or alternative base sets used in computer science		
void	alternative name for $\varnothing$	0
unit	unit type, $\{()\}$, equivalent to $\mathbb{Z}_1$	1
$\mathbb{B}$	booleans, $\{false, true\}$, equivalent to $\mathbb{Z}_2$	2
int	primitive integers, $-2^{n-1}, \ldots, 2^{n-1}-1$ for machine-dependent n, equivalent to $\mathbb{Z}_{2^n}$ [5]	2^n
float	IEEE floating point approximations of real numbers	C
char	characters	finite[6]
string	lists of characters	C

[3] All of mathematics can be built by using $\varnothing$ as the only base set because the others are definable. But it is common to assume at least the number sets as primitives.

[4] $\mathbb{Z}_0$ also exists but is trivial: $\mathbb{Z}_0 = \mathbb{Z}$.

[5] Primitive integers are the 2^n possible values for a sequence of n bits. Old machines used $n = 8$ (and the integers were called "bytes"), later machines used $n = 16$ (called "words"). Modern machines typically use 32-bit or 64-bit integers. Modern programmers usually—but dangerously—assume that 2^n is much bigger than any number that comes up in practice so that essentially $int = \mathbb{Z}$. Some programming languages (e.g., Python) correctly implement $int = \mathbb{Z}$.

[6] The ASCII standard defined 2^7 or 2^8 characters. Nowadays, we use Unicode characters, which is a constantly growing set containing the characters of virtually any writing system, many scientific symbols, emojis, etc. Many programming languages assume that there is one character for every primitive integers, e.g., typically 2^{32} characters.

A.5.2 Functions on the Base Sets

For every base set, we can define some basic operations. These are usually built-in features of programming languages whenever the respective base set is built-in.
We only list a few examples here.

Numbers

For all number sets, we can define addition, subtraction, multiplication, and division in the usual way.
Some care must be taken when subtracting or dividing because the result may be in a different set. For example, the difference of two natural numbers is not in general a natural number but only an integer (e.g., $3 - 5 \notin \mathbb{N}$). Moreover, division by 0 is always forbidden.

Quotients of the Integers

The function $modulus_m$ (see Sect. A.3.3) for $m \in \mathbb{N}$ maps $x \in \mathbb{Z}$ to $x \bmod m \in \mathbb{Z}_m$.
In programming languages, the set $\mathbb{Z}_m$ is usually not provided. Instead, $x \bmod y$ is built-in as a function on int. [7]

Booleans

On booleans, we can define the usual boolean operations conjunction (usually written & or &&), disjunction (usually written | or ||), and negation (usually written !).
Moreover, we have the equality and inequality functions, which take two objects x, y and return a boolean. These are usually written $x == y$ and $x! = y$ in text files and $x = y$ and $x \neq y$ on paper.

A.5.3 Set Constructors

From the base sets, we build all other sets by applying set constructors. Those are operations that take sets and return new sets.
The following table gives an overview, where we also list the size of each set according to Def. A.36:

[7] Some care must be taken if x is negative because not all programming languages agree.

set	description/definition	size
	typical constructors in mathematics	
$A \uplus B$	disjoint union	$\|A\| + \|B\|$
$A \times B$	(Cartesian) product	$\|A\| * \|B\|$
A^n for $n \in \mathbb{N}$	n-dimensional vectors over A	$\|A\|^n$
B^A or $A \to B$	functions from A to B	$\|B\|^{\|A\|}$
$\mathcal{P}(A)$	power set, equivalent to $\mathbb{B}^A$	$2^{\|A\|} = \begin{cases} 2^n & \text{if } \|A\| = n \\ U & \text{otherwise} \end{cases}$
$\{x \in A \mid P(x)\}$	subset of A given by property P	$\leq \|A\|$
$\{f(x) : x \in A\}$	image of operation f when applied to elements of A	$\leq \|A\|$
A/r	quotient set for an equivalence relation r on A	$\leq \|A\|$
	selected additional constructors often used in computer science	
A^*	lists over A	$\begin{cases} 1 & \text{if } \|A\| = 0 \\ U & \text{if } \|A\| = U \\ C & \text{otherwise} \end{cases}$
$A^?$	optional element[8] of A	$1 + \|A\|$
	for new names $l_1, \ldots, l_n$	
$enum\{l_1, \ldots, l_n\}$	enumeration: like $\mathbb{Z}_n$ but also introduces named elements l_i of the enumeration	n
$l_1(A_1) \mid \ldots \mid l_n(A_n)$	labeled union: like $A_1 \uplus \ldots \uplus A_n$ but also introduces named injections l_i from A_i into the union	$\|A_1\| + \ldots + \|A_n\|$
$\{l_1 : A_1, \ldots, l_n : A_n\}$	record: like $A_1 \times \ldots \times A_n$ but also introduces named projections l_i from the record into A_i	$\|A_1\| * \ldots * \|A_n\|$
inductive data types[9]		C
classes[10]		U

A.5.4 Characteristic Functions of the Set Constructors

Every set constructor comes systematically with characteristic functions into and out of the constructed sets C. These functions allow building elements of C or using elements of C for other computations.

For some sets, these functions do not have standard notations in mathematics. In those cases, different programming languages may use slightly different notations.

The following table gives an overview:

set C	build an element of C	use an element x of C
$A_1 \uplus A_2$	$inj_1(a_1)$ or $inj_2(a_2)$ for $a_i \in A_i$	pattern-matching
$A_1 \times A_2$	(a_1, a_2) for $a_i \in A_i$	$x.i \in A_i$ for $i = 1, 2$
A^n	$(a_1, \ldots, a_n)$ for $a_i \in A$	$x.i \in A$ for $i = 1, \ldots, n$
B^A	$(a \in A) \mapsto b(a)$	$x(a)$ for $a \in A$
A^*	$[a_0, \ldots, a_{l-1}]$[11] for $a_i \in A$	pattern-matching
$A^?$	$None$ or $Some(a)$ for $a \in A$	pattern-matching
$enum\{l_1, \ldots, l_n\}$	l_1 or $\ldots$ or l_n	switch statement or pattern-matching
$l_1(A_1) \mid \ldots \mid l_n(A_n)$	$l_1(a_1)$ or $\ldots$ or $l_n(a_n)$ for $a_i \in A_i$	pattern-matching
$\{l_1 : A_1, \ldots, l_n : A_n\}$	$\{l_1 = a_1, \ldots, l_n = a_n\}$ for $a_i \in A_i$	$x.l_i \in A_i$
inductive data type A	$l(u_1, \ldots, u_n)$ for a constructor l of A	pattern-matching
class A	**new** A	$x.l(u_1, \ldots, u_n)$ for a field l of A

[8] An optional element of A is either absent or an element of A.

[9] These are too complex to define at this point. They are a key feature of functional programming languages like SML.

[10] These are too complex to define at this point. They are a key feature of object-oriented programming languages like Java.

[11] Mathematicians start counting at 1 and would usually write a list of length n as $[a_1, \ldots, a_n]$. However, computer scientists always start counting at 0 and therefore write it as $[a_0, \ldots, a_{n-1}]$. We use the computer science numbering here.

Bibliography

[ALR01] A. Algirdas, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001. University of Newcastle upon Tyne, Computing Science.

[BF14] P. Bourque and R. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*. IEEE, 2014. https://www.computer.org/web/swebok.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.