Lectures Notes on Knowledge Representation and Processing

Florian Rabe and Michael Kohlhase

2022

These notes were originally prepared for our CS course at University Erlangen-Nuremberg (FAU) in Summer 2020. They are directed at 3rd semester CS undergraduates and master students but should be intelligible even for earlier students and could be interesting also for PhD students and for students from adjacent majors. The course is recommended both as a first course in the specialization area Artificial Intelligence as well as a one-off overview on on knowledge representation.

The course was developed in Summer 2020 from scratch and materials were built along the way. It integrated current directions and recent results in research on knowledge representation pulling together materials in an entirely new and original way.

Contents

Chapter 0

Meta-Information

0.1 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. On the one handle, they may contain more details than mentioned in the lectures. On the other hand, important material such as background, diagrams, and examples may be part of the lecture but not mentioned in these notes.
- 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. As an experiment in teaching, I am inviting all of you to collaborate on these lecture notes with me. This would require familiarity with LaTeX as well as Git and GitHub — that is not part of this lecture, but it is an essential skill for you. Ask in the lecture if you have difficulty figuring it out on your own.

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!

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. Make sure your git commits carry a user name that I can connect to you.)

Other Advice I maintain a list of useful advice for students at https://github.com/florian-rabe/Teaching/blob/master/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.

0.2 Overview

Structure The subsequent *parts* of this course follow the Tetrapod model with one part per aspect. Each of these will describe the concepts, languages, and tools of the respective aspect as well as their relation to other aspects.

The aspects of the Tetrapod are typically handled in individual courses, which describe highly specialized languages and tools in depth. On the contrary, the overall goal of this course will be seeing all of them as different approaches to semantics and knowledge representation. The course will focus on universal principles and their commonalities and differences as well as their advantages and disadvantages.

The subsequent *chapters* of this first part will be dedicated to aspect-independent material. These will not necessarily be taught in the order in which they appear in these notes. Instead, some of them will be discussed in connection to how they are relevant in individual aspects.

Exercises and Running Example Typical practical projects, e.g., the ones that a strong CS graduate might be put in charge of, involve heterogeneous data and knowledge that must be managed using a variety of optimized aspect-specific languages and tools. Interoperability between these is often a major source of inefficiency and bugs.

The exercises accompanying the course will mimic this situation: they will be designed around a single large project that requires choosing and integrating methods, languages, and tools from all aspects.

Concretely, this project will be the development of a univis-like system for a university. It will involve heterogeneous data such as course and program descriptions, legal texts, websites, grade tables, and transcript generation code.

Over the course of the semester students will implement a completely functional system applying the lessons of the course. This is very unusual and often impossible for other courses: as any university course must teach many different things from a wide area, it is rarely possible to find a project that requires many and only lessons from a single course. Here KRP is special because its material pervades all aspects of system development.

Chapter 1

Fundamental Concepts

1.1 Abbreviations

knowledge representation and processing	KRP	the general area of this course
knowledge representation language	KRL	a languages used in KRP
knowledge representation tool	KRT	a tool implementing a KPL and processing algorithms for it

1.2 Motivation

1.2.1 Knowledge

Human knowledge pervades all sciences including computer science, mathematics, natural sciences and engineering. That is not surprising: "science" is derived from the Latin word "scire" meaning "to know". Similarly, philosophy, from which all sciences derive, is named after the Greek words "philo" meaning loving and "sophia" meaning wisdom, and the for common ending "-logy" is derived from Greek "logos" meaning word (i.e., a representation of knowledge).

In regards to knowledge, computer science is special in two ways: Firstly, many branches of computer science need to understand KRP as a prerequisite for teaching computers to do knowledge-based tasks. In some sense, KRP is the foundation and ultimate goal of all artificial intelligence.[1] Secondly, modern information technology enables all sciences to apply computer-based KRP in order to vastly expand on the domain-specific tasks that can be automated. Currently all sciences are becoming more and more computerized, but most non-CS scientists (and many computer scientists for that matter) lack a systematic education and understanding of IT-KRP. That often leads to bad solutions when domain experts cannot see which KRP solutions are applicable or how to apply them.

1.2.2 Representation and Processing

It is no coincidence that this course uses the phrase "Representation and Processing". In fact, this is an instance of a universal duality. Consider the following table of analogous concept pairs, which could be extended with many more examples:

Representation	Processing
Static	Dynamic
Situation	Change
Be	Become
Data Structures	Algorithms
Set	Function
State	Transition
Space	Time

[1] Indeed, a major problem with the currently very successful machine learning-based AI technology is that it remains unclear when and how it does KRP. That can be dangerous because it leads to AI systems recommending decisions without being able to explain why that decision should be trusted.

Again and again, we distinguish a static concept that describes/represents what is a situation/state is and a dynamic concept that describes how it changes. If that change is a computer doing something with or acting on that representation, we speak of "processing".

It is particular illuminating to contrast KRP to the standard CS course on Data Structures and Algorithms (DA).[2] Generally speaking, DA teaches the methods, and KRP teaches how to apply them. Data structures are a critical prerequisite for representing knowledge. But data structures alone do not capture what the data means (i.e., the knowledge) or if a particular representation makes any sense. Similarly, algorithms are the critical prerequisite for processing knowledge. But while algorithms can be systematically analyzed for efficiency, it is much harder to analyze if an algorithm processes knowledge correctly. The latter requires understanding what the input and output data means.

Capturing knowledge in computers is much harder than developing data structures and algorithms. It is ultimately the same challenge as figuring out if a computer system is working correctly — a problem that is well-known to be undecidable in general and very difficult in each individual case.

1.3 Components of Knowledge

1.3.1 Syntax and Semantics, Data and Knowledge

Four concepts are of particular relevance to understanding knowledge. They form a 2×2-quadruple of concepts:

Syntax	Data
Semantics	Knowledge

All four concepts are primitive, i.e., they cannot be defined in simpler terms. All sciences have few carefully-chosen primitives on which everything builds. This is done most systematically in mathematics (where primitives include set or function). While mathematical primitives as well as some primitives in physics or CS are specified formally, the above four concepts can only be described informally, ultimately appealing to pre-existing human understanding. Moreover, this description is not standardized — different courses may use very different descriptions even they ultimately try to capture the same elusive ideas.

Data (in the narrow sense of computer science) is any object that can be stored in a computer, typically combined with the ability to input/output, transfer, and change the object. This includes bits, strings, numbers, files, etc.

Data by itself is useless because we would have no idea what to do with it. For example, the object $O = ((49.5739143, 11.0264941), "2020-04-21T16:15:00CEST")$ is useless data without additional information about its syntax and semantics. Similarly, a file is useless data unless we know which file format it uses.

Syntax is a system of rules that describes which data is **well-formed**. For O above the syntax could be "a pair of (a pair of two IEEE double precision floating point numbers) and a string encoding of an time stamp". For a file, the syntax is often indicated by the file name extension, e.g., the syntax of an `html` file is given in Section 12 of the current HTML standard[3].

Syntax alone is useless unless we know what the semantics, i.e., what the data means and thus how to correctly interpret and process the data. For example, the syntax of O allows to check that O is well-formed, i.e., indeed contains two numbers and a timestamp string. That allows rejecting ill-formed data such as $((49.5739143, 11.0264941), "foo")$. The HTML syntax allows us to check that a file conforms to the standard.

Semantics is a system of rules that determines the meaning of well-formed data. For example, ISO 8601 specifies that timestamp string refer to a particular date and time in a particular time zone. Further semantics for O might be implicit in the algorithms that produce and consume it: such as "the first component of the pair contains two numbers between 0 and 180 resp. 0 and 360 indicating latitude resp. longitude of a location on earth". Semantics might be multi-staged, and further semantics about O might be that O indicates the location and time of the first lecture of this course. Similarly, Section 14 of the HTML standard specifies the semantics of well-formed HTML files by describing how they are to be rendered in a web browser.

Knowledge is the combining of some data with its syntax and semantics. That allows applying the semantics to obtain the meaning of the data (if syntactically well-formed and signaling an error otherwise). In computer systems,

[2] The course is typically called "Algorithms and Data Structures", but that is arguably awkward because algorithms can only exist if there are data structure to work with. Compare my notes on that course in this repository, where I emphasize data structures much more than is commonly done in that course.

[3] https://html.spec.whatwg.org/multipage/

- data is represented using primitive data (ultimately the bits provided by the hardware) and encodings of more complex data (bytes, arrays, strings, etc.) in terms of simpler ones,
- syntax is theoretically specified using grammars and practically implemented in programming languages using data structures,
- semantics is represented using algorithms that process syntactically well-formed data,
- knowledge is elusive and often emerges from executing the semantics, e.g., rendering of an HTML file.

1.3.2 Semantics as Syntax Transformation

In order to capture knowledge better in computer systems, we often use two syntax levels: one to represent the data itself and another to represent the knowledge. These can be seen as input and output data. In that case, semantics is a function that translates from the data syntax to the knowledge syntax, and knowledge is the pair of the data and the result of applying the semantics. The following table gives some examples.

Data syntax	Semantics function	Knowledge syntax
SPARQL query	evaluation	result set
SQL query	evaluation	result table
program	compiler	binary code
program expression	interpreter	result value
logical formula	interpretation in a model	mathematical object
HTML document	rendering	graphical representation

Thus, the role of syntax vs. semantics may depend on the context: just like one function's output can be another function's input, one interpretation's knowledge can be another one's syntax. For example, we can first compile a program into binary and then execute it to returns its value.

Such hierarchies of evaluation levels are very common in computer systems. In fact, most state-of-the-art compilers are subdivided into multiple phases each further interpreting the output of the previous one. Thus, if knowledge is represented in computers, it is invariably data itself but relative to a different syntax.

1.3.3 Heterogeneity of Semantics and Knowledge

While it is easy to design languages to represent data in general, it is very difficult to designing KRLs that capture the human-level quality of knowledge. Over the last few decades, the KRP area in computer science has diversified into different subareas that approach this research problem in fundamentally different ways. In fact, KRP in the very general sense of this course is usually not even studied by itself — instead the subareas are so different, specialized, and large that they all sustain their respective university courses and research conferences.

This is related to the fact that the data naturally comes in fundamentally different forms such as graphs, arrays, tables in the sense of relational databases, programs in a programming language, logical formulas, or natural language texts. We speak of **heterogeneous** data. These different forms of data are supported by highly specialized KPTs: graph databases, array databases, relational databases, package databases for programming languages, theorem databases for logics (e.g., the Isabelle Archive of Formal Proofs), databases of research papers (such as the arXiv), and so on.

All of these are very successful for their respective kind of data. And all of them include specifications of semantics and KP algorithms that implement this semantics. But it can vary massively how the semantics is specified and implemented. This has caused major practical problems for tool interoperability: many projects require data in multiple formats and algorithms from multiple tools. But the respective tools are often islands that assume that all data is represented in the tool's language and users do not use outside tools. Therefore, the import/export capabilities of the tools are often limited.

Moreover, transporting data across systems is usually ignorant of the semantics: while each tool takes relatively good care to implement the semantics correctly, there is much less certainty that the semantics is preserved when exchanging data across tools. For a trivial example, consider a tool that measures length in inches vs. a tool that uses centimeters, both using floating point numbers for the data: if they exchange the data, i.e., just the numbers, they may miscommunicate the semantics.[4]

This problem is not easy to fix though. The heterogeneity of data and semantics is so extreme that it is, in some cases, an open theoretical problem how knowledge can be shared at all across tools. The basic idea — exchange the

[4]Problems like this have been involved in major disasters such as the Mars Climate Orbiter.

data in a way that preserves semantics — can be difficult to implement if both tools use entirely different paradigms to specify semantics.

1.4 The Tetrapod Model of Knowledge

The Tetrapod model of knowledge is an ongoing research project by the instructors of this course. A first publication was made in [CFKR21]. The structure of this course will draw heavily on the Tetrapod model to get an overview of the different approaches to KPR and their interoperability problems.

1.4.1 Five Aspects of Knowledge

The Tetrapod model distinguishes five basic **aspects** of knowledge and KPR as described below. For each aspect, there is a variety dedicated KRLs supported by highly optimized KPTs as indicated in the following table:

Aspect	KRLs (examples)	KPTs (examples)
ontologization	ontology languages (OWL), description logics (ALC)	reasoners, SPARQL engines (Virtuoso)
concretization	relational databases (SQL, JSON)	databases (MySQL, MongoDb)
computation	programming languages (C)	interpreters, compilers (gcc)
deduction	logics (HOL)	theorem provers (Isabelle)
narration	document languages (HTML, LaTeX)	editors, viewers

Ontologization focuses on developing and curating a coherent and comprehensive ontology of concepts. This focuses on identifying the central concepts in a domain and their relations. For example, a medical ontology would define concepts for every symptom, disease, and medication and then define relations for which symptoms and medications are related to which disease.

Ontologies typically abstract from the knowledge: they standardize identifiers for the concepts and spell out some properties and relations but do not try to capture all details of the knowledge. Well-designed ontologies can capture exactly the knowledge that different KPTs must share and can thus serve as interoperability layers between them.

While organization can use ontology languages such as OWL or RDF, the inherent complexity of formal objects in computer science and mathematics usually requires going beyond general purpose ontology languages (similar to how the programming languages underlying computer algebra systems usually go beyond general purpose programming languages).

Concretization uses languages based on numbers, strings, lists, and records to obtain concrete representations of datasets in order to store and query their properties efficiently. Because concrete objects are so simple and widely used, it is possible and common to build concrete datasets on top of general purpose data representation languages and tools such as JSON or SQL.

Computation uses specification and programming languages to represent algorithmic knowledge.

Deduction uses logics and theorem provers to obtain verifiable correctness.

Narration uses natural language to obtain texts that are easy to understand for humans. Because narrative languages are not well-standardized (apart from general purpose languages such as free text or LaTeX), it is common to develop narrative libraries on top of ad-hoc languages that impose some formal structure on top of informal text, such as a fixed tree structure whose leafs are free text or a particular set of LaTeX macros that must be used. Narrative libraries can be classified based on whether entries are derived from publications (e.g., one abstract per paper in zbMATH) or mathematical concepts (e.g., one page per concept in *n*Lab).

1.4.2 Relations between the Aspects

The aspects can be visualized as the corners of tetrahedron with ontologization in the center and edges and faces representing solutions that mix two or three aspects as seen in Figure 1.1.

Most approaches try to incorporate all or multiple aspects. But all languages and tools tend to be heavily biased towards and optimized for a single one of the four corner aspects. This is not due to ignorance but because each aspect provides characteristic advantages that are extremely hard to capture at once. In fact, every combination of aspects shares characteristic advantages and disadvantages as sketched in Figure 1.2. For example, deductive and narrative definitions of a function involved well-definedness arguments, and a function defined by a concrete table

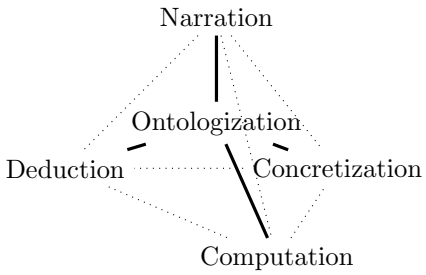

Figure 1.1: Tetrapod model of knowledge

Aspect	characteristic objects	advantage	joint advantage of the other aspects	application
deduction	formal proofs	correctness	ease of use	verification
computation	programs	efficiency	well-definedness	execution
concretization	concrete objects	tangibility	abstraction	storage/retrieval
narration	texts	flexibility	formal semantics	human understanding

Aspect pair	characteristic advantage
ded./comp. narr./conc.	rich meta-theory simple languages
ded./narr. comp./conc.	theorems and proofs normalization
ded./conc. comp./narr.	decidable well-definedness Turing completeness

Figure 1.2: Shared properties and advantages of aspects

is trivially well-defined, but a computational definition of a function may throw exceptions when running; but only the latter can store and compute functions efficiently. Consequently, dedicated and mostly disjoint communities have evolved that have produced large aspect-specific datasets.

Chapter 2

Ontologies

2.1 General Principles

Motivation An ontology is an abstract representation of the main concepts in some domain. Here *domain* refers to any area of the real world such as mathematics, biology, diseases and medications, human relationships, etc. Many examples can be found at https://bioportal.bioontology.org/, including the Gene ontology one of the biggest.

Contrary to the other four aspects, ontological knowledge representations do not aim at capturing the entire semantics of the domain objects. Instead, they focus on defining unique identifiers for those objects and describing some of their properties and relations to each other.

We use the word **ontologization** to refer to the process of organizing the knowledge of a domain in ontologies.

Ontologies are most valuable when they are *standardized* (either sanctioned through a formal body or a quasi-standard because everyone uses it). A standard ontology allows everybody in the domain to use the identifiers defined by the ontology in a way that avoids misunderstandings. Thus, in the simplest form, an ontology can be seen as a dictionary defining the technical terms of a domain. For example, the Gene ontology defines identifier `GO:0000001` to have the formal name "mitochondrion inheritance" and the informal definition "The distribution of mitochondria, including the mitochondrial genome, into daughter cells after mitosis or meiosis, mediated by interactions between mitochondria and the cytoskeleton.".

Ontology Languages An ontology is written in **ontology language**. Common ontology languages are

- description logics such as ALC,
- the W3C ontology language OWL, which is the standard ontology languages of the semantic web,
- the entity-relationship model, which focuses on modeling rather than formal syntax,
- modeling languages like UML, which is the main ontology language used in software engineering.

Ontology languages are not committed to a particular domain — in the Tetrapod model, they correspond to programming languages and logics, which are similarly uncommitted. Instead, an ontology language is a formal language that standardizes the syntax of how ontologies can be written as well as their semantics.

Ontologies The details of the syntax vary between ontology languages. But as a general rule, every **ontology** declares

- **individual** — concrete objects that exist in the real world, e.g., "Florian Rabe" or "WuV"
- **concept** — abstract groups of individuals, e.g., "instructor" or "course"
- **relation** — binary relations between two individuals, e.g., "teach"
- **properties** — binary relations between an individuals and a concrete value (a number, a date, etc.), e.g., "creditValue"
- **concept assertions** — the statement that a particular individual is an instance of a particular concept
- **relation assertions** — the statement that a particular relation holds about two individuals
- **property assertions** — the statement that a particular individual has a particular value for a particular property
- **axioms** — statements about relations between concepts, typically in the form subconcept of statements like

"instructor" $\sqsubseteq$ "person"

All assertions can be understood and spoken as subject-predicate-object **triples** as follows:

Assertion	Triple		
	Subject	Predicate	Object
concept assertion	"Florian Rabe"	`is-a`	"instructor"
relation assertion	"Florian Rabe"	"teach"	"WuV"
property assertion	"WuV"	"creditValue"	7.5

This uses a special relation `is-a` between individuals and concepts. Some languages group `is-a` with the other binary relations between individuals for simplicity although it is technically a little different.

The possible values of properties must be fixed by the ontology language. Typically, it includes at least standard types such as integers, floating point numbers, and strings. But arbitrary extensions are possible such as dates, RGB-colors, lists, etc. In advanced languages, it is possible that the ontology even introduces its own basic types and values.

Ontologies are often divided into two parts:

- The **abstract** part contains everything that holds in general independent of which individuals: concepts, relations, properties, and axioms. It describes the general rules how the worlds works without committing to a particular set of inhabitants of the world. This part is commonly called the **TBox** (T for terminological).
- The **concrete** part contains everything that depends on the choice of individuals: individuals and assertions. It populates the world with inhabitants. This part is commonly called the **ABox** (A for assertional).

A separate division into two parts is the following:

- The **signature** part contains everything that introduces a **named entity**: individuals, concepts, relations, and properties.
- The **theory** part contains everything that describes which statements about the named entities are true: assertions and axioms.

Synonyms Because these principles pervade all formal languages, many competing synonyms are used in different domains. Common synonyms are:

Here	OWL	Description logics	ER model	UML	semantics via logics
individual	instance	individual	entity	object, instance	constant
concept	class	concept	entity-type	class	unary predicate
relation	object property	role	role	association	binary predicate
property	data property	(not common)	attribute	field of base type	binary predicate

In particular, the individual-concept relation occurs everywhere and is known under many names:

domain	individual	concept
type theory, logic	constant, term	type
set theory	element	set
database	row	table
philosophy[1]	object	property
grammar	proper noun	common noun

2.2 A Basic Ontology Language

Vocabularies: Ontologies

O	$::= D^*$	

Declarations

D	$::=$ **individual** `ID`	atomic individual
	| **concept** `ID`	atomic concept
	| **relation** `ID`	atomic relation
	| **property** `ID` : T	atomic property
	| I `is-a` C	concept assertion
	| $I\ R\ I$	relation assertion
	| $I\ P\ V$	property assertion
	| F	other axioms

Formulas

F	$::= C \equiv C$	concept equality
	| $C \sqsubseteq C$	concept subsumption
	| I `is-a` C	concept formula
	| $I\ R\ I$	relation formula
	| $I\ P\ V$	property formula

Individual expressions

I	$::=$ `ID`	atomic individuals

Concept expressions

C	$::=$ `ID`	atomic concepts
	| $\top$	universal concept
	| $\bot$	empty concept
	| $C \sqcup C$	union of concepts
	| $C \sqcap C$	intersection of concepts
	| $\forall R.C$	universal relativization
	| $\exists R.C$	existential relativization
	| $\mathrm{dom}R$	domain of a relation
	| $\mathrm{rng}R$	range of a relation
	| $\mathrm{dom}P$	domain of a property

Relation expressions

R	$::=$ `ID`	atomic relations
	| $R \cup R$	union of relations
	| $R \cap R$	intersection of relations
	| $R; R$	composition of relations
	| R^*	transitive closure of a relation
	| R^{-1}	dual relation
	| Δ_C	identity relation of a concept

Property expressions

P	$::=$ `ID`	atomic properties

Identifiers

`ID` $::=$ alphanumeric string

Basic types and values

T	$::=$ `int` | `float` | `bool` | `string`	types
V	$::=$ (omitted)	values

Figure 2.1: Syntax of BOL

We could study practical ontology languages like ALC or OWL now. But those feature a lot of other details that can block the view onto the essential parts. Therefore, we first define a basic ontology language ourselves in order to have full control over the details.

Definition 2.1 (Syntax of BOL)**.** A BOL-ontology is given by the grammar in Fig. 2.1. It is well-formed if

- no identifier is declared twice,
- every property assertion assigns a value of the type required by the property declaration,
- every reference to an atomic individual/concept/relation/property is declared as such.

The above grammar exhibits some general structure that we find throughout formal KR languages. In particular, an ontology consists of **named declarations** of four different kinds of entities as well as some assertions and axioms about them. Each entity declaration clarifies which kind it is (in our case by starting with a keyword) and introduces a new entity identifier. For each kind, there are complex expressions. These are anonymous and built inductively; their base cases are references to the corresponding identifiers. Sometimes (in our case: individuals and properties), the references are the only expressions of the kind. Sometimes (in our case: concepts and relations), there can be many productions for complex expressions. The complex expressions are used to build axioms; in our case, these are the three kinds of assertions and other formulas.

Remark 2.2 (Formulas vs. Assertions). In Fig. 2.1, the three productions in gray are duplicated: they occur both as assertions and as formulas.

We could remove the three productions for assertions and treat them as special cases of axioms. But We keep the duplication here because assertions are often treated differently from the other axioms. They are grouped with the individuals in the ABox whereas the other axioms are seen as part of the TBox. Moreover, when used as assertions, they may have to be interpreted differently than when used as formulas as we will see in Ch. 6.

Alternatively, we could remove the three productions in gray. But then we would lose the ability to talk about formulas that are not true. That will become relevant in Ch. 7.

Axioms The role of the axioms is two-fold:

- They can be used to perform **consequence closure**: a formula may expresses a closure operation that defines assertions that are automatically added to the ontology. That can be difficult as some kind of exhaustive reasoning is needed. For example, if there is a subconcept axiom "instructor" $\sqsubseteq$ "person" and a concept assertion "FlorianRabe" is-a "instructor", we have to add the implied concept assertion "Florian Rabe" is-a "person".
- They can be used to perform **consistency conditions** that must not violated by the ontology. For example, ontologies may contain contradictory assertions or violations of uniqueness constraints such as a person should only have one father or fathers should be male. The axioms exclude such cases. That should succeed if the assertions are already consequence-closed.

But spelling out how that works is part of the semantics, not the syntax.

Example 2.3. We give a simple ontology that could be used to represent knowledge in the context of a university:

```
individual FlorianRabe
individual WuV
concept person
concept male
concept instructor
concept course
relation teach
property creditValue: float
FlorianRabe is-a instructor ⊓ male
WuV is-a course
FlorianRabe teach WuV
WuV creditValue 7.5
male ⊑ person
instructor ⊑ person
dom teach ⊑ instructor
```

$\mathrm{rng}\ \mathrm{teach} \sqsubseteq \mathrm{course}$
$\mathrm{dom}\ \mathrm{creditValue} \equiv \mathrm{course}$
$\mathrm{course} \sqsubseteq \exists\ \mathrm{teach}^{-1}\ \mathrm{instructor}$

The axioms are meant to state that males and instructors are persons, teaching is done by instructors to courses, exactly the courses have credits, and (the last axiom) every course is taught by at least one instructor. Whether they actually do mean that, depends on the semantics.

The consequence closure (as defined by the semantics) should add the assertion `FlorianRabe is-a person`. Alternatively, if we use the axioms for consistency checking, we should add that assertion from the beginning. Otherwise, the axioms would not be true.

If we use axioms for the consequence closure, we can even omit the two concept assertions — they should be inferred using the domain and range axioms for the relation.

The assertion FlorianRabe is−a instructor $\sqcap$ male could also be split into two assertions FlorianRabe is−a instructor and FlorianRabe is−a male. That will be important as some semantics might have difficulties handling all cases. So it can be helpful to use a variant that does not need $\sqcap$ operator.

2.3 Representing Ontologies as Triples

It is common to represent an entire ontology as a set of subject-predicate-object triples. That makes handling ontologies very simple and efficient. This is the preferred representation of the semantic web.

However, while, e.g., relation assertions are naturally triples, not all declarations are, and some tricks may be necessary.

Inferring the Entity Declarations The entity declarations are not naturally triples. But we can usually infer them from the assertions as follows: any identifier that occurs in a position where an entity of a certain kind is expected is assumed to be declared as an entity for that kind.

For example, the individuals are what occurs as the subject of a concept, relation, or property assertion or as the object of a relation assertion. It is conceivable that there are individuals that occur in none of these. But that is unusual because they would be disconnected from everything in the ontology.

If we give TBox and ABox together, this inference approach usually works well. But if we only give a TBox, this would often not allow inferring all entities. The only place where they could occur in the TBox is in the axioms, and it is quite possible to have concept, relation, and property declarations that are not used in the axioms. In fact, it is not unusual not to have any axioms.

Special Predicates To turn declarations into triples, we can use reflection, i.e., the process of talking about our language constructs as if they were data.

Reflection requires introducing some built-in entities that represent the features of the language. In the semantic web area, this is performed using the following entities:

- "rdfs:Resource": a built-in concept of which all individuals are an instance and thus of which every concept is a subconcept
- "rdf:type": a special predicate that relates an entity to its type:
  - an individual to its concept (corresponding to `is-a` above)
  - other entities to their special type (see below)
- "rdfs:Class": a special class to be used as the type of classes
- "rdf:Property": a special class to be used as the type of properties
- "rdfs:subClassOf": a special relation that relates a subconcept to a superconcept
- "rdfs:domain": a special relation that relates a relation to the concepts of its subjects
- "rdfs:range": a special relation that relates a relation/property to the concept/type of its objects

Here "rdf" and "rdfs" refer to the RDF (Resource Description Framework) and RDFS (RDF Schema) namespaces, which correspond to W3C standards defining those special entities.

Thus, we can represent many and in particular the most important entity declarations as triples:

Assertion	Triple		
	Subject	Predicate	Object
individual	individual	"rdf:type"	"rdfs:Resource"
concept	concept	"rdf:type"	"rdf:Class"
relation	relation	"rdf:type"	"rdf:Property"
property	property	"rdf:type"	"rdf:Property"
concept assertion	individual	"rdf:type"	concept
relation assertion	individual	relation	individual
property assertion	individual	property	value
for special forms of axioms			
$c \sqsubseteq d$	c	"rdfs:subClassOf"	d
$\mathrm{dom}\, r \equiv c$	r	"rdfs:domain"	c
$\mathrm{rng}\, r \equiv c$	r	"rdfs:range"	c

This is subject to the restriction that only atomic concepts and relations can be handled. For example, only concept assertions can be handled that make an individual an instance of an *atomic* concept. This is particularly severe for axioms, where complex expressions occur most commonly in practice. Here, the special relations allow capturing the most common axioms as triples.

Problems Reflection is subtle and can easily lead to inconsistencies. We can see this in how the approach of RDF(S) special entities breaks the semantics via FOL.

For example, it treats classes both as concepts (when they occur as the object of a concept assertion) and as individuals (when they occur as subject or object of a "rdfs:subClassOf" relation assertion). Similarly, "rdfs:Class" is used both as an individual and as a class. In fact, the standard prescribes that "rdfs:Class" is an instance of itself.

In practice, this is handled pragmatically by using ontologies that make sense. A formal way to disentangle this is to assume that there are two variants of "rdfs:Class", one as an individual and one as a class. The translation must then translate "rdfs:Class" differently depending on how it is used.

It would be better if RDFS were described in a way that is consistent under the implicitly intended FOL semantics. But the more pragmatic approach has the advantage of being more flexible. For example, being able to treat every class, relation, or property also as an individual makes it easy to annotate metadata to them. Metadata is a set of properties such as "rdfs:seeAlso" or "owl:versionInfo", whose subjects can be any entity.

Subject-Centered Representations When giving a set of triples, there are usually a lot of triples with the same subject. For example, we could use a simple concrete syntax with one triple per line and whitespace separating subject, predicate, and object:

```
"FlorianRabe" is-a "instructor"
"FlorianRabe" is-a "male"
"FlorianRabe" "teach" "WuV"
"FlorianRabe" "teach" "KRMT"
"FlorianRabe" "age" 40
"FlorianRabe" "office" "11.137"
```

It is more human-friendly to group these triples in such a way that the subject only has to be listed once. For example, we could use a concrete syntax like this, where the subject occurs first and then predicate-object pairs occur on indented lines:

```
"Florian Rabe"
  is-a "instructor"
  is-a "male"
  "teach" "WuV"
  "teach" "KRMT"
  "age" 40
  "office" "11.137"
```

If the same predicate occurs with multiple values, we can group those as well. For example, we could give the objects for the same predicates as a list following the predicate:

```
"Florian Rabe"
  is-a "instructor" "male"
  "teach" "WuV" "KRMT"
  "age" 40
  "office" "11.137"
```

Concrete syntaxes based on the triple representation of ontologies will usually adopt some kind of structure like this. The details may vary.

2.4 Writing Ontologies

2.4.1 The OWL Language

Abstract Syntax and Semantics Due to their central in knowledge representation, a number of languages for ontology writing exist. Most importantly, the syntax and semantics of OWL, including several sublanguages, are standardized by the W3C.

OWL includes a number of built-in special entities. Most importantly, "owl:Thing" corresponds to "rdfs:Resource" as the concept of all individuals.

Concrete Syntax Several concrete syntaxes have been defined and are commonly used for OWL. The OWL2 primer[2] systematically describes examples in five different concrete syntaxes.

APIs for OWL implement the abstract syntax along with good support for reading/writing ontologies in any of the concrete syntaxes.

2.4.2 The Protege Tool

A widely used tool for writing ontologies in OWL is Protege[3].

To get started with Protege without getting confused, we need to continue understand how its key terminology maps to other contexts.

Here	Protege	Edited in WebProtege via
individual	individual	listed in "Individuals" tab
concept	class	listed in "Classes" tab
relation	object property	listed in "Properties" tab
property	data property	listed in "Properties" tab
concept assertion	Type	detail area of the individual in "Individuals" tab
relation assertion	Relationship	detail area of the subject in "Individuals" tab
property assertion	Relationship	detail area of the subject in "Individuals" tab

Protege's interface treats some parts of the ontology specially:

- The "Classes" tab organizes concepts using a tree view based on the subconcept relationship. Superclasses of a class can also be edited directed by listing parents.
- The "Properties" tab organizes properties using a tree view based on the subproperty (i.e., implication, subset) relationship.
- Axioms describing the domain and range of a property can be given directly in its details view.

Note that classes can be in relationships with other classes as well even though that was not considered in the course so far.

2.4.3 Exercise 1

The topic of Exercise 1 is to use Protege to write an OWL ontology for a university.

[2] https://www.w3.org/TR/2012/REC-owl2-primer-20121211/
[3] https://protege.stanford.edu/

Protege is a graphical editor for the abstract syntax of OWL. Familiarize yourself with the various concrete syntaxes of OWL by writing an ontology that uses every feature once, downloading it in all available concrete syntaxes, and comparing those.

The minimal goal of the exercise session is to get a Hello World example going, at which point the task transitions into homework. There will be no homework submission, but you will use your ontology throughout the course.

You should make sure you understand and setup the process in a way that supports you when you revisit and change your ontology many times throughout the semester.

Other than that, the task is deliberately unconstrained to mimic the typical situation at the beginning of a big project, where it is unclear what the ultimate requirements will be.

Chapter 3

Syntax

3.1 The 4 Layers of a Formal Language

Languages are designed, implemented, and evaluated in layers. Fig. 3.1 shows the most common design.

- First, we define a grammar and implement it using a set of inductive types. This is called the AST (abstract syntax tree) of the language. A parser translates string into the corresponding syntax trees, i.e., objects of those inductive types. Objects conforming to the grammar are called **well-formed**.
- Most processing of the language is implemented by inductive functions that traverse the AST. In particular, a context-sensitive inference system is used for type-checking. The type-checking functions take well-formed objects and return booleans, to indicate which are **well-typed**. This defines a context-sensitive sublanguage. That is the language we are really interested in.
- Now the semantics can be implemented: it traverses the AST with the precondition that the input in well-typed. If that succeeds, the objects are called **well-defined**. Ideally, our language satisfies the invariant that for all well-typed objects, the semantics exists. But in some languages well-definedness proofs may be necessary.
- Finally humans evaluate the language by using it. They may report design flaws that feed back into the AST design.

At every layer we detect different issues. Fig. 3.3 and 3.2 give some examples.

Layer	Specified by	Implemented by	Possible error	
Context-Free Syntax	grammar	parser	not derivable from grammar	
Context-Sensitive Syntax	inference system	type checker	symbols not used as declared	KRP
Semantics	inference system, interpretation, or translation		undefined semantics	
Pragmatics	human preferences	human judgment	not useful	not KRP

Figure 3.1: The 4 Layers of a Language

Layer	Expression	Issue	Explanation
Context-Free Syntax	1/	syntax error	argument missing
Context-Sensitive Syntax	1/"2"	typing error	argument has wrong type
Semantics	1/0	run-time error	undefined semantics
Pragmatics	1/1	code review comment	unnecessarily complex expression

Figure 3.2: Typical Errors by Layer in a Programming Language

3.2 Context-Free Syntax

Abstractly, context-free syntax is specified using grammars. Concretely, it is implemented using inductive types.

In the sequel, we will start with the standard definitions and then make a series of variation to each of these definitions until they become equivalent. The intended equivalence is as follows:

Layer	Expression	Issue	Explanation
Context-Free Syntax	$\forall x$	not well-formed	body missing
Context-Sensitive Syntax	$\forall x.P(y)$	not well-typed	y not declared
Semantics	the $x \in \mathbb{N}$ such that $x < 0$	not well-defined	no such x exists
Pragmatics	$\exists x.x \neq x$	inconsistent	no model exists

Figure 3.3: Typical Errors by Layer in Logic

CFG	IDT
non-terminal	type
production	constructor
non-terminal on left of production	return type of constructor
non-terminals on right of production	arguments types of constructor
terminals on right of production	notation of constructor
words derived from non-terminal N	expressions of type N

Remark 3.1 (Chomsky Hierarchy). The Chomsky hierarchy of grammars has been quite influential and is still taught a lot in introductory courses. But it has little relevance for modern language design.

- CH-0, regular grammars: These are equivalent to regular expressions and finite automata. The latter two are usually better ways to work with these languages than a grammar.
- CH-1, context-free grammars: There are what we consider here.
- CH-2, context-sensitive grammars: Context-sensitive languages are very important. But it is more practical to use context-free grammars and then define a context-sensitive subset later in a different way.
- CH-3, unrestricted grammars: These are Turing-complete and are only relevant theoretically.

3.2.1 Context-Free Grammars

We start with the usual definition:

Definition 3.2 (Context-Free Grammar)**.** Given a set Σ of characters (containing the terminal symbols), a **context-free grammar** consists of

- a set N of names called **non-terminal symbols**
- a set of **productions** each consisting of
  - an element of N, called the **left-hand side**
  - a word over $\Sigma \cup N$, called the **right-hand side**

Example 3.3. Let $\Sigma = \{0, 1, +, \cdot, \doteq, \leq\}$. We give a grammar for arithmetic expressions and formulas about them:

$$\begin{aligned} E ::= & \ 0 \\ | & \ 1 \\ | & \ E + E \\ | & \ E \cdot E \\ F ::= & \ E \doteq E \\ | & \ E \leq E \end{aligned}$$

Here we use the BNF style of writing grammars, where the productions are grouped by their left-hand side and written with $::=$ and $|$. We have $N = \{E, F\}$.

First, we give a name to each production of a CFG:

Definition 3.4 (Context-Free Grammar with Named Productions)**.** Given a set Σ of characters (containing the terminal symbols), a **context-free grammar** consists of

- a set N of names called *non-terminal symbols*
- a set of *productions* each consisting of
  - a name

  - an element of N, called the **left-hand side**
  - a word over $\Sigma \cup N$, called the **right-hand side**

Example 3.5. The grammar from above with names written to the right of each production

$$\begin{array}{rll} E ::= & 0 & \text{zero} \\ | & 1 & \text{one} \\ | & E + E & \text{sum} \\ | & E \cdot E & \text{product} \\ F ::= & E \doteq E & \text{equality} \\ | & E \leq E & \text{lessOrEqual} \end{array}$$

This is not common BNF anymore.

Then we add base types to the productions:

Definition 3.6 (Context-Free Grammar with Named Productions and Base Types)**.** Given a set Σ of characters (containing the terminal symbols) and a set T of names (containing the base types allowed in productions), a **context-free grammar** consists of

- a set N of names called *non-terminal symbols*
- a set of *productions* each consisting of
  - a name
  - an element of N, called the **left-hand side**
  - a word over $\Sigma \cup T \cup N$, called the **right-hand side**

The intuition behind base types is that we commonly like to delegate some primitive parts of the grammar to be defined elsewhere. A typical example are literals such as numbers $0, 1, 2, \ldots$: We could give regular expression syntax for digit-strings. Instead, it is nicer to just assume we have a set of base types that we can use to insert an infinite set of literals into the grammar.

Example 3.7. Let Nat be the type of natural numbers and let $T = \{Nat\}$. Then we can improve the grammar from above as follows:

$$\begin{array}{rll} E ::= & Nat & \text{literal} \\ | & E + E & \text{sum} \\ | & E * E & \text{product} \\ F ::= & E \doteq E & \text{equality} \\ | & E \leq E & \text{lessOrEqual} \end{array}$$

3.2.2 Inductive Data Types

We start with the usual definition:

Definition 3.8 (Inductive Data Type)**.** Given a set of names T (containing the types known in the current context), an *inductive data type* consists of

- a name t, called the **type**,
- a set of **constructors** each consisting of
  - a name n
  - a list of elements of $T \cup \{t\}$, called the **argument** types

Example 3.9. Let Nat be the type of natural numbers and $T = \{Nat\}$. We give an inductive type for arithmetic expressions:

$$E \;=\; \texttt{literal}\,\textbf{of}\,Nat \quad | \quad \texttt{sum}\,\textbf{of}\,E * E \quad | \quad \texttt{product}\,\textbf{of}\,E * E$$

Here we use ML-style notation for inductive data types, which separates constructors by | and writes them as `name of argument-type-product`.

First we generalize to mutually inductive types:

Definition 3.10 (Mutually Inductive Data Types)**.** Given a set T of names (containing the types known in the current context), a family of **mutually inductive data type** consists of

- a set N of names, called the **types**,
- a set of *constructors* each consisting of
  - a name
  - an element of N, called the **return type**
  - a list of elements of $N \cup T$, called the **argument** types

Example 3.11. We extend the type definition from above by adding a second type for formulas. Thus, $N = \{E, F\}$.

$$\begin{array}{lll} E = \texttt{literal}\,\textbf{of}\,Nat & | & \texttt{sum}\,\textbf{of}\,E * E \quad | \quad \texttt{product}\,\textbf{of}\,E * E \\ F = \texttt{equality}\,\textbf{of}\,E * E & | & \texttt{lessOrEqual}\,\textbf{of}\,E * E \end{array}$$

Then we add notations to the constructors:

Definition 3.12 (Mutually Inductive Data Types with Notations)**.** Given a set Σ of characters (containing the terminal symbols) and a set T of names (containing the types known in the current context), a family of **mutually inductive data type with notations** consists of

- a set N of names, called the **types**,
- a set of *constructors* each consisting of
  - a name
  - an element of N, called the **return type**
  - a list of elements of $T \cup N$, called the **argument** types
  - a word over the alphabet $\Sigma \cup T \cup N$ containing the argument types in order and only elements from Σ otherwise, called the **notation** of the constructor

The intuition behind notations is that it can get cumbersome to write all constructor applications as $Name(arguments)$. It is more convenient to attach a notation to them such as

Example 3.13. We extend the type definitions from above by adding notations to each constructor. We use the set $\Sigma = \{+, \cdot, \doteq, \leq\}$ as terminals in the notations.

$$\begin{array}{lll} E = \texttt{literal}\,\textbf{of}\,Nat\,\#\,Nat & | & \texttt{sum}\,\textbf{of}\,E * E\,\#\,E + E \quad | \quad \texttt{product}\,\textbf{of}\,E * E\,\#\,E \cdot E \\ F = \texttt{equality}\,\textbf{of}\,E * E\,\#\,E \doteq E & | & \texttt{lessOrEqual}\,\textbf{of}\,E * E\,\#\,E \leq E \end{array}$$

Here we write the constructors as `name of argument-type-product # notation`. It is easy to see that this has introduced redundancy: we can infer the argument types from the notation. So we can just drop the argument types:

$$\begin{array}{lll} E = \texttt{literal}\,\#\,Nat & | & \texttt{sum}\,\#\,E + E \quad | \quad \texttt{product}\,\#\,E \cdot E \\ F = \texttt{equality}\,\#\,E \doteq E & | & \texttt{lessOrEqual}\,\#\,E \leq E \end{array}$$

3.2.3 Merged Definition

With the variation from above we have arrived at the following equivalence:

Theorem 3.14. *Given a set Σ of characters and a set T of names, the following notions are equivalent:*

- *a family of mutually inductive data types in the context of types T with notations using characters from Σ,*
- *a context-free grammar with named productions, terminal symbols from Σ, and base types T.*

Proof. The key idea is that

- the types and constructors of the former correspond to the non-terminals and productions of the latter

- for each constructor-production pair
  - the right-hand side of the latter corresponds to the notation of the former,
  - the argument types of the former correspond to the non-terminals occurring on the right-hand side of the latter.

□

In implementations in programming languages, we often drop the notations. Instead, those are handled, if needed, by special parsing and serialization functions.
However, in an implementation, it is often helpful to additionally give names to each argument of a production/-constructor. That yields the following definition:

Definition 3.15 (Context-Free Syntax)**.** Given a set Σ of characters and a set T of names, a context-free syntax consists of

- a set N of names, called the **non-terminals/types**,
- a set of **productions/constructors** each consisting of
  - a name
  - an element of N, called the **left-hand side/return type**
  - a sequence of objects, called the **right-hand side/arguments** which are one of the following
    - an element of Σ
    - a pair written $(n : t)$ of a name n, called the **argument name**, and an element $t \in T \cup N$ called the **argument type**.

Example 3.16. Using ad hoc language to write the constructors, our example from above as a context-free syntax could look as follows:

$$E = \texttt{literal} \,\#\, (value : Nat) \quad | \quad \texttt{sum} \,\#\, (left : E) + (right : E) \quad | \quad \texttt{product} \,\#\, (left : E) \cdot (right : E)$$
$$F = \texttt{equality} \,\#\, (left : E) \doteq (right : E) \quad | \quad \texttt{lessOrEqual} \,\#\, (left : E) \leq (right : E)$$

3.2.4 Contexts

We assume a context-free language l.

Definition 3.17 (Context)**.** A **context** is a list of

- grammar terminology: productions $N ::= x$
- type terminology: declarations $x : N$

where each x is a unique name and each N is non-terminal symbol.
The x are called **variables**.

Remark 3.18. Sometimes the grammar itself has specific productions for contexts and variables. In that case, we speak of *meta-variable contexts* and **meta-variables** to distinguish them from those of the language.

Definition 3.19 (Expressions in Context)**.** Given a context Γ, a word E derived from non-terminal N that may additionally use the productions of the context is called an *expression of type N in context Γ*.
We write this as $\Gamma \vdash_l E : N$.

Definition 3.20 (Substitution)**.** Given two contexts $\vdash_l \Gamma$ and $\vdash_l \Delta$, a **substitution** γ from $\Gamma = x_1 : N_1, \ldots, x_n : N_n$ to Δ is a list $x_1 := e_1, \ldots, x_n := e_n$ where every e_i is an expression of type N_i in context Δ (i.e., $\Delta \vdash_l e_i : N_i$).
We write this as $\Delta \vdash_l \gamma : \Gamma$ or as $\vdash_l \gamma : \Gamma \to \Delta$.

Definition 3.21 (Substitution Application)**.** Given an expression $\Gamma \vdash_l E : N$ and a substitution $\vdash_l \gamma : \Gamma \to \Delta$ where $\Gamma = x_1 : N_1, \ldots, x_n : N_n$ and $\gamma = x_1 := e_1, \ldots, x_n := e_n$, we write $E[\gamma]$ for the result of replacing every x_i in E with e_i.

Theorem 3.22. *If* $\Gamma \vdash_l E : N$ *and* $\vdash_l \gamma : \Gamma \to \Delta$*, then* $\Delta \vdash_l E[\gamma] : N$.

We often want to substitute only a single variable $x : N$ even though E may be defined in a larger context Γ. This is often written $E[x := e]$. That is just an abbreviation for $E[\gamma]$, where γ contains $x := e$ as well as $y := y$ for every other variable y of Γ.

Notation 3.23. There are many different notations for substitutions that are in common use. Examples include $E(x)$, $E[x := e]$, $E[x/e]$, or $[e/x]E$. Usually authors of a paper or textbook briefly mention their preferred notation briefly at the beginning.

3.3 Context-Sensitive Syntax

3.3.1 Principles

It is common to define a language as the set of words that can be produced from the syntax, i.e., from a distinguished non-terminal (the start symbol) of the context-free grammar. It is common to define a context-sensitive language as a special case: the set of words that can be produced from a context-sensitive grammar.

This is, however, not helpful in practice. While the above remains the official definition of what the context-sensitive languages are, all practical definitions are entirely different. In fact, context-sensitive grammars are virtually never used to define a specific language. Instead, more restrictive definitions are used that capture more properties of practical languages.

A typical context-sensitive syntax includes the following:

- a context-free syntax,
- a distinguished non-terminal symbol $\mathcal{V}$, whose words are called **vocabularies**,
- a set of distinguished non-terminal symbols $\mathcal{E}$, whose words are called $\mathcal{E}$**-expressions**,
- a unary predicate $\mathrm{wfv}(\Theta)$ on vocabularies Θ,
- for every vocabulary Θ and every $\mathcal{E}$, a unary predicate $\mathrm{wff}_\Theta(E)$ on $\mathcal{E}$-expressions E.

In case of $\mathrm{wfv}(\Theta)$, we call Θ **well-formed**. In case of $\mathrm{wff}_\Theta(E)$, we call E a **well-formed** $\mathcal{E}$-expression over Θ.

In practice, it is most useful to think of a context-sensitive language as a family of context-sensitive languages: for every vocabulary, one language containing the expressions.

Remark 3.24 (Terminology). "Well-formed $\mathcal{E}$-expression over Θ" can be a mouthful. Therefore, it is common to simply say that E is an $\mathcal{E}$-expression, or that E is a Θ-expression, and expect readers to fill in the details.

It is also common to give the non-terminal $\mathcal{E}$ names, such as "term", "type", "formula", or (confusingly) "expression". Then we simply say "term" instead of "term-expression" and so on.

Example 3.25. We extend the context-free language from Ex. 3.3 to a context-sensitive one as follows:

Vocabularies

$Voc \ ::= Decl^*$	list of declarations

Declarations

$Decl \ ::= id : Type^* \to Type$	typed function symbols
$\mid \ id : Type^* \to Form$	typed predicate symbols

Types

$Type \ ::= Nat \mid String$	base types

Expressions

$Expr \ ::= 0 \mid 1 \mid Expr + Expr \mid Expr * Expr$	as before
$\mid \ id(Expr^*)$	application of a function symbol

Formulas

$Form ::= Expr \doteq Expr \mid Expr \leq Expr$	as before
$\mid \ id(Expr^*)$	application of a predicate symbol

Voc is the special non-terminal $\mathcal{V}$. The special non-terminals for expressions are $Type$, $Expr$, and $Form$.
The well-formedness predicates check that every identifier is used according to its declaration.

3.3.2 Vocabularies

The vocabularies are typically lists of typically named declarations. They introduce the names that can be used to form expressions. The expression kinds almost always include formulas.
Often declarations contain additional expressions, most importantly types or definitions. In general, all expressions may occur in declarations, but many language systems do not use all of them.
Very different names are used for the vocabularies in different communities. The following table gives an overview:

Aspect	vocabulary Θ	expression kinds $\mathcal{E}$
Ontologization	ontology	individual, concept, relation, property, formula
Concretization	database schema	cell, row, table, formula
Computation	program	term, type, object, class, ...
Logic	theory	term, type, formula, ...
Narration	dictionary	phrases, sentences, texts

The standard library is a fixed vocabulary that is always present. Shifting operators from the grammar into the standard library is a common technique to simplify the language.

Example 3.26. We can rewrite Ex. 3.25 into a much shorter grammar, in which all name-like productions are declared in the standard library:

Voc	$::= Decl^*$	list of declarations
$Decl$	$::= id : Type^* \to Type$	typed function symbols
	$\mid id : Type^* \to FORM$	typed predicate symbols
	$\mid id : TYPE$	type symbols
$Type$	$::= id$	reference to a type symbol
$Expr$	$::= id(Expr^*)$	application of a function symbol
$Form$	$::= id(Expr^*)$	application of a predicate symbol

The standard library is the vocabulary containing the following declarations:

- type symbols: $Nat : TYPE$, $String : TYPE$
- function symbols: $0 : Nat$, $1 : Nat$, $sum : Nat\,Nat \to Nat$, $product : Nat\,Nat \to Nat$,
- predicate symbols: $equals : Nat\,Nat \to FORM$, $lesseq : Nat\,Nat \to FORM$

3.3.3 Contexts and Variable Binding

In addition to the vocabulary, there is often a second kind of declarations that is part of the context: variables. Variables are usually introduced by *binders*. These are special expression productions that declare a variable and take another expressions, which is the scope of the variable. Somewhat confusingly, the list of variables that are currently in scope is usually (also) called the *context* even though the the overall context consists of the vocabulary and the variable context.
Typical binders are

- Quantifiers like $\forall$ and $\exists$.
- The let binder.
- Description/choice operators, which return an object with certain properties.
- The block-sequence operator from programming languages often written as a ;-separated sequence of statements wrapped in {}. Here each statement may be a variable declaration, in which case the remainder of the current block is its scope.

If a type system is used, variable declarations V are often typed and written as $x : A$. Otherwise, they consist only of the variable name x.

Example 3.27. We can extend Ex. 3.26 with expression productions that bind variables:

Voc	$::=$	$Decl^*$	list of declarations
$Decl$	$::=$	$id : Type^* \to Type$	typed function symbols
	$\mid$	$id : Type^* \to FORM$	typed predicate symbols
	$\mid$	$id : TYPE$	type symbols
$VarDecl$	$::=$	$id : Type$	typed variables
$Context$	$::=$	$VarDecl^*$	contexts
$Type$	$::=$	id	reference to a type symbol
$Expr$	$::=$	$id(Expr^*)$	application of a function symbol
	$\mid$	id	reference to a variable
	$\mid$	**the** $VarDecl$ **such that** $Form$	description
	$\mid$	**let** $VarDecl = Expr$ **in** $Expr$	let binding
$Form$	$::=$	$id(Expr^*)$	application of a predicate symbol
	$\mid$	$\forall VarDecl.Form$	universal quantification
	$\mid$	$\exists VarDecl.Form$	existential quantification

For example, in $\forall x : a.F$, the variable x may occur in the formula F. In $let\, x : int = E_1\, in\, E_2$, the variable x may occur in E_2 but not in E_1. This must be enforced by the type checker, which must move the variable declaration into the context when traversing into the respective subexpression.

3.3.4 Vocabularies vs. Contexts

The context is often split into two components:

- the vocabulary containing the symbol declarations,
- the context containing the variable declarations.

They are summarized in Fig. 3.4.

Terminology 3.28. The word *context* is overloaded. It is used both as a generic term for

- any data structure that is carried through a syntax traversal and
- for the special case of such a data structure that collects variable declarations.

Both vocabularies and contexts store declarations that are used later on. Thus the overall context is typically a pair of a vocabulary and a variable context. During context-sensitive traversal, any declaration that is found is remembered in the overall context. Identifier references are only legal if the referenced identifier is in the Context with the appropriate arity, type, etc.

The vocabulary stores all toplevel declarations of the current development. These are often called *symbols* or *constants*. It is often split over multiple files and packages using imports. Users often package up a vocabulary in order to share it with other users. Consequently, it is usually a *set* of declarations and declarations are identified via unique names. To make declaration names from different packages unique, namespaces are used to form qualified names — the namespace is a name for the entire vocabulary, and declarations are referred to by the pair of vocabulary and declaration name. Because the declarations in a vocabulary can be referenced by anybody using that vocabulary, the names cannot be changed easily. Because the vocabulary can become big, it is often implemented as a hash table mapping qualified names to declarations. When checking a vocabulary, each declaration is checked relative to the vocabulary of preceding declarations. (In languages that allow mutual recursion, it may even be that each declaration is checked relative to the entire vocabulary.)

The variable context stores all local *variable* declarations encountered while traversing a single declaration or expression. The scope of each declaration is limited to the remainder of the declaration or subexpression. Consequently, variables can usually be renamed easily without changing the semantics. When checking a declaration, the context is initially empty and grows as variable declarations are traversed. In particular, each expression is checked relative to a vocabulary and a context.

The vocabulary is almost always more expressive than the context: For every a variable declaration, there is a corresponding symbol declaration but not the other way round. Fig. 3.5 gives some examples.

	vocabulary	context
Structure	list of declarations	
Purpose	declare identifiers to be used later on	
Traversal	accumulated during traversal	
Content	previous declarations	variables traversed in current expression
Needed for	processing declarations and expressions	processing expressions
Distribution	split over files, packages	local to current declaration
Modularity	imports, package manager	only local declarations
Visibility of declarations	entire project	local scope
Order	usually inessential	essential
Renaming	change management problem	no change of semantics (α-renaming)
Duplicate names	disambiguated through namespaces	shadowing
Size	often large	typically small
Typical implementation	hash table	list

Figure 3.4: Commonalities and Differences between the two kinds of Context

	vocabulary	context
BOL	concepts, etc.	none
SFOL	type, function, predicate	only nullary functions
SQL	table, column, row	none
Scala	class, type, function, value	type (restricted), value

Figure 3.5: Kind of declarations in some Languages

3.4 Implementation

Data structures for context-free syntax can be implemented systematically in all programming languages. But, depending on the style of the language, they make drastically different. Below we give the two most important paradigms as examples.

In each case, context-sensitive syntax is implemented as a set of traversal functions over these data structures that return the well-formedness property, i.e., a boolean.

3.4.1 Functional Programming Languages

In a function programming language, inductive data types are a primitive feature. However, notations and named arguments are not available. So helper functions must be used.

The basic recipe is as follows:

- The types and constructors (without the notations and named arguments) are implemented as family of mutually inductive data types.
- For each argument of each constructor, a partial projective function is defined.
- A set of mutually recursive string rendering functions are defined, one for each constructor, that implement the notations.

Example 3.29. We define our example syntax in ML.

First the inductive types (assuming a type Nat already exists in the context):

$$\begin{array}{l}\textbf{data}\ E\ =\ \texttt{literal}\ \textbf{of}\ Nat \quad | \quad \texttt{sum}\ \textbf{of}\ E * E \quad | \quad \texttt{product}\ \textbf{of}\ E * E \\ \textbf{and}\ F\ =\ \texttt{equality}\ \textbf{of}\ E * E \quad | \quad \texttt{lessOrEqual}\ \textbf{of}\ E * E\end{array}$$

Now the projection functions:

$$\begin{array}{l}\textbf{fun}\ \texttt{literal\_value}(\texttt{literal}(v)) = SOME\ v \\ |\quad \texttt{literal\_value}(\_) = NONE \\ \textbf{fun}\ \texttt{sum\_left}(\texttt{sum}(x, \_)) = SOME\ x \\ |\quad \texttt{sum\_left}(\_) = NONE \\ \textbf{fun}\ \texttt{sum\_right}(\texttt{sum}(\_, x)) = SOME\ x \\ |\quad \texttt{sum\_right}(\_) = NONE\end{array}$$

and so on for each constructor argument.
Finally, the string rendering functions (assuming a function *natToString* already exists in the context):

$$\begin{aligned}
&\textbf{fun } \texttt{E_toString}(\texttt{literal}(v)) = \mathit{natToString}\ v \\
&\mid\ \ \texttt{E_toString}(\texttt{sum}(x,y)) = \texttt{E_toString}(x) + \text{"+"} + \texttt{E_toString}(y) \\
&\mid\ \ \texttt{E_toString}(\texttt{product}(x,y)) = \texttt{E_toString}(x) + \text{"}\cdot\text{"} + \texttt{E_toString}(y) \\
&\textbf{and } \texttt{F_toString}(\texttt{equality}(x,y)) = \texttt{E_toString}(x) + \text{"}\doteq\text{"} + \texttt{E_toString}(y) \\
&\mid\ \ \texttt{F_toString}(\texttt{lessOrEqual}(x,y)) = \texttt{E_toString}(x) + \text{"}\leq\text{"} + \texttt{E_toString}(y)
\end{aligned}$$

Because ML has inductive data types as primitives, pattern-matching on our syntax comes for free. We will get back to that when defining the semantics.

3.4.2 Object-Oriented Programming Languages

In a object-oriented programming language, inductive data types are not available. Therefore, they must be mimicked using classes. On the positive side, this supports arguments names, and notations are a bit easier.
The basic recipe is as follows:

- Each type is implemented as an abstract class.
- Each constructor of type t is implemented as a concrete class that extends the abstract class t.
- The arguments names and type of each constructor c are exactly the argument names and types of the class c. The constructor arguments are stored as fields in the class.
- The abstract classes require a `toString` method, which is implemented in every concrete class according to its notation.

Example 3.30. We define our example syntax in a generic OO-language somewhat similar to Scala.[1]
In particular, we assume that the sy

```
abstract class E {
  def toString: String
}
class literal extends E {
  field value: Nat
  constructor (value: Nat) {
    this.value = value
  }
  def toString = value.toString
}
class sum extends E {
  field left: Nat
  field right: Nat
  constructor (left: E, right: E) {
    this.left = left
    this.right = right
  }
  def toString = left.toString + "+" + right.toString
}
class product extends E {
  field left: Nat
  field right: Nat
  constructor (left: E, right: E) {
    this.left = left
    this.right = right
  }
  def toString = left.toString + "·" + right.toString
}
```

```
abstract class F {
  def toString: String
}
class equality extends E {
  field left: Nat
  field right: Nat
  constructor (left: E, right: E) {
    this.left = left
    this.right = right
  }
  def toString = left.toString + "≐" + right.toString
}
class product extends E {
  field left: Nat
  field right: Nat
  constructor (left: E, right: E) {
    this.left = left
    this.right = right
  }
  def toString = left.toString + "≤" + right.toString
}
```

Because OO-languages do not have inductive data types as primitives, pattern-matching on our syntax requires awkward switch statements. We will get back to that when defining the semantics.

3.4.3 Combining Paradigms

The Scala language combines ideas from functional and OO-programming. That makes its representation of context-free syntax particularly elegant.

In Scala, the constructor arguments are listed right after the class name. These are automatically fields of the class, and a default constructor always exists that defines those fields. That gets rid of a lot of boilerplate.

If we want to make those fields public (and we do because those are the projection functions, we add the keyword `val` in front of them. But even that is too much boilerplate. So Scala defines a convenience modifier: if we put `case` in front of the classes corresponding to constructors of our syntax, Scala puts in the `val` automatically. It also generates a default implementation of `toString`, which we have to override if we want to implement notations, too. Finally, Scala also generates pattern-matching functions so that we can pattern-match in the same way as in ML.

Then our example becomes (as usual, assuming a class `Nat` already exists):

```
abstract class E {
  def toString: String
}
case class literal(value: Nat) extends E {
  override def toString = value.toString
}
case class sum(left: Nat, right: Nat) extends E {
  override def toString = left.toString + "+" + right.toString
}
case class product(left: Nat, right: Nat) extends E {
  override def toString = left.toString + "·" + right.toString
}

abstract class F {
  def toString: String
}
case class equality(left: Nat, right: Nat) extends E {
```

```
  override def toString = left.toString + "≐" + right.toString
}
case class lessOrEqual(left: Nat, right: Nat) extends E {
  override def toString = left.toString + "≤" + right.toString
}
```

3.4.4 Issues Regarding Equivalence

There are a few subtle issues.

Equality Forming the same value twice should yield the same result, e.g., `literal`(1) = `literal`(1). In ML-style inductive types, that is automatic. In OO-style class-based implementations, we have **new** `literal`(1) = **new** `literal`(1) only if we override the equality check. Scala allows adding the keyword **case** to a class representing a constructor/production. Among other things, it allows dropping the **new** and overrides the equality method automatically.

Null Value Object-oriented language automatically provide the `null` value for every class. Therefore, if we implement a context-free syntax via classes, we always get one spurious value that should not exist. Programmers have to use programming discipline to avoid using it. Occasionally, static analysis tools can non-null annotations to check this discipline.

Sealing When we seal a context-free syntax, it is impossible to add new constructors to it. That guarantees implementors of inductive functions that they know all possible cases and can therefore pattern-match on them. An inductive function on an unsealed datatype can fail when new constructors are added later.

ML-style inductive data types are always sealed. OO-style class-based implementations are never sealed. In Scala, the keyword **sealed** can be added to an abstract class — it forbids any constructors other than the ones present in the same source file.

3.4.5 Implementing Context-Sensitive Checking

We extend the implementation of our example language in Scala from Sect. 3.4.3 to cover the context-sensitive language from Ex. 3.25.

```
abstract class NonTerminal {
    def print(): String
}

// At the toplevel, every formal language has a vocabulary, which is a list of declarations
// (we can skip the abstract class if the non-terminal has exactly one production)
case class Vocabulary(decls: List[Declaration]) extends NonTerminal {
  // print all declarations and concatenate with separator ", "
  def print() = decls.map(_.print()).mkString(", ")
}

// there can be many different kinds of declarations
abstract class Declaration extends NonTerminal {
  // most declarations have a name, but not all e.g., axioms
  def nameO: Option[String]
}
case class FunctionSymbolDeclaration(name: String, inputs: List[Type], output: Type) extends
  def nameO = Some(name)
  // print as "name: inputs -> output"
  def print() = name + ": " + inputs.map(_.print()).mkString(" ") + " -> " + output.print()
}
```

```
abstract class Type extends NonTerminal
abstract class Expr extends NonTerminal
abstract class Form extends NonTerminal

case class Nat() extends Type {
  def print() = "Nat"
}
case class String() extends Type {
  def print() = "String"
}

// for every declaration, there is a production to refer to the declared identifer, i.e., to
case class FunctionSymbolReference(name: String, arguments: List[Expr]) extends Expr {
  // print as "name(arguments)"
  def print() = name + "(" + arguments.map(_.print()).mkString(",") + ")"
}

case class Sum(left: Expr, right: Expr) extends Expr {
  def print() = left.print() + "+" + right.print()
}

case class Product(left: Expr, right: Expr) extends Expr {
  def print() = left.print() + "*" + right.print()
}

case class Zero() extends Expr {
  def print() = "zero"
}

case class One() extends Expr {
  def print() = "one"
}

case class Equals(left: Expr, right: Expr) extends Form {
  def print() = left.print() + "=" + right.print()
}

case class LessEq(left: Expr, right: Expr) extends Form {
  def print() = left.print() + "<=" + right.print()
}
```

Then we implement a context-sensitive type-checker for it. It implements the wff-predicates from Def. 8.2.

A type-checker traverses the AST. Any such traversal consists of

- one function for every non-terminal (mutually recursive)
- one case for every constructor
- one recursive call for every constructor argument

We first do the simple case where we only have a single base type. In this case, the base types in the inputs and outputs of the symbols are always the same and only the number of arguments must be checked. In that case, the type-checker is a traversal that

- takes a Vocabulary as an extra argument - that's the context that makes everything context-sensitive
- returns Boolean

```
object TypeChecker {

  // check every declaration relative to the vocabulary before it
  // this is the same for virtually every formal language
  def check_Voc(voc: Vocabulary): Boolean = {
```

```
  var seenSofar: List[Declaration] = List()
  voc.decls.forall {d =>
     // check the current declaration relative to the ones seen before
     val r = check_Decl(Vocabulary(seenSofar), d)
     // append it to the list of seen declarations
     seenSofar = seenSofar ::: List(d)
     r
  }
}

def check_Decl(voc: Vocabulary, d: Declaration): Boolean = {
  d match {
    case FunctionSymbolDeclaration(n,ins,out) =>
      if (voc.decls.exists(e => e.nameO == Some(n)))
        false // in practice, throw "name already defined" error
      else {
        ins.forall(i => check_Type(voc, i)) && check_Type(voc, out)
       }
  }
}

// compared the the lecture, I've removed function types for simplification
def check_Type(voc: Vocabulary, tp: Type) = {
  tp match {
    case Nat() => true
    case String() => true
  }
}

// In our simplified variant, where we only check arity, expressions are well-typed if
// all subexpressions are and applications use the right number of arguments.
// We will expand on that later.
def check_Expr(voc: Vocabulary, n: Expr): Boolean = {
   n match {
     case Zero() => true
     case One() => true
     case Sum(l,r) => check_Expr(voc, l) && check_Expr(voc, r)
     case Product(l,r) => check_Expr(voc, l) && check_Expr(voc, r)

     case FunctionSymbolReference(s, args) =>
       // for the reference to the identifiers in the vocabulary, we have to lookup the ide
       voc.decls.find(d => d.nameO == Some(s)) match {
         case None => false // in practice, throw "symbol not declared" error
         case Some(d) => d match {
           case FunctionSymbolDeclaration(_, ins,out) =>
             if (ins.length != args.length) {
               false // in practice, throw "wrong number of argumetns" error
             } else {
               // check every argument recursively
               args.forall(a => check_Expr(voc, a))
             }
         }
       }
   }
}
```

```
  def check_Form(voc: Vocabulary, f: Form): Boolean = {
    f match {
      case Equals(l,r) => check_Expr(voc, l) && check_Expr(voc, r)
      case LessEq(l,r) => check_Expr(voc, l) && check_Expr(voc, r)
    }
  }
}
```

The following code shows a simple test of the type checker:

```
object Test {
    def main(args: Array[String]) = {
        // because we do not have a parser, we need to build some objects manually for testin

        // an example vocabulary
        // fib: Nat -> Nat
        val fibDecl = FunctionSymbolDeclaration("fib", List(Nat()), Nat())
        val voc = Vocabulary(List(fibDecl))
        System.out.println(voc.print())
        // check the vocabulary
        System.out.println(TypeChecker.check_Voc(voc))

        // example expressions relative to that vocabulary
        val x = Sum(Zero(), One())
        val y = FunctionSymbolReference("fib", List(x))
        val z = LessEq(x,y)
        System.out.println(z.print())
        // check the expressions
        System.out.println(TypeChecker.check_Form(voc, z))
    }
}
```

3.4.6 Implementing Context-Sensitive Checking with Variables

In languages with variable binding, we have to extend the above implementation: Every checking function for expressions, which used to take a vocabulary, now takes a vocabulary and a variable context.

Chapter 4

Type Systems

4.1 Intrinsic vs. Extrinsic Typing

4.1.1 Overview

We write $x : A$ to say that x has type A. There are two fundamentally different methods for introducing the types A, which are summarized in the following table:

	intrinsic	extrinsic
goes back to λ-calculus by	Church	Curry
general idea	objects carry their type with them	types are designated by the environment
typing is a	function from objects to types	relation between objects and types
objects have	unique type	any number of types
types often interpreted as	disjoint sets	unary predicates on a universal set
type inference for x	uniquely infer A from x	try to find minimal A such that $x : A$
type checking	compare inferred and expected type	prove $x : A$
subtyping $A <: B$	mimicked by casting from A to B	defined by $x : A$ implies $x : B$ for all x
typing decidable	yes unless too expressive	no unless expressivity restricted
typing errors are detected	usually statically (compile-time)	dynamically (run-time)
type of name introduced as	part of declaration	additional axiom
example	**individual** "WuV":"course"	**individual** "Wuv", "WuV" `is-a` "course"
advantages	easy	flexible
	unique type inference	allows subtyping
examples	SFOL, SQL	OWL, Scala, English
	most logics, functional PLs	ontology, OO, natural languages
	many type theories	set theories

Example 4.1 (Extrinsically Typed Ontology Language). In BOL, the objects are the individuals, the types are the concepts, and `is-a` is the typing relation between them. The typing is extrinsic:

- Individuals and their concept assertions are introduced in separate declarations.
- An individual may be an instance of any number of concepts.
- There is no primary concept that could be returned as the inferred type of an individual.
- Concepts are subject to subtyping $C \sqsubseteq C'$.
- Whether an individual is an instance of a concept, must be checked by reasoning about the `is-a` relation.

Therefore, all semantics must interpret individuals as elements of a universal collection, and types as unary predicates on that. Specifically, we have

semantics in	universal collection	unary predicate	typing relation i `is-a` c
FOL	type ι	predicate $c \subseteq \iota$	$c(i)$ true
SQL	table Individuals	table containing ids	id of i in table c
Scala	String	hash set of strings	c.contains(i)
English	proper nouns	common nouns	"i is a c" is true

We can also think of relations as objects. However, BOL cannot express relation types at all, and there is no intrinsic typing. Instead, the domain and range of a relation r are given extrinsically via axioms about $\operatorname{dom} r$ and $\operatorname{rng} r$. Like for individuals that allows flexibility as the same relations may have multiple types.

Example 4.2 (Intrinsically Typed Ontology Language). We could define TOL, a typed ontology language that arises as a variant of BOL. The main differences would be

- Individuals are declared with a concept that serves as their type: **individual** $i : C$.
- Concept assertions are dropped. They are now part of the individual declarations.
- Relations are declared with two concepts for their domain D and range R: **relation** $r <: D \times R$.
- Properties are declared with a concept for their domain C: **property** $p <: C \times T$.

TOL would make many ontologies more concise. For example, we could simply write

```
concept instructor
concept course
individual FlorianRabe : instructor
teach <: instrctor × course
```

However, we would lose flexibility. If we want to add the concept "male", it would be difficult to make `FlorianRabe` have both types. We might be able to remedy that by allowing intersections and declaring individual FlorianRabe: instructor $\sqcap$ male. But even then, we would have to commit to the type of each individual right away — we cannot add different concept assertions for the same individual in different places, a common occurrence in building large ontologies.

Allowing $\sqcap$ would also introduce subtyping. If we are careful in the design of TOL, that may still result in an elegant scalable language. In particular, typing may remain decidable (depending on what other operations we allow). But if we go too far, it may end up so complex that it would have been easier to go with extrinsic typing.

That is why we use intrinsic typing only in two related places in BOL:

- The base types and values use an intrinsic type system (whose details we omitted).
- The range of properties is given intrinsically by a base type.

Remark 4.3 (Subtyping). Languages with subtyping usually have to use extrinsic type systems. Typical sources of subtyping are

- explicit subtyping as in $\mathbb{N} <: \mathbb{Z}$
- comprehension/refinement as in $\{x : \mathbb{N} | x \neq 0\}$
- operations like union and intersection on types
- inheritance between classes, in which case subclass = subtype
- anonymous record types as in $\{x : \mathbb{N}, y : \mathbb{Z}\} <: \{x : \mathbb{N}\}$

4.1.2 Combined Definition

Neither intrinsic nor extrinsic typing is strictly better than the other. The choice of type system is a very difficult trade-off when designing a language.

Many practical languages even combine both methods. In that case, an intrinsic system is used for the most important high-level types and an extrinsic system is used to refine (some of) the high-level types:

Definition 4.4 (Type System)**.** A **type system** for a context-free syntax consists of provides for some expression symbols $\mathcal{E}$

- a non-terminal $\mathcal{T}$ — $\mathcal{T}$-expressions are called the **types** of $\mathcal{E}$-expressions
- a binary predicate $\vdash_\Theta E : T$ on $\mathcal{E}$-expression E and $\mathcal{T}$-expressions T — if satisfied, we say that E **has type** T.

We can now recover the intuitions from above as special cases:

- A purely intrinsic type system is one in which the types correspond to the expression symbols themselves. Here each $\mathcal{E}$-expression has type $\mathcal{E}$, and the grammar already separates expressions by their (intrinsic) types.
- A purely extrinsic type system has two non-terminals $\mathcal{E}$ and $\mathcal{T}$. Here all expressions are merged into the same non-terminal, i.e., the grammar does not distinguish expressions by their types. All typing is done extrinsically by the typing predicate.

Example 4.5. We can think of BOL as a combined type system. The intrinsic types are the non-terminals I, C, R, P, and F, which separate the expressions into the five kinds of individuals, concepts, relations, properties, and formulas.
An extrinsic typing relation exists for I: the types for $\mathcal{E} = I$ are those derived from $\mathcal{T} = C$, i.e., the individuals are extrinsically typed by the concepts. The typing relation $\vdash_\Theta E : T$ is given by the formula $E\,\texttt{is-a}\,T$ is true relative to Θ.

Example 4.6. In set theory, only a few intrinsic types are used for the high-level grouping of objects. These include at least *set* and *prop*. Objects of these intrinsic types are called sets and propositions. Some set theories also use an intrinsic type *class*. Moreover, types like $set \to prop$ can be allowed as the types of unary predicates on sets.
Extrinsic typing is used only for $\mathcal{E} = set$: we have $\mathcal{T} = set$ and the typing relation is the elementhood relation $\in$ between sets.

Example 4.7. In some sense every language has a type system. What is commonly called an untyped language, is technically an intrinsically typed language with only one type.
However, in practice, almost all of this languages have more than one type or some kind weak of additional extrinsic typing. For example, untyped first-order logic really has two intrinsic types: terms and propositions. Python has one intrinsic type of expressions, but these are extrinsically subdivided into lists, dictionaries, integers, etc.

4.2 Variable Contexts

In the presence of a type system, we often use typing in context.

Definition 4.8 (Context)**.** A variable declaration is of the form $x : A$ for an identifier x and a type A. A variable context is a list of variable declarations.
An expression in context Γ may additionally use the variable from Γ. We write $\Gamma \vdash_\Theta E : A$ to say that expression E has type A in context Γ over vocabulary Θ.

Example 4.9. Consider the language from Ex. 3.27 with the standard library from Ex. 3.26. Then $x : Nat, y : Nat \vdash equals(x, y) : FORM$ is a formula in context.

Example 4.10. We can also use the non-terminal expression symbols as types. For example, BOL has no variable binding and therefore no contexts in the grammar. But we still have at least contexts for the intrinsic type system given by the grammar. For example, we can use a context $c : C, d : C$ for two concept variables and give the formula $c : C, d : C \vdash c \sqsubseteq d : F$.

In general, for any context-free grammar, we can use the non-terminals as intrinsic types and form contexts. Then $x : N$ means that x is an arbitrary word derived from the non-terminal N. Such contexts are very common when talking about a language. We speak of meta-contexts. For example, in BOL, we might say "Given a concept c and a relation r, ..." — that corresponds to using the context $c : C, r : R$.

Definition 4.11 (Substitution)**.** Given two contexts Γ and Δ, a substitution provides for every variable $x : A$ exactly one variable assignment x/e where e is an expression from Δ.
For a Γ-expression E, we write $E[\gamma]$ for the Δ-expression resulting from substituting every x in E with the corresponding assignment from γ.
A substitution is well-typed, written $\gamma : \Gamma \to \Delta$, if every pair of and $x : A$ in Γ and x/e in γ satisfies $\Delta \vdash e : A[\gamma]$.

Example 4.12 (Identity). The identity substitution id_Γ from Γ to itself assigns x/x for every x in Γ.
The mapping $E \mapsto E[id_\Gamma]$ is the identity mapping on Γ-expressions.

Example 4.13 (Composition). Given substitutions $\gamma_1 : \Gamma_1 \to \Gamma_2$ and $\gamma_2 : \Gamma_2 \to \Gamma_3$, the composition $\gamma_1; \gamma_2 : \Gamma_1 \to \Gamma_3$ assigns $x/x[\gamma_1][\gamma_2]$ for every x.
The mapping $E \mapsto E[\gamma_1][\gamma_2]$ is the composition of the mappings $E \mapsto E[\gamma_1]$ and $E \mapsto E[\gamma_2]$.

Virtually every type system satisfies the following property. Technically, this must be proved separately for every type system because it is not guaranteed to hold. But in practice, a type system that does not satisfy it is simply broken:

Theorem 4.14 (Type Preservation)**.** *Consider a reasonably defined type system.*
Given a substitution $\gamma : \Gamma \to \Delta$ *and an expression* $\Gamma \vdash E : A$*, we have* $\Delta \vdash E[\gamma] : A[\gamma]$*.*

4.3 Abstract and Concrete Data Types

The general thrust of type systems is to shift more and more information into an increasingly complex type system. This is part of a trade-off: the more the type system can do,

- the more requirements can be expressed and violations thereof detected statically,
- the more complex the type system and its documentation and implementation become.

Concrete and abstract data types have proved to be a particularly interesting trade-off on this expressivity-simplicity spectrum and are — in one way or another — part of many type systems.

4.3.1 Abstract vs. Concrete Types

The words *abstract* and *concrete* do not have standard definitions for types. I like the intuitions described below.
A type is called **concrete** if its values are

- given by their internal form,
- defined along with the type, typically built from already-known pieces.

In other words, a concrete type is given by an *extensional* description of its objects.

Example 4.15. Simple products are concrete types.
They are introduced by (among other rules)

- $A \times B$ is a type if A and B are
- values of type $A \times B$ are of the form (a, b) for $a : A$ and $b : B$.

Example 4.16. **Enumerations** are concrete types that explicitly list a finite set of distinct values.
They can also be seen as the special case of inductive data types in which all constructors take 0 arguments.

Example 4.17. Inductive data types as seen in Def. 3.8 are concrete types. Their values are formed by applying constructors to other values.
I like calling them **concrete data types**.

A type is called **abstract** if its values are

- given by their externally visible properties,
- defined in any environment that understands the type definition.

In other words, an abstract type is given by an *intensional* description of its objects.
This is the case for **abstract data types**.

4.3.2 Concrete Data Types

Concrete data types are the same as inductive data types and were already defined rigorously in Sect. 3.2.2.
It is in the nature of concrete types that they do not depend on the situation in which they are used. The definition of the type once and for all defines the possible values. Thus, they do not differ (or differ at most subtly) between languages. Often the only real difference is whether a language has concrete data types or not. Languages that do not natively provide them may differ substantially in how they simulate them though.

4.3.3 Abstract Data Types

Contrary to concrete types, the values of an abstract data types depend on what features the language provides to construct objects that satisfy the abstract requirements. Therefore, the treatment differs widely across languages. The following table gives an overview:

aspect	language	abstract data type
ontologization	UML	class
concretization	SQL	table schema
computation	Scala	class, interface
deduction	various	theory, specification, module, locale
narration	various	emergent feature

Example 4.18 (Classes). A UML class is an abstract data type. Its values are the instances of implementing classes.

A UML class only defines what methods should be available. How they are implemented by specific values of the type is left to the programming languages.

Thus, different programming languages could have different values for the same abstract data type. They certainly look different, e.g., in Java and Scala implementations of the same UML class. But the languages might also be fundamentally different in expressivity, e.g., a Turing-complete programming language might have strictly more values for the same abstract data type than a non-Turing-complete one.

Moreover, which instances actually exist changes during the run time of the program. If we take this into account, the values of the abstract data type are not even fixed within a programming language.

Example 4.19 (Schemas). An SQL table schema is an abstract data type. Its values are the rows.

The schema only defines what types the columns of a table have. Different database systems might theoretically provide different ways to build rows for the table.

However, this does not happen in practice because SQL table columns are typed by base types, which have the same values across database systems. This would be different if we allowed table columns to have function types.

Example 4.20 (Theories). A logical theory (e.g., Monoid) is an abstract data type. Its values are the models of the theory (e.g., for Monoid: $(\mathbb{N}, +, 0)$ or $(\mathbb{N}, *, 1)$).

The theory only defines what operations a model must provide (for Monoid: binary operation and neutral element) and which axioms it must satisfy (for Monoid: associativity, neutrality). How we build the models is left open.

We usually build models in mathematical language and naively assume that fixes the models once and for all. But that is too naive: depending on which mathematical foundation we use (e.g., set theory with or without axioms of choice), we can build different models. Moreover, we can also build models in type theories (which underly many deduction systems such as Coq or Isabelle). We can even build them in programming languages, e.g., by implementing theories as classes (typically moving the axioms into comments).

The choice of language substantially changes what the values of the abstract data type are.

In the sequel we introduce a fairly general definition that subsumes many practical languages.

Definition 4.21 (Abstract Data Type)**.** Consider an arbitrary type system.

An **abstract data type** (ADT) is

- a **flat** type of the form
$$\{c_1 : T_1[= t_i], \ldots, c_n : T_n[= t_i]\}$$
where the c_i are distinct names, the T_i are types, and the t_i are optional and wherever given must have type T_i, or
- a **mixin** type of the form $A_1 * A_2$ for ADTs A_i.

We say that a type system has **internal ADT**s if all ADTs are types (and thus may in particular occur as the T_i in a record type).

The intuition of a mixin $A * B$ is that we merge the fields of A and B. However, this union dependent: if B is flat, its fields may refer to fields introduced in A.

The most important special case of an ADT are classes:

Definition 4.22 (Class)**.** A class definition defines an ADT abbreviation of the form

$$a = a_1 * \ldots * a_m * \{c_1 : T_1, \ldots, c_n : T_n\}$$

where the a_i are names of previously defined ADTs.
We call the a_i the **superclasses** or **parent classes** and say that a inherits from the a_i. We call the c_i the **fields** or **members** of a.

In an OO-language, a class definition is more commonly written somehow like

```
abstract class a extends a_1 with ... with a_m {
  c_1 : T_1
  ⋮
  c_n : T_n
}
```

The details can vary, and special care must be taken in programming languages where initialization may have side effects.
Flat ADTs are the standard case, and all mixin ADTs can be simplified into flat ones. This can be seen as a semantics in the sense that the language of flat and mixin ADT is translated to the language of flat ADTs.

Definition 4.23 (Mixin Semantics)**.** The **flattening** $A^\flat$ of an ADT A is defined as follows:

- if A is flat: $A^\flat = A$
- if A is of the form $A_1 * \ldots * A_n$: $A^\flat$ arises by concatenating the fields of all $A_i^\flat$ where duplicate field names are handled as follows:
  - if the same field (same name, types equal, definitions equal or both absent) occurs more than once, only the first occurrence is kept,
  - if the fields $c : T_1[= t_i]$ and $c : T_2[= t_2]$ occur for inequal types T_i, A is ill-formed,
  - if the fields $c : T = t_1$ and $c : T = t_2$ occur for inequal objects t_i, A is ill-formed,
  - if the fields $c : T = t$ and $c : T$ occur, only the defined one is kept $(*)$.

Remark 4.24 (Dependency Between Fields)*.* Our definition sweeps a very important but subtle detail under the rug: in a flat ADT with a field $c : T = t$, may T and/or t refer to fields declared later? We sketch a few possible answers.
In the simplest case, we forbid such forward references. Then ADTs are very well-behaved. But we have a problem with the case $(*)$ in Def. 4.23: if $c : T$ occurs before $c : T = t$, we cannot simply drop the former because intermediate fields may refer to c. A straightforward solution would be to declare the ADT to be ill-formed. But unfortunately, this case is very important in practice — it occurs whenever $c : T$ is declared in an abstract class and $c : T = t$ in a concrete class implementing it.
A more common solution is to allow the fields to be mutually recursive. Consider a flat ADT with fields $\Gamma, c : T[= t], \Delta$ where Γ and Δ are lists of fields. Let Γ' and Δ' arise by dropping all definitions. Then we require that

- T must be a well-formed type in context Γ'. Thus, the types may only refer to previous fields.
- t must have type T in context $\Gamma', c : T, \Delta'$. Thus, the definitions may be mutually recursive.

This makes the case $(*)$ work. But it comes at the price of recursion, which allows writing non-terminating fields (a feature in a programming language, but potentially undesirable in other settings).
Even so, the mutual-recursion solution is problematic in the presence of dependent types. Here, dropping definitions is not always allowed: T might be well-formed in context Γ, but Γ' might not even be a well-formed context at all. Because OO-languages are usually not dependently-typed, this is not an issue in most settings.

4.4 Abstract Data Types in Ontologies

4.4.1 Motivation

Recall the subject-centered representation of individuals described in Sect. 2.3. Here we introduce an individual together with all assertions of which it is the subject as in

```
individual "FlorianRabe"
  is-a "instructor" "male"
  "teach" "WuV" "KRMT"
  "age" 40
  "office" "11.137"
```

It is often desirable to use types to force the presence of such assertions. We might wish require that every instructor teaches a list of things, and has an office. Moreover, we can use types to specify the objects of the respective assertions: we can specify that only courses are taught and that the office is a string. Rather than the relations with subjects "FlorianRabe" just happening to be around as well, the type system would now force their existence and the type of the object. Forgetting to give such an assertion or giving it with the wrong object could be detected statically (i.e., without applying the semantics) and flagged as a typing error.

This leads to the idea of **subject-centered types**. This could looks as follows:

```
concept instructor
  teach course*
  age: int
  office: string

individual "FlorianRabe": "instructor"
  is-a "male"
  "teach" "WuV" "KRMT"
  "age" 40
  "office" "11.137"
```

Now the type "instructor" forces the presence of a list of taught courses (The * is meant to indicate a list.), an integer for the age, and a string for the office.

We can now see that, in fact, every person should have an age, and not just every instructor. Because every instructor is meant to be a person, we could try to capture this as well to avoid redundancy. Moreover, every male is meant to be a person, too.

That leads to the idea of **modular types**. This could look as follows:

```
concept person
  age: int

concept male <: person

concept instructor <: person
  teach course*
  office: string

individual "FlorianRabe": "instructor" ⊓ "male"
  "teach" "WuV" "KRMT"
  "age" 40
  "office" "11.137"
```

Incidentally, that eliminates the need to independently declare relations and properties. Instead, we can treat their occurrences inside the concept definitions as their declarations.

That has the added benefit that two relations/properties of the same name declared in different concepts can be distinguished and can have different types.

4.4.2 Database Schemas as Typed Ontologies

It is very typical that optimized systems are developed for one of the corner aspects. These systems then subsume an ontology language. From the Tetrapod perspective, it is instructive to tease apart the ontology.

In particular, what is typically called an ontology follows the semantic web community. Tetrapodally, they are working on the combination of ontology and concrete data. The former defines the rules of the world (TBox), the latter populates the world (ABox).

We can see the same pattern in the relational database community. Tetrapodally, the SQL schema defines the ontology, and the database holds the concrete data. Fig. 4.1 shows the analogy in detail.

	semantic web	relational databases
ontology aspect	TBox of ontology	SQL schema
conceptual model	knowledge graph	set of tables
concrete data aspect	ABox of ontology	SQL database
concrete data storage	set of triples	set of rows of the tables
concrete data formats	RDF	CSV
concrete data tool	triple store	database implementation
typing	soft/Curry	hard/Church
query language	SPARQL	SQL SELECT query
openness of world	tends to be open	tends to be closed

Figure 4.1: The Two Kinds of Concrete Data Systems

We can then understand relational databases as an ontology+concrete data approach that focuses on abstract data types. Each table is one such type, and each column is a field of the type. The database holds the concrete data, i.e., the rows of the tables.

For example, the SQL schema might declare an ADT by

```
TABLE person(id: ID, name: string, age: int)
TABLE male(id: Ref person)
TABLE instructor(id: ID, parent: Ref person, teach: course*, office: string)
```

Here we write Ref T for the type ID of identifiers if we want to emphasize that it should hold an identifier that is present in the table T.

Concepts like male can be represented as single-column tables holding the identifiers of individuals that are in the concept. Inheritance can be represented by using two separate tables for parent and child type with the latter holding a special column parent holding an identifier in the parent table. For example, concrete data could be inserted as

```
INSERT VALUES (id = X, name = "FlorianRabe", age = 40) INTO person
INSERT VALUES X INTO male
INSERT VALUES (id = Y, parent = X, teach = ["WuV", "KRMT"], office = "11.137")
  INTO instructor
```

Usually, relations where the same subject can have multiple objects are not stored as a list-valued column but as a separate two-column table. For example, we would have

```
TABLE instructor(id: ID, parent: Ref person, office: string)
TABLE course(id: ID, name: string)
TABLE teach(subject: Ref person, object: Ref Course)
INSERT VALUES (id = W, name = "WuV") INTO course
INSERT VALUES (id = K, name = "KRMT") INTO course
INSERT VALUES (subect = X, object = W) INTO teach
INSERT VALUES (subect = X, object = K) INTO teach
```

4.4.3 Exercise

The topic of Exercise is to build a relational database schema for a univis-like system.

Chapter 5

Data

5.1 Overview

This chapter section presented ongoing research on developing infrastructure for the semantic representation and interchange of data across systems. It is presented on the slides and the lecture videos.

Chapter 6

Semantics

This chapter introduces different kinds of semantics. The emphasis will be on relative semantics by compositional translations.

We use BOL as a running example and give four different semantics of BOL — using the four other aspects:

Section	Aspect	kind of semantic language	semantic language
6.3	deduction	logic	SFOL
6.4	concretization	database language	SQL
6.5	computation	programming language	Scala
6.6	narration	natural language	English

6.1 Kinds of Semantics

To define a language system, we need to define the well-formedness predicates. Moreover, we need to define the semantics of the well-formed vocabularies and expressions. One way to do that is by translating the syntax into another language and then use existing definitions of well-formedness there.

But we also need to be able to get off the ground, i.e., to define a semantics from scratch when we do not have another language available. This is usually done by giving an inference system for the well-formedness predicates and other judgments such as truth.

We can also do both: if we have a semantics via an inference system and another one via translation, we can show that the latter respects the former. That leads to the concepts of soundness (everything well-formed is translated to something well-formed) and its dual completeness.

6.1.1 Absolute Semantics

Absolute semantics is very hard to define in general. A typical approach is to single out one property of the syntax and then use an inference system to define when that property holds.

For example, for BOL we could use the property whether a formula can be proved from the axiom. The inference system then is the proof calculus that define the provable formulas.

6.1.2 Relative Semantics

Relative semantics uses an additional component as a reference point for semantics.

By Translation Semantics by translation uses a second language as the target of a translation. Both the vocabularies and the expressions are translated, namely to vocabularies and expressions of the target language. Usually, the target language already has a semantics, and the semantics of the original language is obtained by composing the two.

The correspondence for the syntax between context-free grammars and inductive data types can be extended to the semantics. Now we have a correspondence between case-based function definitions and inductive functions.

Aspect	Example
Deductive	FOL → set theory
Computational	C++ → assembly
Concrete	SPARQL → SQL
Narrative	English → German

Figure 6.1: Examples of Semantics by Translation

Definition 6.1. A **semantics by translation** consists of the following parts:

- syntax: a language system l
- semantic language: a language system L (from a different or the same aspect as l)
- semantic prefix: a vocabulary P in L that is prefixed to the translation of all vocabularies of l
- vocabulary translation: a function that translates every l-vocabulary T to an L-vocabulary $P, [\![T]\!]$

Critically, the semantic language (which is itself a language system and can thus have a semantics itself) must be a language whose semantics we already know. Therefore, it is often important to give multiple equivalent semantics — choosing a different semantics for different audiences, who might be familiar with different languages.

The role of the semantic prefix P is to define once and for all the L-material that we need in general to interpret l-vocabularies (in the case of BOL: ontologies). It occurs at the beginning of all interpretations of ontologies. In particular, it is equal to the interpretation of empty vocabulary.

By Interpretation Semantics by interpretation uses a situation relative to which expressions are evaluated. The role of the situation is to supply the meaning of all identifiers declared in the vocabulary. Given a situation, inductive functions map all expressions to their semantics. Usually, the result of interpretation is an extremely simple object, whose semantics is self-evident.

Aspect	Situation	Provides
Deductive	model	interpretations of the function/predicate/... symbols
Computational	environment	e.g., standard input/output
Concrete	database	set of objects for each type
Narrative	semantic dictionary	meanings of the words

Figure 6.2: Examples of Situational Semantics by Interpretation

Situational semantics can be seen as a special case of semantics by translation as follows:

- The target language is an implicit assumed background language such as mathematics or the computer hardware. Once this is made explicit, situations can be described as vocabularies of the background language.
- The semantic prefix describes what a situation is in general, i.e., the possible situations for the empty vocabulary.
- The semantics of a vocabulary extends the semantic prefix with the possible ways to interpret the identifiers.
- The semantics of an expression E is the function mapping the situation S to the situational semantics of E under S. This can be expressed as an expression of the target language.

6.2 Compositionality

Introduction There are some general principles shared by all translations:

- Every l-declaration is translated to an L-declaration for the same name, and ontologies are translated declaration-wise.
- For every non-terminal N of l, there is one inductive function $[\![-]\!]_N$ mapping complex l-expressions derived from N to L-expressions.

- The base cases of references to declared l-identifiers are translated to themselves, i.e., to the identifiers of the same name declared in L.
- The other cases are compositional: every case for a complex l-expression recurses only into the semantics of the direct subexpressions.

The notion of *compositionality* captures these properties. An interpretation function is compositional if the interpretation of any kind of expression $E(e_1, \ldots, e_n)$ with subexpressions e_i only depends on E and the interpretation of the e_i, i.e.,

$$[\![E(e_1, \ldots, e_n)]\!] = [\![E]\!]([\![e_1]\!], \ldots, [\![e_n]\!])$$

for some semantic operation $[\![E]\!]$. Compositionality is also called the substitution property or the homomorphism property. See also Def. 6.11.

More rigorously, we define a compositional translation as follows:

Definition 6.2 (Compositional Semantics)**.** Consider a semantics for syntax grammar l and interpretation function $[\![-]\!]$.

$[\![-]\!]$ is compositional if it is defined as follows:

- a family of functions $[\![-]\!]_N$, one for every non-terminal N of l
- for every expressions E derived from N, we put $[\![E]\!] = [\![E]\!]_N$
- each $[\![-]\!]_N$ is defined by induction on the productions for N
- for each production $N ::= *(N_1, \ldots, N_r)$ and all expressions e_i derived from N_i

$$[\![*(e_1, \ldots, e_r)]\!]_N = [\![*]\!]([\![e_1]\!]_{N_1}, \ldots, [\![e_r]\!]_{N_r})$$

  for some L-expression $[\![*]\!]$

Without loss of generality, we can assume that every production is of the form $N ::= *(N_1, \ldots, N_r)$ where the N_i are all the non-terminals on the right-hand side and $*$ is a stand-in for all the terminal symbols.

Compositional Translations of Contexts We can extend every compositional translation to contexts, substitutions, and expressions in contexts:

Definition 6.3. Given a translation $[\![-]\!]$ as above, for a non-terminal N, we define $[\![N]\!]$ as the non-terminal from which the translations of N-expressions are derived.

Then we define:

$$[\![x_1 : N_1, \ldots, x_n : N_n]\!] := x_1 : [\![N_1]\!], \ldots, x_n : [\![N_n]\!]$$

$$[\![x_1 := w_1, \ldots, x_n := w_n]\!] := x_1 := [\![w_1]\!], \ldots, x_n := [\![w_n]\!]$$

$$[\![x]\!] := x$$

The requirement of compositionality is critical for two reasons:

- A non-compositional translation could translate l-expressions derived from the same non-terminal N to L-expressions derived from different non-terminals. Then we would not be able to define $[\![N]\!]$.
- The definition $[\![x]\!] := x$ adds a case to the case distinction in the compositional translation function. Without compositionality, this would not make sense.

Theorem 6.4 (Type Preservation)**.** *For a compositional translation as above, we have*

$$\Gamma \vdash_l w : N \quad \text{implies} \quad [\![\Gamma]\!] \vdash_L [\![w]\!] : [\![N]\!]$$

Substitution Theorem The main value of compositionality is the following:

Theorem 6.5 (Substitution Theorem)**.** *Consider a compositional semantics.*
For every context $\Gamma = x_1 : N_1, \ldots, x_r : N_r$, every syntax expression $\Gamma \vdash_l E : N$, and every substitution $\vdash_l \gamma : \Gamma$

$$[\![E[\gamma]]\!] = [\![E]\!][[\![\gamma]]\!]$$

Formulated without substitutions, this means that for every syntax expression $E(e_1, \ldots, e_r)$ derived from N, where the e_i are subexpression derived from non-terminal N_i, we have

$$[\![E(e_1, \ldots, e_n)]\!]_N = [\![E]\!]([\![e_1]\!]_{N_1}, \ldots, [\![e_n]\!]_{N_r})$$

Simply put, a semantics is compositional iff it is defined by mutually inductive translation functions with only compositional cases. The latter is very easy to check by inspecting the shape of the finitely many cases of the definition. The former is a powerful property because it applies to any of the infinitely many expressions of the syntax.

Non-Compositional Semantics It is highly desirable but not always possible to give a compositional translation. Sometimes a feature of the syntactic language cannot be directly interpreted in the semantic language. In that case, it may still be possible to give a non-compositional translation.

Example 6.6 (Non-Compositional Translation via Sub-Induction). A simple example of non-compositionality is the translation of natural numbers based on zero, one, and addition (i.e., $N ::= 0 \mid 1 \mid N + N$) into natural numbers based on zero and successor (i.e., $N ::= 0 \mid \texttt{succ}(N)$): It is straightforward to translate zero and one compositionally:

$$[\![0]\!] = 0 \qquad [\![1]\!] = \texttt{succ}(0)$$

Now we would like to translate

$$[\![m + n]\!] = [\![+]\!]([\![m]\!], [\![n]\!]),$$

but there is no way to define $[\![+]\!]$ in terms of zero and successor. Instead, we need subcases:

$$[\![m + n]\!] = \begin{cases} [\![m]\!] & \text{if } n = 0 \\ \texttt{succ}([\![m]\!]) & \text{if } n = 1 \\ [\![(m + n_1) + n_2]\!] & \text{if } n = n_1 + n_2 \end{cases}$$

This corresponds to the usual definition of addition, i.e., $[\![+]\!]$, by induction.

Other common examples of non-compositional translations are

- several important logical theorems such as
  - cut elimination, which is the translation from sequent calculus with cut to sequent calculus without cut,
  - the deduction theorem, which is the translation from natural deduction to Hilbert calculus,
- almost anything done by an optimizing compiler, e.g., loop unrolling or function inlining,
- query optimization done by a database, e.g., turning a WHERE of a JOIN into a JOIN of WHEREs,
- almost all translations between natural languages, e.g., when words are ambiguous and a different translation must be chosen for the same word based on the context (The introduction of richer intermediate structures like ASTs and functions as values into the translation can recover some compositionality here).

Typical sources of non-compositionality in formal language translations are:

- A case in the translation function requires subcases which inspect the e_i and treat them differently.
- A case in the translation function requires subcases which translate an expression differently based on the context in which it occurs.
- The translation function requires nested inductions, i.e., a case in the translation function (which is already inductive) requires a sub-induction on one of the sub-expressions.
- The semantic prefix is not fixed but depends on the translated object, i.e, the top-level case of the translation scans through the entire argument X to collect all occurrences of a particular feature and then custom-builds the semantic prefix of $[\![X]\!]$.

See also Ex. 6.12.

Such non-compositional translations are undesirable for multiple reasons:

- The implementation is more complicated and error-prone.
- Reasoning about the translation is more difficult.
- The custom semantic prefix can be large.

But most importantly, non-compositional translations are less robust. Firstly, if we add a production to the syntax, a compositional translation is easy to extend: just add a case to the translation. But a non-compositional translation

may additionally require a new subcase wherever subcases/subinductions are used. Moreover, if a custom semantic prefix is used, its definition may have to be amended, at least it must be rechecked.

Secondly, in practice there are two sources of complex expressions: the ones already mentioned in the language, and the ones used later for other reasons. For example, some complex expressions occur already *statically* in the definition of a vocabulary V. But others might be appear *dynamically* later, e.g., when talking about V, proving properties of V, or running queries on V. Thus, the definition of V and the use of complex expressions are decoupled: V is defined statically once and for all, and complex expressions relative to V can be created and used dynamically. But if a custom semantic prefix is used, only the static occurrences inside V can be considered for building the prefix. Thus, it is not possible to translate expressions dynamically unless the semantic prefix is extended all the time while V is used.

6.3 Deductive Semantics

We fix one language that we have already understood and define an interpretation function that maps all complex expression of BOL to the semantic language. For simple ontology languages like BOL, ALC, OWL, etc., it is common to use first-order logic (FOL) as the deductive semantic language. More specifically, we use SFOL, the typed variant of FOL:

6.3.1 A Basic Semantic Language: SFOL

We give typed first-order logic (SFOL) as a language system.

Definition 6.7. Fig. 6.3 gives the context-free grammar. The vocabulary symbol is Thy. The expression symbols are Y, T, and F.

6.3.2 Semantics

Definition 6.8 (Deductive Semantics of BOL)**.** The semantic prefix is the SFOL-theory containing

- a type ι (for individuals),
- additional types and constants corresponding to base types and values of BOL.

Every BOL-ontology O is interpreted as the SFOL-theory $P, [\![O]\!]$, where $[\![O]\!]$ is defined in Fig. 6.4.

As foreshadowed above, we can observe some general principles: Every BOL-declaration is translated to an SFOL-declaration for the same name, and ontologies are translated declaration-wise. For every kind of complex BOL-expression, there is one inductive function mapping BOL-expressions to SFOL-expressions. The base cases of references to declared BOL-identifiers are translated to themselves, i.e., to the identifiers of the same name declared in the SFOL-theory. The other cases are compositional: every case for a complex BOL-expression recurses only into the semantics of the direct subexpressions.

Vocabularies: theories

$Thy ::= D^*$

Declarations

D	$::=$ **type** `ID`	type declaration
	$\mid$ **fun** `ID` $: Y^* \to Y$	function symbol declaration
	$\mid$ **pred** `ID` $\subseteq Y^*$	predicate symbol declaration
	$\mid$ **axiom** F	axiom

type expressions

Y	$::=$ `ID`	atomic type

term expressions

T	$::=$ `ID`(T^*)	function symbol applied to arguments
	$\mid$ `ID`	term variables

formula expressions

F	$::=$ `ID`(T^*)	predicate symbol applied to arguments
	$\mid T \doteq_Y T$	equality of terms at a type
	$\mid \top$	truth
	$\mid \bot$	falsity
	$\mid F \wedge F$	conjunction
	$\mid F \vee F$	disjunction
	$\mid F \Rightarrow F$	implication
	$\mid F \Leftrightarrow F$	equivalence
	$\mid \neg F$	negation
	$\mid \forall$`ID` $: Y.F$	universal quantification at a type
	$\mid \exists$`ID` $: Y.F$	existential quantification at a type

Identifiers

`ID` $::=$ alphanumeric string

Figure 6.3: Syntax of SFOL

BOL Syntax X	Semantics $[\![X]\!]$ in SFOL
ontology	SFOL theory
$D_1, \ldots, D_n$	$[\![D_1]\!], \ldots, [\![D_n]\!]$
BOL declaration	FOL declaration
individual i	nullary function symbol $i : \iota$
concept i	unary predicate symbol $i \subseteq \iota$
relation i	binary predicate symbol $i \subseteq \iota \times \iota$
property $i : T$	binary predicate symbol $i \subseteq \iota \times T$
I `is-a` C	axiom $[\![C]\!]([\![I]\!])$
$I_1\ R\ I_2$	axiom $[\![R]\!]([\![I_1]\!], [\![I_2]\!])$
$I\ P\ V$	axiom $[\![P]\!]([\![I]\!], [\![V]\!])$
F	axiom $[\![F]\!]$
Formula	Formula without free variables
$C_1 \equiv C_2$	$\forall x : \iota.[\![C_1]\!](x) \Leftrightarrow [\![C_2]\!](x)$
$C_1 \sqsubseteq C_2$	$\forall x : \iota.[\![C_1]\!](x) \Rightarrow [\![C_2]\!](x)$
I `is-a` C	$[\![C]\!]([\![I]\!])$
$I_1\ R\ I_2$	$[\![R]\!]([\![I_1]\!], [\![I_2]\!])$
$I\ P\ V$	$[\![P]\!]([\![I]\!], [\![V]\!])$
Individual	Terms of type ι
i	i
Concept	Formula with free variable $x : \iota$
i	$i(x)$
$\top$	*true*
$\bot$	*false*
$C_1 \sqcup C_2$	$[\![C_1]\!](x) \vee [\![C_2]\!](x)$
$C_1 \sqcap C_2$	$[\![C_1]\!](x) \wedge [\![C_2]\!](x)$
$\forall R.C$	$\forall y : \iota.[\![R]\!](x, y) \Rightarrow [\![C]\!](y)$
$\exists R.C$	$\exists y : \iota.[\![R]\!](x, y) \wedge [\![C]\!](y)$
$\mathrm{dom}\, R$	$\exists y : \iota.[\![R]\!](x, y)$
$\mathrm{rng}\, R$	$\exists y : \iota.[\![R]\!](y, x)$
$\mathrm{dom}\, P$	$\exists y : T.[\![P]\!](x, y)$ (T is type of P)
Relation	Formula with free variables $x : \iota, y : \iota$
i	$i(x, y)$
$R_1 \cup R_2$	$[\![R_1]\!](x, y) \vee [\![R_2]\!](x, y)$
$R_1 \cap R_2$	$[\![R_1]\!](x, y) \wedge [\![R_2]\!](x, y)$
$R_1; R_2$	$\exists m : \iota.[\![R_1]\!](x, m) \wedge [\![R_2]\!](m, y)$
R^{-1}	$[\![R]\!](y, x)$
R^*	(tricky, omitted)
Δ_C	$x \doteq y \wedge [\![C]\!](x)$
Property of type T	Formula with free variables $x : \iota, y : T$
i	$i(x, y)$

Figure 6.4: Interpretation Function for BOL into SFOL

The consequence closure of SFOL, using the usual semantics of SFOL, induces the desired consequence closure for BOL:

Definition 6.9 (Consequence Closure)**.** We say that a BOL-statement F is a consequence of an ontology O if $[\![F]\!]$ is an SFOL-theorem of $P, [\![O]\!]$.

Example 6.10. We interpret the example ontology from Ex. 2.3. Excluding the semantic prefix, it results in

$\texttt{FlorianRabe} : \iota$, $\texttt{WuV} : \iota$,
$\texttt{person} \subseteq \iota$, $\texttt{male} \subseteq \iota$, $\texttt{instructor} \subseteq \iota$, $\texttt{course} \subseteq \iota$,
$\texttt{teach} \subseteq \iota \times \iota$, $\texttt{creditValue} \subseteq \iota \times \texttt{float}$
$\texttt{instructor}(\texttt{FlorianRabe}) \wedge \texttt{male}(\texttt{FlorianRabe})$, $\texttt{course}(\texttt{WuV})$, $\texttt{teach}(\texttt{FlorianRabe}, \texttt{WuV})$, $\texttt{creditValue}(\texttt{WuV}, 7.5)$
$\forall x : \iota.\texttt{male}(x) \Rightarrow \texttt{person}(x)$,
$\forall x : \iota.\texttt{instructor}(x) \Rightarrow \texttt{person}(x)$,
$\forall x : \iota.(\exists y : \iota.\texttt{teach}(x, y)) \Rightarrow \texttt{instructor}(x)$,
$\forall x : \iota.(\exists y : \iota.\texttt{teach}(y, x)) \Rightarrow \texttt{course}(x)$,
$\forall x : \iota.(\exists y : \iota.\texttt{teach}(y, x)) \Leftrightarrow \texttt{course}(x)$,
$\forall x : \iota.\texttt{course}(x) \Rightarrow \exists y : \iota.\texttt{teach}(y, x) \wedge \texttt{instructor}(y)$

Example 6.11 (Compositionality). The interpretation of BOL is compositional.
For example, consider the case of composition of relations:

$$[\![R_1; R_2]\!] = \exists m : \iota.[\![R_1]\!](x, m) \wedge [\![R_2]\!](m, y)$$

Here we have an expression $E(e_1, \ldots, e_n)$ with $n = 2$ and E is the ;-operator mapping $(e_1, e_2) \mapsto e_1; e_2$, i.e., R_1 and R_2 are the direct subexpressions of $R_1; R_2$. The semantics is a relatively complicated FOL-formula, but it only depends on $[\![R_1]\!]$ and $[\![R_2]\!]$ — everything else is fixed. We have $[\![;]\!] = (p_1, p_2) \mapsto \exists m : \iota.p_1(x, m) \wedge p_2(m, y)$, i.e., the interpretation of the ;-operator is the function that maps two predicates p_1, p_2 to the formula $\exists m : \iota.p_1(x, m) \wedge p_2(m, y)$. Then we have

$$[\![R_1; R_2]\!] = [\![;]\!]([\![R_1]\!], [\![R_2]\!]).$$

Example 6.12 (Non-Compositional Translation via Custom Semantic Prefix). In Fig. 6.4, we omitted the case for the transitive closure. That was because it is not possible to translate it compositionally into FOL. We can only do it non-compositionally with a custom semantic prefix:
We define the FOL-interpretation of an ontology O by $[\![O]\!] = P_O, [\![O]\!]$, where P_O is a custom semantic prefix. P_O is different for every ontology O and is defined as follows:

1. We scan through O and collect all occurrences of R^* for any (not necessarily atomic) relation R.
2. P_O contains the following declarations for each R:
   - A binary predicate symbol $C_R \subseteq i \times i$. Note that R may be a complex expression; so we have to generate a fresh name C_R here.
   - The axiom $\forall x : \iota, y : \iota.\, R(x, y) \Rightarrow C_R(x, y)$, i.e., C_R extends R.
   - The axiom $\forall x : \iota, y : \iota, z : \iota.\, C_R(x, y) \wedge C_R(y, z) \Rightarrow C_R(x, z)$, i.e., C_R is transitive.
3. We add the case $[\![R^*]\!] = C_R(x, y)$ to the interpretation function.

Intuitively, every occurrence of the *-operator is removed from the language and replaced with a fresh name that is axiomatized to have the needed properties. All of these axioms are added to the semantic prefix.

6.4 Concretized Semantics

We give an alternative semantics using a semantic language for concrete data. Specifically we use the database language SQL.

6.4.1 An SQL-Inspired Basic Database Language

We give an SQL-like database language as a formal system.

Definition 6.13. Fig. 6.5 gives the context-free grammar. The vocabulary symbol is S. The expression symbols are T, R, V, and F.

6.4.2 Semantics

Even though this is a very different knowledge aspect, the general principles of the semantics are the same: Every BOL-declaration is translated to an SQL declaration, and ontologies are translated declaration-wise. For every kind of complex expression, there is one inductive function mapping BOL-expressions to SQL-expressions.

In SQL, we can nicely see the difference between declarations and expressions: the former are translated to side effect-ful statements, the latter to side effect-free queries.

Definition 6.14 (Concretized Semantic of BOL)**.** The **semantic prefix** consists of the following SQL-statements

- a type ID of identifiers (if not already supported anyway by the underlying database)
- declarations of all base types and values of BOL (if not already supported anyway by the underlying database)
- TABLE individuals (id: ID, name: string), where the id field is unique and automatically generated when inserting values

Every BOL-ontology O is interpreted as the sequence $P, [\![O]\!]$ of SQL statements, where $[\![O]\!]$ is defined in Fig. 6.6.

Vocabularies: Schemas

S ::= D^*

Declarations

D	::= **TABLE** `ID` TT	table
	| **INSERT** R **INTO** `ID`	row in a table

table types

TT	::= CT^*	list of column types
CT	::= `ID` : Y	column type

table expressions (i.e., table-valued queries)

T	::= `ID`	atomic tables
	| **JOIN** T^*	join of tables
	| **UNION** T^*	union of tables
	| **INTER** T^*	intersection of tables
	| **SELECT**[**DISTINCT**] CL **FROM** T	selection of columns
	| T **WHERE** F	selection of rows
	| T **AS** `ID`	prefix for column names
	| T **AGGREGATE** CA^*	aggregation
CA	::= **COUNT**(`ID`)	count values in a column
	| **MAX**(`ID`)	maximum value in a column
	| ...	minimum, sum, etc.
CL	::= $*$ | (`ID` | `ID` **AS** `ID`)*	list of column names

row expressions (i.e., single row–valued queries)

R	::= **VALUES** (CD^*)	explicit row
	| T	table, but may only contain one row
CD	::= `ID` = V	column definition

cell expressions (i.e., single column, single row–valued queries)

V	::= R	row, but may only contain one column
	| `ID`(V^*)	built-in function symbol applied to values
	| (*base values*)	as for BOL

base types

Y	::= (*base types*)	as for BOL

formulas

F	::= V	boolean value
	| $V = V$	equality of values
	| R **IN** T	containment of rows in tables
	| R **NOT IN** T	opposite of containment
	| ...	boolean operators

Identifiers

`ID` ::= alphanumeric string

Figure 6.5: Syntax of SQL

BOL Syntax X	Semantics $[\![X]\!]$ in SQL
ontology	SQL schema (list of statements)
$D_1, \ldots, D_n$	$[\![D_1]\!], \ldots, [\![D_n]\!]$
BOL declaration (C, R, P atomic)	SQL statement
individual i	INSERT VALUES (name="i") INTO individuals
concept i	TABLE i (id: ID)
relation i	TABLE i (subject: ID, object: ID)
property $i : Y$	TABLE i (subject: ID, value: Y)
I `is-a` C	INSERT VALUES (id=$[\![I]\!]$) INTO C
I_1 R I_2	INSERT VALUES (subject=$[\![I_1]\!]$, object=$[\![I_2]\!]$) INTO R
I P V	INSERT VALUES (subject=$[\![I]\!]$, value=V) INTO P
F	consistency check, consequence closure (omitted)
Formula	Query that returns empty result iff formula is true
$C_1 \equiv C_2$	($[\![C_1]\!] \setminus [\![C_2]\!]$) UNION ($[\![C_2]\!] \setminus [\![C_1]\!]$)
$C_1 \sqsubseteq C_2$	$[\![C_1]\!] \setminus [\![C_2]\!]$
I `is-a` C	$[\![I]\!]$ IN $[\![C]\!]$
I_1 R I_2	($[\![I_1]\!]$, $[\![I_2]\!]$) IN R
I P V	($[\![I]\!]$, V) IN P
Individual	an identifier from the table individuals
i	SELECT id FROM individuals WHERE name="i"
Concept	SQL table expression for ($id : ID$)
i	SELECT * FROM i
$\top$	individuals
$\bot$	individuals WHERE false
$C_1 \sqcup C_2$	$[\![C_1]\!]$ UNION $[\![C_2]\!]$
$C_1 \sqcap C_2$	$[\![C_1]\!]$ INTERSECT $[\![C_2]\!]$
$\forall R.C$	individuals \ (SELECT subject FROM $[\![R]\!]$ WHERE object NOT IN $[\![C]\!]$)
$\exists R.C$	SELECT DISTINCT subject FROM $[\![R]\!]$, $[\![C]\!]$ WHERE object=id
dom R	SELECT DISTINCT subject FROM $[\![R]\!]$
rng R	SELECT DISTINCT object FROM $[\![R]\!]$
dom P	SELECT DISTINCT subject FROM $[\![P]\!]$
Relation	SQL table expression for (subject: ID, object: ID)
i	SELECT * FROM i
$R_1 \cup R_2$	$[\![R_1]\!]$ UNION $[\![R_2]\!]$
$R_1 \cap R_2$	$[\![R_1]\!]$ INTERSECT $[\![R_2]\!]$
$R_1; R_2$	SELECT DISTINCT l.subject, r.object FROM $[\![R_1]\!]$ AS l, $[\![R_2]\!]$ AS r WHERE l.object = r.subject
R^{-1}	SELECT object AS subject, subject AS object FROM $[\![R]\!]$
R^*	(tricky, omitted)
Δ_C	SELECT id AS subject, id AS object FROM $[\![C]\!]$
Property of type Y	SQL table expression for (subject: ID, value: Y)
i	SELECT * FROM i

Using the abbreviations: $S \setminus T$ = SELECT id FROM S WHERE id NOT IN T
S, T = JOIN S, T

Figure 6.6: Interpretation Function for BOL into SQL

Remark 6.15 (Limitations). Our interpretation of BOL in SQL is restricted to assertions using only atomic expressions. For example, in the case for I `is-a` C, we assume that I and C are names. Thus, we have already created an individual for I and a table for C, and we can thus insert the former into the latter. The general case would be more complicated but is much less important in practice. But other expressions very quickly become more difficult.

The interpretation of formulas into SQL is less obvious because SQL is not a logic and therefore does not define a consequence closure. Thus, we can only use axioms for consistency checks in SQL. But that requires first carrying out an explicit consequence closure that adds all implied assertions to the database.

Example 6.16. We interpret the example ontology from Ex. 2.3. Excluding the semantic prefix, the entity declarations and assertions result in the following

```
INSERT VALUES (name="FlorianRabe") INTO individuals
INSERT VALUES (name="WuV") INTO individuals
TABLE person (id: ID)
TABLE male (id: ID)
TABLE instructor (id: ID)
TABLE course (id: ID)
TABLE teach (subject: ID, object: ID)
TABLE creditValue (subject: ID, value: float)
INSERT VALUES (id=2) INTO course
INSERT VALUES (subject=1, object=2) INTO teach
INSERT VALUES (subject=1, value=7.5) INTO creditValue
```

Here we assume that inserting into the table individuals has automatically assigned the ids 1 and 2 to our two individuals.

The concept assertion about `FlorianRabe` using $\sqcap$ cannot be handled by this semantics. Therefore, we skip that assertion. The two missing assertions

```
INSERT VALUES (id=1) INTO instructor
INSERT VALUES (id=1) INTO male
```

must then be provided by performing the consequence closure.

Moreover, the axioms result in the following consistency checks, i.e., queries that should be empty:

```
SELECT * FROM male \ SELECT * FROM person
SELECT * FROM instructor \ SELECT * FROM person
SELECT * FROM (SELECT DISTINCT subject FROM teach) \ SELECT * FROM instructor
SELECT * FROM (SELECT DISTINCT object FROM teach) \ SELECT * FROM course
(SELECT * FROM (SELECT DISTINCT subject FROM creditValue) \ SELECT * FROM course)
   UNION (SELECT * FROM course \ SELECT DISTINCT subject FROM creditValue)
SELECT * FROM course \
  (SELECT DISTINCT subject
   FROM (SELECT object AS subject, subject AS object FROM teach), instructor
   WHERE object=id)
```

Some of these checks will only succeed after performing the consequence closure. In particular, the table `person` misses the entry 1 for the individual `FlorianRabe` because the assertion `FlorianRabe is-a person` is only present as a consequence

6.5 Computational Semantics

We give an alternative semantics using computation, i.e., by using a programming language as the semantic language. Specifically, we use the programming language Scala.

6.5.1 A Scala-Inspired Basic Programming Language

We give a simple programming language as a language system.

Definition 6.17. Fig. 6.7 gives the context-free grammar. The vocabulary symbol is P. The expression symbols are Y, V, and F.

Vocabularies: Programs

P ::= D^*

Declarations

D ::=	**class** `ID`$[X^*]$ **extends** `ID`$^*\{d^*\}$	class definition
|	**object** `ID` **extends** `ID`$^*\{d^*\}$	object definition
d ::=	**val** `ID` : $Y[=T]$	immutable field in a class/object, possibly abstract
|	**var** `ID` : $Y = T$	mutable field in a class, with initial value

Type expressions

Y ::=	`ID`$[X^*]$	atomic type (class) applied to type arguments
|	`ID`	built-in type (booleans, int, etc.)
|	X	type variable
|	Y => Y	function types

Term expressions

T ::=	`ID`	atomic value (class, value, variable)
|	`ID`	built-in value (boolean operators, etc.)
|	T.`ID`	field access in an object
|	T.`ID` $= T$	assignment to a mutable field in an object
|	`ID` $= T$	assignment to a local variable
|	$T : Y$	instance check
|	**new** `ID` $\{d^*\}$	new instance of class
|	$T \doteq_Y T$	equality of terms at a type
|	(`ID` : Y) => T	function
|	$T(T^*)$	function applied to values
|	$\{T^*\}$	sequencing (;-operator)
|	d	local declaration
|	**if** $(T)\,T$ **else** T	if-then-else
|	**while** $(T)\,T$	while-loop

Formula expressions

F ::= T terms of boolean type

Identifiers

`ID` ::= alphanumeric string

Figure 6.7: Syntax of BPL

6.5.2 Semantics

Again, the general principles are the same: Every BOL-declaration is translated to a Scala-declaration, and ontologies are translated declaration-wise to Scala-programs. For every kind of complex expression, there is one inductive function mapping BOL-expressions to Scala-objects.

Definition 6.18 (Computational Semantic of BOL)**.** The **semantic prefix** consists of the following Scala statements

- classes for all BOL-base types and values for them (if not already present in Scala)
- classes for individuals and hash sets of objects:

```
import scala.collection.mutable.HashSet
val individuals = new HashSet[String]
```

Every BOL-ontology O is interpreted as the Scala program $P, [\![O]\!]$, where $[\![O]\!]$ is defined in Fig. 6.8.

Remark 6.19 (Scala Syntax). In Scala, val $x = e$ evaluates e and stores the result in x. $\{d_1; \ldots; d_n\}$ is evaluated by executing all d_i in order and returning the result of d_n.
(A, B) is the product type $A \times B$ with pairing operator (x, y) and projection functions _1 and _2. $x \Rightarrow F(x)$ is $\lambda x.F(x)$.
The class HashSet is part of the standard library and offers function += and -= to add/remove elements, contains to test elementhood, and forall, foreach to quantifiy/iterate over elements.
Types of variables are inferred if omitted.

Remark 6.20 (Limitations). Our interpretation of BOL in Scala has similar problems as the one in SQL. We restrict entities in assertions to be atomic. And we assume that all assertions implied by the consequence closure have already been obtained and added to the ontology.

Example 6.21. We interpret the example ontology from Ex. 2.3. Excluding the semantic prefix, the entity declarations and assertions result in the following

```
individuals += "FlorianRabe"
individuals += "WuV"
val person = new HashSet[String]
val male = new HashSet[String]
val person = new HashSet[String]
val course = new HashSet[String]
val teach = new new HashSet[(String,String)]
val creditValue = new HashSet[(String,float)]
course += WuV
teach += ("FlorianRabe", WuV)
creditValue += (WuV, 7.5)
```

Like for SQL, the two statements

```
instructor += "FlorianRabe"
male += "FlorianRabe"
```

must be obtained by consequence closure because we cannot handle the $\sqcap$ assertion. Note that we could easily compute the hash set `instructor.diff(male)` and add to it. But that would not add anything to the two constituent sets.
If we thing of the axioms as consistency checks, we can translate them to assertions, i.e., Boolean expressions that must be true. We only give some examples:

```
{val c1 = male; val c2 = person; c1.forall(x ⇒ c2.contains(x))}
```

```
{
  val c1 = course;
  val c2 = {
    val c = instructor;
    val r = {
      val r = new HashSet[(String,String)];
      teach.foreach(x ⇒ r += (x._2,x._1));
```

BOL Syntax X	Semantics $[\![X]\!]$ in Scala
ontology	Scala program
$D_1, \ldots, D_n$	**object** $Program$ {P, $[\![D_1]\!], \ldots, [\![D_n]\!]$}
BOL declaration (C, R, P atomic)	Scala declaration
individual i	val i = "i"; individuals += i
concept i	val i = new HashSet[String]
relation i	val i = new HashSet[(String,String)]
property $i : T$	val i = new HashSet[(String,T)]
I `is-a` C	$[\![C]\!]$ += $[\![I]\!]$
$I_1\ R\ I_2$	$[\![R]\!]$ += $([\![I_1]\!], [\![I_2]\!])$
$I\ P\ V$	$[\![P]\!]$ += $([\![I]\!], [\![V]\!])$
F	assertions, consequence closure (omitted)
Formula	Program that evaluates the formula to a Boolean
$C_1 \equiv C_2$	{val $c1$ = $[\![C_1]\!]$; val $c2$ = $[\![C_2]\!]$; c_1.forall(x ⇒ c_2.contains(x)) && c_2.forall(x ⇒ c_1.contains(x))}
$C_1 \sqsubseteq C_2$	{val $c1$ = $[\![C_1]\!]$; val $c2$ = $[\![C_2]\!]$; c_1.forall(x ⇒ c_2.contains(x))}
I `is-a` C	$[\![C]\!]$.contains($[\![I]\!]$)
$I_1\ R\ I_2$	$[\![R]\!]$.contains(($[\![I_1]\!], [\![I_2]\!]$))
$I\ P\ V$	$[\![P]\!]$.contains(($[\![I]\!], [\![V]\!]$))
Individual	String object
i	i
Concept	HashSet[String] object
i	i
$\top$	individuals
$\bot$	HashSet[String].empty()
$C_1 \sqcup C_2$	$[\![C_1]\!]$.union($[\![C_2]\!]$)
$C_1 \sqcap C_2$	$[\![C_1]\!]$.inter($[\![C_2]\!]$)
$\forall R.C$	{val c = $[\![C]\!]$; val r = $[\![R]\!]$; val e = individuals.clone; r.foreach(x ⇒ if (!c.contains(x._2)) e -= x._1); e}
$\exists R.C$	{val c = $[\![C]\!]$; val r = $[\![R]\!]$; val e = new HashSet[String]; r.foreach(x⇒ if (c.contains(x._2)) e += x._1); e}
dom R	{val c = new HashSet[String]; $[\![R]\!]$.foreach(x ⇒ c += x._1); c}
rng R	{val c = new HashSet[String]; $[\![R]\!]$.foreach(x ⇒ c += x._2); c}
dom P	{val c = new HashSet[String]; $[\![P]\!]$.foreach(x ⇒ c += x._1); c}
Relation	HashSet[(String,String)] object
i	i
$R_1 \cup R_2$	$[\![R_1]\!]$.union($[\![R_2]\!]$)
$R_1 \cap R_2$	$[\![R_1]\!]$.inter($[\![R_2]\!]$)
$R_1; R_2$	{val r1 = $[\![R_1]\!]$; val r2 = $[\![R_2]\!]$; val e = new HashSet[(String,String)]; r1.foreach(x ⇒ r2.foreach(y ⇒ if (x._2 == y._1) e += (x._1,y._2))); e}
R^{-1}	{val r = new HashSet[(String,String)]; $[\![R]\!]$.foreach(x ⇒ r += (x._2,x._1)); r}
R^*	{val r = $[\![R]\!]$; var e = $[\![\Delta_\top]\!]$; **while** (!equal(comp(e,r), e)) {e = comp(e,r)}; e}
Δ_C	{val r = new HashSet[(String,String)]; $[\![C]\!]$.foreach(x ⇒ r += (x,x)); r}
Property of type T	HashSet[(String,T)] object
i	i

Figure 6.8: Interpretation Function for BOL into Scala (assuming functions equal and comp on relations)

```
      r
    }
    val e = new HashSet[String];
    r.foreach(x ⇒ if (c.contains(x._2)) e += x._1);
    e
  };
  c1.forall(x ⇒ c2.contains(x))
}
```

6.6 Narrative Semantics

We give an alternative semantics using narration, i.e., by using a natural language as the semantic language. Specifically, we use the natural language English.

Again, the general principles are the same: Every BOL-declaration is translated to an English sentence, and ontologies are translated declaration-wise to English texts. For every kind of complex expression, there is one inductive function mapping BOL-expressions to English phrases.

Definition 6.22 (Narrative Semantic of BOL)**.** The **semantic prefix** consists of English statements explaining

- the base types of BOL (if they are not universally known),
- that we rely on a lexicon to correctly form plurals (indicated by -s) and verb forms (indicated by -s, -ing, -ed).

Every BOL-ontology O is interpreted as the English text $P, [\![O]\!]$, where $[\![O]\!]$ is defined in Fig. 6.9.

Natural language defines a consequence closure by appealing to consequence in natural language. That is well-defined as long as we express ourselves precisely enough.

Definition 6.23 (Consequence Closure)**.** We say that a BOL-statement F is a consequence of an ontology O if $[\![F]\!]$ is a consequence of $P, [\![O]\!]$.

Example 6.24. We interpret the example ontology from Ex. 2.3. Excluding the semantic prefix and the lexicon lookup, it results in the following text:

FlorianRabe is a proper noun.
WuV is a proper noun.
person is a common noun.
male is a common noun.
instructor is a common noun.
course is a common noun.
teach is a transitive verb.
creditValue is a common noun for a property that can take `float`-values.
FlorianRabe is a instructor and is a male.
WuV is a course.
FlorianRabe teachs WuV.
WuV has creditValue 7.5.
everything that is a male also is a person.
everything that is a instructor also is a person.
everything that teachs is a instructor.
everything that is teached by something is a instructor.
has some creditValue is the same as is a course.
everything that is a course also is teached by something that is a instructor.

This English is very clunky of course. Multiple tweaks would be needed to get the grammar right:

- It is "teaches" and "taught" instead of "teachs" and "teached",
- It is "an instructor" instead of "a instructor",

BOL Syntax X	Semantics $[\![X]\!]$ in English
ontology	English text
$D_1, \ldots, D_n$	$[\![D_1]\!], \ldots, [\![D_n]\!]$
BOL declaration	dictionary entry or true sentence
individual i	i is a proper noun.
concept i	i is a common noun.
relation i	i is a transitive verb.
property $i : T$	i is a common noun for a property that can take T-values.
I `is-a` C	$[\![I]\!]$ $[\![C]\!]$.
$I_1\ R\ I_2$	$[\![I_1]\!]$ $[\![R]\!]$ $[\![I_2]\!]$.
$I\ P\ V$	$[\![I]\!]$ has $[\![P]\!]$ $[\![V]\!]$.
F	$[\![F]\!]$.
Formula	sentence
$C_1 \equiv C_2$	$[\![C_1]\!]$ is the same as $[\![C_2]\!]$.
$C_1 \sqsubseteq C_2$	everything that $[\![C_1]\!]$s also $[\![C_2]\!]$s.
I `is-a` C	$[\![I]\!]$ $[\![C]\!]$.
$I_1\ R\ I_2$	$[\![I_1]\!]$ $[\![R]\!]$s $[\![I_2]\!]$.
$I\ P\ V$	$[\![I]\!]$ has $[\![P]\!]$ $[\![V]\!]$.
Individual	noun phrase (to be used as a subject or object)
i	i
Concept	intransitive verb phrase (to be plugged after a subject)
i	is a i
$\top$	is anyone
$\bot$	is no one
$C_1 \sqcup C_2$	$[\![C_1]\!]$ or $[\![C_2]\!]$
$C_1 \sqcap C_2$	$[\![C_1]\!]$ and $[\![C_2]\!]$
$\forall R.C$	$[\![R]\!]$s only things that $[\![C]\!]$
$\exists R.C$	$[\![R]\!]$s something that $[\![C]\!]$
$\mathrm{dom}\, R$	$[\![R]\!]$s something
$\mathrm{rng}\, R$	is $[\![R]\!]$ed by something
$\mathrm{dom}\, P$	has some $[\![P]\!]$
Relation	transitive verb phrase (to be plugged between subject and object)
i	i
$R_1 \cup R_2$	$[\![R_1]\!]$s or $[\![R_2]\!]$s
$R_1 \cap R_2$	$[\![R_1]\!]$s and $[\![R_2]\!]$s
$R_1; R_2$	$[\![R_1]\!]$s something that $[\![R_2]\!]$s
R^{-1}	is $[\![R]\!]$ed by
R^*	$[\![R]\!]$s something that $[\![R]\!]$s something and so on that $[\![R]\!]$s
Δ_C	$[\![C]\!]$s and is the same as
Property of type T	property phrase
i	i

Figure 6.9: Interpretation Function for BOL into English (intransitive VP version)

- Sentences start with upper case letters.
- Proper nouns often have different names in the ontology than in reality, e.g., it should be "Florian Rabe" and "credit value" instead of "FlorianRabe" and "creditValue".

Moreover, the language could be polished in many places. For example, "is a instructor and is a male" could become "is a instructor and a male" with a relatively easy special case treatment, or it could become "is a male instructor" with a more complex semantics that interprets some concepts via nouns and some via adjectives.

Remark 6.25 (Variants of English). We are relatively open as to what kind of English we want to use as the semantic language. The simplest choice would be to use plain English as you could find in a novel or newspaper article. But for many applications (e.g., formal ontologies in the STEM fields), we would rather use STEM English, i.e., English interspersed with formulas, diagrams, and epistemic cues like "Definition", "Theorem", "Proof", and even □. For this kind of English, LaTeX is a good target format. We can even use special LaTeX dialects like sTeX [Koh08] where we can capture more of the semantic properties.

Remark 6.26 (Better Language Generation). While the target languages in the other translations are formal languages engineered for regularity and simplicity (in terms of language primitives), natural languages have evolved in practical human communication. As a consequence, the translation in Def. 6.22 results in English that is clumsy at best and non-grammatical in general. We can think of the result as **BOL-pidgin** English.

Let us have a look at some of the problems that appear in both translations:

- We need a lexicon to obtain inflection information and the translation tries to remedy that by appending "s" in various places. This works in some cases but not in others.
- there are many linguistic devices that serve an important role in natural language, but which we are not targeting. An example is plural objects for aggregation. Say we have $P\texttt{is}-\texttt{a}C$, $M\texttt{is}-\texttt{a}C$, this would translate to "P is a $[\![C]\!]$, M is a $[\![C]\!]$" in BOL-pidgin, whereas in natural English we would aggregate this to "P and M are $[\![C]\!]$s".

A way out is to utilize special systems for dealing with the surface structure of natural language. An example of this is the Grammatical Framework (GF, [Ran11]): it allows specifying a rich formal language of **abstract syntax trees** for natural language (ASTs) together with language-specific **linearizations**, which amount to recursive functions that translate ASTs to language-specific strings. GF comes with a large resource library that provides a comprehensive, language-independent AST specification and linearizations for over 35 languages. We will not pursue this here, but there is a special course "Logic-based Natural Language Semantics" at FAU in the Winter Semesters that covers these and related topics. One of the major issues that need to be addressed there and here is the notion of compositionality, which is central to all processing and semantics. We will address it next, and come back to it time and again later.

Chapter 7

Querying via a Semantics

7.1 Overview

Let us assume we have a semantics for our syntax. We again write l for the syntax, L for the semantic, and $[\![-]\!]$ for the translation function.

We can now use the semantics to answer questions asked in the syntax. Here we use the syntax to phrase a question and the semantics to determine the answer.

We call this *querying*. Contrary to standard practice, we will use that word in a very broad sense that covers all aspects. It is more common to use the word only for concretized querying, where SQL has been developed, which has shaped many intuitions about querying.

Usually, querying requires the syntax to designate some non-terminals as *propositional*. A non-terminal is propositional if the semantics can make its words true. Without a notion of propositions, it is impossible to define what questions and answers even are.

Definition 7.1 (Propositions)**.** A context-free **syntax with propositions** is a context-free syntax with some designated non-terminal symbols.

A **semantics with theorems** is one that additionally defines some propositions to be theorems. We write $\vdash F$ if F is a theorem.

That definition does not mean that any kind of logic is needed for querying. Many languages use highly restricted notions of propositions that would not generally be considered as logic. For example, languages might use equalities between objects or even equalities between certain objects as the only propositions. The following table gives an overview:

aspect	typical propositions	proposition operators
ontology language	assertions, concept equality/subsumption	
programming language	equality for some types	boolean operators
database language	equality for base types	boolean operators
logic	equality for all types	boolean operators, quantifiers
natural language	sentences	and, or, some, every, ...

Often the development of querying for a language leads to the discovery of omissions in the syntax: certain objects that are helpful to ask questions were omitted from the syntax because they were not needed to describe the data. Then sometimes the syntax is extended with non-terminals or productions that seem like dead code: they are not needed for or not allowed in the official data. The following table gives some examples:

aspect	typical extensions
ontology language	conjunction of assertions
programming language	quantifiers
database language	membership in a table
logic	(already tries to capture all possible propositions)
natural language	(already captures all possible propositions)

Example 7.2 (Propositions in BOL). The obvious choice of propositions for BOL are the formulas.

In Rem. 2.1, we mentioned that the BOL syntax from Fig. 2.2 had some redundant parts that were grayed out. Assertions are needed for writing ontologies only in such that they behave like axioms, i.e., they are automatically true. But for querying BOL, we also need them to behave like formulas so that we can use them as questions, i.e., we must allow them to be true or false.

Moreover, it is common to also allow conjunctions. Therefore, the BOL propositions are the conjunctions of formulas.

In the sequel, we will use each of the four kinds of semantics to

Section	builds on Section	aspect	query	result
7.2	6.3	deduction	proposition	yes/no
7.3	6.4	concretization	proposition with free variables	true ground instances
7.4	6.5	computation	term	value
7.5	6.6	narration	question	answer

Remark 7.3 (Meta-Level Questions). Finally, any semantics admits a meta-level where additional questions can be asked. Examples are asking for the consistency of a theory or the equivalence of two theories/programs/queries. At the next-higher meta-level, we can ask about the completeness of a semantics or the equivalence of two semantics (of which completeness is a special case). These meta-questions can usually not be expressed in the syntax, and we do not consider them a part of querying here. But it is worth mentioning that the need to use yet another language (a meta-language) to ask these questions can be annoying, and some advancements in language design are about trying to integrate them into the syntax. For example, reflection is the process of representing a language in itself so that the language can talk about itself. That way meta-questions become regular questions.

7.2 Deductive Querying

7.2.1 Method

We assume that l is a syntax with propositions and that L is a logic (and thus in particular has propositions) whose semantics has theorems.

It is not guaranteed $[\![-]\!]$ translates l-propositions to L-propositions. If not, we assume there is some operation `True` in L that we can use to lift the translations of l-propositions to L-propositions.

A **deductive query** consists of a proposition. The answer to the query is yes or no.

Thus, the deductive semantics must determine which propositions are theorems, i.e., whether $\vdash_L \texttt{True}([\![F]\!])$. This is usually done in one of two ways.

Proof theory uses a calculus for L to derive true propositions. Thus, we can say that an l-proposition F is true if the calculus derives $\texttt{True}([\![F]\!])$. Accordingly, if L has a negation operator $\neg$, we can say that F is false if the calculus derives $\texttt{True}(\neg[\![F]\!])$.

Model theory uses a second deductive semantics, namely a translation from L to an even stronger deductive language M, usually some form of set theory. Then, we can say that l-propositions are true if the composition of the two translations yields a true M-proposition.

Either way, it is determined whether to answer an l-proposition with yes or no.

7.2.2 Challenges

Consistency The L-calculus might derive both F and $\neg F$. In that case L is inconsistent and usually every formula. We usually assume L to be consistent even though we do not always prove that.

Decidability Deductive semantics is usually undecidable, i.e., there is no algorithm that takes in F and always returns yes or no in finite time.

Therefore, deductive querying is very difficult in general. One has to run heuristics (theorem provers) to see if a proof of F or $\neg F$ can be found.

A common compromise is to allow only a restricted set of propositions as queries for which decision procedures exist. However, it can be tricky to find good restrictions, especially if the syntax allows for function symbols and equality.

For example, SFOL is undecidable. But many fragments of SFOL are decidable, such as propositional logic and various fragments in between.

When giving a deductive semantics into SFOL, it is therefore important to check whether the image of $[\![-]\!]$ falls inside a decidable fragment. This is typically the case for ontology languages.

Completeness Deductive semantics is usually incomplete, i.e., there are unanswered questions. More precisely, the L-calculus typically derives F for some propositions, $\neg F$ for some, but neither for some others. The third kind of proposition cannot be answered by the semantics.

> *Remark* 7.4. The work "complete" is used for two different things in logic.
>
> Firstly, it can be a relation between two semantics, typically proof theory and a model theory. That is the dominant meaning of the word as in, e.g., the completeness theorem for SFOL and Gödel's incompleteness theorem.
>
> Secondly, it can mean that a logic proves or disproves every proposition, i.e., there is no F such that neither F nor $\neg F$ are derivable. That is the sense we use above. This kind of completeness rarely holds, usually only in very restricted circumstances.

Decidability and completeness are essentially the same problem. Specifically, if completeness holds, we already obtain a decision procedure for the logic: to decide the truth of F, enumerate all proofs until a proof of F or $\neg F$ is found. Vice versa, if we have a sound decision procedure, running it on F will prove either F or $\neg F$.

Efficiency Independent of whether the semantics is complete/decidable, theorem proving is typically very expensive.

Therefore, in addition to identifying decidable fragments of a logic, it is desirable to identify *efficiently* decidable fragments. Typically, a semantics meant for efficient practical querying aims for polynomially decidable fragments. This is the case for very simple ontology languages. But it can quickly become exponential if the language of propositions becomes more expressive.

7.3 Concretized Querying

This was discussed on the slides.

7.4 Computational Querying

This was discussed on the slides.

7.5 Narrative Querying

This was discussed on the slides.

Chapter 8

Formal Systems

8.1 Formal Systems

8.1.1 Syntax

Definition 8.1. A **formal system** consists of

- a set Voc of vocabularies,
- for any two vocabularies V, W, a set $\mathrm{Voc}(V, W)$ of vocabulary morphisms from V to W
- for any vocabulary V, a set Exp_V of expressions
- for any vocabulary morphism $m \in \mathrm{Voc}(V, W)$, a mapping $\mathrm{Exp}_m : \mathrm{Exp}_V \to \mathrm{Exp}_W$

A formal system with typing

- additionally provides a relation $\vdash_V E : E'$ between expression $E, E' \in \mathrm{Exp}_V$.
- such that for $m \in \mathrm{Voc}(V, W)$, we have that if $\vdash_V E : E'$, then $\mathrm{Exp}_m(E) : \mathrm{Exp}_m(E')$.

The vocabularies are usually lists of named declarations. In that case, the vocabulary morphism from V to W are usually lists of assignments $c := E$ where c is the name of a declaration in V and E is a W-expression. In that case, the mapping Exp_m usually arises by replacing every V-identifier with the W-expression provided by m.

8.1.2 Translation

Definition 8.2. A **translation** from formal system l to formal system L consists of several mappings, all written $[\![-]\!]$:

- a vocabulary translation $\mathrm{Voc}_l \to \mathrm{Voc}_T$,
- for any two l-vocabularies V, W, a morphism translation $\mathrm{Voc}(V, W) \to \mathrm{Voc}([\![V]\!], [\![W]\!])$
- for any l-vocabulary V, an expression translation $\mathrm{Exp}_V \to \mathrm{Exp}_{[\![V]\!]}$
- such that for any vocabulary morphism $m \in \mathrm{Voc}(V, W)$ and expression $E \in \mathrm{Exp}_V$, we have $[\![\mathrm{Exp}^l_m(E)]\!] = \mathrm{Exp}^L_{[\![m]\!]}([\![E]\!])$

If l and L have typing, the translation **preserves typing** if $\vdash^l_V E : E'$ implies $\vdash^L_{[\![V]\!]} [\![(]\!]E) : [\![(]\!]E')$.

The vocabulary translation usually consists of a semantics prefix and a declaration-wise translation: A V-vocabulary $D_1, \dots D_n$ is translated to $P, D'_1, \dots, D'_n$ where the D'_i are the translations of the the D_i.

8.1.3 Interpretation

Definition 8.3. An **interpretation** of a formal system l consists of the following parts:

- for every l-vocabulary V a set $Sit(V)$, whose elements are called situations,
- for every l-vocabulary V, every situation $S \in Sit(V)$, a function mapping every $E \in \mathrm{Exp}_V$ to its interpretation $[\![E]\!]_S$.

8.2 Semantics

Both translations and interpretations can be used to assign semantics to a formal system. Both are relative either to another formal system or to a situation.
But we can also distinguish semantics by the kinds of questions they answer.

8.2.1 Deductive

8.2.2 Computational

8.2.3 Concrete Data

8.3 Kinds of Problems

8.3.1 Problems as Intensional Descriptions

Remark 8.4 (Intensional vs. Extensional). Consider a set S of objects. An intensional description of S
An extensional description of S

Remark 8.5 (Single vs. Multi-Variable Problem). We can think of

Definition 8.6. A **problem** is a theory.
A **solution** is a model of the theory.

Consistency (or satisfiability)	Does P have a solution?
Decision	Is S a solution of P?
Solving	Find a solution of P.
Enumeration	List all solutions of P.

Example 8.7. A constraint satisfaction problem

Example 8.8. A search problem

Example 8.9. The satisfiability problem

8.3.2 Families of Problems

Bibliography

[CFKR21] J. Carette, W. Farmer, M. Kohlhase, and F. Rabe. Big Math and the One-Brain Barrier: The Tetrapod Model of Mathematical Knowledge. *The Mathematical Intelligencer*, 43(1):78–87, 2021.

[Koh08] M. Kohlhase. Using LaTeX as a Semantic Markup Format. *Mathematics in Computer Science*, 2(2):279–304, 2008.

[Ran11] A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, 2011.