Lectures Notes on Knowledge Representation and Processing

Florian Rabe (for a course given with Michael Kohlhase)

2020

These notes were originally prepared for my CS course at University Erlangen-Nuremberg (FAU) given with Michael Kohlhase 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

Part I

Introduction

Chapter 1

Meta-Remarks

Important stuff that you should read carefully!

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

- I am developing these notes in parallel with the lecture — they can grow or change throughout the semester.
- These notes are neither a subset nor a superset of the material discussed in the lecture. 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.

Chapter 2

Fundamental Concepts

2.1 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

2.2 Motivation

2.2.1 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.

2.2.2 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.

2.3 Components of Knowledge

2.3.1 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 primitive 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 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.

2.3.2 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.

2.3.3 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 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 very massively how the semantics is specified and implemented. This has cause 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 measure 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 mis-communicate 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.

2.4 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 [CFKR20]. 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.

2.4.1 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 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.

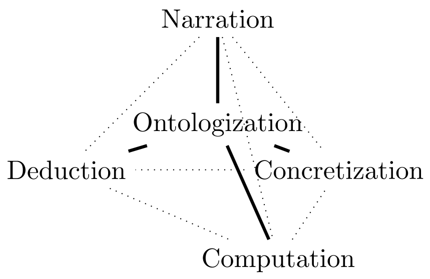
Computation uses specification and programming languages to represent algorithmic knowledge.

Deduction uses logics and theorem provers to obtain verifiable correctness.

Narration uses natural language to obtain texts 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 nLab).

2.4.2 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 **??**.



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

Figure 2.1: Shared properties and advantages of aspects

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 2.1. For example, deductive and narrative definitions of a function involved well-definedness arguments, and a function defined by a concrete table 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 3

Overview of This Course

3.1 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.

3.2 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 4

Representing Syntax and Semantics

4.1 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:

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

4.1.1 Context-Free Grammars

We start with the usual definition:

Definition 4.1 (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 4.2. Let $\Sigma = \{0, 1, +, \cdot, \doteq, \leq\}$. We give a grammar for arithmetic expressions and formulas about them:

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

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

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

Definition 4.3 (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 4.4. The grammar from above with names written to the right of each production

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

This is not common BNF anymore.

Then we add base types to the productions:

Definition 4.5 (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 4.6. Let Nat be the type of natural numbers and let $T = \{Nat\}$. Then we can improve the grammar from above as follows:

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

4.1.2 Inductive Data Types

We start with the usual definition:

Definition 4.7 (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, called the **type**,
- a set of **constructors** each consisting of
  - a name
  - a list of elements of T, called the **argument** types

Example 4.8. Let Nat be the type of natural numbers and $T = \{Nat\}$. We give an inductive type for arithmetic expressions:

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

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 4.9 (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 I, called the **return type**
  - a list of elements of $N \cup I$, called the **argument** types

Example 4.10. We extend the type definition from above by adding a second type for formulas. Thus, $N = \{E, F\}$.

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

Then we add notations to the constructors:

Definition 4.11 (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 such as

Example 4.12. 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.

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

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:

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

4.1.3 Merged Definition

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

Theorem 4.13. *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 4.14 (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 4.15. Using ad hoc language to write the constructors, our example from above as a context-free syntax could look as follows:

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

This uses an ad hoc

4.2 Implementation

Context-free syntax can be implemented systematically in all programming languages. But, depending on the style of the language, they make drastically different. We give the two most important paradigms as examples.

4.2.1 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 define, one for each constructor, that implement the notations.

Example 4.16. We define our example syntax in ML.

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

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

Now the projection functions:

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

and so on for each constructor argument.

Finally, the string rendering functions (assuming a function *natToString* already exists in the context):

$$\begin{array}{l}
\textbf{fun } \texttt{E\_toString}(\texttt{literal}(v)) = natToString\ v \\
\mid \quad \texttt{E\_toString}(\texttt{sum}(x, y)) = \texttt{E\_toString}(x) + \text{"} + \text{"} + \texttt{E\_toString}(y) \\
\mid \quad \texttt{E\_toString}(\texttt{product}(x, y)) = \texttt{E\_toString}(x) + \text{"} \cdot \text{"} + \texttt{E\_toString}(y) \\
\textbf{and } \texttt{F\_toString}(\texttt{equality}(x, y)) = \texttt{E\_toString}(x) + \text{"} \doteq \text{"} + \texttt{E\_toString}(y) \\
\mid \quad \texttt{F\_toString}(\texttt{lessOrEqual}(x, y)) = \texttt{E\_toString}(x) + \text{"} \leq \text{"} + \texttt{E\_toString}(y)
\end{array}$$

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.

4.2.2 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 types 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 4.17. 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 + "\cdot" + 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 + "\doteq" + 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 + "\leq" + 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.

4.2.3 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 if 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
}
```

4.3 Semantics as a Recursive Function

Chapter 5

Encoding Data

5.1 Data Representation Languages

5.2 Typed Data

5.3 Encoding Typed Data in Untyped Representation Languages

Part II

Ontological Knowledge

Chapter 6

Ontologies

6.1 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 the 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., "teaches"
- **properties** — binary relations between an individuals and a concrete value (a number, a date, etc.), e.g., "has-credits"
- **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"	"teaches"	"WuV"
property assertion	"WuV"	"has credits"	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

6.2 A Basic Ontology Language

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

Individual expressions

I	::=	`ID`	atomic individuals

Concept expressions

C	::=	`ID`	atomic concepts
	|	$C \sqcup C$	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

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
T	::=	(omitted)	values

Figure 6.1: Grammar 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.

6.2.1 Syntax

Definition 6.1 (Syntax of BOL)**.** A BOL-ontology is given by the grammar in Fig. 6.1. 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.

6.2.2 Deductive Semantics

We give a semantics of BOL as an example of a semantics by translation. We fix one language that we have already understood and define an interpretation function that maps all complex expression of the syntax into the semantic language. Specifically, we give a deductive/logical semantics, i.e., the semantic language is a logic.

For simple ontology languages like BOL, ALC, OWL, etc., it is common to use first-order logic (FOL) as the semantic language. More specifically, we use SFOL, the typed variant of FOL with

Definition 6.2 (Logical Semantics of BOL)**.** The **semantic prefix** P is the FOL-theory containing

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

Then every BOL-ontology O is interpreted as the FOL-theory $P, [\![O]\!]$, where $[\![O]\!]$ is defined in Fig. 6.2.

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

The role of the semantic prefix P is to define once and for all the FOL material that we need in general to interpret ontologies. It occurs at the beginning of all interpretations of ontologies. In particular, it is equal to the interpretation of empty ontology.

6.2.3 Concretized Semantics

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

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.

BOL Syntax X	Semantics $[\![X]\!]$ in FOL
ontology	FOL 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
$C_1 \equiv C_2$	$\forall x : \iota.[\![C_1]\!](x) \Leftrightarrow [\![C_2]\!](x)$
$C_1 \sqsubseteq C_2$	$\forall x : \iota.[\![C_1]\!](x) \Rightarrow [\![C_2]\!](x)$
Individual	Terms of type ι
i	i
Concept	Formula with free variable $x : \iota$
i	$i(x)$
$C_1 \sqcup C_2$	$[\![C_1]\!](x) \vee [\![C_2]\!](x)$
$C_1 \sqcap C_2$	$[\![C_1]\!](x) \wedge [\![C_2]\!](x)$
$\forall R.C$	$\forall y : \iota.[\![R]\!](x, y) \Rightarrow [\![C]\!](y)$
$\exists R.C$	$\exists y : \iota.[\![R]\!](x, y) \wedge [\![C]\!](y)$
$\mathrm{dom}\, R$	$\exists y : \iota.[\![R]\!](x, y)$
$\mathrm{rng}\, R$	$\exists y : \iota.[\![R]\!](y, x)$
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 6.2: Interpretation Function for BOL into FOL

BOL Syntax X	Semantics $[\![X]\!]$ in SQL
ontology	SQL statements
$D_1, \ldots, D_n$	$[\![D_1]\!], \ldots, [\![D_n]\!]$
BOL declaration (I, C, R atomic)	SQL statement
individual i	INSERT INTO individuals (name) VALUES (i)
concept i	CREATE TABLE i (id ID)
relation i	CREATE TABLE i (subject ID, object ID)
property $i : T$	CREATE TABLE i (subject ID, object T)
I `is-a` C	INSERT INTO C VALUES ($[\![I]\!]$)
$I_1\ R\ I_2$	INSERT INTO R (subject, object) VALUES ($[\![I_1]\!]$, $[\![I_2]\!]$)
$I\ P\ V$	INSERT INTO P (subject, object) VALUES ($[\![I]\!]$, V)
F	consistency check, consequence closure (omitted)
Formula	Formula without free variables
$C_1 \equiv C_2$	check equality of query results
$C_1 \sqsubseteq C_2$	check subset of query results
Individual	an identifier from the table individuals
i	SELECT id FROM individuals WHERE name=i
Concept	SQL query for one-column table
i	SELECT * FROM i
$C_1 \sqcup C_2$	$[\![C_1]\!]$ UNION $[\![C_2]\!]$
$C_1 \sqcap C_2$	$[\![C_1]\!]$ INTERSECT $[\![C_2]\!]$
$\forall R.C$	$[\![R]\!]$???[2] $[\![C]\!]$
$\exists R.C$	SELECT DISTINCT subject FROM $[\![R]\!]$, $[\![C]\!]$ WHERE object= id
dom R	SELECT DISTINCT subject FROM $[\![R]\!]$
rng R	SELECT DISTINCT object FROM $[\![R]\!]$
Relation	SQL query for two-column table
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, subject FROM $[\![R]\!]$
R^*	(tricky, omitted)
Δ_C	SELECT id AS subject, id AS object FROM $[\![C]\!]$
Property of type T	SQL query for two-column table
i	SELECT * FROM i

Figure 6.3: Interpretation Function for BOL into SQL

Definition 6.3 (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)
- CREATE 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 a sequence $P, [\![O]\!]$ of SQL statements, where $[\![O]\!]$ is defined in Fig. 6.3.

Translation of Formulas The interpretation of formulas into SQL is less obvious because SQL is not a logic. We have to consider two subtleties.

Firstly, a formula may express a consistency condition 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. In FOL, this amounts to $[\![O]\!]$ being an inconsistent theory, which can

[2]This case is a mini homework.

be reasoned about as usual. In SQL, we can mimic it by checking the consistency of the database. (Consistency is undecidable for FOL in general but often decidable for ontology languages.)

Secondly, a formula may express a closure operation that must be mimicked by adding implied assertions to the ontology. For example, if there is a subconcept axiom "instructor" $\sqsubseteq$ "person" and a concept assertion "Florian Rabe" is-a "instructor", we have to add the implied concept assertion "Florian Rabe" is-a "person". In FOL, this happens automatically by the calculus of FOL, which adds all theorems of a theory. In SQL, we have to mimic it manually. (Consequence is undecidable for FOL in general but often decidable for ontology languages.)

6.2.4 Compositionality

Definition An interpretation function is compositional if the interpretation of any kind of expression $E(e_1, \ldots, e_n)$ with direct subexpressions e_i only depends on the interpretation of the e_i, i.e.,

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

for some semantic operation $[\![E]\!]$.

The interpretations of BOL in FOL and SQL are compositional. For example, consider the case of composition of relations:

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

We have $n = 2$ 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

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

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 6.4 (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 usually 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 he 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.

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 $[\![X]\!]$.

In Fig. 6.2, 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:

Example 6.5 (Non-Compositional Translation via 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 $\forall x : \iota, y : \iota, z : \iota. C_R(x, y) \wedge C_R(y, z) \Rightarrow C_R(x, x)$, 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.

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, in BOL some complex expressions occur already *statically* in the definition of an ontology O. But others might be appear *dynamically* later, e.g., when talking about O, proving properties of O, or running queries on O. Thus, the definition of O and the use of complex expressions are decoupled: O is defined statically once and for all, and complex expressions relative to O can be created and used dynamically. But if a custom semantic prefix is used, only the static occurrences inside O can be considered for building the prefix. Thus, it is not possible to translate the dynamic occurrences of the transitive closure unless the semantic prefix is extended all the time as O is used.

6.3 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:

```
"Florian Rabe" is-a "Instructor"
"Florian Rabe" is-a "male"
"Florian Rabe" "teaches" "WuV"
"Florian Rabe" "teaches" "KRMT"
"Florian Rabe" "age" 40
"Florian Rabe" "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"
  "teaches" "WuV"
  "teaches" "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"
  "teaches" "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.

6.4 Type Systems for Ontologies

6.4.1 Concepts as Types

6.4.2 Record Types

Chapter 7

Writing Ontologies

7.1 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[1] 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.

7.2 The Protege Tool

A widely used tool for writing ontologies in OWL is Protege[2].

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.

[1] https://www.w3.org/TR/2012/REC-owl2-primer-20121211/

[2] https://protege.stanford.edu/

7.3 Exercise

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

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.

Part III

Concretized Knowledge

Part IV

Computational Knowledge

Part V

Deductive Knowledge

Various methods have been developed to represent and perform inferences. We structure our presentation by how each method relates to computation, the aspect most whose integration with inference has drawn the most attention. In general, the ubiquity of underspecified function symbols and quantified variables means that logical expressions usually do not normalize to unique values. At best, computations like $y := f(x)$ can be represented as open-ended conjectures where different options for y are produced, each together with a proof of the respective equality. Therefore, inference systems usually sacrifice computation or at least its efficiency.

Proof assistants sit at the extreme end of this spectrum. They employ strong logics and high-level declarations to provide a convenient way to formalize domain knowledge and reason about it. The reasoning is usually interactive in order to represent inferences that are too difficult to be fully automated. Most proof assistants integrate at least some of the other methods to overcome this weakness.

Further along the spectrum, *automated theorem provers* use simpler logics than interactive proof assistants. They are fully automatic and much faster, but can handle much fewer theorems, and typically do not check their proofs. *Satisfiability checkers* continue this progression by aiming at decidable automation support, whereas theorem proving is usually an semi-decidable search problem. That limits them to propositional logic or specific theories of more expressive logics (usually of first-order logic) that are complete, i.e., where every formula can be proved or disproved. In the special cases, where satisfiability checkers are applicable, they come close to verified computation systems.

Orthogonal to the above triplet, there are several methods for realizing Turing-complete computation naturally inside a logic. Here imperative and object-oriented computation are usually avoided in favor of other programming paradigms that are easier to reason about. *Rewriting* aims at optimizing the $f(x) \rightsquigarrow y$ progression, allowing users to mark specific transformations as rewrite steps. *Terminating recursion* is the method of adding recursive functions to a logic in order to make it a pure functional programming language. Finally, *logic programming* restricts attention to theorems of a special form, for which proof search is simple and predictable so that users can represent computations by supplying axioms that guide the proof search.

Part VI

Narrative Knowledge

Part VII

Conclusion

Bibliography

[CFKR20] J. Carette, W. Farmer, M. Kohlhase, and F. Rabe. Big Math and the One-Brain Barrier. *The Mathematical Intelligencer*, 2020. to appear.