Knowledge Representation and Processing

Florian Rabe (for a course given with Michael Kohlhase)

Computer Science, University Erlangen-Nürnberg, Germany

Summer 2020

Administrative Information

Format

Zoom

- lectures and exercises via zoom
- participants muted by default for simplicity
- interaction strongly encouraged

We don't want to lecture — we want to have a conversation during which you learn

- let's try out zoom
  - use reactions to say yes no, ask for break etc.
  - feel free to annotate my slides
  - talk in the chat

Recordings

- maybe prerecorded video lectures or recorded zoom meeting
- to be decided along the way

Background

Instructors

- Prof. Dr. Michael Kohlhase
  Professor of Knowledge Representation and Processing
- PD Dr. Florian Rabe — same research group

Course

- This course is given for the first time
- Always a little bit of an experiment — cutting edge vs. unpolished
- Could become signature course of our research group — same name!

Prerequisites

Required

- basic knowledge about formal languages, context-free grammars
  but we'll do a quick revision here

Helpful

- Algorithms and Data Structures mostly as a contrast to this lecture
- Basic logic we'll revise it slightly differently here
- all other courses as examples of how knowledge pervades all of CS

General

- Curiosity this course is a bit unusual
- Interest in big picture
  this course touches on lots of things from all over CS

Examination and Grading

Suggestion

- grade determined by single exam
- written or oral — depends on number of students
- some acknowledgment for practical exercises

to be finalized next week

Exam-relevant

- anything mentioned in notes
- anything discussed in lectures

neither is a superset of the other!

Materials and Exam-Relevance

Textbook

- does not exist
- normal for research-near specialization courses

Notes

- textbook-style but not as comprehensive
- developed along the way

Slides

- not comprehensive
- used as visual aid, conversation starters

Communication

Open for questions

- open door policy in our offices — if the lockdown ever ends
- always room for questions during lectures
- for personal questions, contact me during/after lecture or by email
- forum at `https://fsi.cs.fau.de/forum/154-Wissensrepraesentation-und-Verarbeitung`

Materials

- official notes and slides as pdf: `https://kwarc.info/teaching/WuV/`
  will be updated from time to time
- watch me prepare the materials: `https://github.com/florian-rabe/Teaching/tree/master/WuV`
  pull requests and issues welcome

Exercises

Learning Goals

- Get acquainted with state of the art of practice
- Try out real tools

Homeworks

- one major project as running example
- homeworks building on each other

build one large knowledge-based system

details on later slides

Overview and Essential Concepts

Representation and Processing

Common pairs of concepts:

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

Data and Knowledge

2×2 key concepts

Syntax	Data
Semantics	Knowledge

- Data: any object that can be stored in a computer
  Example: $((49.5739143, 11.0264941), "2020-04-21T16:15:00CEST")$
- Syntax: a system of rules that describes which data is **well-formed**
  Example: “a pair of (a pair of two IEEE double precision floating point numbers) and a string encoding of a time stamp”
- Semantics: system of rules that determines the meaning of well-formed data
- Knowledge: combination of some data with its syntax and semantics

Knowledge is Elusive

Representation of key concepts

- Data: using primitive objects
  implemented as bits, bytes, strings, records, arrays, ...
- Syntax: (context-free) grammars, (context-sensitive) type systems implemeted as inductive data structures
- Semantics: functions for evaluation, interpretation, of well-formed data
  implemented as recursive algorithms on the syntax
- Knowledge: elusive
  emerges from applying and interacting with the semantics

Semantics as Translation

- Knowledge can be captured by a higher layer of syntax
- Then semantics is translation into syntax

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	graphics context

Heterogeneity of Data and Knowledge

- Capturing knowledge is difficult
- Many different approaches to semantics
  - fundamental formal and methodological differences
  - often captured in different fields, conferences, courses, languages, tools
- Data formats equally heterogeneous
  - ontologies
  - programs
  - logical proofs
  - databases
  - documents

Challenges of Heterogeneity

Challenges

- collaboration across communities
- translation across languages
- conversion between data formats
- interoperability across tools

Sources of problems

- interoperability across formats/tools major source of
  - complexity
  - bugs
- friction in project team due to differing preferences, expertise
- difficult choice between languages/tools with competing advantages
  - reverting choices difficult, costly
  - maintaining legacy choices increases complexity

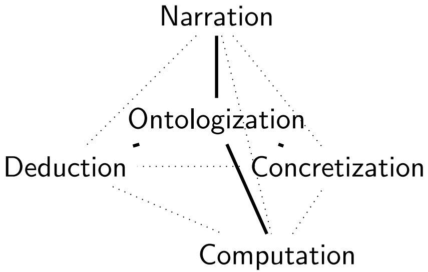
Aspects of Knowledge

- Tetrapod model of knowledge active research by our group
- classifies approaches to knowledge into five aspects

Aspect	KRLs (examples)
ontologization	ontology languages (OWL), description logics (ALC)
concretization	relational databases (SQL, JSON)
computation	programming languages (C)
deduction	logics (HOL)
narration	document languages (HTML, LaTeX)

Relations between the Aspects

Ontology is distinguished: capture the knowledge that the other four aspects share



Complementary Advantages of the Aspects

Aspect	objects	characteristic		
		advantage	joint advantage of the other aspects	application
ded.	formal proofs	correctness	ease of use	verification
comp.	programs	efficiency	well-definedness	execution
concr.	concrete objects	tangibility	abstraction	storage/retrieval
narr.	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

Structure of the Course

Aspect-independent parts

- general methods that are shared among the aspects
- to be discussed as they come up

Aspects-specific parts

- one part (about 2 weeks) for each aspect
- high-level overview of state of the art
- focus on comparison/evaluation of the aspect-specific results

Structure of the Exercises

One major project

- representative for a project that a CS graduate might be put in charge of
- challenging heterogeneous data and knowledge
- requires integrating/combining different languages, tools

unique opportunity in this course because knowledge is everywhere

Concrete project

- develop a univis-style system for a university
- lots of heterogeneous knowledge
  - course and program descriptions
  - legal texts
  - websites
  - grade tables
  - transcript generation code
- build a completely functional system applying the lessons of the course

Ontological Knowledge

Components of an Ontology

8 main declarations

- **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, e.g., "instructor" $\sqsubseteq$ "person"

Divisions of an Ontology

Abstract vs. concrete

- TBox: concepts, relations, properties, axioms
  everything that does not use individuals
- ABox: individuals and assertions

Named vs. unnamed

- Signature: individuals, concepts, relations, properties
  together called entities or resources
- Theory: assertions, axioms

Comparison of Terminology

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

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

[1] as in `https://plato.stanford.edu/entries/object/`

Ontologies as Sets of Triples

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

Efficient representation of ontologies using RDF and RDFS standardized special entities.

Special Entities

RDF and RDFS define special entities for use in ontologies:

- "rdfs:Resource": concept of which all individuals are an instance and thus of which every concept is a subconcept
- "rdf:type": 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": special class for the type of classes
- "rdf:Property": special class for 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

Goal/effect: capture as many parts as possible as RDF triples.

Declarations as Triples using Special Entities

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

An Example Ontology Language

see syntax of BOL in the lecture notes

Semantics as Translation

Example: Syntax of Arithmetic Language

Syntax: represented as formal grammar

$\boxed{\text{Numbers}}$

$$
\begin{array}{rclll}
N & ::= & 0 \mid 1 & & \text{literals} \\
 & \mid & N + N & & \text{sum} \\
 & \mid & N * N & & \text{product}
\end{array}
$$

$\boxed{\text{Formulas}}$

$$
\begin{array}{rclll}
F & ::= & N \doteq N & & \text{equality} \\
 & \mid & N \leq N & & \text{ordering by size}
\end{array}
$$

Implementation as inductive data type

Example: Semantics of Arithmetic Language

Semantics: represented as translation into known language

Problem: Need to choose a known language first
Here: unary numbers represented as strings
Built-in data (strings and booleans):

$$
\begin{array}{rcll}
S & ::= & \varepsilon & \text{empty} \\
  & | & \texttt{(Unicode)} & \text{characters} \\
B & ::= & \texttt{true} & \text{truth} \\
  & | & \texttt{false} & \text{falsity}
\end{array}
$$

Built-in operations to work on the data:

- concatenation of strings $S ::= \texttt{conc}(S, S)$
- replacing all occurrences of c in S_1 with S_2
  $S ::= \texttt{replace}(S_1, c, S_2)$
- equality test: $B ::= S_1 == S_2$
- prefix test: $B ::= \texttt{startsWith}(S_1, S_2)$

Example: Semantics of Arithmetic Language

Represented as function from syntax to semantics

- mutually recursive, inductive functions for each non-terminal symbol
- compositional: recursive call on immediate subterms of argument

For numbers n: semantics $[\![n]\!]$ is a string

- $[\![0]\!] = \varepsilon$
- $[\![1]\!] = "|"$
- $[\![m + n]\!] = \mathtt{conc}([\![m]\!], [\![n]\!])$
- $[\![m * n]\!] = \mathtt{replace}([\![m]\!], "|", [\![n]\!])$

For formulas f: semantics $[\![f]\!]$ is a boolean

- $[\![m \doteq n]\!] = [\![m]\!] == [\![n]\!]$
- $[\![m \leq n]\!] = \mathtt{startsWith}([\![n]\!], [\![m]\!])$

Semantics of BOL

Aspect	kind of semantic language	semantic language
deduction	logic	SFOL
concretization	database language	SQL
computation	programming language	Scala
narration	natural language	English

see details of each translation in the lecture notes

General Definition

A semantics by translation consists of

- syntax: a formal language l
- semantic language: a formal language L

  different or same aspect as l
- semantic prefix: a theory P in L

  formalizes fundamentals that are needed to represent l-objects
- interpretation: translates every l-theory T to an L-theory $P, [\![T]\!]$

Common Principles

Properties shared by all semantics of BOL

not part of formal definition, but best practices

- l-declaration translated to L-declaration for the same name
- ontologies translated declaration-wise
- one inductive function for every kind of complex l-expression
  - individuals, concepts, relations, properties, formulas
  - maps l-expressions to L-expressions
- atomic cases (base cases): l-identifier translated to L-identifier of the same name — or something very similar
- complex cases (step cases): compositional

Compositionality

Case for operator $*$ in interpretation function compositional iff interpretation of $*(e_1, \ldots, e_n)$ only depends on on the interpretation of the e_i

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

for some function $[\![*]\!]$

Example: ;-operator of BOL in translation to FOL

- translation: $[\![R_1; R_2]\!] = \exists m : \iota.[\![R_1]\!](x, m) \wedge [\![R_2]\!](m, y)$
- special case of the above via
  - $* =;$
  - $n = 2$
  - $[\![;]\!] = (p_1, p_2) \mapsto \exists m : \iota.p_1(x, m) \wedge p_2(m, y)$
- Indeed, we have $[\![R_1; R_2]\!] = [\![;]\!]([\![R_1]\!], [\![R_2]\!])$

Compositionality (2)

Translation compositional iff

- one translation function for each non-terminal all written $[\![-]\!]$
- each defined by one induction on syntax

  i.e., one case for production

  mutually recursive
- all cases compositional

Substitution theorem: a compositional translation satisfies

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

for

- every expression $E(N_1, \ldots, N_n)$ with non-terminals N_i
- some function $[\![E]\!]$ that only depends on E

Compositionality (3)

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

for every expression $E(N_1, \ldots, N_n)$ with non-terminals N_i

Now think of

- variable x_i of type N_i instead of non-terminal N_i
- $E(x_1, \ldots, x_n)$ as expression with free variables x_i of type N_i
- expressions e derived from N as expressions of type N
- $E(e_1, \ldots, e_n)$ as result of substituting e_i for x_i
- $[\![E]\!](x_1, \ldots, x_n)$ as (semantic) expression with free variables x_i

Then both sides of equations act on $E(x_1, \ldots, x_n)$:

- left side yields $[\![E(e_1, \ldots, e_n)]\!]$ by
  - first substitution e_i for x_i
  - then semantics $[\![-]\!]$ of the whole
- right side yields $[\![E]\!]([\![e_1]\!], \ldots, [\![e_n]\!])$ by
  - first semantics $[\![-]\!]$ of all parts
  - then substitution $[\![e_i]\!]$ for x_i

semantics commutes with substitution

Non-Compositionality

Examples

- deduction: cut elimination, translation from natural deduction to Hilbert calculus
- computation: optimizing compiler, e.g., loop unrolling
- concretization: query optimization, e.g., turning a WHERE of a join into a join of WHEREs,
- narration: ambiguous words are translated based on context

Typical sources

- subcases in a case of translation function
  - based on inspecting the arguments, e.g., subinduction
  - based on context
- custom-built semantic prefix

Type Systems

Breakout Question

Is this an improvement over BOL?

$\boxed{\text{Declarations}}$

D ::=	**individual** $\mathtt{ID} : C$	typed atomic individual
|	**concept** $\mathtt{ID}$	atomic concept
|	**relation** $\mathtt{ID} \subseteq C \times C$	typed atomic relation
|	**property** $\mathtt{ID} \subseteq C \times T$	typed atomic property

rest as before

Actually, when is a language an improvement?

Criteria: orthogonal, often mutually exclusive

- syntax design trade-off
  - expressivity: easy to express knowledge
    e.g., big grammar, extra production for every user need
  - simplicity: easy to implement/interpret
    e.g., few, carefully chosen productions
- semantics: specify, implement, document
- intended users
  - skill level
  - prior experience with related languages
  - amount of training needed
- long-term plans: re-answer the above question but now
  - maintainability: syntax was changed, everything to be redone
  - scalability: expressed knowledge content has reached huge sizes

Church vs. Curry Typing

	intrinsic	extrinsic
λ-calculus by	Church	Curry
type is	carried by object	given by environment
typing is a	function objects $\rightarrow$ types	relation objects $\times$ types
objects have	unique type	any number of types
types interpreted as	disjoint sets	unary predicates
type given by	part of declaration	additional axiom
example	**individual** "WuV":"course"	**individual** "Wuv",
		"WuV" `is-a` "course"
examples	SFOL, SQL	OWL, Scala, English
	most logics, functional PLs	ontology, OO,
		natural languages
	many type theories	set theories

Type Checking

	intrinsic	extrinsic
type is	carried by object	given by environment
typing is a	function objects $\rightarrow$ types	relation objects $\times$ types
objects have	unique type	any number of types
type given by	part of declaration	additional axiom
example	**individual** "WuV":"course"	**individual** "Wuv", "WuV" `is-a` "course"
type inference for x	uniquely infer A from x	find minimal A with $x : A$
type checking	inferred=expected	prove $x : A$
subtyping $A <: B$	cast from A to B	$x : A$ implies $x : B$
typing decidable	yes unless too expressive	no unless restricted
typing errors	static (compile-time)	dynamic (run-time)
advantages	easy unique type inference	flexible allows subtyping

Curry Typing in BOL

language	objects	types	typing relation
Syntax	individuals	concepts	i `is-a` c
Semantics in			
FOL	type ι	predicates $c \subseteq \iota$	$c(i)$ true
SQL	table Individuals	tables containing ids	id of i in table c
Scala	String	hash sets of strings	c.contains(i)
English	proper nouns	common nouns	"i is a c" is true

Subtyping

Subtyping works best with Curry Typing

- 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 $\{x : \mathbb{N}, y : \mathbb{Z}\} <: \{x : \mathbb{N}\}$

A General Definition of a Type System

A **type system** consists of

- a collection, whose elements are called **objects**,
- a collection, whose elements are called **intrinsic types**,
- a function assigning to every object x its **intrinsic type** I, in which case we write $x : I$,
- for some intrinsic types I
  - an intrinsic type E_I
  - a relation $\in_I$ between objects with intrinsic types I and E_I, called the **extrinsic typing** relation for I.

Examples

System	intrinsic types	E_I	$\in_I$
pure Church	one per type	none	none
pure Curry	objects O, types T	$E_O = T$	$\in_O =:$
FOL	one per type	none	none
Scala	$AnyRef$, $Class$	$E_{Any} = Class$	$\in_{Any} = \mathtt{isInstance}$
BOL	Ind, $Conc$	$E_{Ind} = Conc$	$\in_{Ind} = \mathtt{is} - \mathtt{a}$
set theory	Set, $Prop$	$E_{Set} = Set$	$\in_{Set} = \in$

Breakout Question

What do the following have in common?

- Java class
- SQL schema for a table
- logical theory (e.g., Monoid)

Breakout Question

What do the following have in common?

- Java class
- SQL schema for a table
- logical theory (e.g., Monoid)

all are (essentially) abstract data types

Abstract Data Types: Motivation

Recall subject-centered representation of assertion triples:

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

Can we use types to force certain assertions to occur together?

- Every instructor should teach a list of courses.
- Every instructor should have an office.

Abstract Data Types: Motivation

Inspires **subject-centered types**, e.g.,

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

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

Incidental benefits:

- no need to declare relations/properties separately
- reuse relation/property names
  distinguish via qualified names: instructor .age

Abstract Data Types: Motivation

Natural next step: inheritance

```
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"
```

our language quickly gets a very different flavor

Abstract Data Types: Examples

Prevalence of abstract data types:

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

same idea, but may look very different across languages

Abstract vs. Concrete Types

Concrete type: values are

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

examples: products, inductive data types

Abstract type: values are

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

main example: abstract data types

Abstract Data Types: Examples

aspect	type	values
computation	abstract class	instances of implementing classes
concretization	table schema	table rows
deduction	theory	models

Values depend on the environment in which the type is used:

- class defined in one specification language (e.g., UML), implementations in programing languages Java, Scala, etc.
  available values may depend on run-time state
- theory defined in logic, models defined in set theories, type theories, programming languages
  available values may depend on philosophical position

Abstract Data Types: Definition

Given some type system, an **abstract data type** (ADT) is

- a **flat** type

  $$\{c_1 : T_1[= t_1], \ldots, c_n : T_n[= t_n]\}$$

  where
  - c_i are distinct names
  - T_i are types
  - t_i are optional definitions; if given, $t_i : T_i$ required
- or a **mixin** type

  $$A_1 * \ldots * A_n$$

  for ADTs A_i.

Languages may or may not make ADTs additional types of the type system

Abstract Data Types: Class Definitions

A class definition in OO:

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

Corresponding ADT definition:

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

The usual terminology:

- a **inherits** from a_i
- a_i are **super**-X or **parent**-X of a where X is whatever the language calls its ADTs (e.g., X=class)

Abstract Data Types: Flattening

The **flattening** $A^\flat$ of an ADT A is

- if A is flat: $A^\flat = A$
- $(A_1 * \ldots * A_n)^\flat$ is union of all $A_i^\flat$
  where duplicate field names are handled as follows
  - same name, same type, same or omitted definition: merge
    details may be much more difficult
  - otherwise: ill-formed

Abstract Data Types: Subtleties

We gloss over several major issues:

- How exactly do we merge duplicate field names? Does it always work? implement abstract methods, override, overload
- Is recursion allowed, i.e., can I define an ADT $a = A$ where a occurs in A?

  common in OO-languages: use a in the types of its fields
- What about ADTs with type arguments?

  e.g., generics in Java, square-brackets in Scala
- Is mutual recursion between fields in a flat type allowed?

  common in OO-languages
- Is * commutative? What about dependencies between fields?

no unique answers

incarnations of ADTs subtly different across languages

Context-Sensitive Syntax

Definition

A **language system** consists of

- context-free syntax
- distinguished non-terminal symbol $\mathcal{V}$

  words called **vocabularies**
- some distinguished non-terminal symbols $\mathcal{E}$

  words called $\mathcal{E}$-**expressions**
- unary predicate $\mathrm{wft}(\Theta)$ on vocabularies Θ

  well-formed vocabulary Θ
- unary predicates $\mathrm{wff}^{\mathcal{E}}_{\Theta}(E)$ well-formed $\mathcal{E}$-expressions E

Typical Structure

Vocabularies

- lists of declarations

Declarations

- named
- at least one for each expression kind
- may contain other expressions — e.g., type, definition
- may contain nested declarations — e.g., fields in an ADT

Expressions

- inductive data type
- relative to vocabulary — names occur as base cases
- formulas as special case

Vocabularies and Expressions

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	signature, theory	term, type, formula, ...
Narration	dictionary	phrases, sentences, texts

Examples

See notes made during the lecture for examples

Concrete Knowledge

Motivation

Main ideas

- Ontology abstractly describes concepts and relations
- Tool maintains concrete data set
- Focus on efficiently
  - identifying (i.e., assign names)
  - representing
  - processing
  - querying

  large sets of concrete data

Recall: TBox-ABox distinction

- TBox: general parts, abstract, fixed
  main challenge: correct modeling of domain
- ABox: concrete individuals and assertions about them, growing
  main challenge: aggregate them all

Concrete Data

Concrete is

- Base values: integers, strings, booleans, etc.
- Collections: sets, multisets, lists (always finite)
- Aggregations: tuples, records (always finite)
- User-defined concrete data: enumerations, inductive types
- Advanced objects: finite maps, graphs, etc.

Concrete is not

- Free symbols to be interpreted by a model
  exception: foreign function interfaces
- Variables (free or bound) λ-abstraction, quantification
- Symbolic expressions formulas, algorithms
  Exceptions:
  - expressions of inductive type
  - application of built-in functions
  - queries that return concrete data

Breakout question

What is the difference between

- an OWL ontology
- an SQL database

Two Approaches

Based on untyped (Curry-typed) ontology languages

- Representation based on knowledge graph
- Ontology written in BOL-like language
- Data maintained as set of triples — tool = triple store
- Typical language/tool design
  - ontology and query language separate — e.g., OWL, SPARQL
  - triple store and query engine integrated — e.g., Virtuoso tool

Based on typed (Church-typed) ontology languages

- Representation based on abstract data types
- Ontology written as database schema
- Data maintained as tables — tool = (relational) database
- Typical language/tool design
  - ontology and query language integrated — e.g., SQL
  - table store and query engine integrated — e.g., SQLite tool

Evolution of Approaches

Our usage is non-standard

- Common
  - ontologies = untyped approach, OWL, triples, SPARQL
  - databases = typed approach, tables, SQL
- Our understanding: two approaches evolved from same idea
  - triple store = untyped database
  - SQL schema = typed ontology

Evolution

- Typed-untyped distinction minor technical difference
- Optimization of respective advantages causes speciation
- Today segregation into different
  - jargons
  - languages, tools
  - communities, conferences
  - courses

Curry-typed concrete data

Central data structure = knowledge graph

- nodes = individuals i
  - identifier
  - sets of concepts of i
  - key-value sets of properties of i
- edges = relation assertions
  - from subject to object
  - labeled with name of relation

Processing strengths

- store: as triple set
- edit: Protege-style or graph-based
- visualize: as graph — different colors for concepts, relations
- query: match, traverse graph structure
- untyped data simplifies integration, migration

Church-typed concrete data

Central data structure = relational database

- tables = abstract data type
- rows = objects of that type
- columns = fields of ADT
- cells = values of fields

Processing strengths

- store: as CSV text files, or similar
- edit: SQL commands or table editors
- visualize: as table view
- query: relational algebra
- typed data simplifies selecting, sorting, aggregating

Identifiers

Curry-Typed Knowledge graph

- concept, relation, property names given in TBox
- individual names attached to nodes

Church-Typed Database

- table, column names given in schema
- row identified by distinguished column (= key)
  options
  - preexistent characteristic column
  - added upon insertion
    - UUID string
    - incremental integers
    - concatenation of characteristic list of columns
- column/row identifiers formed by qualifying with table name

Axioms

Curry-Typed Knowledge Graph

- traditionally very expressive axioms
- yields inferred assertions
- triple store must do consequence closure to return correct query results
- not all axioms supported by every triple store

Church-Typed Database

- typically no axioms
- instead consistency constraints, triggers
- allows limited support for axioms without calling it that way
- stronger need for users to program the consequence closure manually

Breakout question

When using typed concrete data,
how to fully realize abstract data types

- nesting: ADTs occurring as field types
- inheritance between ADTs
- mixins

ADTs in Typed Concrete Data

Nesting: field $a : A$ in ADT B

- field types must be base types, $a : A$ not allowed
- allow ID as additional base type
- use field $a : ID$ in table B
- store value of b in table A

Inheritance: B inherits from A

- add field $parent_A$ to table B
- store values of inherited fields of B in table A

general principle: all objects of type A stored in same table

Mixin: $A * B$

- essentially join of tables A and B on common fields
- some subtleties depending on ADT flattening

Open/Closed World

- Question: is the data complete?
  - closed world: yes
  - open world: not necessarily
- Dimensions of openness
  - existence of individual objects
  - assertions about them
- Sources of openness
  - more exists but has not yet been added
  - more could be created later
- Orthogonal to typed/untyped distinction, but in practice
  - knowledge graphs use open world
  - databases use closed world

Open world is natural state, closing adds knowledge

Closing the World

Derivable consequences

- induction: prove universal property by proving for each object
- negation by failure: atomic property false if not provable
- term-generation constraint: only nameable objects exist

Enabled operations

- universal set: all objects
- complement of concept/type
- defaults: assume default value for property if not otherwise asserted

Monotonicity problem

- monotone operation: bigger world = more results
- examples: union, intersection, $\exists R.C$, join, IN conditions
- counter-examples: complement, $\forall R.C$, NOT IN conditions

technically, non-monotone operations in open world dubious

Primitive Types and Encoding Data

Primitive vs. Declared

Primitive Types

- built-into the language
- assumed to exist a priori — fundamentals of nature
- fixed semantics (usually interpreted by identity function)

Triple Structure: 3 kinds of named objects

- the type — eg: 'int'
- values of the type — eg: 0, 1, -1, ...
- operations on type — eg: addition, multiplication, ...

	primitive	declared
introduced by	language designer	user
introduced in	grammar	vocabulary V
visible in	all vocabularies	V only
semantics given	explicitly	implicitly
... by	translation function	axioms

Examples

Typical primitive types

- natural numbers ($= \mathbb{N}$)
- arbitrary precision integers ($= \mathbb{Z}$)
- fixed precision integers (32 bit, 64 bit, ...)
- floating point (float, double, ...)
- Booleans
- characters (ASCII, Unicode)
- strings

Observation:

- essentially the same in every language
  including whatever language used for semantics
- semantics by translation trivial

Quasi-Primitive = Declared in standard library

Standard library

- present in every language — assumed empty vocabulary by default
- one fixed vocabulary
  - implicitly included into every other vocabulary
  - implicitly fixed by any translation between vocabularies
- objects technically declared
- but practically part of primitive objects

Examples

- sufficiently expressive languages
  - push many primitive objects to standard library — never all
  - simplifies language, especially when defining operations

  strings in C, BigInteger in Java, inductive type for $\mathbb{N}$
- inexpressive languages
  - many primitives — SQL, spreadsheet software
  - few (quasi)-primitives — few operations available in OWL

Treatment in this Course

BOL syntax and semantics so far

- primitive objects omitted in syntax
- assumed reasonable collection available
- assumed same (quasi-)primitive objects in semantic languages
  irrelevant if interpreting primitive objects as primitive or quasi-primitive

largely justified by practical languages

But what exactly is the standard?

- will present possible solution
- uses special ontology language just for specifying primitive objects
  - name
  - type
  - semantics
    typically narrative; alternatively deductive, computational
- current research, not standard practice

Encoding Primitive Types

Problem

- quickly encounter primitive types not supported by common languages
- need to encode them using existing types

  typically as strings, ints, or prodcuts/lists thereof

Examples

- date, time, color, location on earth
- graph, function
- picture, audio, video
- gene, person

Breakout questions: What primitive types do we need for univis?

Failures of Encodings

Genes in Excel

- 2016 study found errors in 20% of spreadsheets accompanying genomics journal papers
- gene names encoded as strings but auto-converted to other types by Excel
  - "SEPT2" (Septin 2) converted to September 02)
  - REKIN identifiers, e.g., "2310009E13", converted to float $2.31E + 1$

https://genomebiology.biomedcentral.com/articles/10.1186/s13

Y2K bug

- date encoded as tuple of integers, using 2 digits for year
- needed fixing in year 2000
- estimated $300 billion spent to change software
- possible repeat: in 2038, number of seconds since 1970-01-01 (used by Unix to encode time as integer) overflows 32-bit integers

Research Goal for Aspect-Independent Data in Tetrapod

Standardization of Common Data Types

- Ontology language optimized for declaring types, values, operations

  semantics must exist but can be extra-linguistic
- Vocabulary declaring such objects

  should be standardized, modular, extensible

Standardization of Codecs

- Fixed small set of primitive objects

  should be (quasi-)primitive in every language

  not too expressive, possibly untyped
- Standard codecs for translating common types to primitive ones

Codec for type A

- coding function A-values $\to$ primitive objects
- partial decoding function primitive objects $\to$ A-values
- inverse to each other — in some sense

Complex Data Types

- relatively easy if all primitive types atomic   int, string, etc.
- but need to allow for complex types

Finite Collection Types

- lists
- fixed-length lists (= Cartesian power, n-tuple)
- sets
- multisets (= bags)

all homogeneous, e.g. *list* A is type of lists over A

Aggregation Types

- Cartesian product of some types $A \times B$
- disjoint union of some types $A \cup B$
- labeled Cartesian product (= record) $\{a : A, b : B\}$

  unordered, named projections
- labeled disjoint union $a(A)|b(B)$

  uncommon, only mentioned for systematics

Subtyping

- relatively easy if all data types disjoint
- better with subtyping — open problem how to do it nicely

Subtyping Atomic Types

- $\mathbb{N} <: \mathbb{Z}$
- ASCII $<:$ Unicode

Subtyping Complex Types

- Covariance subtyping (= vertical subtyping) same for disjoint unions

$$A <: A' \Rightarrow \mathit{list}\ A <: \mathit{list}\ A'$$

$$A_i <: A'_i \Rightarrow \{\ldots, a_i : A_i, \ldots\} <: \{\ldots, a_i : A'_i, \ldots\}$$

- structural subtyping (= horizontal subtyping)

$$\{a : A, b : B\} :> \{a : A, b : B, c : C\}$$

$$a(A)|b(B) <: a(A)|b(B)|c(C)$$