

# 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

$2 \times 2$  key concepts

Syntax	Data
Semantics	Knowledge

- ▶ Data: any object that can be stored in a computer  
Example:  $((49.5739143, 11.0264941), "2020 - 04 - 21 T16 : 15 : 00 CEST")$
- ▶ 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	advantage	characteristic joint advantage of the other as- pects	application
ded. comp.	formal proofs programs	correctness efficiency	ease of use well- definedness	verification execution
concr. narr.	concrete objects texts	tangibility flexibility	abstraction formal seman- tics	storage/retrieval human understanding

Aspect pair	characteristic advantage
ded./comp. narr./concr.	rich meta-theory simple languages
ded./narr. comp./concr.	theorems and proofs normalization
ded./concr. comp./narr.	decidable well-definedness 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 <sup>1</sup>	object	property
grammar	proper noun	common noun

<sup>1</sup>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"	$d$
$\text{dom } r \equiv c$	$r$	"rdfs:domain"	$c$
$\text{rng } r \equiv c$	$r$	"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

### Numbers

$N ::= 0$		$1$	literals
		$N + N$	sum
		$N * N$	product

### Formulas

$F ::= N \doteq N$	equality
$N \leq N$	ordering by size

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):

$S ::= \varepsilon$	empty
(Unicode)	characters
$B ::= \text{true}$	truth
false	falsity

Built-in operations to work on the data:

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



## 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  $\llbracket n \rrbracket$  is a string

- ▶  $\llbracket 0 \rrbracket = \varepsilon$
- ▶  $\llbracket 1 \rrbracket = \text{"|"}$
- ▶  $\llbracket m + n \rrbracket = \text{conc}(\llbracket m \rrbracket, \llbracket n \rrbracket)$
- ▶  $\llbracket m * n \rrbracket = \text{replace}(\llbracket m \rrbracket, \text{"|"}, \llbracket n \rrbracket)$

For formulas  $f$ : semantics  $\llbracket f \rrbracket$  is a boolean

- ▶  $\llbracket m \dot{=} n \rrbracket = \llbracket m \rrbracket == \llbracket n \rrbracket$
- ▶  $\llbracket m \leq n \rrbracket = \text{startsWith}(\llbracket n \rrbracket, \llbracket m \rrbracket)$

## 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  $I$
- ▶ semantic language: a formal language  $L$   
different or same aspect as  $I$
- ▶ semantic prefix: a theory  $P$  in  $L$   
formalizes fundamentals that are needed to represent  $I$ -objects
- ▶ interpretation: translates every  $I$ -theory  $T$  to an  $L$ -theory  $P, \llbracket T \rrbracket$

## Common Principles

Properties shared by all semantics of BOL

not part of formal definition, but best practices

- ▶  $I$ -declaration translated to  $L$ -declaration for the same name
- ▶ ontologies translated declaration-wise
- ▶ one inductive function for every kind of complex  $I$ -expression
  - ▶ individuals, concepts, relations, properties, formulas
  - ▶ maps  $I$ -expressions to  $L$ -expressions
- ▶ atomic cases (base cases):  $I$ -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  $*(e_1, \dots, e_n)$  only depends on the interpretation of the  $e_i$

$$\llbracket *(e_1, \dots, e_n) \rrbracket = \llbracket * \rrbracket (\llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket)$$

for some function  $\llbracket * \rrbracket$

Example:  $;$ -operator of BOL in translation to FOL

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

## Compositionality (2)

Translation compositional iff

- ▶ one translation function for each non-terminal all written  $\llbracket - \rrbracket$
- ▶ each defined by one induction on syntax
  - i.e., one case for production
  - mutually recursive
- ▶ all cases compositional

Substitution theorem: a compositional translation satisfies

$$\llbracket E(e_1, \dots, e_n) \rrbracket = \llbracket E \rrbracket(\llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket)$$

for

- ▶ every expression  $E(N_1, \dots, N_n)$  with non-terminals  $N_i$
- ▶ some function  $\llbracket E \rrbracket$  that only depends on  $E$

## Compositionality (3)

$$\llbracket E(e_1, \dots, e_n) \rrbracket = \llbracket E \rrbracket(\llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket)$$

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

Now think of

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

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

- ▶ left side yields  $\llbracket E(e_1, \dots, e_n) \rrbracket$  by
  - ▶ first substitution  $e_i$  for  $x_i$
  - ▶ then semantics  $\llbracket - \rrbracket$  of the whole
- ▶ right side yields  $\llbracket E \rrbracket(\llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket)$  by
  - ▶ first semantics  $\llbracket - \rrbracket$  of all parts
  - ▶ then substitution  $\llbracket e_i \rrbracket$  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?

Declarations
--------------

$D ::=$	<b>individual</b>	$ID : C$	typed atomic individual
	<b>concept</b>	$ID$	atomic concept
	<b>relation</b>	$ID \subseteq C \times C$	typed atomic relation
	<b>property</b>	$ID \subseteq C \times T$	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
$\lambda$ -calculus by type is typing is a objects have types interpreted as	Church carried by object function objects $\rightarrow$ types unique type disjoint sets	Curry given by environment relation objects $\times$ types any number of types unary predicates
type given by example	part of declaration <b>individual</b> "WuV" : "course"	additional axiom <b>individual</b> "Wuv", "WuV" is-a "course"
examples	SFOL, SQL most logics, functional PLs  many type theories	OWL, Scala, English ontology, OO, natural languages set theories

# Type Checking

	intrinsic	extrinsic
type is typing is a objects have	carried by object function objects $\rightarrow$ types unique type	given by environment relation objects $\times$ types any number of types
type given by example	part of declaration <b>individual</b> "WuV" : "course"	additional axiom <b>individual</b> "Wuv", "WuV" is-a "course"
type inference for $x$ type checking subtyping $A <: B$ typing decidable typing errors	uniquely infer $A$ from $x$ inferred=expected cast from $A$ to $B$ yes unless too expressive static (compile-time)	find minimal $A$ with $x : A$ prove $x : A$ $x : A$ implies $x : B$ no unless restricted 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 $\iota$	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	$c.\text{contains}(i)$
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  $\{x : \mathbb{N} \mid x \neq 0\}$
- ▶ 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	<i>AnyRef</i> , <i>Class</i>	$E_{Any} = Class$	$\in_{Any} = \text{isInstance}$
BOL	<i>Ind</i> , <i>Conc</i>	$E_{Ind} = Conc$	$\in_{Ind} = \text{is} - a$
set theory	<i>Set</i> , <i>Prop</i>	$E_{Set} = Set$	$\in_{Set} = \in$

## 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"  $\sqcap$  "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 programming 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

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

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

$$A_1 * \dots * A_n$$

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  $a_m$  {  
   $c_1 : T_1$   
   $\vdots$   
   $c_n : T_n$   
}
```

Corresponding ADT definition:

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

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.,  $X=\text{class}$ )

# Abstract Data Types: Flattening

The **flattening**  $A^b$  of an ADT  $A$  is

- ▶ if  $A$  is flat:  $A^b = A$
- ▶  $(A_1 * \dots * A_n)^b$  is union of all  $A_i^b$   
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  $\text{wft}(\Theta)$  on vocabularies  $\Theta$   
well-formed vocabulary  $\Theta$
- ▶ unary predicates  $\text{wff}_{\Theta}^{\mathcal{E}}(E)$  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 $\Theta$	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)  $\lambda$ -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 Types and Encoding Data

## Motivation

# Data Interoperability

## Situation

- ▶ languages systems focus on different aspects  
frequent need to exchange data
- ▶ generally, lots of aspect/language-specific objects  
proofs, programs, tables, sentences
- ▶ but same/similar **primitive** data types used across systems  
should be easy to exchange

## Problem

- ▶ crossing system barriers usually require interchange language  
serialize as string and repars
- ▶ interchange languages typically untyped XML, JSON, YAML, ...

## Solution

- ▶ standardize primitive data types
- ▶ standardize encoding in interchange languages

## 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
  - ▶ semanticstypically 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 products/lists thereof

## Examples

- ▶ date, time, color, location on earth
- ▶ graph, function
- ▶ picture, audio, video
- ▶ physical quantities (1*m*, 1*in*, etc.)
- ▶ gene, person

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



## Failures of Encodings

### 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

### 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  $2.31E + 1$

## Failures of Encodings (2)

### Mars Climate Orbiter

- ▶ two components exchanged physical quantity
- ▶ specification required encoding as number using unit Newton seconds
- ▶ one component used wrong encoding (with pound seconds as unit)
- ▶ led to false trajectory and loss of \$300 million device

### Shellshock

- ▶ bash allowed gaining root access from 1998 to 2014
- ▶ function definitions were encoded as source code
- ▶ not decoded at all; instead, code simply run (as root)
- ▶ allowed appending ";" ... to function definitions

SQL injection similar: complex data encoded as string, no decoding

# 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 interchange languages

## Codec for type $A$ and int. lang. $L$

- ▶ coding function  $A$ -values  $\rightarrow L$ -objects
  - ▶ partial decoding function  $L$ -objects  $\rightarrow A$ -values
  - ▶ inverse to each other
- in some sense

# Overview

Next steps

1. Data types
2. Data interchange languages
3. Codecs

# Primitive Types and Encoding Data

## Data Types

## Breakout Question

What types do we need?

## Atomic Data Types: basic

### typical in IT systems

- ▶ fixed precision integers (32 bit, 64 bit, ...)
- ▶ IEEE float, double
- ▶ Booleans
- ▶ Unicode characters
- ▶ strings                      could be list of characters but usually bad idea

### typical in math

- ▶ natural numbers ( $= \mathbb{N}$ )
- ▶ arbitrary precision integers ( $= \mathbb{Z}$ )
- ▶ rational, real, complex numbers
- ▶ graphs, trees

clear: language must be modular, extensible

## Atomic Data Types: advanced

### general purpose

- ▶ date, time, color, location on earth
- ▶ picture, audio, video

### domain-specific

- ▶ physical quantities (*1m*, *1in*, etc.)
- ▶ gene, person
- ▶ semester, course id, ...

clear: language must be modular, extensible



## Complex Data Types

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

Two kinds

- ▶ type operators: take **only type arguments**, return types
  - ▶ type operator  $\times$
  - ▶ takes two types  $A, B$
  - ▶ returns type  $A \times B$
- ▶ dependent types: take **also data arguments**, return types
  - ▶ dependent type operator *vector*
  - ▶ takes natural number  $n$ , type  $A$
  - ▶ returns type  $A^n$  of  $n$ -tuples over  $A$

dependent types much more complicated, less uniformly used  
harder to standardize

# Collection Data Types

## Homogeneous Collection Types

- ▶ sets
- ▶ multisets (= bags)
- ▶ lists      all unary type operators, e.g. *list A* is type of lists over *A*
- ▶ fixed-length lists (= Cartesian power, vector *n*-tuple)  
dependent type operator

## Heterogeneous Collection Types

- ▶ lists
- ▶ fixed-length lists (= Cartesian power, *n*-tuple)
- ▶ sets
- ▶ multisets (= bags)  
all atomic types, e.g., *list* is type of lists over any objects

## Aggregation Data Types

### Products

- ▶ Cartesian product of some types  $A \times B$   
values are pairs  $(x, y)$     numbered projections  $_1, _2$  — order relevant
- ▶ labeled Cartesian product (= record)  $\{a : A, b : B\}$   
values are records  $\{a = x, b = y\}$   
named projections  $a, b$  — order irrelevant

### Disjoint Unions

- ▶ disjoint union of some types  $A \uplus B$   
values are  $inj_1(x), inj_2(y)$  numbered injections  $_1, _2$  — order relevant
- ▶ labeled disjoint union  $a(A) \mid b(B)$   
values are constructor applications  $a(x), b(y)$   
named injections  $a, b$  — order irrelevant

labeled disjoint unions uncommon  
but recursive labeled disjoint union = inductive data type

## 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

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

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

- ▶ structural subtyping (= horizontal subtyping)

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

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

# Primitive Types and Encoding Data

## Data Representation Languages

## Overview

### General Properties

- ▶ general purpose or domain-specific
- ▶ typed or untyped
  - typical: Church-typed but no type operators, quasi untyped
- ▶ text or binary serialization
- ▶ libraries for many programming languages
  - ▶ data structures
  - ▶ serialization (data structure to string)
  - ▶ parsing (string to data structure, partial)

### Candidates

- ▶ XML: standard on the web, notoriously verbose
- ▶ JSON: JavaScript objects, more human-friendly text syntax
  - older than XML, probably better choice than XML in retrospect
- ▶ YAML: line/indentation-based

## Breakout Question

What is the difference between JSON, YAML, XML?

# Typical Data Representation Languages

XML, JSON, YAML essentially the same

except for concrete syntax

## Atomic Types

- ▶ integer, float, boolean, string
- ▶ need to read fine-print on precision

## (Not Very) Complex Types

- ▶ heterogeneous lists
- ▶ records

a single type for all lists

a single type for all records



## Example: JSON

JSON:

```
{  
  "individual" : "FlorianRabe",  
  "age" : 40,  
  "concepts" : ["instructor", "male"],  
  "teach" : [  
    {"name" : "Wuv", "credits" : 7.5},  
    {"name" : "KRMT", "credits" : 5}  
  ]  
}
```

Weirdnesses:

- ▶ atomic/list/record = basic/array/object
- ▶ record field names are arbitrary strings, must be quoted
- ▶ records use : instead of =

## Example: YAML

inline syntax: same as JSON but without quoted field names

alternative: indentation-sensitive syntax

```
individual : "FlorianRabe"  
age : 40  
concepts :  
  - "instructor"  
  - "male"  
teach :  
  - name : "WuV"  
    credits : 7.5  
  - name : "KRMT"      credits : 5
```

Weirdnesses:

- ▶ atomic/list/record = scalar/collection/structure
- ▶ records use : instead of =

## Example: XML

Weird structure but very similar

- ▶ elements both record (= attributes) and list (= children)
- ▶ elements carry name of type (= tag)

```
<Person individual="Florian Rabe" age="40">  
  <concepts>  
    <Concept>instructor </Concept>  
    <Concept>male</Concept>  
  </concepts>  
  <teach>  
    <Course name="WuV" credits="7.5" />  
    <Course name="KRMT" credits="5" />  
  </teach>  
</Person>
```

- ▶ Good: Person, Course, Concept give type of object  
easier to decode
- ▶ Bad: value of record field must be string  
concepts cannot be given in attribute  
integers, Booleans, whitespace-separated lists coded as strings

## A Basic Data Representation Language

Let BDL be given by

Data

$D ::=$	<i>int</i>	64 bit integers
	<i>float</i>	IEEE double precision floating point
	<i>string</i>	Unicode strings
	<i>bool</i>	true or false
	$D^*$	lists
	$(ID = D)^*$	records

Exercise 4: Implement BDL using JSON as concrete syntax

- ▶ data structures
- ▶ serializer to JSON strings
- ▶ parser from JSON strings

you may/should use an existing JSON library

# Structure Sharing

## Problem

- ▶ Large objects are often redundant specially when machine-produced
- ▶ Same string, URL, mathematical objects occurs in multiple places
- ▶ Handled in memory via pointers
- ▶ Size of serialization can explode

## Solution 1: in language

- ▶ Add definitions to language common part of most languages anyway
- ▶ Users should introduce name whenever object used twice
- ▶ Problem: only works
  - ▶ duplication anticipated
  - ▶ users introduced definition
  - ▶ duplication within same context

structure-sharing most powerful if across contexts

## Structure Sharing (2)

### Solution 2: in tool

- ▶ Use factory methods instead of constructors
- ▶ Keep huge hash set of all objects
- ▶ Reuse existing object if already in hash set
- ▶ Advantages
  - ▶ allows optimization
  - ▶ transparent to users
- ▶ Problem: only works
  - ▶ for immutable data structures
  - ▶ if no occurrence-specific metadata e.g., source reference

### In data representation language

- ▶ Allow any subobject to carry identifier
- ▶ Allow identifier references as subobjects
  - allows preserving structure-sharing in serialization

supported by XML, YAML

# Primitive Types and Encoding Data

## Codecs