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

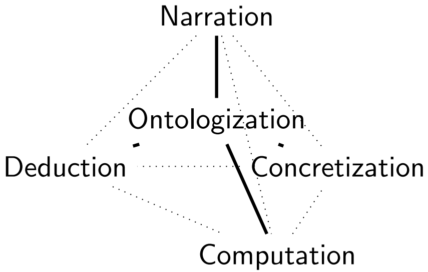
Aspects of Knowledge

- 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

$\boxed{\text{Numbers}}$

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

$\boxed{\text{Formulas}}$

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

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 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 $P, [\![T]\!]$

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?

$\boxed{\text{Declarations}}$

$D ::= \textbf{individual}\ \mathtt{ID} : C$	typed atomic individual
$\mid\ \textbf{concept}\ \mathtt{ID}$	atomic concept
$\mid\ \textbf{relation}\ \mathtt{ID} \subseteq C \times C$	typed atomic relation
$\mid\ \textbf{property}\ \mathtt{ID} \subseteq C \times T$	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 $\to$ 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 $\to$ 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	flexible
	unique type inference	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 a1 with ... with am {
  c1 : T1
  ⋮
  cn : Tn
}
```

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 and Typed Ontologies

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 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 reparse
- 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
  - 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
- physical quantities ($1m$, $1in$, 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 $2.31E+1$

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

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 (1*m*, 1*in*, 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 starndardize

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 $a(A)|b(B)$
  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

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

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

- structural subtyping (= horizontal subtyping)

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

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

A Basic Language for Typed Data

Let BDL be given by

$\boxed{\text{Types}}$

$T ::= \mathit{int} \mid \mathit{float} \mid \mathit{string} \mid \mathit{bool}$	base types
$\mid\ \texttt{list}\ T$	homogeneous lists
$\mid\ (\texttt{ID} : T)^*$	record types
$\mid\ \ldots$	additional types

$\boxed{\text{Data}}$

$D ::= (64\ \mathit{bit\ integers})$	
$\mid\ (\mathit{IEEE\ double})$	
$\mid\ ''(\mathit{Unicode\ strings})''$	
$\mid\ \texttt{true} \mid \texttt{false}$	
$\mid\ D^*$	lists
$\mid\ (\texttt{ID} = D)^*$	records
$\mid\ \ldots$	constructors for additional types

BDL Extended with Named ADTs

$V ::= D^*$		Vocabularies
$D ::= \textbf{adt}\ t\ \{\texttt{ID} : T^*\}$		ADT definitions
$\quad\mid\ \textbf{datum}\ d : T = D$		data definitions

$\boxed{\text{Types}}$

$T ::= \ldots$	as before
$\quad\mid\ t$	reference to a named ADT

$\boxed{\text{Data}}$

$D ::= \ldots$	as before
$\quad\mid\ d$	reference to a named datum
$\quad\mid\ t\{(\texttt{ID} = D)^*\}$	ADT elements

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 — a single type for all lists
- records — a single type for all records

Example: JSON

JSON:

$$
\begin{array}{l}
\{ \\
\quad "individual" : "FlorianRabe", \\
\quad "age" : 40, \\
\quad "concepts" : ["instructor", "male"], \\
\quad "teach" : [\\
\quad\quad \{"name" : "Wuv", credits : 7.5\}, \\
\quad\quad \{"name" : "KRMT", credits : 5\} \\
\quad] \\
\}
\end{array}
$$

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

$$
\begin{array}{l}
individual : "FlorianRabe" \\
age : 40 \\
concepts : \\
\quad - "instructor" \\
\quad - "male" \\
teach : \\
\quad - name : "WuV" \\
\quad credits : 7.5 \\
\quad - name : "KRMT" \qquad credits : 5
\end{array}
$$

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

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 if
  - 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 if
  - 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

General Definition

Throughout this section, we fix a data representation language L.

L-words called codes

Given a data type T, a codec for T consists

- coding function: $c : T \to L$
- partial decoding function: $d : L \to^? T$
- such that

$$d(c(x)) = x$$

Codec Operators

Given a data type operator T taking n type arguments,
a codec operator C for T

- takes n codecs C_i for T_i
- returns a codec $C(C_1, \ldots, C_n)$ for $T(T_1, \ldots, T_n)$

Exercise 4

We fix strings as the data representation language L.

Then,

1. Jointly specify
   - additional BDL types and constructors for univis-specific data
   - codecs and codec operators for all types resp. type operators
2. Individually, in any programming language, implement
   - data structures for BDL
   - string codecs (operators) for all BDL base types (operators)
3. Use your codecs to exchange example data with your fellow students, who used different implementations and different programming languages.

Codecs for Base Types

We define codecs for the base types using strings as the data representation language L.

Easy cases:

- StandardFloat: as specified in IEEE floating point standard
- StandardString: as themselves, quoted
- StandardBool: as *true* or *false*
- StandardInt (64-bit): decimal digit-sequences as usual

Breakout Question

How to encode unlimited precision integers?

Codecs for Unlimited Precision Integers

Encode $z \in \mathbb{Z}$

- L is strings: decimal digit sequence as usual
- L is JSON:
  - IntAsInt: decimal digit sequence as usual

    JSON does not specify precision
    but target systems may get in trouble
  - IntAsString: string containing decimal digit sequence

    safe but awkward
  - IntAsDecList: list of decimal digits safe but awkward
  - IntAsList1: as list of digits for base 2^{64}

    OK, but we can do better
  - IntAsList2: as list of
    - integer for the number of digits, sign indicate sign of z
    - list of digits of $|z|$ for base 2^{64}

    Question: Why is this smart?

Codecs for Unlimited Precision Integers

Encode $z \in \mathbb{Z}$

- L is strings: decimal digit sequence as usual
- L is JSON:
  - IntAsInt: decimal digit sequence as usual

    JSON does not specify precision
    but target systems may get in trouble
  - IntAsString: string containing decimal digit sequence

    safe but awkward
  - IntAsDecList: list of decimal digits — safe but awkward
  - IntAsList1: as list of digits for base 2^{64}

    OK, but we can do better
  - IntAsList2: as list of
    - integer for the number of digits, sign indicate sign of z
    - list of digits of $|z|$ for base 2^{64}

    Question: Why is this smart?
    Can use lexicographic ordering for size comparison

Codecs for Lists

Encode list x of elements of type T

- L is strings: e.g., comma-separated list of T-encoded elements of x
- L is JSON:
  - ListAsString: like for strings above
  - ListAsArray: lists JSON array of T-encoded elements of x

Additional Types

Examples: semester

Extend BDL:

$\boxed{\text{Types}}$

T ::= *Sem* semester

$\boxed{\text{Data}}$

D ::= $sem(int, bool)$ i.e., year + summer?

Define standard codec:

$$sem(y, true) \rightsquigarrow "SSY"$$

$$sem(y, false) \rightsquigarrow "WSY"$$

where Y is encoding of y

Additional Types (2)

Examples: timestamps

Extend BDL:

$\boxed{\text{Types}}$

$T ::= \textit{timestamp}$

$\boxed{\text{Data}}$

$D ::=$ (productions for dates, times, etc.)

Standard codec: encode as string as defined in ISO 8601

Primitive Types and Encoding Data

Data Interchange

Design

1. Specify type system, e.g., BDL
   - types
   - constructors
   - operations

   can be done in appropriate type theory

2. Pick data representation language L
3. Specify codecs for type system and L
   - at least one codec per base type
   - at least one codec operator per type operator

   on paper

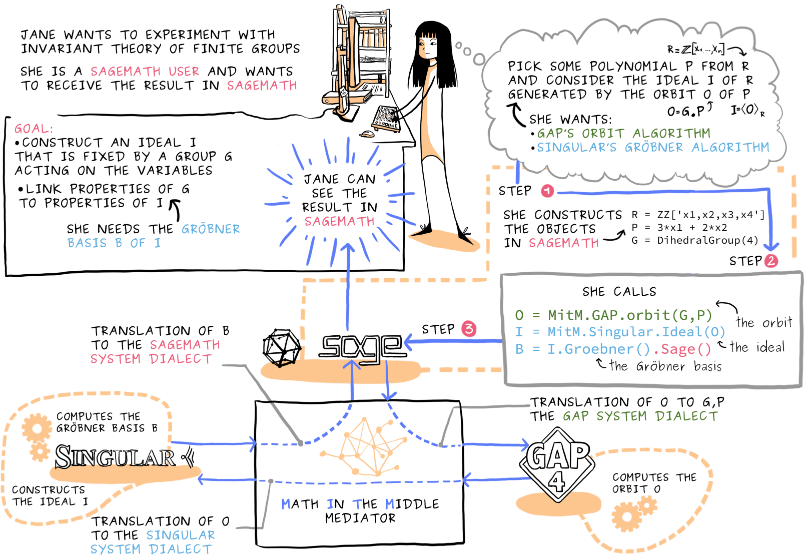
4. Every system implements
   - type system (as they like) typically aspect-specific constraints
   - codecs as specified
   - function mapping types to codecs
5. Systems can exchange data by encoding-decoding

   type-safe because codecs chosen by type

Example

Implementation in Scala part of course resources

Example Application: OpenDreamKit research project



Integrating BOL and BDL

OWL-near option

- use BDL to define the primitive types of BOL
- use those as types of BOL properties
- Curry-typing throughout

easy: just merge the grammars

SQL-near option

- use BDL to define the primitive types of BOL
- also add ADTs
- Church typing more prominent

open question: ADTs in addition to or instead of BOL concepts

We assume the latter for now without spelling out the details.

BDL-Mediated Interoperability

Idea

- define data types in BDL   or similar typed ontology language
- use ADTs
- generate corresponding
  - class definitions for programming languages PL
    one class per ADT
  - table definitions in SQL   one table per ADT
- use codecs to convert automatically when interchanging data between PL and SQL

Open research problem

no shiny solution yet that can be presented in lectures

Codecs in ADT Definitions

SQL table schema = list of fields where field is

- name
- type — only types of database supported

BDL semantic table schema = list of fields where field is

- name
- type T of type system — independent of database
- codec for T using primitive objects of database as codes
  see research paper https://kwarc.info/people/frabe/Research/WKR_

Codec could be chosen automatically, but we want to allow multiple users a choice of codecs for the same type.

Example

Ontology based on BDL-ADTs with additional codec information:

```
schema Instructor
  name:     string       codec StandardString
  age:      int          codec StandardInt
  courses:  list Course  codec CommaSeparatedList CourseAsName
schema Course
  name:     string    codec StandardString
  credits:  float     codec StandardFloat
  semester: Semester  codec SemesterAsString
```

Generated SQL tables:

```
CREATE TABLE Instructor
  (name string, age int, courses string)
CREATE TABLE Course
  (name string, credits float, semester string)
```

Open Problem: Non-Compositionality

Sometimes optimal translation is non-compositional

- example translate *list*-type in ADT to comma-separated string in DB
- better break up *list* B fields in type A into separate table with columns for A and B

Similar problems

- a pair type in an ADT could be translated to two separate columns
- an option type in an ADT could translated to a normal column using SQL's NULL value

Open Problem: Querying

- General setup
  - write SQL-style queries using at the BDL level
  - automatically encode values when writing to database from PL
  - automatically decode query results when reading from DB
- But queries using semantic operations cannot always be translated to DB
  - operation $IsSummer : Semester \to bool$ in BDL
  - query SELECT $*$ FROM course WHERE$IsSummer$(semester)
  - how to map $IsSummer$ to SQL?
- Ontology operations need commuting operations on codes
  - given $f : A \to B$ in BDL, codecs C, D for A and B
  - SQL function f' commutes with f iff

$$B.decode(f'(C.encode\, a)) = f(a)$$

for all $a : A$

Exercise 5, part 1

We build on the implementation of BDL and codecs from Exercise 4 and on the database schemas from Exercise 3.

1. Extend the implementation to BDL+ADT (see Slide 101).
2. Extend
   - codecs and codec operators with identifiers $I ::= (\mathit{strings})$
   - ADT fields with codec expressions $c ::= I \mid I(c_1 \ldots, c_n)$

   and write a function that maps c to the corresponding codec.

Exercise 5, part 2

3. Write a function that takes a vocabulary (= a list of ADT definitions with codec expressions) and generates an SQL schema for it. Use the type returned by the codec as the database type.
4. Write a function that takes an element d of an ADT and generates the SQL (or CSV) representation of d with all field values encoded by the corresponding codec.
5. Write a function that takes an ADT name and a SQL or CSV object and applies decoding to build the corresponding ADT element.
6. Test this by
   - writing some of your univis table schemas as ADTs and some example values as ADT elements,
   - exchanging these with a database and/or via CSV with fellow students' implementations.

Querying

Querying

Overview

General Ideas

- Recall
  - syntax = context-free grammar
  - semantics = translation to another language
- Example: BOL translated to SQL, SFOL, Scala, English
- Querying = use semantics to answer questions about syntax

Note:

- Not the standard definition of querying
- Design of a new Tetrapod-level notion of querying

  ongoing research
- Subsumes concepts of different names from the various aspects

Propositions

syntax with propositions =
designated non-terminals for propositions

Examples:

aspect	basic propositions
ontology language	assertions, concept equality/subsumption
programming language	equality for some types
database language	equality for base types
logic	equality for all types
natural language	sentences

Aspects vary critically in how propositions can be formed

- any program in computation
- quantifiers in deductions undecidable
- *IN* in databases

Propositions as Queries

Propositions allow defining queries

	Query	Result
deduction	proposition	yes/no
concretization	proposition with free variables	true ground instances
computation	term	value
narration	question	answer

Semantics of Propositions

syntax with propositions =
designated non-terminals for propositions

needed to ask queries

semantics with theorems =
designates some propositions as theorems or contradictions

needed to answer queries

Note:

- A propositions may be neither theorem nor contradiction.
- We say that language has negation if:
  F theorem iff $\neg F$ contradiction and vice versa.

We write $\vdash F$ if F is theorem.

Querying

Deductive Queries

Definition

We assume

- a semantics $[\![-]\!]$ from I to L
- I has propositions
- there is an operation `True` that maps translations of I-propositions to L-propositions
- L has semantics with propositions

We define

- a deductive query is an I-proposition p
- the result is
  - yes if $\texttt{True}[\![p]\!]$ is a theorem of L
  - no if $\texttt{True}[\![p]\!]$ is a contradiction in L

Breakout question

What can go wrong?

Problem: Inconsistency

In general, (in)consistency of semantics

- Some propositions may be both a theorem and a contradiction.
- In that case, queries do not have a result.

In practice, however:

- If this holds for some propositions, it typically holds for all of them.
- In that, we call L inconsistent.
- We usually assume L to be consistent.

Problem: Incompleteness

In general, (in)completeness of semantics

- We cannot in general assume that every proposition in L is either a theorem or a contradiction.
- In fact, most propositions are neither.
- So, queries do not necessarily have a result.
- We speak of incompleteness.

  Note: not the same as the usual (in)completeness of logic

In practice, however:

- It may be that L is complete for all propositions in the image of $\mathtt{True}[\![-]\!]$.
- This is the case if I is simple enough

  typical for ontology languages

Problem: Undecidability

In general, (un)decidability of semantics:

- We cannot in general assume that it is decidable whether a proposition in L is a theorem or a contradiction.
- In fact, it usually isn't.
- So, we cannot necessarily compute the result of a query.
- However: If we have completeness, decidability is likely.

  run provers for F and $\neg F$ in parallel

In practice, however:

- It may be that L is decidable for all propositions in the image of $\texttt{True}[\![-]\!]$.
- This is the case if I is simple enough

  typical for ontology languages

Problem: Inefficiency

In general, (in)efficiency of semantics:

- Answering deductive queries is very slow.
- Even if we are complete and decidable.

In practice, however:

- Decision procedures for the image of $\mathtt{True}[\![-]\!]$ may be quite efficient.
- Dedicated implementations for specific fragments.
- This is the case if I is simple enough

typical for ontology languages

Querying

Contexts and Free Variables

Concepts

Recall the analogy between grammars and typing:

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

We will now add contexts and substitutions.

Contexts

Given a context-free language l, we define:

- A context Γ is of the form $x_1 : N_1, \ldots, x_n : N_n$ where the
  - x_i are names
  - N_i are non-terminals

  We write this as $\vdash_l \Gamma$.
- A substitution for Γ is of the form $x_1 := w_1, \ldots, x_n := w_n$ where the
  - x_i are as in Γ
  - w_i derived from the corresponding N_i

  We write this as $\vdash_l \gamma : \Gamma$.
- An expression in context Γ of type N is a word w derived from N using additionally the productions $N_i ::= x_i$.
  We write this as $\Gamma \vdash_l w : N$.
- Given $\Gamma \vdash w : N$ and $\vdash \gamma : \Gamma$ as above, the substitution of γ in w is obtained by replacing every x_i in w with w_i. We write this as $w[\gamma]$.

Contexts under Compositional Translation

Consider a compositional semantics $[\![-]\!]$ from I to L between context-free languages.

- Every $\vdash_I w : N$ is translated to some $\vdash_L [\![w]\!] : N'$ for some N'.
- Compositionality ensures that N' is the same for all w derived from N.
- We write $[\![N]\!]$ for that N'.
- Then we have

$$\vdash_I w : N \qquad \text{implies} \qquad \vdash_L [\![w]\!] : [\![N]\!]$$

Now we translate contexts, substitutions, and variables as well:

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

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

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

Then we have

$$\Gamma \vdash_I w : N \qquad \text{implies} \qquad [\![\Gamma]\!] \vdash_L [\![w]\!] : [\![N]\!]$$

Substitution under Compositional Translation

From previous slide:

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

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

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

$$\Gamma \vdash_I w : N \qquad \text{implies} \qquad [\![\Gamma]\!] \vdash_L [\![w]\!] : [\![N]\!]$$

We can now restate the substitution theorem as follows:

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

Querying

Concretized Queries

Definition

We assume

- as for deductive queries
- semantics must be compositional

We define

- a concretized query is an l-proposition p in context Γ
- a single result is a
  - a substitution $\vdash_l \gamma : \Gamma$
  - such that $\vdash_L \mathtt{True}[\![p[\gamma]]\!]$
- the result set is the set of all results

Example

1. BOL ontology:

$$\begin{array}{l} \textit{concept male},\ \textit{concept person},\ \textit{axiom male} \sqsubseteq \textit{person}, \\ \textit{individual FlorianRabe},\ \textit{assertion FlorianRabe isa male} \end{array}$$

2. Query $x : \textit{individual} \vdash_{BOL} x$ isa $\textit{person}$
3. Translation to SFOL: $x : \iota \vdash_{SFOL} \textit{person}(x)$
4. SFOL calculus yields theorem $\vdash_{SFOL} \textit{person}(\textit{FlorianRabe})$
5. Query result $[\![\gamma]\!] = x := \textit{FlorianRabe}$
6. Back-translating the result to BOL: $\gamma = x := \textit{FlorianRabe}$

back translation is deceptively simple:
translates SFOL-constant to BOL-individual of same name

Breakout question

What can go wrong?

Problem: Open World

In general, semantics uses open world:

- open world: result contains all known results

  same query might yield more results later

- closed world: result set contains all results

  always relative to concrete database for L

In practice, however,

- system explicitly assumes closed world typical for databases
- users aware of open world and able to process results correctly

Problem: Infinity of Results

In general, there may be infinitely many results:

- e.g., query for all x such that $\vdash x$,

In practice, however,

- systems pull results from finite database e.g., SQL, SPARQL
- systems enumerate results, require user to explicitly ask for more e.g., Prolog

Problem: Back-Translation of Results

In general, $[\![-]\!]$ may be non-trivial to invert

- easy to obtain $[\![p]\!]$ in context $[\![\Gamma]\!]$ just apply semantics
- possible to find substitutions

$$\vdash_L \delta : [\![\Gamma]\!] \qquad \text{where} \qquad [\![\Gamma]\!] \vdash_L \mathtt{True}[\![p]\!][\delta]$$

easiest case: just look them up in database

- but how to translate δ to I-substitutions γ with

$$\vdash_I \gamma : \Gamma \qquad \text{where} \qquad [\![\Gamma]\!] \vdash_L \mathtt{True}[\![p[\gamma]]\!]$$

substitution theorem: pick such that $[\![\gamma]\!] = \delta$
the more $[\![-]\!]$ does, the harder to invert

In practice, however:

- often only interested in concrete substitutions
- translation of concrete data usually identity

But: practice restricted to what works even if more is needed

Querying

Computational Queries

Definition

We assume

- the same as for deductive queries
- semantics has equality/equivalence $\doteq$

We define

- a computational query is an l-expression e
- the result is an l-expression e' so that $\vdash_L [\![e]\!] \doteq [\![e']\!]$

intuition: e' is the result of evaluating e

If semantics is compositional, e may contain free variables

evaluate to themselves

Problem: Back-Translation of Results

In general, $[\![-]\!]$ may be non-trivial to invert

- easy to obtain $E := [\![e]\!]$
- possible to find E' with $\vdash_L E' \doteq E$ by working in the semantics
- non-obvious how to obtain e' such that $[\![e']\!] = E'$

In practice, however:

- evaluation meant to simplify, i.e., only useful if E' very simple
- simple E' usually in the image of $[\![-]\!]$
- typical case: E' is concrete data and $e' = E'$ called a value

Problem: Non-Termination

In general, computation of E' from E might not terminate

- while-loops
- recursion
- $(\lambda x.x\,x)\,(\lambda x.x\,x)$ with β-rule
- simplification rule $x \cdot y \rightsquigarrow y \cdot x$

  similar: distributivity, associativity

In practice, however:

- image of $[\![-]\!]$ part of terminating fragment

But: if I is Turing-complete or undecidable, general termination not possible

Problem: Lack of Confluence

In general, there may be multiple E' that are simpler than E

- there may be multiple rules that apply to E
- e.g., $f(g(x))$
  - call-by-value: first simplify $g(x) \rightsquigarrow y$, then $f(y) \rightsquigarrow z$
  - call-by-name: first plug $g(x)$ into definition of f, then simplify
- Normal vs. canonical form
  - normal: $\vdash_L E \doteq E'$
  - canonical: normal and $\vdash_L E_1 \doteq E_2$ iff $E'_1 = E'_2$

equivalent expressions have identical evaluation
allows deciding equality

In practice, however:

- image of $[\![-]\!]$ part of confluent fragment
- typical: evaluation to a value is canonical form

works for BDL-types but not for, e.g., function types

Querying

Narrative Queries

Definition

We assume

- semantics into natural language

We define

- a narrative query is an *L*-question about some *l*-expressions
- the result is the answer to the question

Problem: Unimplementable

very expressive = very difficult to implement

- Natural language understanding
  - no implementable syntax of natural language
    needs restriction to controlled natural language
  - specifying semantics hard even when controlled
- Knowledge base for question answering needed
  - very large — must include all common sense
  - might be inconsistent — common sense often is
  - finding answers still very hard

In practice, however:

- accept unreliability — attach probability measures to answers
- implement special cases
  e.g., lookup in databases like Wikidata
- search knowledge base for related statements — Google, Watson

Querying

Absolute Semantics

Motivation

So far

- relative semantics = semantics by translation

  relative to semantics of target language
- obvious problem: what is semantics of target language?
- our answer so far: it's in our heads, in reality
- obvious problem: does not work well with computers

Also need semantics that is

- absolute — no reference to other languages
- machine-actionable

  requires an imperfect model of application domain

Imperfect Modeling

- Machine-actionable requires reduction to finite set of rules

  whatever a rule is
- Does not work for most domains
  - practical argument: any practically interesting system has too many rules

    cf. physics, e.g., three-body problem already chaotic
  - theoretical argument: no language can fully model itself

    cf. Gödel's incompleteness theorems
- Imperfect models
  - language focuses on some aspects of domain
  - vocabulary approximates domain

Big question: what aspects to focus on?

Querying as a Guide for Semantics

Idea

- Very difficult to choose aspects for imperfect models
- Turn problem around
  - ask what the practical purpose of the semantics could be
  - then choose aspects that allow realizing that purpose

Querying as the Purpose

- Before: identified different kinds of querying
  focussing on different aspects of knowledge
- Now: each induces a kind of absolute semantics

Problems

Next

- four kinds of absolute semantics — one per aspect
- Each motivated by one kind of querying
- Each defines the aspect

  e.g., a logic is a language with deductive semantics

Relation to previous slides

- before: querying via relative semantics
- just the special case where target language has corresponding absolute semantics

  e.g., deductive querying possible given deductive semantics
  no matter if relative or absolute
- conceptually, absolute semantics comes first, but easier to understand after querying
- discussed problems apply to absolute semantics accordingly

Deductive Absolute Semantics

Definition

- A set of rules that determines which propositions are theorems
  called a calculus
- Languages called logics
- Systems called theorem provers

Examples

- Natural deduction for first-order logic
- Axiomatic set theory

Most logics have multiple semantics

- Proof theory: absolute semantics — to be self-contained
- Model theory: relative semantics by translation to set theory
  common reference language
- Logic translation: relative by translation into standard logics, e.g., SFOL

Concrete Absolute Semantics

Definition

- A set of rules that determines true ground instances of a proposition
- Languages typically called query languages

  inspired our, more general use of the word
- Systems called triple stores, databases

Examples

- SQL for Church-typed ontologies with ADTs
- SPARQL for Curry-typed ontologies
- Prolog for first-order logic

Computational Absolute Semantics

Definition

- A set of rules that determines evaluation of expressions
- Languages typically called programming languages
- Systems called interpreters

Examples

- Any interpreted language — Python, bash, ...
- Machine language — interpretation rules built into microchips

Many languages have multiple semantics

- Interpreter: absolute semantics — for flexible experimentation
- Compiler: relative semantics — for fast execution