Knowledge Representation and Processing

Florian Rabe (for a course given with Michael Kohlhase)

Computer Science, University Erlangen-Nürnberg, Germany

2022

Administrative Information

Format

Zoom

- lectures and exercises in person — if we have a room
- zoom for now
- interaction strongly encouraged — We don't want to lecture — we want to have a conversation during which you learn
- make use of 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 second time
- First time was a little bit of an experiment
  polishing and revising the materials this year
- To 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
- oral, 30 minutes
- exercises indirectly graded through conversation during exam

Exam-relevant

- anything mentioned in notes
- anything discussed in lectures
- anything done in exercises

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
- systematic write-up, not necessarily in lecture order

Slides

- not comprehensive
- used as visual aid, conversation starters

Communication

Open for questions

- open door policy in our offices
- always room for questions during lectures
- for personal questions, contact me during/after lecture
- forum at studon

Materials

- notes and slides are at: `https://github.com/florian-rabe/Teaching/tree/master/WuV`
- currently last year's version, will change throughout the semester
  you can read ahead, but maybe don't print everything right away
- 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
  implemented 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

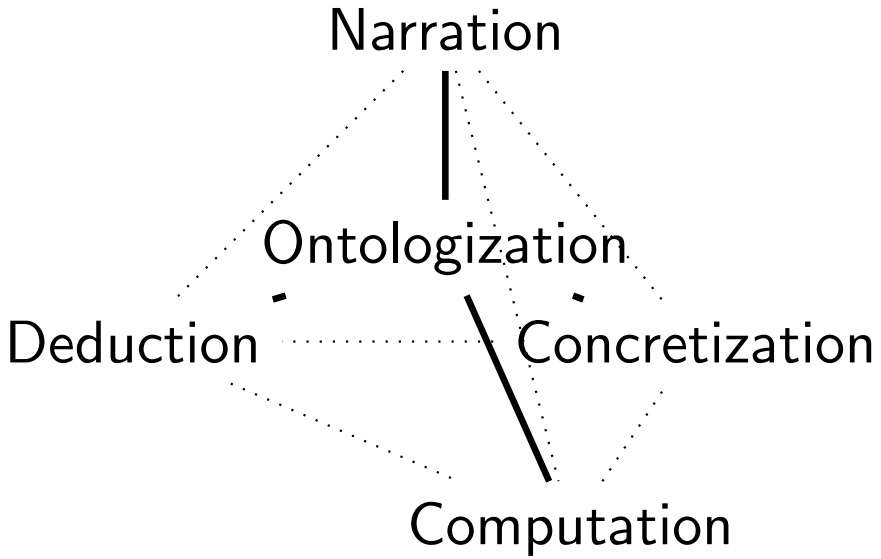
Aspects of Knowledge

- 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	queriability	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

- shared characteristics
- general methods

Aspects-specific parts

- one part 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 campo/studnon-style KRP system for a university
- lots of heterogeneous knowledge
  - course and program descriptions
  - legal texts
  - websites
  - grade tables
  - code to generate diplomas
- build a functional system applying the lessons of the course

we'll see how far we get — priority is learning

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

Exercise 1

As a team, build an ontology for a university.

Using git, OWL, and WebProtege are good ways to start.

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

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

Summary

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

Exercise 2-3

Extend your ontology with an ABox and axioms. Export it in RDF format to a triple store like Virtuoso and run a concrete query in SPARQL. Export it in OWL format to a reasoner like FaCT++ and run a deductive query.

Language Layers

Layers of Language Design

Layer	Specified by	Implemented by
Syntax		
Context-Free	grammar	AST+parser+printer
Context-Sensitive	inference system	type checker
Semantics	inference system, interpretation, or translation	
Pragmatics	human preferences	human judgment

KRP = syntax + semantics

Layered Processing

Data is processed in phases

1. data representation format, e.g., string, JSON, XML, binary
2. parsed — well-formed syntax tree
3. type-checked by traversal of the syntax tree — well-typed syntax tree
4. computation by traversal of well-typed AST — semantics

Possible Errors

Layer	Error
CFS	not derivable from grammar
CSS	symbols not used as declared, other conditions
Sem.	ambiguous/undefined semantics
Pragmatics	not useful

Typical Errors by Layer

In a programming language:

Layer	Expression	Issue	Explanation
CFS	1/	syntax error	argument missing
CSS	1/"2"	typing error	wrong type
Sem.	1/0	run-time error	undefined semantics
Pragm.	1/1	code review	unnecessarily complex

Typical Errors by Layer

In a logic:

Layer	Expression	Issue	Explanation
CFS	$\forall x$	not well-formed	body missing
CSS	$\forall x.P(y)$	not well-typed	y not declared
Sem.	the $x \in \mathbb{N}$ with $x < 0$	not well-defined	no such x exists
Pragm.	$\exists x.x \neq x$	not useful	no model exists

Context-Free Grammars

The Chomsky Hierarchy

- CH-0, regular grammars:
  - equivalent to regular expressions and finite automata
  - not used much as grammars
- CH-1, context-free grammars (CFGs) our focus
- CH-2, context-sensitive grammars
  - important as languages, but awkward as grammars
  - instead: type system determines subset of context-free language
- CH-3, unrestricted grammars
  - Turing-complete, theoretically important
  - not used much as grammars

Definitions

- An alphabet is a set of symbols.
- A word is a list of symbols from the alphabet.
- A production is pair of words.
  - A production is written $lhs ::= rhs$.
  - Multiple productions for the same left-hand side are abbreviated $lhs ::= rhs_1 \mid \ldots \mid rhs_n$.
  - Right-hand side may also use regular expressions like * for repetition and [] for optional parts.
- A CFG is a set of productions where *lhs* is a single symbol.
  - If there is a production $N ::= rhs$, N is called non-terminal, otherwise terminal.
  - If a word contains non-terminal symbols, it is called non-terminal, otherwise terminal.
- A syntax tree is a tree with nodes are labeled with productions $N ::= rhs$ where the non-terminals in rhs are exactly the lhs's of the children.
- The word produced by a syntax tree is read off by exhaustively replacing every lhs with the respective rhs.

Example: Syntax of Arithmetic Language

Numbers

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

Formulas

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

Implementing CFGs via Inductive Data Types

Correspondence

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

Classes of Languages

Functional languages:

- pure: ML, Haskell
- with OO: F#, Scala

inductive types are primitive

OO-languages:

- C#, Java, C++

inductive types simulated via classes

Untyped languages:

- Python, Javascript

inductive types simulated ad hoc

Implementing the Example

Done interactively. See the examples in the repository. See also the notes.

Context-Sensitive Syntax

Vocabularies and Declarations

Generic structure of a context-sensitive language

- a vocabulary is a list of declarations
  - named: type/function/predicate symbol etc.
  - unnamed: axioms etc.
  - structural: inclusion/import, datatype definitions
- named declarations introduce atomic objects of different kinds
- for each kind, a non-terminal for complex expressions of that kind
- references to names introduced by declarations are base cases of expressions

Example: Typed Expressions

Vocabularies

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

Declarations

$Decl ::= id : Type^* \to Type$ — typed function symbols

$\mid \; id : Type^* \to FORM$ — typed predicate symbols

Types

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

Expressions

$Expr ::= 0 \mid 1 \mid Expr + Expr \mid Expr * Expr$ — as before

$\mid \; id(Expr^*)$ — application of a function symbol

Formulas

$Form ::= Expr \doteq Expr \mid Expr \leq Expr$ — as before

$\mid \; id(Expr^*)$ — application of a predicate symbol

Example: Vocabularies and Expressions

Example vocabulary V containing the following declarations:

- $fib : Nat \to Nat$
- $length : String \to Nat$
- $mod : Nat\ Nat \to Nat$
- $prime : Nat \to FORM$

Example expressions relative to V

- expressions: $fib(0)$, $mod(fib(fib(1)), 1 + 1)$
- formulas: $fib(0) = 0$, $prime(fib(1))$

Primitive vs. Declared

Primitive

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

	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

more expressive declarations $\rightarrow$ fewer primitives needed

paradoxical: more complex language may have simpler grammar

Quasi-Primitive = Declared in standard library

Standard library: a vocabulary *StdLib*

- present in every language — empty vocabulary by default
- one fixed vocabulary
  - implicitly included into every other vocabulary
  - implicitly fixed by any translation between vocabularies

  Combination of advantages
  - from the user's perspective: like a primitive
  - from the theory's/system's perspective: no special treatment

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

Example: Removing Built-in Operations

Grammar without built-in operations

Voc	$::= Decl^*$	list of declarations
$Decl$	$::= id : Type^* \to Type$	typed function symbols
	$\mid id : Type^* \to FORM$	typed predicate symbols
$Type$	$::= Nat \mid String$	base types
$Expr$	$::= id(Expr^*)$	application of a function symbol
$Form$	$::= id(Expr^*)$	application of a predicate symbol

Standard library:

- $0 : Nat$, $1 : Nat$, $sum : Nat\ Nat \to Nat$, $product : Nat\ Nat \to Nat$,
- $equals : Nat\ Nat \to FORM$, $lesseq : Nat\ Nat \to FORM$

Example: Removing more Primitive Operations

If we add type declarations, we can remove *Nat* as well

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

Note: *Type* and *Form* are non-terminals, *TYPE* and *FORM* are not

Add to default vocabulary: $Nat : TYPE$, $String : TYPE$

Context-Sensitivity

A reference to a declared name must respect the way in which it was declared in the vocabulary examples errors relative to V above

- occur in a position where an expression of the right kind is expected example error: $prime(1) = 1$
- be applied to the right number of arguments example error: $prime(1, 1)$
- if a type system is used
  - arguments must have the right types $length(1)$
  - return type must match what is expected $fib(1)$ if a string is expected

Syntax Traversal

Context-free traversal — mutually recursive functions

- one function for each non-terminal/inductive type
- for each such function, one case for each production/constructor
- for each such case, one recursive call for each non-terminal on the rhs/constructor argument
- Examples
  - printer/serializer (return, e.g., string)
  - test if a feature is used (return boolean)
  - find set of used identifiers (return set of names)

Context-sensitive traversal: as above but

- functions take extra argument for vocabulary
- cases for identifier references look up declaration in vocabulary
- Examples: anything that looks up identifier declarations
  - substitution of identifiers with new expressions
  - typical syntax transformations, e.g., in compilers (return same expression kind)
  - semantics by translation

Type Checker

Syntax checker

- context-sensitive traversal where all functions return booleans
- case for vocabulary checks each declaration relative to the preceding vocabulary
- cases for identifier references check correct use of identifier

Type checker: as above but additionally

- functions for typed expression additionally take expected type as argument
- cases for identifiers references
  - check each argument against declared input type
  - compare output to expected type

Exercise 4

Individually, using any programming language, implement the AST for the BOL language. Allow for integers and strings as basic types.

Implement a type-checker for BOL. BOL is untyped, and not much type-checking is needed. But the type-checker must check at least that all identifier references are correct. The main check needed is that all property assertion use values according to the property type.

Exercise 5

As a group, define an XML schema for BOL using the RelaxNG compact syntax. See `https://relaxng.org/compact-tutorial-20030326.html`.

Individually, extend your implementations of BOL with XML importers and exporters relative to that schema. The exporter is a context-free traverser. For the importer, use an XML library for your chosen programming language to parse the input. Then write a context-free traverser over the XML data structures that translates the XML into your BOL data structures.

After importing a vocabulary, type-check it.

Write some example vocabularies for BOL and export-import them to each other.

Extrinsic and Intrinsic Typing

Breakout Question

Is this an improvement over BOL?

Declarations

$D ::=$	**individual** $\texttt{ID} : C$	typed atomic individual
$\mid$	**concept** $\texttt{ID}$	atomic concept
$\mid$	**relation** $\texttt{ID} \subseteq C \times C$	typed atomic relation
$\mid$	**property** $\texttt{ID} \subseteq C \times T$	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, complex type system
  - simplicity: easy to implement/interpret
    e.g., few, carefully chosen productions, types
- semantics: specify, implement, documents
- intended users
  - skill level
  - prior experience with related languages
  - amount of training needed
  - innovation height, differential evaluation against existing languages

Actually, when is a language an improvement? (2)

More criteria:

- long-term plans: re-answer the above question but now
  - maintainability: syntax was changed, everything to be redone
  - backwards compatibility: support for legacy input
  - scalability: expressed knowledge content has reached huge sizes
- support software ecosystem
  - optional tool support: IDEs, debuggers, heuristic checkers, alternative implementations, interpreter/REPL
  - many/large well-crafted vocabularies and package managers to find them
  - integrations with other languages: translations, common run-time platforms, foreign function interface

Church vs. Curry Typing

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

Type Checking

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

Examples: Curry Typing in BOL Semantics

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** for a syntax consists of

- some non-terminals $\mathcal{E}$, whose words are called $\mathcal{E}$-**expressions**,
  coarse, easy to check classification into disjoint sets
- for some (possibly no) intrinsic types $\mathcal{E}$
  - a non-terminal $\mathcal{T}$
  - a typing relation between $\mathcal{E}$-expressions and $\mathcal{T}$-expressions, called the **extrinsic typing** relation for $\mathcal{E}$

  fine-granular subclassification of the expressions

Examples

System	intrinsic types	extrinsic types	extr. typing
pure Church	one per type	none	none
pure Curry	one for all expressions	types T	:
FOL	one per type	none	none
OO	primitive types, Object	classes	`isInstanceOf`
BOL	*Ind*, *Conc*, ...	concepts	`is-a`
set theory	*Set*, *Prop*	sets	$\in$

Breakout Questions

What kind of extrinsic types could we choose?

Abstract vs. Concrete Types

Concrete type: values are

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

product types, enumeration types, collection types

main example: concrete (inductive/algebraic) data types

Abstract type: values are

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

structures, records, classes, aggregation types

main example: abstract data types

Plain vs. Recursive Types

Plain types

- given by some expressions
- can be anonymous
- values given directly

integers, lists of strings, ...

Recursive types

- definition of the type must refer to the type itself
  so type must have a name
- type typically defines other named operations
- values obtained by fixed-point constructions

optional property of concrete and abstract data types

Atomic vs. Complex Types

Atomic type

- given by its name
- values are a set

integers, strings, booleans, ...

Complex types

- arise by applying type symbol to arguments
- separate set of values for each tuple of arguments

two kinds of complex types (next slide)

Type operators vs. Dependent type Families

Both are symbols that take arguments and return a type.

Type operators take only type arguments, e.g.,

- type operator $\times$
- takes two types A, B
- returns type $A \times B$

Dependent types take also value arguments, e.g.,

- 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

Non-Recursive Data Types

Basic Atomic Types

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

Advanced Atomic Types

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

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)

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

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

Concrete Data Types

Motivation

Idea

- describe infinite type in finite way
- exploit inductive structure to catch all values

Name: usually called **inductive** data type, especially when recursive

Examples

Natural numbers Nat given by

- $zero$
- $succ(n)$ for every $n : Nat$

Lists $list\ A$ over type A given by

- empty list nil
- $cons(a, l)$ for every $a : A$ and $l : list\ A$

Arithmetic expressions E given by

- natural number $literal(n)$ for $n : Nat$
- sum $plus(e, f)$ for every $e, f : E$
- product $times(e, f)$ for every $e, f : E$

Rigorous Definition

Let T be the set of types that are known in the current context.

An **inductive data type** is given by

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

Notation:

```
inductive n = c(A,...) | ...

inductive Nat = zero | succ(Nat)
inductive E = Number(Nat) | sum(E,E) | times(E,E)
```

Induction Principle

The values of the inductive type are exactly the ones that can be built by the constructors.

- No junk: the constructors are jointly-surjective
  - no other values but union of their images
  - closed world
- No confusion: the constructors are jointly-injective in the following sense
  - each constructor is an injective function
  - images of the constructor are pairwise disjoint

Inductive definition: define function out of inductive type by giving one case per constructor pattern matching

- total because jointly-surjective
- well-defined because jointly-injective (no overlap between cases)

Special case: No recursion

A concrete data types without recursive constructor arguments are called a **labeled union**. They are isomorphic to the union of the products of the constructor arguments.

Example:

```
inductive Value = Number(Nat) | true | false
inductive Product(A,B) = Pair(A,B)
```

A concrete data types without any constructor arguments is called an **enumeration**. They have exactly one element per constructor.

Example:

```
inductive Boolean = true | false
inductive Color = red | blue | green
```

Generalization: Mutual Induction

Multiple inductive types whose definitions refer to each other.

Example:

```
inductive E = literal(Nat) | sum(E,E) | times(E,E)
inductive F = equal(E,E) | less(E,E)
```

Abstract Data Types

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

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.

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

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

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

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

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

Class Definitions

A class definition in OO:

$$
\begin{array}{l}
\texttt{abstract class } a \texttt{ extends } a_1 \texttt{ with } \ldots \texttt{ with } a_m \texttt{ \{} \\
\quad c_1 : \ T_1 \\
\quad \vdots \\
\quad c_n : \ T_n \\
\texttt{\}}
\end{array}
$$

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)

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

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

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

Motivation

Data Interoperability

Situation

- languages 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 requires interchange language
  serialize as string and reparse
- interchange languages typically untyped XML, JSON, YAML, ...

Solution

- standardize primitive data types
- standardize encoding in interchange languages

Treatment in this Course

BOL syntax and semantics so far

- primitive types 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 practice in common languages

No universal set of types standardized, but some types much more common than others.

Common Primitive Types

Typical (quasi-)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

Less Common 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

Types and their Operations

Types usually come with additional operations:

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, ...

Specifying a Primitive Type

Components

- name
- set of values
- string encoding — injective function from values to strings

Examples

- IEEE floating point numbers
- ISO 8601 for date/time

Exercise 6

As a group, specify additional primitive datatypes for BOL that are helpful the university knowledge ontology.

For each new type, you have to give

- name
- values
- string encoding of each value

The name is added by an additional BOL-production for non-terminal, the values by additional productions for V. The string encodings are used by the import/export functions.

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 interchange languages
2. Data types
3. Codecs

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:

```
{
    "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

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

Data Types

A Basic Language for Typed Data

Let BDL be given by

Types

$T ::=$	*int* | *float* | *string* | *bool*	base types
	$\mid$ `list` T	homogeneous lists
	$\mid$ (`ID` : T)*	record types
	$\mid$ ...	additional types

Data

$D ::=$	(64 *bit integers*)	
	$\mid$ (*IEEE double*)	
	$\mid$ "(*Unicode strings*)"	
	$\mid$ `true` | `false`	
	$\mid$ D^*	lists
	$\mid$ (`ID` $= D$)*	records
	$\mid$ ...	constructors for additional types

BDL Extended with Named ADTs

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

Types

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

Data

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

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 $T(T_1, \ldots, T_n)$

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:

Types	
$T ::= Sem$	semester

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:

Types

$T ::= timestamp$

Data

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

Standard codec: encode as string as defined in ISO 8601

Data Interchange

Design

1. Specify types
   - 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

BDL-Mediated Interoperability

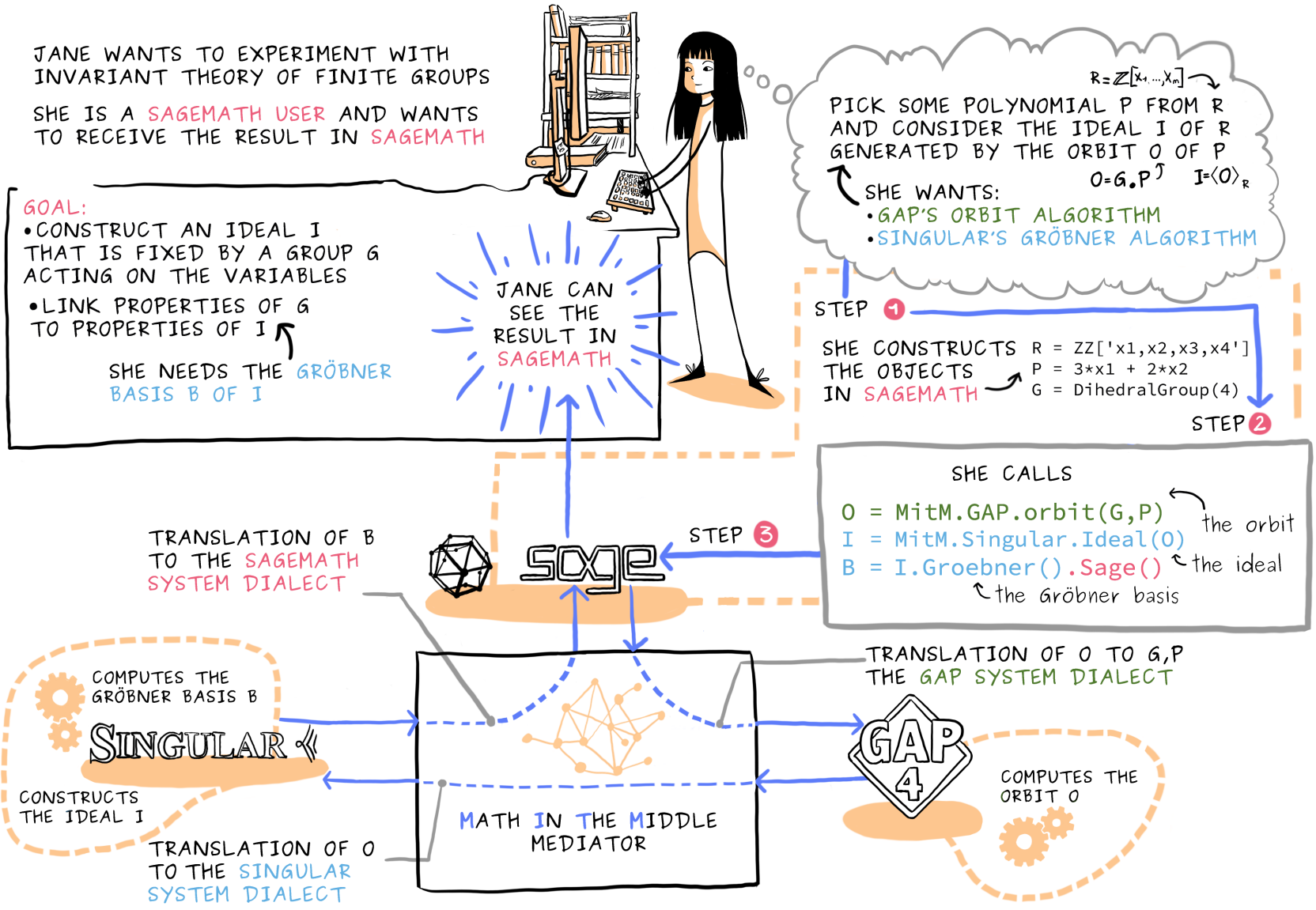
Idea

- 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

Example Application: OpenDreamKit research project



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 $\mathit{IsSummer} : \mathit{Semester} \to \mathit{bool}$ in BDL
  - query SELECT $*$ FROM course WHERE$\mathit{IsSummer}$(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

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

for all $a : A$

Kinds of Semantics

Recall

Recall:

Syntax	Data
Semantics	Knowledge

Representing

- syntax = formal language
  - grammar — context-free part
  - type system — context-sensitive well-formedness
- data = words in the syntax
  - set of vocabularies
  - set of typed expressions for each vocabulary
- semantics = ???
- knowledge = emergent property of having well-formed words with semantics

Relative Semantics by Translation

Components:

- Two syntaxes
  - object-language l e.g., BOL
  - meta-language L e.g., SFOL, Scala, SQL, English
- Semantics of L assumed fixed captures what we already know
- Semantics of l by translation into L

semantics of l relative to to existing semantics of L

Problem: just kicking the can?

Discussion of Semantics by Translation

Advantages

- a few meta-languages yield semantics for many languages
- easy to develop new languages
- good connection between syntax and semantics via compositionality, substitution theorem

Disadvantages

- does not solve the problem once and for all
- impractical without implementation of semantics of meta-language
- meta-languages typically much more expressive than needed for object-languages
- translations can be difficult, error-prone

Also needed: absolute semantics

Absolute vs. Relative Semantics

Absolute = self-contained, no use of meta-language L

Get off the ground

- semantics for a few important meta-languages
  e.g., FOL, assembly language, set theory
- relative semantics for all other languages, e.g.,
  - model theory: logic $\rightarrow$ set theory
  - compilation: Scala $\rightarrow$ JVM $\rightarrow$ assembly

Redundant semantics

- common to give
  - relative and absolute semantics for same syntax
  - multiple relative semantics translations to different aspects
  - sometimes even maybe multiple absolute ones
- Allows understanding syntax from multiple perspectives
- Allows cross-checking show equivalence of two semantics

Example: Recall Syntax of Arithmetic Language

Syntax: represented as formal grammar

Numbers

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

Formulas

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

Implementation as inductive data type

Example: Absolute Semantics

Represented as judgments defined by sets of rules

- unclear what judgments to use
- here: computation $\vdash N \rightsquigarrow N$ and truth $\vdash F$

For numbers n: Rules to normalize numbers into values

$$\frac{}{\vdash N + 0 \rightsquigarrow N} \qquad \frac{}{\vdash N * 0 \rightsquigarrow 0} \qquad \frac{}{\vdash N * 1 \rightsquigarrow N}$$

$$\frac{}{\vdash N * (R + S) \rightsquigarrow N * R + N * S}$$

and their commutative variants as well as

$$\frac{}{\vdash L + (M + N) \rightsquigarrow (L + M) + N}$$

For formulas f: rules to determine true formulas

$$\frac{}{\vdash N \doteq N} \qquad \frac{}{\vdash N \leq N} \qquad \frac{\vdash L \leq M}{\vdash L + N \leq M + N}$$

Example: Absolute Semantics (2)

Checking if an absolute semantics works as intended is hard.

Here: number rules allow

1. eliminating all cases where arguments of $*$ are 0, 1, or $+$; thus, no more $*$
2. eliminating all cases where arguments of $+$ are 0
3. shift brackets of nested $+$ to the left
4. left: 0 or $(\ldots(1+1)\ldots+1)$ — isomorphic to natural numbers

formula rules allow

1. concluding equality if identical normal forms
2. reducing $M+1 \leq N+1$ to $M \leq N$, repeat until $N \leq N$

Example: Relative Semantics

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 ::=	""	empty
|	(Unicode)	character sequence
B ::=	`true`	truth
|	`false`	falsity

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: Relative Semantics

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

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

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

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

Example: Equivalence of Semantics

For formulas

- if $\vdash F$, then $[\![F]\!] = \mathit{true}$ — usually called soundness
- if $[\![F]\!] = \mathit{true}$, then $\vdash F$ — usually called completeness

For numbers

- $\vdash N \rightsquigarrow 0$ iff $[\![N]\!] = ""$
- $\vdash N \rightsquigarrow (\ldots(1+1)\ldots+1)$ iff $[\![N]\!] = "|\ldots|"$

No Perfect Model for Absolute Semantics

- 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 representation of intended semantics required
  focus on some aspect

Big question: what aspects to focus on?

Querying as a Guide

Idea

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

Meta-remark: We do relative semantics first even though absolute semantics conceptually comes first.

Querying as the Purpose

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

Relative Semantics for BOL

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 system I
- semantic language: a formal system L
  different or same aspect as I
- semantic prefix: a vocabulary P in L
  formalizes fundamentals that are needed to represent I-objects
- interpretation: translates every I-vocabulary T to an L-vocabulary P, $[\![T]\!]$

Common Principles

Properties shared by all semantics by translation

not part of formal definition, but best practices

- *l*-declaration translated to *L*-declaration for the same name
- vocabularies 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

Translation vs. Embedding

Translation

- as above, *I* and *L* are at the same level
- *I*-declarations represented as *L*-declarations

also called shallow embedding

Embedding

- *L* is used as meta-language to represent *I*

e.g., *L* is programming language to implement *I*

- *I*-declarations represented as *L*-objects using an inductive type

also called deep embedding

Compositionality

Case for operator $*$ in translation 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

Absolute Semantics for BOL

Judgments

Typing:

$$\Gamma \vdash_V^{BOL} e : E$$

Deduction:

$$\Gamma \vdash_V^{BOL} F$$

Propositions `prop`:

- $C \sqsubseteq D$, $C \equiv D$
- all three kinds of assertions

Notation:

- We drop the superscript ^{BOL} everywhere.
- We drop the subscript $_V$ unless we need to use V.
- We drop the context Γ unless we need to use/change Γ.

Typing

Trivial intrinsic typing (Church) $\vdash e :^{int} E$

- E is a non-terminal
- e an expression derived from E

Refined by extrinsic typing (Curry) $\vdash e :^{ext} E$

- e is an individual, i.e., $\vdash e :^{int} I$
- E is a concept, i.e., $\vdash E :^{int} C$

  where I and C are the non-terminals from the grammar
- e has concept E, i.e., $\vdash e \texttt{ is-a } E$

Propositions as Types

Say also $\vdash p : f$ for proofs p of proposition f

in particular: $x : f$ in contexts to make local assumptions

Notation:

$$\Gamma,\ f \quad \text{instead of} \quad \Gamma,\ p : f$$

sufficient if we only state the rules, not build proofs

Lookup Rules

The main rules that need to access the vocabulary:

$$\frac{f \text{ in } V}{\vdash_V f}$$

for assertions or axioms f

Assumptions in the context are looked up accordingly:

$$\frac{x : f \text{ in } \Gamma}{\Gamma \vdash f}$$

Rules for Subsumption and Equality

Subsumption is an order with respect to equality:

$$\frac{}{\vdash c \sqsubseteq c}$$

$$\frac{\vdash c \sqsubseteq d \qquad \vdash d \sqsubseteq e}{\vdash c \sqsubseteq e}$$

$$\frac{\vdash c \sqsubseteq d \qquad \vdash d \sqsubseteq c}{\vdash c \equiv d}$$

Equal concepts can be substituted for each other:

$$\frac{\vdash c \equiv d \qquad x : C \vdash f(x) : \mathtt{prop} \qquad \vdash f(c)}{\vdash f(d)}$$

This completely defines equality.

Rules relating Instancehood and Subsumption

$$\frac{\vdash i \texttt{ is-a } c \qquad \vdash c \sqsubseteq d}{\vdash i \texttt{ is-a } d}$$

Read:

- if
  - $i \texttt{ is-a } c$
  - $c \sqsubseteq d$
- then $i \texttt{ is-a } d$

$$\frac{x : I, x \texttt{ is-a } c \vdash x \texttt{ is-a } d}{\vdash c \sqsubseteq d}$$

Read:

- if
  - assuming an individual x and $x \texttt{ is-a } c$, then $x \texttt{ is-a } d$
- then $c \sqsubseteq d$

Induction

Consider from before

$$\frac{x : I,\, x \,\texttt{is-a}\, c \vdash x \,\texttt{is-a}\, d}{\vdash c \sqsubseteq d}$$

Question: Do we allow proving the hypothesis by checking for each individual x? induction

Induction

Consider from before

$$\frac{x : I,\, x \,\texttt{is-a}\, c \vdash x \,\texttt{is-a}\, d}{\vdash c \sqsubseteq d}$$

Question: Do we allow proving the hypothesis by checking for each individual x? induction

- Open world: no

Induction

Consider from before

$$\frac{x : I,\ x\ \texttt{is-a}\ c \vdash x\ \texttt{is-a}\ d}{\vdash c \sqsubseteq d}$$

Question: Do we allow proving the hypothesis by checking for each individual x? induction

- Open world: no
- Closed world: yes

$$\frac{\Gamma[x = i] \vdash f[x = i] \ \text{ for every individual } i}{\Gamma, x : I \vdash f(x)}$$

effectively applicable if only finitely many individuals

Rules for Union and Intersection of Concepts

Union as the least upper bound:

$$\frac{}{\vdash c \sqsubseteq c \sqcup d} \qquad \frac{}{\vdash d \sqsubseteq c \sqcup d}$$

$$\frac{\vdash c \sqsubseteq h \qquad \vdash d \sqsubseteq h}{\vdash c \sqcup d \sqsubseteq h}$$

Dually, intersection as the greatest lower bound:

$$\frac{}{\vdash c \sqcap d \sqsubseteq c} \qquad \frac{}{\vdash c \sqcap d \sqsubseteq d}$$

$$\frac{\vdash h \sqsubseteq c \qquad \vdash h \sqsubseteq d}{\vdash h \sqsubseteq c \sqcap d}$$

Rules for Existential and Universal

Easy rules:

- Existential

$$\frac{\vdash i\ r\ j \qquad \vdash j\ \texttt{is-a}\ c}{\vdash i\ \texttt{is-a}\ \exists r.c}$$

- Universal

$$\frac{\vdash i\ \texttt{is-a}\ \forall r.c \qquad \vdash i\ r\ j}{\vdash j\ \texttt{is-a}\ c}$$

Other directions are trickier:

- Existential

$$\frac{\vdash i\ \texttt{is-a}\ \exists r.c \qquad j : I,\ i\ r\ j,\ j\ \texttt{is-a}\ c \vdash f}{\vdash f}$$

- Universal

$$\frac{j : I,\ i\ r\ j \vdash j\ \texttt{is-a}\ c}{\vdash i\ \texttt{is-a}\ \forall r.c}$$

Selected Rules for Relations

Inverse:

$$\frac{\vdash i\, r\, j}{\vdash j\, r^{-1}\, i}$$

Composition:

$$\frac{\vdash i\, r\, j \qquad \vdash j\, s\, k}{\vdash i\, (r; s)\, k}$$

Transitive closure:

$$\frac{}{\vdash i\, r^*\, i} \qquad \frac{\vdash i\, r\, j \qquad \vdash j\, r^*\, k}{\vdash i\, r^*\, k}$$

Identity at concept c:

$$\frac{\vdash i\, \texttt{is-a}\, c}{\vdash i\, \Delta_c\, i}$$

Formal Systems

Typical Structure of a Formal System

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

Example: Vocabularies and Expressions

Aspect	vocabulary Θ	expression kinds
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

Components and Well-Formedness

Components of formal system *I*

- context-free syntax
- distinguished non-terminal symbol $\mathcal{V}$ — words called **vocabularies**
- some distinguished non-terminal symbols — words called **expressions**
- unary predicate $\mathrm{wfv}(V)$ on vocabularies V — well-formed vocabulary V
- unary predicates $\mathrm{wff}_V(e)$ — well-formed expressions e

Intuition

- context-free syntax generates more than needed
- context-sensitive well-formedness defines the exact subset

Question: How do we define the well-formedness predicates?

use an inference system with context-sensitive rules

Inference System

Define well-formedness via

- contexts Γ of the form $x_1 : E_1, \ldots, x_n : E_n$ for expressions E_i
- a set of judgments including
  - a judgment $\vdash^I V$ on vocabularies V $\quad \mathrm{wfv}(V) \text{ iff } \vdash^I_V$
  - a judgment $\Gamma \vdash^I_V e : E$ between expressions e, E

    $\mathrm{wff}_V(e) \text{ iff } \vdash^I_V e : E \text{ for some } E$
- a set of rules for the judgments, each one of the form

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

where the J's are judgments

conventions: leave out superscript I, subscript V if clear

leave out Γ if empty

Terminology

For an inference system, we define

- derivation: tree of judgments such that for every node J with children $J_1, \ldots, J_n$, there is a rule

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

- derivation of J: a derivation with root J
- J holds: there is a derivation of J

$$\mathrm{Voc}^I = \{V \mid \vdash^I V\}$$

$$\mathrm{Exp}^I_V(E) = \{e \mid \vdash^I_V e : E\}$$

$$\mathrm{Exp}^I_V = \bigcup_E \mathrm{Exp}^I_V(E)$$

Special Cases

A formal system with propositions

- additionally has a distinguished expression `prop`
- define F is proposition if $\vdash_V F : \texttt{prop}$

A formal system with equality

- additionally has a distinguished proposition $e_1 \doteq_E e_2$ whenever $\vdash e_i : E$

in the sequel: fix I as above

Deductive Semantics

Deductive Semantics

Definition

- a system that determines which propositions are theorems
- for every V, a subset $\mathrm{Thm}^I_V \subseteq \mathrm{Exp}^I_V(\texttt{prop})$ of theorems

write $\vdash^I_V F$ for $F \in \mathrm{Thm}^I_V$

Terminology

- Logic: language plus deductive semantics
- Calculus: set of rules defining absolute deductive semantics
- Theorem prover: implementation of deductive semantics
- Decision procedure: special case of theorem prover when decidable

Examples

- Natural deduction for first-order logic
- Axiomatic set theory for (most of) mathematics

Redundant Deductive Semantics

Multiple deductive semantics

- Proof theory: absolute
- Model theory: relative via translation to set theory L

  write $\models F$ for $\vdash_L \texttt{True}[\![F]\!]$
- Logic translation: relative via translation into standard logics, e.g., SFOL

Equivalence Theorems

- Soundness: $\vdash F$ implies $\models F$
- Completeness: $\models F$ implies $\vdash F$ accordingly for other translations

Relationship to Typing

Define deductive semantics as a special case of typing

- propositions as types
- proofs as expressions
- extend grammar so that there are expressions for proofs
- add typing rules such that $\vdash P : F$ captures the statement "P is proof of F"
- define: $\vdash F$ iff there is P such that $\vdash P : F$

often called Curry-Howard representation

Computational Semantics

Computational Semantics

Definition

- determines how expressions evaluate to values
- for every V, a function $\mathrm{Eval}^I_V : \mathrm{Exp}^I_V \to \mathrm{Exp}^I_V$

write $\vdash^I_V e \rightsquigarrow e'$ for $e' = \mathrm{Eval}^I_V(e)$

Terminology

- Programming language: languages plus computational semantics
- Operational semantics: rules defining computational semantics
- Interpreter: implementation of absolute semantics
- Compiler: implementation of relative semantics

Examples

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

Caveat

Evaluation $\vdash^{I}_{V} e \rightsquigarrow e'$ insufficient in general

special case of interpreter when evaluation pure and terminating

Actual programming languages more complex

- IO channels
- Object creation/destruction
- Mutable variables
- Non-termination (needed for Turing completeness)

Semantics requires environment, heap, stack, references, ...

Redundant Computational Semantics

Multiple computational semantics

- Specification: absolute as rules on paper
- Interpreter: absolute as implementation
- Compiler: relative via translation to assembly L

  write $\models E \rightsquigarrow V$ for $\vdash_L [\![E]\!] \rightsquigarrow [\![V]\!]$
- Cross-compilation: relative via translation into other languages

  Church-Turing thesis: always possible

Equivalence Theorems

- Correctness of compiler: $\vdash E \rightsquigarrow V$ iff $\models E \rightsquigarrow V$

  accordingly for other translations

Relationships to other judgments

Big Step vs. Small Step

- big step: $\vdash^I_V e \rightsquigarrow e'$ is the entire evaluation
- small step: $\vdash^I_V e \rightsquigarrow e'$ is just one step and semantics requires exhaustive chaining of steps

Typing

- subject reduction: if $\vdash e : E$, then $\vdash \mathrm{Eval}(e) : E$

Deductive semantics with equality

- normal forms:
  - Eval^I_V idempotent, i.e., $\mathrm{Eval}^I_V(x) = x$ if x value
  - $\vdash^I_V e \doteq_E \mathrm{Eval}^I_V(e)$
- canonical forms: $\vdash^I_V e_1 \doteq_E e_2$ iff $\mathrm{Eval}^I_V(e_1) = \mathrm{Eval}^I_V(e_2)$

Interdefinability

Given a computational semantics, define a deductive one:

- distinguished expression $\vdash$ *true* : `prop`,
- $\vdash F$ iff $\mathrm{Eval}(F) = \mathit{true}$

  implies decidability, so usually only possible for some F

Given a deductive semantics, define computational one:

- $\mathrm{Eval}(e)$ is some e' such that $\vdash e \doteq e'$

  trivially normal, but usually not canonical

Both kinds of semantics add different value. We usually want both.

Contexts and Substitutions

Syntax with Contexts

If we want to talk about contexts, too, we need to expand all of the above.

Syntax with contexts

- contexts: for every V, a set Cont'_V write $\vdash_V \Gamma$
- substitutions: for $\Gamma, \Delta \in \mathrm{Cont}_V$, a set $\mathrm{Subs}_V(\Gamma, \Delta)$ write $\vdash_V \gamma : \Gamma \to \Delta$

Expressions in context

- expressions: sets $\mathrm{Exp}_V(\Gamma)$
- substitution application: functions $\mathrm{Exp}(\gamma) : \mathrm{Exp}(\Gamma) \to \mathrm{Exp}(\Delta)$ for $\gamma \in \mathrm{Subs}(\Gamma, \Delta)$ write $\mathrm{Exp}(\gamma)(e)$ as $e[\gamma]$

Typing in context

- expressions: sets $\mathrm{Exp}_V(\Gamma, E)$, written as $\Gamma \vdash_V e : E$
- substitution preserves types: if $\Gamma \vdash e : E$ and $\vdash \gamma : \Gamma \to \Delta$, then $\Delta \vdash e[\gamma] : E[\gamma]$

Contexts: General Definition

We can leave contexts abstract or spell out a concrete definition:

- contexts Γ are of the form

$$x_1 : E_1, \ldots, x_n : E_n$$

  where $E_i \in \mathrm{Exp}(x_1 : E_1, \ldots, x_{i-1} : E_{i-1})$
- for Γ as above, substitutions $\Gamma \to \Delta$ are of the form:

$$x_1 = e_1, \ldots, x_n = e_n$$

  where $\Delta \vdash e_i : E_i[x_1 = e_1, \ldots, x_{i-1} = e_{i-1}]$

This works uniformly for any formal system. But most formal systems are a bit more restrictive, e.g., by requiring that all E_i are types.

Semantics with Contexts

Deductive semantics

- define: theorem sets $\mathrm{Thm}_V(\Gamma)$ write $F \in \mathrm{Thm}_V(\Gamma)$ as $\Gamma \vdash_V F$
- such that theorems are preserved by substitution:
  if $\Gamma \vdash_V F$ and $\vdash \gamma : \Gamma \to \Delta$, then $\Delta \vdash_V F[\gamma]$

Computational semantics

- define: evaluation functions $\mathrm{Eval}_V(\Gamma) : \mathrm{Exp}_V(\Gamma) \to \mathrm{Exp}_V(\Gamma)$
  write $e' = \mathrm{Eval}_V(\Gamma)(e)$ as $\Gamma \vdash_V e \rightsquigarrow e'$
- extend to substitutions:
  $\mathrm{Eval}_V(\Delta)(\ldots, x = e, \ldots) \;=\; \ldots, x = \mathrm{Eval}_V(\Delta)(e), \ldots$
- require that evaluation is preserved by substitution $\vdash \gamma : \Gamma \to \Delta$
  $\mathrm{Eval}_V(\Delta)(e[\gamma]) = \mathrm{Eval}_V(\Delta)(e)[\mathrm{Eval}_V(\Delta)(\gamma)]$
  substitution theorem for Eval as a translation from I to itself

Definitions for Substitutions

- write $\cdot$ for empty context/substitution
- ground expression is expression in empty context
  also called closed; then opposite is open
- ground substitution: $\vdash \gamma : \Gamma \to \varnothing$
  no free variables after substitution
- if we have computational semantics:
  value substitution is ground substitution where all expressions are values
- if we have deductive semantics:
  true instance of $\Gamma \vdash F : \texttt{prop}$ is γ such that $\vdash F[\gamma]$

Concrete Semantics

Concrete Semantics

Definition

- determines the true instances of propositions
- for every $\Gamma \vdash^{I}_{V} F : \texttt{prop}$, a set $\mathrm{Inst}^{I}_{V}(\Gamma, F)$ of ground substitutions

write $\vdash^{I}_{V} \gamma : \Gamma$ and $\vdash F[\gamma]$ for $\gamma \in \mathrm{Inst}_{V}(\Gamma, F)$

Terminology

- Query languages (in the usual, narrower sense than used here): languages plus concrete semantics
- Database: implementation of concrete semantics

usually optimized for fast query answering

Examples

- SQL for Church-typed ontologies with ADTs (relational databases)
- SPARQL for Curry-typed ontologies (triple stores)
- Prolog for first-order logic

Yes/No vs. Wh-Questions

Deductive/concrete semantics may be a bit of a misnomer

- Queries about $\vdash F$ are yes/no questions
  - specialty of deductive semantics
  - but maybe only because everything else is ever harder to do deductively
- Queries about ground instances of $\Gamma \vdash F$ are Wh questions
  - specialty of concrete databases
  - for the special case of retrieving finite results sets from a fixed concrete store
  - only situation where Wh questions are easy

But Yes/no and Wh questions exist in all aspects.

Redundant Concrete Semantics

Multiple concrete semantics

- Specification: absolute as rules on paper
- Database: absolute by custom database
- Database: relative via translation to assembly L

Equivalence Theorems

- typically: choose one, no redundancy, no equivalence theorems
- infinite results: easy on paper, hard in database
- open world: are all known ground instances in database?

Interdefinability

Given concrete semantics, define a deductive one

- for ground F, $\mathrm{Inst}(\cdot, F)$ is either $\{\cdot\}$ or $\{\}$
- $\vdash F$ iff $\mathrm{Inst}(\cdot, F) = \{\cdot\}$
  but concrete semantics usually cannot find all substitutions for all F

Given concrete semantics, define a computational one

- $\vdash e \rightsquigarrow e'$ iff $(x = e') \in \mathrm{Inst}(x : E, e \doteq_E x)$
  but concrete semantics usually cannot find that substitution for all e

Given deductive semantics, define a concrete one

- $\mathrm{Inst}(\Gamma, F) = \{\vdash \gamma : \Gamma \to \cdot \mid \vdash F[\gamma]\}$
  but deductive semantics usually does not allow computing that set

Given computational semantics, define a concrete one

- $\mathrm{Inst}(\Gamma, F) = \{\mathrm{Eval}(\cdot, \gamma) \mid \vdash \gamma : \Gamma \to \cdot, \vdash F[\gamma] \rightsquigarrow \mathit{true}\}$
- allows restricting results to value substitutions
  composition of previous inter-definitions, inherits both problems

Narrative Semantics

Narrative Semantics

Definition

- Describes how to answer (some) questions
- Implementations tend to be AI-complete, hypothetical
- In practice, information retrieval = find related documents

More precisely?

- Not much theory, wide open research problem
- Some natural language document with interspersed definitions, formulas
- Maybe judgment: $\vdash Q?A$ for "A is answer to Q"

Examples

- "W3C Recommendation OWL 2" and Google
- "ISO/IEC 14882: 1998 Programming Language C++" and Stroustrup's book
- Mathematics textbooks and mathematicians

Relative Semantics

Translations

A translation T from formal system I to formal system L consists of

- function $\mathrm{Voc}^T : \mathrm{Voc}^I \to \mathrm{Voc}^L$
- family of functions $\mathrm{Exp}^T_V : \mathrm{Exp}^I_V \to \mathrm{Exp}^L_{\mathrm{Voc}^T(V)}$

Desirable properties

- Should satisfy type preservation:

$$\vdash^I_V e : E \quad \text{implies} \quad \vdash^L_{\mathrm{Voc}^T(V)} \mathrm{Exp}^T_V(e) : \mathrm{Exp}^T_V(E)$$

intuition: what we have, is preserved

- Might satisfy type reflection/conservativity:

$$\vdash^L_{\mathrm{Voc}^T(V)} e' : \mathrm{Exp}^T_V(E) \quad \text{implies} \quad \vdash^I_V e : E \text{ for some } e$$

intuition: nothing new is added

Translation of Contexts

Translations extend to contexts and substitutions

- $\mathrm{Cont}^T(\ldots, x : E, \ldots) = \ldots, x : \mathrm{Exp}^T(E), \ldots$
- $\mathrm{Subs}^T(\ldots, x = e, \ldots) = \ldots, x = \mathrm{Exp}^T(e), \ldots$
- $\mathrm{Exp}^T(x) = x$ for all variables

Desirable properties for arbitrary contexts

- Type preservation:

  $$\Gamma \vdash^I_V e : E \quad \text{implies} \quad \mathrm{Cont}^T_V(\Gamma) \vdash^L_{\mathrm{Voc}^T(V)} \mathrm{Exp}^T_V(e) : \mathrm{Exp}^T_V(E)$$

- Conservativity:

  $$\mathrm{Cont}^T_V(\Gamma) \vdash^L_{\mathrm{Voc}^T(V)} e' : \mathrm{Exp}^T_V(E) \quad \text{implies} \quad \Gamma \vdash^I_V e : E \text{ for some } e$$

Compositionality

Define: a translation is compositional iff we can show the substitution theorem for it

Given

$$\Gamma \vdash^{I}_{V} e : E \qquad \vdash^{I}_{V} \gamma : \Gamma \to \Delta$$

we have that

$$\mathrm{Cont}^{T}_{V}(\Delta) \vdash^{L}_{\mathrm{Voc}^{T}(V)} \mathrm{Exp}^{T}_{V}(e[\gamma]) \doteq_{\mathrm{Exp}^{T}_{V}(E[\gamma])} \mathrm{Exp}^{T}_{V}(e)[\mathrm{Subs}^{T}_{V}(\gamma)]$$

Simplify: write $T(-)$ for $\mathrm{Voc}^{T}(-), \mathrm{Exp}^{T}_{V}(-)$, $\mathrm{Cont}^{T}_{V}(-)$, $\mathrm{Subs}^{T}_{V}(-)$

$$T(\Delta) \vdash^{L}_{T(V)} T(e[\gamma]) \doteq_{T(E[\gamma])} T(e)[T(\gamma)]$$

Relative Semantics

Given

- formal systems I and L
- semantics for L
- translation T from I to L

define semantics for I

- deductive: define

$$\vdash^{I}_{V} F \quad \text{iff} \quad \vdash^{L}_{\mathrm{Voc}^{T}(V)} \mathrm{Exp}^{T}_{V}(F)$$

- computational: define

$$\vdash^{I}_{V} e \rightsquigarrow e' \quad \text{iff} \quad \vdash^{L}_{\mathrm{Voc}^{T}(V)} \mathrm{Exp}^{T}_{V}(e) \rightsquigarrow \mathrm{Exp}^{T}_{V}(e')$$

both work accordingly with a context Γ

- concrete: define

$$\Gamma \vdash^{I}_{V} \gamma : F \quad \text{iff} \quad \mathrm{Cont}^{T}_{V}(\Gamma) \vdash^{L}_{\mathrm{Voc}^{T}(V)} \mathrm{Subs}^{T}_{V}(\gamma) : \mathrm{Exp}^{T}_{V}(F)$$

Equivalence of Semantics

Definition

Two semantics $\vdash^1$ and $\vdash^2$ for l are equivalent if

- deductive: $\vdash^1 F$ iff $\vdash^2 F$
- computational: $\vdash^1 e \rightsquigarrow e'$ iff $\vdash^2 e \rightsquigarrow e'$
- concrete: $\Gamma \vdash^1 \gamma : F$ iff $\Gamma \vdash^2 \gamma : F$

Example for deductive semantics:

- $\vdash^1$ absolute semantics by calculus
  e.g., natural deduction for SFOL
- $\vdash^2$ relatives semantics by translation e.g., L is set theory, T is model theory of SFOL
- Assume proofs-as-expressions
- Then:
  - type preservation = soundness
  - conservativity = completeness

Example: Relative Computational Semantics for BOL

Scala, SQL semantics evaluates

- concept c to
  - SQL: table of individuals — result of running query $[\![c]\!]$
  - Scala: hashset of individuals — result of running program $[\![c]\!]$
- propositions to booleans — accordingly

Technically, results not in image of $[\![-]\!]$
Fix: add productions for all values

$$F ::= \mathit{true} \mid \mathit{false} \quad \text{truth values}$$
$$C ::= \{I, \ldots, I\} \quad \text{finite concepts}$$

Equivalence with respect to Semantics

So far: equivalence of two semantics wrt all queries

Related concept: equivalence of two queries wrt one semantics

- F, G deductively equivalence:

$$\vdash F \quad \text{iff} \quad \vdash G$$

may be internalized by syntax as proposition $F \leftrightarrow G$

- F, G concretely equivalent:

$$\vdash F[\gamma] \quad \text{iff} \quad \vdash G[\gamma]$$

for all ground substitutions γ weaker than $\Gamma \vdash F \leftrightarrow G$

- closed e, e' computationally equivalent:

$$\vdash e \rightsquigarrow v \quad \text{iff} \quad \vdash e' \rightsquigarrow v$$

may be internalized by syntax as proposition $e \doteq e'$

Equivalence with respect to Semantics (2)

Interesting variants of computational semantics

- open e, e' extensionally equivalent:

$$\vdash e[\gamma] \rightsquigarrow v \quad \text{iff} \quad \vdash e'[\gamma] \rightsquigarrow v$$

  for all ground substitutions γ

  equal inputs produce equal outputs

  weaker then $\Gamma \vdash e \doteq e'$ — intensional equivalence

- machines M, M' observationally equivalent:
  produce equal sequences of outputs for the same sequence of inputs e.g., automata, objects in OO-programming

choice of semantics defines legal optimizations in compiler

Equivalence of BOL Semantics

Now 5 semantics for BOL

- absolute deductive via calculus
- relative deductive via SFOL
- relative computational via Scala
- relative concrete via SQL
- relative narrative via English

Moreover, these are interdefinable.

e.g., Scala translation also induces deductive semantics

Can compare equivalence

- for every pair of semantics
- for every kind of equivalence (deductive, concrete, computational)

Question: Which of them hold?

Questions

For example, consider:

- Are the absolute semantics and the Scala semantics deductively equivalent?
- Assuming BOL and SQL have the base types and values: Are the absolute semantics and the SQL semantics concretely equivalent?

Deductive Semantics of BOL

Are these two BOL semantics deductively equivalent

- absolute deductive semantics
- relative deductive semantics via translation $[\![-]\!]$ to SFOL

Soundness: $\vdash^{BOL}_{V} f$ implies $\vdash^{SFOL}_{[\![V]\!]} [\![f]\!]$

- induction on derivations of $\vdash^{BOL}_{V} f$
- one case per rule induction rule from above not sound
- several pages of work but straightforward and relatively easy

Deductive Semantics of BOL

Are these two BOL semantics deductively equivalent

- absolute deductive semantics
- relative deductive semantics via translation $[\![-]\!]$ to SFOL

Completeness: $\vdash^{BOL}_{V} f$ implied by $\vdash^{SFOL}_{[\![V]\!]} [\![f]\!]$

works if we add missing rules

- induction on SFOL derivations does not work
  - SFOL more expressive than BOL
  - $[\![-]\!]$ not surjective
- instead show that $[\![-]\!]$ preserves consistency of vocabularies

no universal recipe how to do that

- then a typical proof uses V extended with $\neg f$
  - if V inconsistent, $\vdash_V f$ for all f, done
  - if V consistent and $V + \neg f$ inconsistent, then $\vdash_V f$, done
  - if $V + \neg f$ consistent, so is $[\![V + \neg f]\!]$, which contradicts $\vdash^{SFOL}_{[\![V]\!]} [\![f]\!]$

Computational Semantics of BOL

Are these two BOL semantics deductively equivalent

- absolute deductive semantics
- relative deductive semantics via translation $[\![-]\!]$ to Scala

Soundness: $\vdash^{BOL}_{V} f$ implies $\vdash^{Scala}_{[\![V]\!]} [\![f]\!] \rightsquigarrow \mathit{true}$

- Problem: Absolute semantics performs consequence closure, e.g.,
  - transitivity of $\sqsubseteq$
  - relationship between $\sqsubseteq$ and `is-a`
- Scala semantics does so only if we explicitly implemented it

we didn't

same problem for SQL semantics

Computational Semantics of BOL

Are these two BOL semantics deductively equivalent

- absolute deductive semantics
- relative deductive semantics via translation $[\![-]\!]$ to Scala

Completeness: $\vdash_{V}^{BOL} f$ implied by $\vdash_{[\![V]\!]}^{Scala} [\![f]\!] \rightsquigarrow \mathit{true}$

- absolute semantics leaves closed world optional
- Scala uses closed worlds

  e.g., used to compute $c \sqsubseteq d$ by checking all individuals
- complete only if we add induction rule

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.

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

The True operator

Problem with type-preserving translation $[\![-]\!]$

- must translate $F : \mathtt{prop}^I$ to $[\![F]\!] : [\![\mathtt{prop}^I]\!]$
- but need not satisfy $[\![\mathtt{prop}^I]\!] = \mathtt{prop}^L$

  I-propositions maybe not translated to L-propositions
- If not, $\vdash^L [\![F]\!]$ cannot be used to answer query $\vdash^I F$

Solution: L-operator $\mathtt{True} : [\![\mathtt{prop}^I]\!] \to \mathtt{prop}^L$

- maps translations of I-propositions to L-propositions
- then use $\vdash^L \mathtt{True}[\![F]\!]$ to answer $\vdash^I F$

Example: standard semantics of SFOL in set theory

- SFOL-propositions: formulas; set theory propositions: truth values
- semantics of formula is function from models to truth values
- True takes $[\![F]\!]$ and returns 1 iff $[\![F]\!](M) = 1$ for all models M

  standard notation for $[\![F]\!](M) = 1$: $[\![F]\!]^M = 1$ or $M \models F$

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

typical for ontology languages

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

Independent of whether I already has contexts/variables, we can 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 $\vdash_I \Gamma$.
- 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 $\vdash_I \gamma : \Gamma$.
- 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 $\Gamma \vdash_I w : N$.
- 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

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

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

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

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:

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

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 $\vdash_L \texttt{True}[\![p[\gamma]]\!]$
- the result set is the set of all results

Example

1. BOL ontology:

   $$\textit{concept male},\ \textit{concept person},\ \textit{axiom male} \sqsubseteq \textit{person},$$
   $$\textit{individual FlorianRabe},\ \textit{assertion FlorianRabe isa male}$$

2. Query x : *individual* $\vdash_{BOL}$ x isa *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

$$\vdash_L \delta : [\![\Gamma]\!] \quad \text{where} \quad [\![\Gamma]\!] \vdash_L \texttt{True}[\![p]\!][\delta]$$

easiest case: just look them up in database

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

$$\vdash_l \gamma : \Gamma \quad \text{where} \quad [\![\Gamma]\!] \vdash_L \texttt{True}[\![p[\gamma]]\!]$$

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

Computational Queries

Definition

We assume

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

We define

- a computational query is an I-expression e
- the result is an I-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

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

Syntactic Querying

Search

- "search" not systematically separated from "querying"
- often interchangeable
- querying tends to imply formal languages for queries with well-specified semantics e.g., SQL
- search tends to imply less targeted process e.g., Google

we will not distinguish between the two

Syntactic vs. Semantic Querying

Semantic querying

- Query results specified by vocabulary V but (usually) not contained in it
- Query answered using semantics of language
- Challenge: apply semantics to find results
  - deductive query $\vdash f : \texttt{prop}$ requires theorem prover
  - computation query $\vdash e : E$ requires evaluator
  - concrete query $\Gamma \vdash f : \texttt{prop}$ requires enumerating all substitutions, running theorem prover/evaluator on all of them

what we've looked at so far

Syntactic querying

- Query is an expression e
- Result is set of occurrences of e in V
- Independent of semantics
- Much easier to realize

Challenges for Syntactic Search

Easier to realize → scale until new challenges arise

- large vocabularies
  - narrative: all text documents in a domain
    e.g., all websites, all math papers
  - deductive: large repositories of formalization in proof assistants
    10^5 theorems
  - computational: package managers for all programming languages
  - concrete: all databases in a domain TBs routine
- incremental indexing: reindex only new/changed parts
- incremental search to handle large result sets pagination
- sophisticated techniques for
  - indexing: to allow for fast retrieval
  - similarity: to select likely results
  - quality: to rank selected results
- integration of some semantic parts

Overview

- Deduction
  - semantic: theorem proving called search
  - syntactic: text search
- Concretization
  - semantic: complex query languages (nestable queries) SQL, SPARQL
  - syntactic: search by identifier (linked data)
- Computation
  - semantic: interpreters called execution
  - mixed: IDEs search for occurrences, dependencies
  - syntactic: search in IDE, package manager
- Narration:
  - semantic: very difficult
  - syntactic: bag of words search

Abstract Definition: Document

Document =

- file or similar resource that contains vocabularies
- often with comments, metadata
- different names per aspect
  - deduction: formalization, theory, article
  - computation: source files
  - concretization: database, ontology ABox
  - narrative: document, web site

Library =

- collection of documents
- usually structured into folders, files or similar
- often grouped by user access e.g., git repository
- vocabularies interrelated within and across libraries

Abstract Definition: Document Fragment

Fragment = subdivision of documents into nested semantic units

Examples

- deductive: theory, section, theorem, definition, proof step, etc.
- computational: class, function, command, etc.
- concrete: table, row, cell
- narrative: section, paragraph, etc.

Assign unique **fragment URI**, e.g., LIB/DOC?FRAG where

- LIB: base URI of library — e.g., repository URL
- DOC: path to document within library — e.g., folder structure, file name
- FRAG: id of fragment within document — e.g., class name/method name

Abstract Definition: Index(er)

Indexer consists of

- data structure O for indexable objects
  specific to aspect, index design
  e.g., words, syntax trees
- function that maps library to index
  the indexing

Index entry consists of

- object that occurred in the library
- URI of the containing fragment
- information on where in the fragment it was found

Index = set of index entries

Abstract Definition: Query and Result

Given

- indexer I with data structure O
- set of libraries
- union of their indexes — computed once, queried often

Query = object $\Gamma \vdash^I q : O$

Result consists of

- index entry with object o
- substitution for Γ such that q matches o

  definition of "match" index-specific, e.g., $q[\gamma] = o$

Result set = set of all results in the index

Bag of Words Search

Definition:

- Index data structure = sequences of words (n-grams) up to a certain length
- Query = bag of words — bag = multiset
- Match: (most) words in query occur in same n-gram or n-grams near each other

Example implementations

- internet search engines for websites
- Elasticsearch: open source engine for custom vocabularies

Mostly used for narrative documents

- can treat concrete values as words — e.g., numbers
- could treat other expressions as words — works badly

Symbolic Search

Definition:

- Index data structure = syntax tree (of any grammar) of expressions o with free/bound variables
- Query = expression q with free (meta-)variables
- Match: $q[\gamma] =_\alpha o$, i.e., up to variable renaming

Example implementation

- MathWebSearch

  see separate slides on MathWebSearch in the repository

Mostly used for formal documents

- deductive
- computational

Knowledge Graph Search

Definition:

- Index data structure = assertion forming node/edge in a knowledge graph
- Index = big knowledge graph G
- Query = knowledge graph g with free variables
- Match: $g[\gamma]$ is part of G

Example implementations

- SPARQL engines without consequence closure

  i.e., the most common case in practice
- graph databases

Mainly used for ABoxes of untyped ontologies

Value Search

Definition:

- Index data structure = BDL values v
- Query = BDL expression q with free variables
- Match: $q[\gamma] = v$

Example implementations

- no systematic implementation yet
- special cases part of most database systems

Could be used for values occurring in any document

- all aspects
- may need to decode/encode before putting in index

Cross-Aspect Occurrences

Observation

- libraries are written in one primary aspect
- indexer focuses on one aspect and kind of object
- but documents may contain indexable objects of any index

Cross-Aspect Occurrences: Examples

- Any library can contain
  - metadata on fragments
    - relation assertions induce knowledge graph structure between fragments
    - property assertions contain values narrative, symbolic objects, or values
  - cross-references to fragments of any other library
  - narrative comments
- Narrative text may contain symbolic expressions

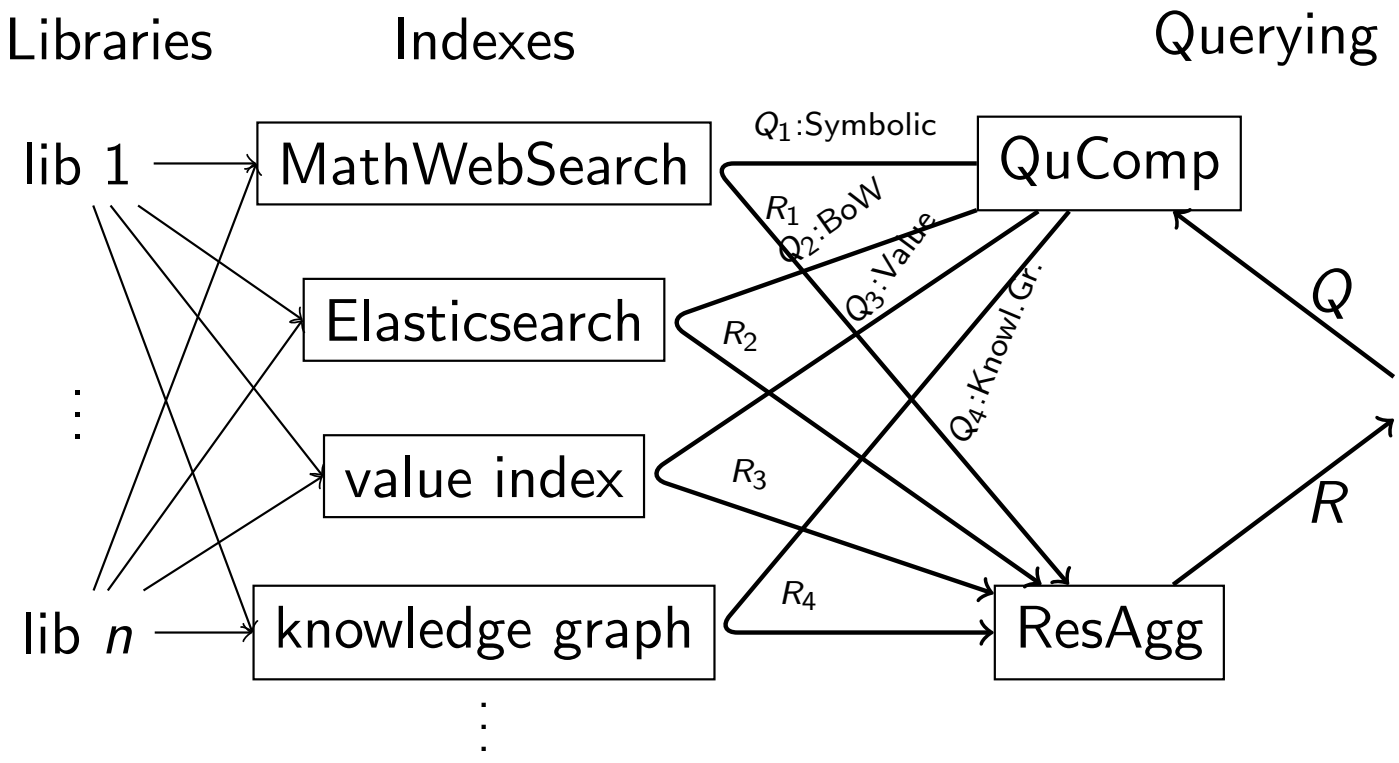
  STEM documents
- Database table may have columns containing
  - text
  - encoded BDL values
  - symbolic expression (often as strings)
- Symbolic fragments may contain database references

  e.g., when using database for persistent memoization

A New Indexing Design

recent paper `https://kwarc.info/people/frabe/Research/BKR_mdql_20.pdf`

with K. Bercic



A New Indexing Design (2)

Tricky question: What is the query language that allows combining queries for each index?

Easy:

- query = conjunction of atomic queries
- each atom queries one index
- QuComp splits into atoms
- ResAgg take intersection of results

Better: allow variables to be shared across atoms

open research question

A New Indexing Design: Example

Consider

- table of graphs with human-recognizable names and arc-transitivity property
  indexed into
  - value index for the graph `sparse6 codec`
  - Boolean computed property for the arc-transitivity in knowledge graph
  - text index for name
- papers from arXiv in narrative index
  indexed into
  - narrative index for text
  - MathWebSearch for formulas
  - knowledge graph for metadata

Query goal: find arc-transitive graphs mentioned by name in articles with h-index greater than 50

Integrating Semantic Querying

Word search

- find multi-meaning words for only one meaning
  "normal" in math
- special treatment of certain queries e.g., "weather" in Google

Symbolic search

- match query $e \doteq e'$ against occurrence $e' \doteq e$
- similarly: associativity, commutativity, etc.
- slippery slope to deductive queries

Value search

- match query 1.5 against interval 1.4 ± 0.2
- match query $5 \cdot x$ against 25
- slippery slope to computational queries

frontiers of research — in our group: for STEM documents