Basic Approaches to the Semantics of Computation (BaSC)

Lecture 1: Introduction

Jorge A. Pérez

Bernoulli Institute for Mathematics, Computer Science, and AI
University of Groningen, Groningen, the Netherlands

November 11, 2025

Part I

Logistics

Overview

A new course this year!
BaSC examines the problem of how to assign meaning to programs.

Overview

A new course this year!
BaSC examines the problem of how to assign meaning to programs.

Understanding program semantics is crucial for computing professionals, who are confronted with programming languages from different paradigms on a daily basis.

Here we study sequential models of computation (imperative and functional) by developing their underlying semantic models and establishing important properties.

Learning Outcomes (Ocasys)

At the end of the course, you will be able to:

1. Identify the distinguishing features of rigorous models of computation for imperative and functional programming.
2. Develop abstract specifications of programs in imperative and functional models, and identify the key differences between them.
3. Formulate formally basic properties of sequential models of computation using mathematical proofs.

Learning Outcomes (Ocasys)

At the end of the course, you will be able to:

1. Identify the distinguishing features of rigorous models of computation for imperative and functional programming.
2. Develop abstract specifications of programs in imperative and functional models, and identify the key differences between them.
3. Formulate formally basic properties of sequential models of computation using mathematical proofs.

Ideally:
After BaSC, you will be prepared (and motivated!) to embark into a research project involving the semantics of programming languages.

Prerequisites

We assume that you know and are comfortable with topics such as:

- **Discrete Mathematics**: Sets, relations (and their properties), the induction principle (and proofs by induction), ordering relations
- **Logic**: Formal reasoning in propositional logic and first-order logic
- **Automata Theory**: Strings, grammars, regular expressions, finite-state machines

If you don't (fully) remember some of these topics, it is a good idea to catch up!

Plan

1. A taste of semantic methods (today!)
2. Inference and unification
3. Well-founded induction
4. IMP, a core imperative language
5. Rule induction
6. IMP: Abstract semantics
7. IMP: Denotational semantics
8. HOFL, a core functional language
9. Rocq, an interactive proof assistant
10. *Optionally, concurrency*: CCS, the Calculus of Communicating Systems

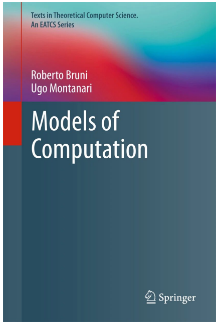
Plan

1. A taste of semantic methods (today!)
2. Inference and unification
3. Well-founded induction
4. IMP, a core imperative language
5. Rule induction
6. IMP: Abstract semantics
7. IMP: Denotational semantics
8. HOFL, a core functional language
9. Rocq, an interactive proof assistant
10. *Optionally, concurrency*: CCS, the Calculus of Communicating Systems

It's a new BSc course: your feedback on the lessons is kindly appreciated!

Material

Textbook:


Texts in Theoretical Computer Science. An EATCS Series
Roberto Bruni
Ugo Montanari
Models of Computation
Springer

- *Models of Computation* by Roberto Bruni and Ugo Montanari.
- Course slides based on those by Roberto Bruni (used with permission)

Many other excellent resources (textbooks, tutorials) around!

Teaching Method

On our side

A combination of **lectures** and **tutorials**.

A typical week:

- Tuesday and/or Thursday: lectures
- Wednesday: tutorial (exercise session, with the TA)

There are variations; check “Schedule 2025” in Brightspace for updates.

On your side

Self-study! Read in advance before each lecture.

Teaching Method

On our side

A combination of **lectures** and **tutorials**.

A typical week:

- Tuesday and/or Thursday: lectures
- Wednesday: tutorial (exercise session, with the TA)

There are variations; check "Schedule 2025" in Brightspace for updates.

Contact:

`basc.rug.contact[at]gmail.com`

Teaching Method

On our side

A combination of **lectures** and **tutorials**.

A typical week:

- Tuesday and/or Thursday: lectures
- Wednesday: tutorial (exercise session, with the TA)

There are variations; check "Schedule 2025" in Brightspace for updates.

Contact:

`basc.rug.contact[at]gmail.com`

On your side

Self-study! Read in advance before each lecture.

Grading (See Ocasys for Details)

Components

1. 3 individual assignments (mandatory, each grade must be at least 3.0).

Important

- Deadlines are always on Friday, 11:59 (noon).
- The use of AI, in any of its wonderful manifestations, is strictly forbidden.

Your Final Grade

If you timely submit the assignments A_1, A_2, A_3 and obtain at least 3.0 in each:

$$F = 0.3 \times A_1 + 0.3 \times A_2 + 0.4 \times A_3$$

Resit

Is $F < 5.75$? You are eligible for a closed-book exam (each A_i must be least 3.0).

Scientific Integrity and Plagiarism

Study Guide Computer Science, section on “Scientific Integrity” — `http://student.portal.rug.nl/infonet/studenten/fse/bachelors/computing-science/`:

> *Plagiarism is not accepted at this university nor elsewhere in the scientific community.*
> *In all cases in which plagiarism is found or suspected, the examiner will inform the Board of Examiners.*
> *When the Board decides that plagiarism has occurred they will sanction in accordance with the “Regulations and Guidelines”.*
> *In general, this will mean that* **a student is excluded from participation in examinations or other forms of testing of the concerning module for the current academic year**.

Part II

A Taste of Semantic Methods

Programming Languages

When we define a programming language, we fix its:

Syntax	Well-formed programs, exclude nonsense
Types	Reduce allowed programs, exclude common mistakes
Pragmatics	How to use constructs and features
Semantics	The meaning of (well-typed) programs

Formal Syntax

The syntax of a formal language rigorously defines

1. The alphabet: which symbols can be used
2. The grammatical structure of programs: which sequences of symbols are valid, which sequences should be discarded

Formal Syntax

The syntax of a formal language rigorously defines

1. The alphabet: which symbols can be used
2. The grammatical structure of programs: which sequences of symbols are valid, which sequences should be discarded

Standard ways for defining syntax: regular expressions, context-free grammars, BNF notation, syntax diagrams, ...

Formal Syntax

The syntax of a formal language rigorously defines

1. The alphabet: which symbols can be used
2. The grammatical structure of programs: which sequences of symbols are valid, which sequences should be discarded

Standard ways for defining syntax: regular expressions, context-free grammars, BNF notation, syntax diagrams, ...

Example

A BNF grammar:

$$
\begin{aligned}
<\texttt{numeral}> &::= <\texttt{digit}> \mid <\texttt{numeral}><\texttt{digit}> \\
<\texttt{digit}> &::= \text{``0''} \mid \text{``1''} \mid \text{``2''} \mid \text{``3''} \mid \text{``4''} \mid \text{``5''} \mid \text{``6''} \mid \text{``7''} \mid \text{``8''} \mid \text{``9''}
\end{aligned}
$$

Type Systems

Type systems can be used to

1. Limit the occurrence of errors
2. Allow compiler optimizations
3. Reduce the presence of bugs
4. Discourage programming malpractices

Type systems are often presented as logic rules

Type Systems

Type systems can be used to

1. Limit the occurrence of errors
2. Allow compiler optimizations
3. Reduce the presence of bugs
4. Discourage programming malpractices

Type systems are often presented as logic rules

Different type systems can be defined for the same language!

Benefits of Formalization

- Standardisation of the language
  programmers write syntactically correct programs
  implementors write correct parsers
- Formal analysis of language properties
  ambiguity, expressiveness, recognizability, comparability
- Automatic implementation of compiler's front-end
  yacc, Bison, Xtext, etc

Pragmatics

Programmers should understand the code they type!

Every language manual also contains

1. natural language descriptions of the various constructs
2. sample code fragments and usage patterns
3. examples of malpractices

Pragmatics

Programmers should understand the code they type!

Every language manual also contains

1. natural language descriptions of the various constructs
2. sample code fragments and usage patterns
3. examples of malpractices

We call them pragmatics:

1. how to exploit the various features
2. how compilers should be designed
3. which auxiliary tools are available

Is it enough?

Natural language descriptions should be

1. as much precise as possible
2. understandable
3. unambiguous but not pedantic

Is it enough?

Natural language descriptions should be

1. as much precise as possible
2. understandable
3. unambiguous but not pedantic

Still, there are issues:

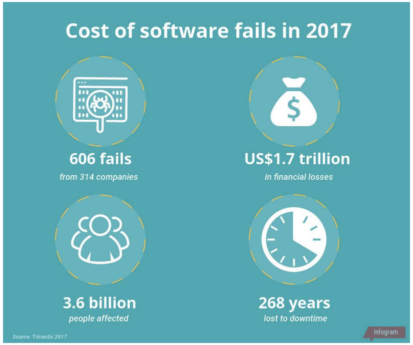
1. it is difficult (nearly impossible) to cover all cases
2. many points will remain open to different interpretations
3. inconsistencies can arise
4. good practices target problems but do not eliminate them (they hide them!)

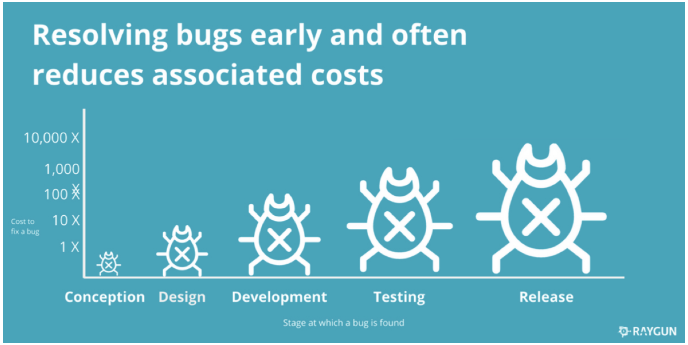
Some Persistent Issues

- How to prove conformance to some specification?
- How to prove absence of problems?
- How to produce reliable code?
- How to prove vendors’ compliance?
- How to prove correctness of an implementation?
- How to define the correct outcomes of test cases?
- How to early detect ambiguities, anomalies, inconsistencies?
- How to expose weaknesses?
- ...

Some Persistent Issues

Practices such as code reviews and test-driven development don't really address the fundamental issues:





Semantics

- The word semantics was introduced in 1900 as
  the study of how words change their meanings (Michel Bréal)
- Ironically its meaning has now changed to
  the study of the attachment between the sentences of a language (written, spoken or formal) and their meanings
- In Computer Science, it is concerned with
  the study of the meaning of (well-typed) programs

Formal semantics assigns rigorous, non-ambiguous meaning: it tells programmers the meaning of the code they type (at some level of abstraction)

Semantics

Someone may always claim

- “(my) implementation is the semantics of the language”

It is correct by definition, but

Semantics

Someone may always claim

- "(my) implementation is the semantics of the language"

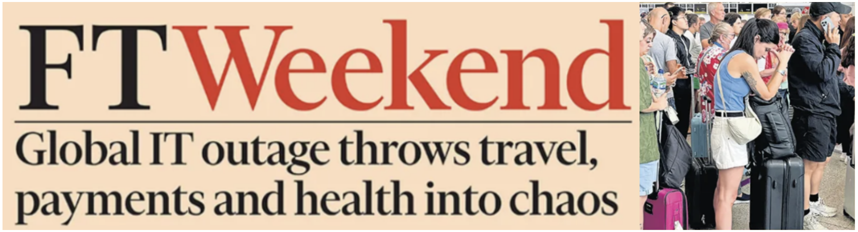
It is correct by definition, but

- machine-independent?
- portability?
- how long it will last?
- useful abstraction for other programmers?
- how to reason on it?
- what about competitors?

Benefits of Formalization (Again)

- Standardization of the reference model of the language
  official, machine-independent
  a mental model for programmers
  a benchmark for implementors
- Formal analysis of language properties
  subtleties, expressiveness, type safety,
  program compliance, subject reduction
- Automatic implementation of compiler's front-end
  prototypical interpreter for experimentation

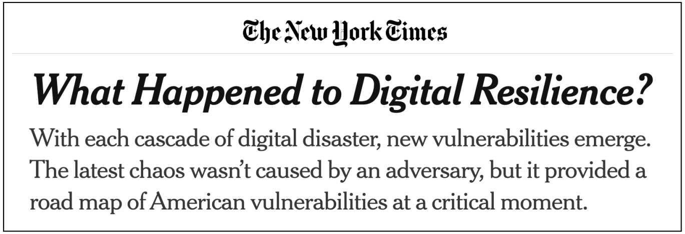
Math and logic? I want to code!



FT Weekend

Global IT outage throws travel, payments and health into chaos

(July 19, 2024)



The New York Times

What Happened to Digital Resilience?

With each cascade of digital disaster, new vulnerabilities emerge. The latest chaos wasn't caused by an adversary, but it provided a road map of American vulnerabilities at a critical moment.

(July 19, 2024)

Different Approaches

Roughly, semantics definition methods fall into three groups:

- Operational
  The interest is in **how** the effect of the computation is achieved
- Denotational
  Here only the **effect** is of interest, not how it is obtained
- Axiomatic
  Focus is on valid **assertions** about the computation (Hoare logic, anyone?)

Operational Semantics

- Idea:
  Define some kind of abstract machine and describe the meaning of a program in terms of the steps that this machine executes to perform the task
- Rationale: Explain computations

$$s_0 \to s_1 \to s_2 \to \cdots \to s_n \to r$$

Important aspects:

- We study 'abstract machines' as opposed to concrete computing devices
- The emphasis is on states and state transformations

Operational Semantics: History

- ['70s] Small-step: Semantics of LISP by John McCarthy (1960) and of Algol 68 (1975)
- ['80s] SOS approach: Gordon Plotkin introduced the structural operational semantics in 1981. ‘Structural’: syntax-oriented and inductively defined.

$$\frac{\langle e_0, \sigma\rangle \rightarrow \langle e_0', \sigma\rangle}{\langle e_0 + e_1, \sigma\rangle \rightarrow \langle e_0' + e_1, \sigma\rangle}$$

- ['90s] Big-step: Gilles Kahn introduced the natural semantics in 1987, where the result is computed in a single step:

$$s \longrightarrow r$$

SOS Semantics: Overview

The transition relation between states is typically defined inductively, by axioms and inference rules according to the syntax of the program.

Advantages:

- immediate implementation (Horn clauses in logic programming);
- prototype Prolog interpreter (almost) for free;
- strong connections to the syntax of the language;
- rules for different constructs are neatly separated;
- useful to detect underspecified behaviors;
- involved mathematics is usually not much complicated;
- SOS descriptions are easy to read, even for non-specialists;
- could appear in any manual (but usually it doesn't)

Denotational Semantics

- Idea:
  A program's meaning is some mathematical object (e.g. a function from input to output) and the steps taken to calculate the result are unimportant:

$$[\![\cdot]\!] : \mathit{Programs} \to \mathit{Domain}$$

- Rationale: Functions are independent of their means of computation; hence, they are simpler than the step-by-step sequence in an operational semantics

Denotational Semantics: History

- ['70s]: Christopher Strachey and Dana Scott

Compositionally principle: the semantics takes the form of a function that assigns an element of some mathematical domain to each individual construct.

Key principle:

> *the meaning of a composite construct does not depend on the particular form of the constituent constructs, but only on their meanings*

Denotational Semantics: Overview

Advantages:

- mathematically elegant;
- useful to detect underspecified behaviors;
- can be used to derive prototype implementations;
- has served as inspiration for many programming languages;
- difficult to apply to concurrent, interactive systems

Axiomatic Semantics

- Idea:
  Describe the constructs in a programming language by providing logical axioms that are satisfied by these constructs
- Rationale: Prove the correctness of a program with respect to a given specification

$$\vdash \{P\}\, c\, \{Q\}$$

Axiomatic Semantics: History

- ['60s]: Robert W. Floyd (1967) and Tony Hoare (1969)

Hoare logic: A statement is accompanied by a precondition (the state before the execution) and a postcondition (after the execution).

Key principle:

> *the meaning of a program is a logical proposition that states some property of the output whenever some properties of the input are met*

Axiomatic Semantics: Overview

Advantages:

- emphasis on proof correctness from the very start;
- strikingly elegant proof systems;
- can be used to prove absence of bugs;
- difficult to apply to concurrent, interactive systems

Which One to Choose?

- Different semantics are often seen in opposition one each other, but this should not be the case! We would gain much more from their combination!
- In this course, we focus on operational and denotational semantics
- We will present the fundamental ideas behind these approaches and stress their relationship, by proving some relevant correspondence theorems.

A Simple Language

Informal syntax of numerical expressions:

- any numeral N is an expression;
- if E_1 and E_2 are expressions, then $\mathsf{E}_1 \oplus \mathsf{E}_2$ is an expression;
- if E_1 and E_2 are expressions, then $\mathsf{E}_1 \otimes \mathsf{E}_2$ is an expression.

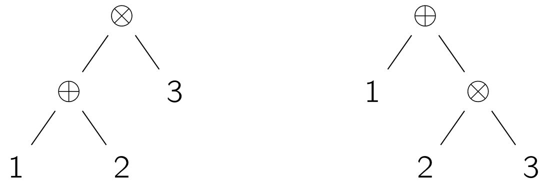
Numerals vs numbers:

- Numerals is a syntax (text) for writing numbers (‘5’, ‘1 0 1’, ‘five’, ‘cinq’)
- Numbers, denoted $\boldsymbol{n}, \boldsymbol{n}', \ldots$, denote mathematical objects (i.e. concepts).

Formal Syntax

$$E ::= N \mid E_1 \oplus E_2 \mid E_1 \otimes E_2$$

- The string ‘$\oplus 3 \otimes 4$’ is not a well-formed numerical expression.
- The string ‘$1 \oplus 2 \otimes 3$’ is well-formed. Two abstract syntax trees:



- To solve ambiguities, we use parenthesis (e.g., ‘$1 \oplus (2 \otimes 3)$’) or fix precedence between operators.

Assigning Meaning to Expressions

$$E ::= N \mid E_1 \oplus E_2 \mid E_1 \otimes E_2$$

Going beyond syntax:

- is N necessarily a number?
- is $\oplus$ necessarily the arithmetic sum?
- is $\otimes$ necessarily the arithmetic product?

Assigning Meaning to Expressions

$$E ::= N \mid E_1 \oplus E_2 \mid E_1 \otimes E_2$$

Going beyond syntax:

- is N necessarily a number?
- is $\oplus$ necessarily the arithmetic sum?
- is $\otimes$ necessarily the arithmetic product?

Some possibilities—maybe we mean

- matrices with addition and multiplication,
- or sets with union and intersection
- or strings with concatenation and least common prefix
- or trees with branching and merging

An Informal Semantics

- a numeral N evaluates to its corresponding number n;

An Informal Semantics

- a numeral N evaluates to its corresponding number n;
- to evaluate $\mathsf{E}_1 \oplus \mathsf{E}_2$ we evaluate E_1 and E_2 and sum their values;

An Informal Semantics

- a numeral N evaluates to its corresponding number n;
- to evaluate $E_1 \oplus E_2$ we evaluate E_1 and E_2 and sum their values;
- to evaluate $E_1 \otimes E_2$ we evaluate E_1 and E_2 and multiply their values.

An Informal Semantics

- a numeral N evaluates to its corresponding number n;
- to evaluate $\mathsf{E}_1 \oplus \mathsf{E}_2$ we evaluate E_1 and E_2 and sum their values;
- to evaluate $\mathsf{E}_1 \otimes \mathsf{E}_2$ we evaluate E_1 and E_2 and multiply their values.

Note:

- These three rules are enough to determine the value of any well-formed expression, no matter how large
- We are not specifying the order in which arguments are evaluated: is it important?

Pragmatics

Example

- 2 evaluates to 2
- $(1 \oplus 2) \otimes 3$ evaluates to 9
- $(1 \oplus 2) \otimes (3 \oplus 4)$ evaluates to 21

Small-step Semantics

Let's define a syntax of runtime numerical expressions:

$$\mathsf{E} ::= n \mid \mathsf{N} \mid \mathsf{E}_1 \oplus \mathsf{E}_2 \mid \mathsf{E}_1 \otimes \mathsf{E}_2$$

Intuition: The state of the abstract machine can mix intermediate results with (not yet evaluated) expressions

Small-step Semantics

Let's define a syntax of runtime numerical expressions:

$$\mathsf{E} ::= n \mid \mathsf{N} \mid \mathsf{E}_1 \oplus \mathsf{E}_2 \mid \mathsf{E}_1 \otimes \mathsf{E}_2$$

Intuition: The state of the abstract machine can mix intermediate results with (not yet evaluated) expressions

We want a small-step semantics, such that:

- A step:

$$\mathsf{E}_0 \to \mathsf{E}_1$$

- An evaluation:

$$\mathsf{E}_0 \to \mathsf{E}_1 \to \mathsf{E}_2 \to \cdots \to \mathsf{E}_k \to n \qquad (\text{also written } \mathsf{E}_0 \to^* n)$$

- We also expect $n \not\to$

Inference Rules: Format

(Rule Name)

$$\frac{\text{premise}_1 \qquad \cdots \qquad \text{premise}_n}{\text{conclusion}} \ \text{(side condition)}$$

Informally: If the premises and the side condition are met then the conclusion holds.

Note:

- The conclusion is a single judgement
- The premises consist of one, none or more judgements
- The side condition is a logical predicate
- The rule name is just a convenient label

SOS Rules - Small-step Semantics

(num)
$$\frac{}{\mathsf{N} \to n}$$

(sum)
$$\frac{}{n_0 \oplus n_1 \to n} \quad n = n_0 + n_1$$

(sumL)
$$\frac{\mathsf{E}_0 \to \mathsf{E}_0'}{\mathsf{E}_0 \oplus \mathsf{E}_1 \to \mathsf{E}_0' \oplus \mathsf{E}_1}$$

(sumR)
$$\frac{\mathsf{E}_1 \to \mathsf{E}_1'}{\mathsf{E}_0 \oplus \mathsf{E}_1 \to \mathsf{E}_0 \oplus \mathsf{E}_1'}$$

SOS Rules - Small-step Semantics

(num)
$$\frac{}{\mathsf{N} \to n}$$

(sum)
$$\frac{}{n_0 \oplus n_1 \to n} \quad n = n_0 + n_1$$

(sumL)
$$\frac{\mathsf{E}_0 \to \mathsf{E}_0'}{\mathsf{E}_0 \oplus \mathsf{E}_1 \to \mathsf{E}_0' \oplus \mathsf{E}_1}$$

(sumR)
$$\frac{\mathsf{E}_1 \to \mathsf{E}_1'}{\mathsf{E}_0 \oplus \mathsf{E}_1 \to \mathsf{E}_0 \oplus \mathsf{E}_1'}$$

What about the rest?

(prod)
$$\frac{}{}$$

(prodL)
$$\frac{}{}$$

(sumR)
$$\frac{}{}$$

Some Derivations

Example

$$
\text{(prodL)}\ \dfrac{\text{(sumL)}\ \dfrac{\text{(num)}\ \dfrac{}{1 \rightarrow 1}}{(1 \oplus 2) \rightarrow (1 \oplus 2)}}{(1 \oplus 2) \otimes (3 \oplus 4) \rightarrow (1 \oplus 2) \otimes (3 \oplus 4)}
$$

Some Derivations

Example

$$\dfrac{\dfrac{\dfrac{}{2 \to 2}\,(\text{num})}{(1 \oplus 2) \to (1 \oplus 2)}\,(\text{sumR})}{(1 \oplus 2) \otimes (3 \oplus 4) \to (1 \oplus 2) \otimes (3 \oplus 4)}\,(\text{prodL})$$

Example

$$\dfrac{\dfrac{}{1 \oplus 2 \to 3}\,(\text{sum}) \qquad 3 = 1 \oplus 2}{(1 \oplus 2) \otimes (3 \oplus 4) \to 3 \otimes (3 \oplus 4)}\,(\text{prodL})$$

A Computation

$$\begin{aligned}(1 \oplus 2) \otimes (3 \oplus 4) &\to (1 \oplus 2) \otimes (3 \oplus 4)\\ &\to (1 \oplus 2) \otimes (3 \oplus 4)\\ &\to 3 \otimes (3 \oplus 4)\\ &\to 3 \otimes (3 \oplus 4)\\ &\to 3 \otimes (3 \oplus 4)\\ &\to 3 \otimes 7\\ &\to 21\\ &\not\to\end{aligned}$$

Therefore, $(1 \oplus 2) \otimes (3 \oplus 4) \to^* 21$.

Note: For each use of '$\to$' above, there is a corresponding derivation.

Another Computation

$$\begin{aligned}
(1 \oplus 2) \otimes (3 \oplus 4) &\to (1 \oplus 2) \otimes (3 \oplus 4) \\
&\to (1 \oplus 2) \otimes (3 \oplus 4) \\
&\to (1 \oplus 2) \otimes (3 \oplus 4) \\
&\to (1 \oplus 2) \otimes 7 \\
&\to (1 \oplus 2) \otimes 7 \\
&\to 3 \otimes 7 \\
&\to 21 \\
&\not\to
\end{aligned}$$

Therefore, $(1 \oplus 2) \otimes (3 \oplus 4) \to^* 21$.

Confluence?

- There are many different evaluation sequences starting from the same expression—this is called non-determinism.
- Are they guaranteed to lead to the same outcome?
- We can optimize the inference rules to impose determinism, i.e., some specific evaluation strategy.
- For example, we can impose a left-to-right evaluation of arguments by changing rules (sumR) and (prodR).

SOS Rules with Evaluation Strategy

(num)
$$\frac{}{\mathsf{N} \to n}$$

(sum)
$$\frac{}{n_0 \oplus n_1 \to n} \; n = n_0 + n_1$$

(sumL)
$$\frac{\mathsf{E}_0 \to \mathsf{E}_0'}{\mathsf{E}_0 \oplus \mathsf{E}_1 \to \mathsf{E}_0' \oplus \mathsf{E}_1}$$

(sumR)
$$\frac{\mathsf{E}_1 \to \mathsf{E}_1'}{n_0 \oplus \mathsf{E}_1 \to n_0 \oplus \mathsf{E}_1'}$$

Where is the difference?

SOS Rules with Evaluation Strategy

(num)
$$\frac{}{\mathsf{N} \rightarrow n}$$

(sum)
$$\frac{}{n_0 \oplus n_1 \rightarrow n}\; n = n_0 + n_1$$

(sumL)
$$\frac{E_0 \rightarrow E_0'}{E_0 \oplus E_1 \rightarrow E_0' \oplus E_1}$$

(sumR)
$$\frac{E_1 \rightarrow E_1'}{n_0 \oplus E_1 \rightarrow n_0 \oplus E_1'}$$

Where is the difference? What about the rest?

(prod)
$$\frac{}{}$$

(prodL)
$$\frac{}{}$$

(sumR)
$$\frac{}{}$$

A Computation

With the adopted evaluation strategy, we have only one possible computation:

$$
\begin{aligned}
(1 \oplus 2) \otimes (3 \oplus 4) &\to ({\color{blue}1} \oplus 2) \otimes (3 \oplus 4) \\
&\to ({\color{blue}1} \oplus {\color{blue}2}) \otimes (3 \oplus 4) \\
&\to {\color{blue}3} \otimes (3 \oplus 4) \\
&\to {\color{blue}3} \otimes ({\color{blue}3} \oplus 4) \\
&\to {\color{blue}3} \otimes ({\color{blue}3} \oplus {\color{blue}4}) \\
&\to {\color{blue}3} \otimes {\color{blue}7} \\
&\to {\color{blue}21} \\
&\not\to
\end{aligned}
$$

Therefore, $(1 \oplus 2) \otimes (3 \oplus 4) \to^* {\color{blue}21}$.

Big-step semantics

- A step represents a whole computation:

$$\mathsf{E}_0 \longrightarrow n$$

- How to define the transition relation?
- Usually simpler than small-step rules.
- Can correspond to an efficient interpreter!
- Cannot express non-terminating computations: derivations are possible only for terminating programs.

SOS Rules - Big-step Semantics

(num)

$$\frac{}{\mathsf{N} \longrightarrow n}$$

(sum)

$$\frac{\mathsf{E}_0 \longrightarrow n_0 \qquad \mathsf{E}_1 \longrightarrow n_1}{\mathsf{E}_0 \oplus \mathsf{E}_1 \longrightarrow n} \; n = n_0 + n_1$$

(prod)

$$\frac{}{}$$

A Derivation

Example

$$\dfrac{(\text{sum})\ \dfrac{(\text{num})\ \dfrac{}{1 \longrightarrow 1} \qquad (\text{num})\ \dfrac{}{2 \longrightarrow 2}}{(1 \oplus 2) \longrightarrow 3} \qquad (\text{sum})\ \dfrac{(\text{num})\ \dfrac{}{3 \longrightarrow 3} \qquad (\text{num})\ \dfrac{}{4 \longrightarrow 4}}{(3 \oplus 4) \longrightarrow 7}}{(1 \oplus 2) \otimes (3 \oplus 4) \longrightarrow 21}\ (\text{prod})$$

Denotational Semantics

- We define a domain and an interpretation function.
- For expressions, the domain is $\mathbb{N}$ and the interpretation is

$$\mathcal{E}[\![\cdot]\!] : \mathit{Exp} \to \mathbb{N}$$

- Given E (a piece of syntax), its denotation $\mathcal{E}[\![\text{E}]\!]$ is a semantic object.

Denotational Semantics

- We define a domain and an interpretation function.
- For expressions, the domain is $\mathbb{N}$ and the interpretation is

$$\mathcal{E}[\![\cdot]\!] : \mathit{Exp} \to \mathbb{N}$$

- Given E (a piece of syntax), its denotation $\mathcal{E}[\![\mathrm{E}]\!]$ is a semantic object.
- The choice of the domain has immediate consequences: We already know that expressions are deterministic (every expression has at most one answer) and normalizing (every expression has an answer)
- Different syntactic categories may require different domains!

Structural Induction

$$\begin{aligned} \mathcal{E}[\![\mathsf{N}]\!] &= n \\ \mathcal{E}[\![\mathsf{E}_0 \oplus \mathsf{E}_1]\!] &= \mathcal{E}[\![\mathsf{E}_0]\!] + \mathcal{E}[\![\mathsf{E}_1]\!] \\ \mathcal{E}[\![\mathsf{E}_0 \otimes \mathsf{E}_1]\!] &= \mathcal{E}[\![\mathsf{E}_0]\!] \cdot \mathcal{E}[\![\mathsf{E}_1]\!] \end{aligned}$$

- Compositionality Principle:

  The meaning of a composite construct does not depend on the particular form of the constituent constructs, but only on their meanings.

An Evaluation

The denotational semantics at work:

$$\begin{aligned}
\mathcal{E}[\![(1 \oplus 2) \otimes (3 \oplus 4)]\!] &= \mathcal{E}[\![1 \oplus 2]\!] \cdot \mathcal{E}[\![3 \oplus 4]\!] \\
&= (\mathcal{E}[\![1]\!] + \mathcal{E}[\![2]\!]) \cdot (\mathcal{E}[\![3]\!] + \mathcal{E}[\![4]\!]) \\
&= (1 + 2) \cdot (3 + 4) \\
&= 21
\end{aligned}$$

Comparison

We have seen different approaches to the semantics of numerical expressions.

Let us consider some angles for comparison:

- Fundamental properties (termination, determinacy)
- Induced equivalences
- Congruences (equivalences under any context)

Comparison

We have seen different approaches to the semantics of numerical expressions:

$\mathsf{E} \to^* n$ $\mathsf{E} \longrightarrow n$ $\mathcal{E}[\![\mathsf{E}]\!] = n$

Consider a key property: termination (aka normalization):

$\forall \mathsf{E}.\exists n.\mathsf{E} \to^* n$	$\forall \mathsf{E}.\exists n.\mathsf{E} \longrightarrow n$	$\forall \mathsf{E}.\exists n.\mathcal{E}[\![\mathsf{E}]\!] = n$
must be proved	must be proved	immediate

Comparison

Now consider determinacy:

$\forall \mathsf{E}, n, m.(\mathsf{E} \rightarrow^* n \wedge \mathsf{E} \rightarrow^* m) \Rightarrow n = m$	must be proved
$\forall \mathsf{E}, n, m.(\mathsf{E} \longrightarrow n \wedge \mathsf{E} \longrightarrow m) \Rightarrow n = m$	must be proved
$\forall \mathsf{E}, n, m.(\mathcal{E}[\![\mathsf{E}]\!] = n \wedge \mathcal{E}[\![\mathsf{E}]\!] = m) \Rightarrow n = m$	immediate

Comparison

Consider consistency:

$$\forall \mathrm{E}.\exists n.(\mathrm{E} \to^* n \;\Leftrightarrow\; \mathrm{E} \longrightarrow n \;\Leftrightarrow\; \mathcal{E}[\![\mathrm{E}]\!] = n)$$

must be proved

Consider induced equivalences:

- Define $\mathrm{E}_0 \equiv_s \mathrm{E}_1$ iff $\forall n.(\mathrm{E}_0 \to^* n \Leftrightarrow \mathrm{E}_1 \to^* n)$
- Define $\mathrm{E}_0 \equiv_b \mathrm{E}_1$ iff $\forall n.(\mathrm{E}_0 \longrightarrow n \Leftrightarrow \mathrm{E}_1 \longrightarrow n)$
- Define $\mathrm{E}_0 \equiv_d \mathrm{E}_1$ iff $\mathcal{E}[\![\mathrm{E}_0]\!] = \mathcal{E}[\![\mathrm{E}_1]\!]$.

Do these equivalences coincide?

Comparison

Can we prove or disprove:

- Properties of specific expressions, such as, e.g.,

$$2 \otimes 6 \equiv_s 3 \otimes 4$$

- Properties of generic expressions, such as, e.g.,

$$\forall E, E_1, E_2 . E \otimes (E_1 \oplus E_2) \equiv_d (E \otimes E_1) \oplus (E \otimes E_2)$$

Comparison

- An equivalence is a congruence when it is preserved by all constructs of a language. A congruence ensures that equality holds under any context.

Comparison

- An equivalence is a congruence when it is preserved by all constructs of a language. A congruence ensures that equality holds under any context.
- A context for numerical expressions is an expression with a 'hole', denoted $\bullet$:

$$\mathsf{C}[\bullet] ::= [\bullet] \mid \mathsf{C}[\bullet] \oplus \mathsf{E} \mid \mathsf{E} \oplus \mathsf{C}[\bullet] \mid \mathsf{C}[\bullet] \otimes \mathsf{E} \mid \mathsf{E} \otimes \mathsf{C}[\bullet]$$

- This way, given a context $\mathsf{C}[\bullet]$ and an expression E, a filled context is $\mathsf{C}[\mathsf{E}]$.

Comparison

- An equivalence is a congruence when it is preserved by all constructs of a language. A congruence ensures that equality holds under any context.
- A context for numerical expressions is an expression with a 'hole', denoted $\bullet$:

$$C[\bullet] ::= [\bullet] \mid C[\bullet] \oplus E \mid E \oplus C[\bullet] \mid C[\bullet] \otimes E \mid E \otimes C[\bullet]$$

- This way, given a context $C[\bullet]$ and an expression E, a filled context is $C[E]$.

Relevant properties:

$\forall E_0, E_1, C[\bullet].(E_0 \equiv_s E_1 \Rightarrow C[E_0] \equiv_s C[E_1])$ must be proven

$\forall E_0, E_1, C[\bullet].(E_0 \equiv_b E_1 \Rightarrow C[E_0] \equiv_b C[E_1])$ must be proven

$\forall E_0, E_1, C[\bullet].(E_0 \equiv_d E_1 \Rightarrow C[E_0] \equiv_d C[E_1])$ immediate

Beyond Numerical Expressions

Suppose that we have expressions with variables, denoted $x, y, \ldots$:

$$\mathsf{E} ::= x \mid \mathsf{N} \mid \mathsf{E} \oplus \mathsf{E} \mid \mathsf{E} \mid \ \otimes \mathsf{E}$$

- How to evaluate expressions such as $(x \oplus 4) \otimes y$?

Beyond Numerical Expressions

Suppose that we have expressions with variables, denoted $x, y, \ldots$:

$$\mathsf{E} ::= x \mid \mathsf{N} \mid \mathsf{E} \oplus \mathsf{E} \mid \mathsf{E} \mid \ \otimes \mathsf{E}$$

- How to evaluate expressions such as $(x \oplus 4) \otimes y$?
- Solution: We need some memories

$$\mathbb{M} \triangleq \{\sigma \mid \sigma : X \to \mathbb{N}\}$$

- The states of the abstract machines and the interpretation function:

$$\langle \mathsf{E}, \sigma \rangle \qquad\qquad \mathcal{E}[\![\cdot]\!] : \mathit{Exp} \to (\mathbb{M} \to \mathbb{N})$$

How to redefine the various semantics and properties?

The End