Semantics and Domain Theory IMC011, Lecture 1

H. Geuvers

Institute for Computing and Information Sciences
Radboud University Nijmegen

Version: fall 2023

Outline

Organisation of the lectures

- Lectures on Monday, 8:30-10:15 in HG00.622, Exercise class on Tuesday, 8:30-10:15 in HFML0220.
- Exceptions:
  - In week 39 (September 25, 26), there will be no lecture / exercise class.
  - I would like to plan an **extra** lecture / exercise class in week 43 (October 23, 24), if that's ok with everyone.
- information:
  `http://www.cs.ru.nl/ herman/onderwijs/semantics2023/`
- but answers to exercises will appear on Brightspace.
- Grade = (3/4 x Written-exam) + (1/4 x Assignment).
- You can redo both.
- Explanation of the assignment will come later.

Semantics and Domain Theory

- Semantics: "assigning meaning to programs" (more generally: to phrases in a *formal* language)
- Domain theory: the mathematical theory of the *sets-with-structure* necessary to achieve this

One contrasts

- operational semantics: "evaluation"
- axiomatic semantics: "logic" (assertions about programs)
- denotational semantics: "model theory"

Operational semantics

- inductive definitions
- grammars
- systems of inference rules defining derivations of judgments
- definition by structural recursion (on syntax, on rules)
- proof by induction
- specification of *behaviour* by execution/evaluation
- exp $\Rightarrow$ val, evaluation judgments, for apppropriate notions of "exp" (expressions) and "val" (values).
- config $\Rightarrow$ st, execution judgemnts, for an abstract machine with *configurations* "config" and *final states* "st"

Operational semantics

There are different styles of operational semantics:

- **big-step**: evaluation/execution all in one go;
- **small-step**: consider intermediate configurations of an abstract machine; take transitive closure to reach a final state

Operational semantics gives an **intensional** theory of behaviour: **how**, not what.

Denotational semantics

- abstract mathematical structures
- behaviour described in terms of mathematical functions operating on (abstract) values
- **extensional** theory of behaviour: **what**, not how
- "equal values map to equal results": need notions of equality of abstract programs
- typically *recursive* (rather than inductive) definitions
- "a general theory of recursive definitions"

Another view on denotational semantics/domain theory is:
"a general theory of partiality" or "of partial recursive definitions"

Programming languages I

The simple imperative language (IMP) or (WHILE):

- assignment to variables (called "locations" by Fiore et al.), conditional execution, sequential composition (need to put smaller program together to make larger ones), potentially unbounded iteration
- origins: Turing/von Neumann model (1930s), ForTran (Backus, 1957)
- meanings given by transformations on states: partial functions from locations (variables) to values, so: ... higher-order functions ...

Programming languages II

Functional languages, essentially varieties of lambda calculus (PCF):

- function application and abstraction, possibly built on top of some primitives.
- origins: Church (untyped 1930s, typed 1940s), LISP (McCarthy, 1957).
- meanings given by... already in terms of ... higher-order functions (again!).
- defined (usually) by (possibly many) syntactic categories (phrase types) T, E, C, ... specified by a (context-free) grammar.
- each phrase type C gives rise to a set of the well-formed phrases of that type. We identify C with this set.

Domains

A domain is ...

- an appropriate target $[\![C]\!]$ for interpreting (giving meaning to) elements of C;
- it will turn out to be analysed in terms of suitable set-with-structure (an order relation reflecting *partial states of knowledge*).

Denotational Semantics

A denotational semantics for the phrase type C is simply a mapping

$$[\![-]\!] : C \to [\![C]\!].$$

So we overload $[\![-]\!]$ in the usual way; these double braces are usually called *Scott brackets* after Dana Scott, who basically founded the subject; he was originally a student of Tarski)

$[\![-]\!]$ should respect/reflect the appropriate structure on C and $[\![C]\!]$.

Features of Denotational Semantics

- **Inductive definitions** of syntax and of relations (e.g. operational semantics).
- Proofs over these by **structural induction**.
- Defining functions (e.g. over syntax) by **recursion**.
- Definitions are **compositional**: There is a clause for each syntactic construct and the semantics of a composite element is defined in terms of the semantics of its constituents.

Summarising

The principal aim of the subject:

to develop the interplay between these two points-of-view on the meaning/behaviour of programs, that is, to relate

- execution of programs,
  e.g. $\langle \text{prog}, \text{config} \rangle \Rightarrow \text{config}$
- mathematical description in terms of functions,
  e.g. $[\![\text{prog}]\!] : [\![\text{config}]\!] \to [\![\text{config}]\!]$

taking into account

- partiality
- recursion
- appropriate notions of equality, and simulation, between programs

Rest of this first lecture

- Section 1.1 of DENS (Fiore et al.); next lecture we will complete Ch.1 of DENS.
- A simple example of the interplay between operational semantics and denotational semantics.

For those who haven't followed the course *Semantics and Correctness, IBC026* (or a similar course): read pages 7-32 (until 2.2) of *Semantics with applications*, by Hanne Riis Nielson and Flemming Nielson (Wiley 1999), see the webpage of the course.