Utrecht University

Concepts of Programming Language Design

Semantics

Gabriele Keller
Tom Smeding

Where we are

- So far
  - Judgements and inference rules
  - Rule induction
  - Inference rules
  - Grammars specified using inference rules
  - Judgements and relations
  - First- and higher-order abstract syntax
  - Substitution
- Next up
  - Static semantics
  - Dynamic semantics

Utrecht University

Static Semantics

- What is static semantics?
  - properties of a program apparent without executing the program
  - can be checked by a compiler (or external tool such as `lint`)
  - depends on the programming language (e.g. scoping)
- Example of static properties
  - does the program contain undefined/out of scope occurrences of variables?
  - is the program type correct?
  - does it contain dead code, usage of uninitialised variables?
- Arithmetic example language
  - there is only one type (Int), so not much to check
  - but we can check scoping (are all variables defined?)

Utrecht University

Scoping

- Inference rules to check scoping
  - judgement ***e ok***: *e* contains no free variables
  - how can we define this using inference rules?

- Recall the rules to check if expressions are syntactically correct:

$$\frac{i \in \mathit{Int}}{(\texttt{Num}\ i\ )\ \mathit{expr}} \qquad \frac{t_1\ \mathit{expr} \quad t_2\ \mathit{expr}}{(\texttt{Plus}\ t_1\ t_2)\ \ \mathit{expr}} \qquad \frac{t_1\ \mathit{expr} \quad t_2\ \mathit{expr}}{(\texttt{Times}\ t_1\ \ t_2)\ \ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \qquad t_2\ \mathit{expr}}{(\texttt{Let}\ \ t_1\ \ (\mathit{id}.t_2))\ \ \mathit{expr}} \qquad \frac{}{\mathit{id}\ \mathit{expr}}$$

Scoping

- Inference rules to check scoping
  - judgement e ***ok***: e contains no free variables
  - we need to remember which variables are defined in the current context
  - key idea: we use an *environment* to keep track of all bound variables
    - for now, the environment is just a set of variable names
  - composite judgement:
    - $\{x_1, x_2, \ldots, x_n\} \vdash e$ ***ok***
    - assuming the variables x_1 to x_n are bound, e ***ok*** holds

$\{y\} \vdash$ `(Let` y `(` x`. Plus` x y`))` ***ok***

$\{\} \vdash$ `(Plus (Num 1) (Num 3))` ***ok***

$\{x,y,z\} \vdash$ `(Let` y `(` x`. Plus` x y`))` ***ok***

Utrecht University

Scoping

- Inference rules:

$$\frac{}{\Gamma \vdash (\texttt{Num}\ i\ )\ ok}$$

$$\frac{\Gamma \vdash t_1\ ok \quad \Gamma \vdash t_2\ ok}{\Gamma \vdash (\texttt{Plus}\ t_1\ t_2)\ \ ok} \qquad \frac{\Gamma \vdash t_1\ ok \quad \Gamma \vdash t_2\ ok}{\Gamma \vdash (\texttt{Times}\ t_1\ t_2)\ \ ok}$$

$$\frac{\Gamma \vdash t_1\ ok \qquad \{x\} \cup \Gamma \vdash t_2\ ok}{\Gamma \vdash (\texttt{Let}\ \ t_1\ \ (x.t_2))\ \ ok} \qquad \frac{x \in \Gamma}{\Gamma \vdash x\ ok}$$

- Example: `Let (Num 5)` $($x$.($`Plus` x $x$$))$

  `Let (Num 5)` $($x$.($`Plus` x $y$$))$

Dynamic Semantics

- What is dynamic semantics?
  - specifies the program execution process
  - may include side effects and computed values
  - there are various kinds of dynamic semantics
    - denotational
    - operational
    - axiomatic
- Denotational Semantics:
  - Idea: syntactic expressions are mapped to mathematical objects, e.g.,
    - mapping to lambda-calculus
    - fix-point semantics over complete partial orders (CPOs)

Semantics

- Axiomatic Semantics
  - Idea: statements over programs in the form of axioms describing logic program properties
  - Hoare's calculus

Hoare triple

$\{P\}\ prgrm\ \{Q\}$

P: precondition
Q: postcondition

$$\frac{}{\{P[x:=E]\}\ x:=E\ \{P\}}$$

rule for assignment

$$\frac{\{P\}\ s_1\{Q\} \quad \{Q\}\ s_2\{R\}}{\{P\}\ s_1;s_2\{R\}}$$

sequence of statements

  - Dijkstra's Weakest Precondition (WP) calculus

$wp\ (prgm, Q) = P$

What is the weakest precondition P such that after executing prgm, Q holds?

$wp\ (x:=E, R) = R[x:=E]$

rule for assignment

$wp\ (s_1;s_2, R) = wp(s_1,wp(s_2,R))$

sequence of statements

  - Traditionally used for program verification

Utrecht University

Semantics

- Operational Semantics
  - Idea: defines semantics in terms of an abstract machine
    - ‘imaginary’ machine with a set of basic instructions and possible states
    - map program constructs to machine instructions, state transitions
  - There are two main forms:
    - small step semantics or structural operational semantics (SOS): step by step execution of a program
    - big step, natural or evaluation semantics: specifies result of execution of complete programs/subprograms
  - we will be looking at both, small step as well as big step semantics

Structural /Single Step Operational Semantics

Definition: Transition Systems

A ***transition system*** specifies the step-by-step evaluation of a program and consists of

- a set of states S of an abstract computing device
- a set of initial states $I \subseteq S$
- a set of final state $F \subseteq S$, and
- a relation $\mapsto \; \subseteq S \times S$ describing the effect of a single evaluation step on state s

Transition Systems

Back to our arithmetic expression example:

- what should evaluation look like?

Let (Num 5) (x.(Plus x x))

Transition Systems

- States:
  - the set of all well-formed arithmetic expressions

$$\boldsymbol{S} = \{e \mid \exists \Gamma . \Gamma \vdash \boldsymbol{e}\ \boldsymbol{ok}\}$$

- Initial States:
  - the set of all closed, well formed arithmetic expressions:

$$I = \{e \mid \{\} \vdash \boldsymbol{e}\ \boldsymbol{ok}\}$$

- Final States:
  - values

$$F = \{(\texttt{Num}\ \ i) \mid i\ \boldsymbol{Int}\}$$

- Operations of the abstract machines:
  - addition & multiplication
  - substitution

Evaluation Strategy

- We need to fix *an evaluation strategy*

- Example: addition

the machine operation 'addition on integers'

$$\frac{}{\texttt{(Plus (Num } n\texttt{) (Num } m\texttt{))} \mapsto}$$

$$\frac{}{\texttt{(Plus (Num } n\texttt{) } e_2\texttt{)} \mapsto}$$

$$\frac{}{\texttt{(Plus } e_1 \; e_2\texttt{)} \mapsto}$$

multiplication can be defined similarly

Evaluation Strategy

- Evaluating let-expressions

`let` x `=` e_1 `in` e_2 `(Let` e_1 `(`x`.`e_2`))`

Eager or strict evaluation:

- evaluate the right-hand side of binding e_1 to value v
- substitute the value v for the bound variable x, and
- evaluate the body e_2`[`x `:=` v`]`

Lazy evaluation

- substitute expression e_1 for the bound variable x, and
- evaluate the body e_2`[`x `:=` e_1`]`

Evaluation Strategy

Eager or strict evaluation:

$$\frac{}{\texttt{(Let (Num } n\texttt{) (}x.e_2\texttt{))} \mapsto e_2\texttt{[}x \texttt{ := Num } n\texttt{]}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(Let } e_1 \texttt{ (}x.e_2\texttt{))} \mapsto \texttt{(Let } e_1' \texttt{ (}x.e_2\texttt{))}}$$

Lazy evaluation

$$\frac{}{\texttt{(Let } e_1 \texttt{ (}x.e_2\texttt{))} \mapsto e_2\texttt{[}x \texttt{ := } e_1\texttt{]}}$$

Small Step Semantics

- One steps corresponds to finding the left-most subtree that can be simplified by using one of the axioms

$$\frac{}{\texttt{(Plus (Num } n\texttt{) (Num } m\texttt{))} \mapsto \texttt{(Num (}n + m\texttt{))}}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{(Plus (Num } n\texttt{)} \; e_2\texttt{)} \mapsto \texttt{(Plus (Num n)} \; e_2'\texttt{)}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(Plus} \; e_1 \; e_2\texttt{)} \mapsto \texttt{(Plus} \; e_1 \; e_2'\texttt{)}}$$

$$\frac{}{\texttt{(Let (Num } n\texttt{) (}x.e_2\texttt{))} \mapsto e_2 \texttt{[}x \texttt{ := Num } n\texttt{]}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(Let} \; e_1 \; \texttt{(}x.e_2\texttt{))} \mapsto \texttt{(Let} \; e_1' \; \texttt{(}x.e_2\texttt{))}}$$

Small Step Semantics

Definition

An execution sequence $s_0, s_1, \ldots, s_n$

- is ***maximal*** if there is no s_{n+1} such that $s_n \mapsto s_{n+1}$
- is ***complete*** if $s_n \in F$

Transition System

- Stuck States:
  - every complete execution sequence in a system is maximal, but
  - not every maximal sequence is complete. Why?
    - there may be states for which no follow up state exists, but which are not in $\boldsymbol{F}$
    - we call such a state a stuck state
    - stuck states correspond to (non-handled) run-time errors in a program

Transition System

- Type safety (preview):
  - a type-safe language does not have stuck states
    - a stuck state in the abstract machine correspond to undefined behaviour of a program
  - every statically correct program evaluates to a final state
  - we look into type safety in more detail later

Evaluation Semantics or Big Step Semantics

- Small step semantics:
  - specify how each evaluation step alters the state of the machine
- Big step semantics:
  - specify how evaluation of a complex program proceeds based on the evaluation of its components

Evaluation Semantics or Big Step Semantics

- Evaluation relation

also called big step or natural semantics. Consists of

- a set of evaluable expressions $\boldsymbol{E}$
- a set of values $\boldsymbol{V}$ (often, but not necessarily, a subset of $\boldsymbol{E}$),
- basic operations, and
- an *"evaluates to"* relation $\Downarrow \subseteq \boldsymbol{E} \times \boldsymbol{V}$ defined in terms of sub results, and how they combine via the basic operations

Utrecht University

Evaluation Semantics

- Arithmetic expression example (basic operations stay the same)
  - Set of evaluable expressions: $E = \{e \mid \{\} \vdash e\ ok\}$
  - Set of values: $V = \{(\texttt{Num}\ i) \mid i\ Int\}$

$$\frac{}{(\texttt{Num}\ n) \Downarrow (\texttt{Num}\ n)}$$

$$\frac{e_1 \Downarrow (\texttt{Num}\ n_1) \qquad e_2 \Downarrow (\texttt{Num}\ n_2)}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (\texttt{Num}(n_1+n_2))}$$

$$\frac{e_1 \Downarrow (\texttt{Num}\ n_1) \qquad e_2 \Downarrow (\texttt{Num}\ n_2)}{(\texttt{Times}\ e_1\ e_2) \Downarrow (\texttt{Num}(n_1 * n_2))}$$

$$\frac{e_1 \Downarrow (\texttt{Num}\ n_1) \quad e_2\ [x := (\texttt{Num}\ n)] \Downarrow (\texttt{Num}\ n_2)}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow (\texttt{Num}\ n_2)}$$

or if we choose lazy evaluation:

$$\frac{e_2\ [x := e_1] \Downarrow (\texttt{Num}\ n)}{(\texttt{Let}\ (\texttt{Num}\ n)\ (x.\ e_2)) \Downarrow (\texttt{Num}\ n)}$$

Utrecht University

Evaluation Semantics

`P (P  (T 5 3) 6) (T 2 4)` $\Downarrow$?

Relating SOS and Evaluation Semantics

- Small step vs bis step
  - two different ways of specifying the operational semantics of a language
  - small step provides more detail
    - ★ order of evaluation beyond data dependency (but this is not always necessary)
    - ★ necessary to model concepts like explicit concurrency
  - big step semantics
    - ★ like a recursive interpreter
    - ★ more compact in general
    - ★ only provides information about terminating evaluations!

Relating SOS and Evaluation Semantics

- Small step vs bis step
  - are both definitions equivalent for our example?
    - is $e \mapsto^! e'$ if and only if $e \Downarrow e'$?

Relating SOS and Evaluation Semantics

- Which cases do we need to consider to show that
  - $e \Downarrow$ (Num n) implies $e \mapsto^!$ (Num n) ?

(1) $e =$ (Num n)

[G] $e \mapsto^!$ (Num n)

$$\frac{e_1 \Downarrow (\texttt{Num}\ n_1) \qquad e_2 \Downarrow (\texttt{Num}\ n_2)}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (\texttt{Num}(n_1+n_2))}$$

(2) $e =$ (Plus e_1 e_2) with (Plus e_1 e_2) $\Downarrow$ (Num (n_1+n_2))

[A1] $e_1 \Downarrow$ (Num n_1)

[A2] $e_2 \Downarrow$ (Num n_2)

[IH1] $e_1 \mapsto^!$ (Num n_1)

[IH2] $e_2 \mapsto^!$ (Num n_2)

[G] (Plus e_1 e_2) $\mapsto^!$ (Num (n_1+n_2))

Utrecht University

Relating SOS and Evaluation Semantics

- Which cases do we need to consider to show that

  - $e \Downarrow$ `(Num` n`)` implies $e \mapsto^{!}$ `(Num` n`)` ?

(1) $e =$ `(Num` n`)`

holds since $e \mapsto^{0}$ `(Num` n`)`

$$\frac{e_1 \Downarrow \texttt{(Num } n_1\texttt{)} \qquad e_2 \Downarrow \texttt{(Num } n_2\texttt{)}}{\texttt{(Plus } e_1\ e_2\texttt{)} \Downarrow \texttt{(Num(}n_1 + n_2\texttt{))}}$$

(2) $e =$ `(Plus` e_1 e_2`)` with `(Plus` e_1 e_2`)` $\Downarrow$ `(Num` n`)`

- A1: $e_1 \Downarrow$ `(Num` n_1`)`
- A1: $e_2 \Downarrow$ `(Num` n_2`)`, $n_1 + n_2 = n$
- I.H.-1: $e_1 \mapsto^{!}$ `(Num` n_1`)`
- I.H.-2: $e_2 \mapsto^{!}$ `(Num` n_2`)`

Relating SOS and Evaluation Semantics

(2) $e =$ `(Plus` e_1 e_2`)` with `(Plus` e_1 e_2`)` $\Downarrow$ `(Num` n`)`, $n_1 + n_2 = n$

- A1: $e_1 \Downarrow$ `(Num` n_1`)`
- A1: $e_2 \Downarrow$ `(Num` n_2`)`, $n_1 + n_2 = n$
- I.H.-1: $e_1 \mapsto^!$ `(Num` n_1`)`
- I.H.-2: $e_2 \mapsto^!$ `(Num` n_2`)`

`(Plus` e_1 e_2`)`

$\mapsto^*$ A1,I.H.-1, def. of $\mapsto^*$

`(Plus (Num` n_1`)` e_2`)`

$\mapsto^*$ A2, I.H.-2, def. of $\mapsto^*$

`(Plus (Num` n_1`) (Num` n_2`))`

$\mapsto$ def. of $\mapsto$

`(Num (`$n_1 + n_2$`))`