CS 456

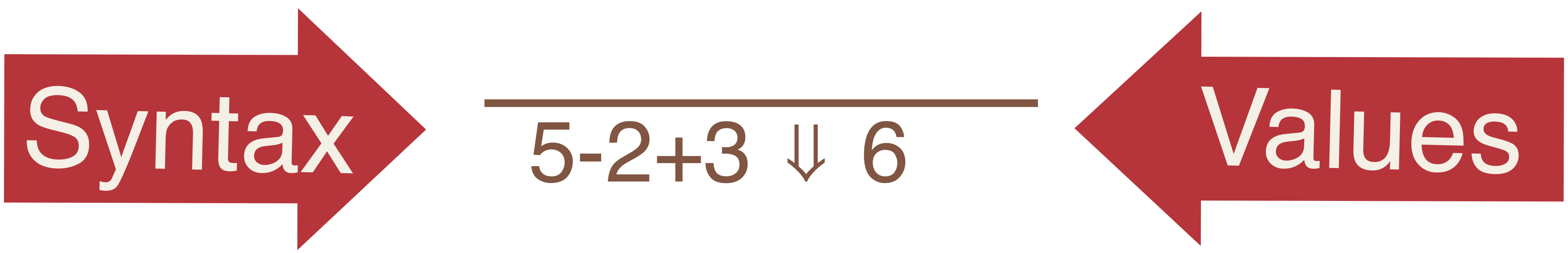
Programming Languages
Fall 2024

Week 11

Smallstep and Denotational Semantics

Big-Step Semantics

- Binary relation on pairs of syntax and values
- Read '⇓' as 'evaluates to'
- Specifies what values program can map to



- Good for whole program reasoning
  - Compiler Correctness; program equivalence;
- Bad for talking about intermediate states
  - Concurrent programs; errors

Concurrent Imp

Consider Imp with a fork operator

```
C ::= c1;c2
    | if b then ct else ce fi
    | skip
    | x := a
    | while b do c end
    | par c1 with c2 end
```

Imp Program


```
C := skip
   | x := A
   | C ; C
   | if B then C
          else C end
   | while B do C end
   | par C with C end
```

```
par
  X := 2;
  Y := 4
with
  Z := 5;
  W := 6
end
```

Small-Step

- Binary relation on pairs of expressions
- Read '$e_1 \longrightarrow e_2$' as 'reduces to'
- Specifies single transition of abstract machine
- Exposes intermediate states

Small-Step

Consider toy language:

$$E ::= C\ \mathbb{N} \mid E +_E E$$

Big-Step Semantics

$$\frac{e_n \Downarrow n \qquad e_m \Downarrow m}{e_n +_E e_m \Downarrow n + m}$$

$$\frac{}{C\ n \Downarrow n}$$

Small-Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

$$e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots \longrightarrow e_n$$

Small-Step

Consider toy language:

$$E ::= C\ \mathbb{N} \mid E +_E E$$

(C 1)+((C 2)+(C 3))+((C 4)+(C 6)))
$\longrightarrow$
(C 1)+(C 5)+((C 4)+(C 6)))
$\longrightarrow$
(C 1)+((C 5)+(C 10))
$\longrightarrow$
(C 1)+(C 15)
$\longrightarrow$
C 16

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

$$e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots \longrightarrow e_n$$

Small-Step

Consider toy language:

$$E ::= C\ \mathbb{N} \mid E +_E E$$

Two "flavors" of rule:

(Boring) Congruence rules

Interesting rules

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

$$e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots \longrightarrow e_n$$

Step Size

Big Step Semantics

$$\frac{e_n \Downarrow n \qquad e_m \Downarrow m}{e_n +_E e_m \Downarrow n + m}$$

$$\frac{}{C\ n \Downarrow n}$$

Big-Step reduction relation is from syntax, to values.

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Small-Step reduction relation is from syntax, to syntax.

Small-Step Termination

- How to tell when we're 'done' evaluating?
- Define a class of syntactic values:

$$\frac{}{\textbf{value}\ C\, n}$$

Now we can talk about making progress

Theorem [STRONG PROGRESS]:

For any term t, either t is a value or there exists a term t' such that t $\longrightarrow$ t'.

Small-Step Termination

- How to tell when we're 'done' evaluating?
- Another style of defining values:

$$v ::= C\ n$$

$$\frac{e_m \longrightarrow e_m'}{v +_E e_m \longrightarrow v +_E e_m'}$$

Example

How many steps does this program need to reach a value?

(C 10)+((C 12) + (C 23))

- 0
- 1
- 2
- 3

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Concept Check

How many steps does this program need to reach a value?

C 10

- 0
- 1
- 2
- 3

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Normal Form

A term e that isn't reducible is in normal form.

$$\neg \exists e'.\ e \longrightarrow e'$$

How is this different from a value?

Syntactic versus semantic.

Do not need to coincide!

Example

How might we change the reduction rules so that there are normal forms that aren't values?

$$\frac{}{\textbf{value}\ C\ n}$$

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

MultiStep Relation

We generically lift single-step to full execution as the *transitive*, *reflexive* closure:

$$\frac{}{e \longrightarrow^{*} e}\ \textsc{Refl} \qquad \frac{e_1 \longrightarrow^{*} e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow^{*} e_3}\ \textsc{Trans}$$

So: (C 1)+((C 2) + (C 3))+((C 4)+(C 6))) $\longrightarrow^{*}$ 16:

1+((2+3)+(4+6)) $\longrightarrow$ 1+(5+(4+6)) $\longrightarrow$ 1+(5+10)
$\longrightarrow$ 6+10 $\longrightarrow$ 16

Evaluation Order

Small step semantics give control over the order in which terms are reduced

$$\frac{e_m \longrightarrow e_m'}{e_n + e_m \longrightarrow e_n + e_m'}$$

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Evaluation orders can be **deterministic** or **nondeterministic**

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

$$\frac{\sigma, a_1 \Downarrow v}{\sigma, x := a_1 \longrightarrow [x \mapsto v]\sigma, \text{skip}} \quad \underline{\text{CSASSN}}$$

$$\frac{\sigma_1, c_1 \longrightarrow \sigma_2, c_3}{\sigma_1, c_1; c_2 \longrightarrow \sigma_2, c_3; c_2} \quad \underline{\text{CSSEQSTEP}}$$

$$\frac{}{\sigma, \text{skip}; c_2 \longrightarrow \sigma, c_2} \quad \underline{\text{CSSEQSKIP}}$$

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

Reduction Rules

$$\frac{}{\sigma, \textbf{if}\ \text{true}\ \textbf{then}\ c_t\ \textbf{else}\ c_f\ \textbf{end} \longrightarrow \sigma, c_t}\ \text{CSIfT}$$

$$\frac{}{\sigma, \textbf{if}\ \text{false}\ \textbf{then}\ c_t\ \textbf{else}\ c_f\ \textbf{end} \longrightarrow \sigma, c_f}\ \text{CSIfF}$$

Congruence Rules

$$\frac{\sigma, b_1 \longrightarrow_B b_2}{\sigma, \textbf{if}\ b_1\ \textbf{then}\ c_t\ \textbf{else}\ c_f\ \textbf{end} \longrightarrow \sigma, \textbf{if}\ b_2\ \textbf{then}\ c_t\ \textbf{else}\ c_f\ \textbf{end}}\ \text{CSIfStep}$$

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

CSWHILE

$$\frac{}{\sigma, \textbf{while } b \textbf{ do } c \longrightarrow \sigma, \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ end} \textbf{ else } \text{skip} \textbf{ end}}$$

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

CSParL

$$\frac{\sigma_1, c_1 \longrightarrow \sigma_2, c_3}{\sigma_1, \textbf{par } c_1 \textbf{ with } c_2 \longrightarrow \sigma_2, \textbf{par } c_3 \textbf{ with } c_2}$$

CSParR

$$\frac{\sigma_1, c_2 \longrightarrow \sigma_2, c_3}{\sigma_1, \textbf{par } c_1 \textbf{ with } c_2 \longrightarrow \sigma_2, \textbf{par } c_1 \textbf{ with } c_3}$$

CSParFinish

$$\frac{}{\sigma, \textbf{par } \text{skip } \textbf{with } \text{skip} \longrightarrow \sigma, \text{skip}}$$

Imp Program


```
C := skip
   | x := A
   | C ; C
   | if B then C
          else C end
   | while B do C end
   | par C with C end
```

σ,

```
par
  X := 2;
  Y := 4
with
  X := 5;
  Y := 6
end
```

→ [X↦2][Y↦4]σ, skip

→ [X↦5][Y↦4]σ, skip

Imp Program


```
while true do
  skip
end
```

⟶

```
if true then
  while true do
    skip
  end
else skip
```

⟶

```
while true do
  skip
end
```

Summary

To recap smallstep operational semantics:

These rules form an abstract reference 'implementation' for a language, entirely at the level of the language itself

You can validate a particular implementation of a language against this specification (or prove that it meets it)

The rules are declarative: they don't necessarily prescribe a specific evaluation order, or even how to implement each step

This approach allows us to reason about properties of the language, e.g. type safety, irrespective of an implementation

Semantics Recap

- We've considered several flavors of Operational Semantics:
  - Abstract machine specifies *how* an expression is executed:
- σ, c $\Downarrow$ σ' reads as 'when run in initial state σ, c produces (i.e. evaluates to) final state σ''
- $e_1 \longrightarrow e_2$ reads as 'e_1 reduces to e_2 in a single step'
- $e_1 \longrightarrow^* e_2$ reads as 'e_1 reduces to e_2 in zero or more steps'

Denotational Semantics

Key Idea: 'Denotation' function translates source program to target mathematical object

- Define a **sem**… language T
- Deno… this domain:
- Denota… am means
- Abstrac… reasoning
  - Natur… program equivalence
- Finding domain can be tricky— Domain Theory

Important: ⟦·⟧ is a total function— every program gets a meaning!

Denotational Semantics

Key Idea: 'Denotation' function translates source program to target mathematical object

$$[\![\cdot]\!]_A : A \rightarrow \mathbb{N}$$

Domain is $\mathbb{N}$

$$[\![n]\!]_A \equiv n$$

$$[\![n+m]\!]_A \equiv [\![n]\!]_A +_{\mathbb{N}} [\![m]\!]_A$$

$$[\![n-m]\!]_A \equiv [\![n]\!]_A -_{\mathbb{N}} [\![m]\!]_A$$

$$[\![n*m]\!]_A \equiv [\![n]\!]_A *_{\mathbb{N}} [\![m]\!]_A$$

Denotational Semantics

Key Idea: 'Denotation' function translates source program to target mathematical object

$$[\![\cdot]\!]_A : A \rightarrow \mathbb{N}$$

$$[\![n]\!]_A \equiv [\![n]\!]_A +_{\mathbb{N}} [\![m]\!]_A$$

$$[\![x]\!]_A \equiv ??$$

$$[\![n+m]\!]_A \equiv [\![n]\!]_A +_{\mathbb{N}} [\![m]\!]_A$$

$$[\![n-m]\!]_A \equiv [\![n]\!]_A -_{\mathbb{N}} [\![m]\!]_A$$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$$[\![\cdot]\!]_A : A \to \mathcal{P}((\mathrm{Id} \to \mathbb{N}) \times \mathbb{N})$$

$$[\![n]\!]_A \equiv \{(\sigma, n)\}$$

$$[\![x]\!]_A \equiv \{(\sigma, \sigma(x))\}$$

$$[\![e_n+e_m]\!]_A \equiv \{(\sigma, v_n +_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

$$[\![e_n-e_n]\!]_A \equiv \{(\sigma, v_n -_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

Denotational Semantics

$$[\![\cdot]\!]_A : A \to \mathcal{P}((\mathrm{Id} \to \mathbb{N}) \times \mathbb{N})$$

$$[\![n]\!]_A \equiv \{(\sigma, n)\}$$

$$[\![x]\!]_A \equiv \{(\sigma, \sigma(x))\}$$

$$[\![e_n + e_m]\!]_A \equiv \{(\sigma, v_n +_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

$$[\![e_n - e_n]\!]_A \equiv \{(\sigma, v_n -_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

- $[\![4]\!]_A \ni (\sigma, 4)$
- $[\![x]\!]_A \ni ([x \mapsto 4]\sigma, 4)$
- $[\![x]\!]_A \ni ([x \mapsto 6]\sigma, 6)$
- $[\![4 + x]\!]_A \ni ([x \mapsto 6]\sigma, 10)$
- $[\![x + 4]\!]_A \ni ([x \mapsto 6]\sigma, 10)$

Concept Check

Which of the following claims are true?

$[\![\cdot]\!]_A : A \to \mathcal{P}((Id \to \mathbb{N})$
$\times\ \mathbb{N})$

$[\![n]\!]_A \equiv \{(\sigma, n)\}$

$[\![x]\!]_A \equiv \{(\sigma, \sigma(x))\}$

$[\![e_n+e_m]\!]_A \equiv \{(\sigma, v_n +_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$

$[\![e_n-e_n]\!]_A \equiv \{(\sigma, v_n -_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$

1. $[\![4+15]\!]_A \ni (\sigma,19)$
2. $[\![4+x+y]\!]_A \ni ([y \mapsto 24][x \mapsto 6]\sigma,30)$
3. $[\![5+x+4-y]\!]_A \ni (\sigma, 9+\sigma(x)-\sigma(y))$

Nondeterminism

This semantic domain can also model nondeterministic semantics:

$$[\![\cdot]\!]_A : A \to \mathcal{P}((Id \to \mathbb{N}) \times \mathbb{N})$$

$$[\![rand]\!]_A \equiv \{(\sigma, m)\}$$

$$[\![n]\!]_A \equiv \{(\sigma, n)\}$$

$$[\![x]\!]_A \equiv \{(\sigma, \sigma(x))\}$$

$$[\![e_n+e_m]\!]_A \equiv \{(\sigma, v_n +_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

$$[\![e_n-e_n]\!]_A \equiv \{(\sigma, v_n -_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

Partial Programs

This semantic domain can also define the meaning of ill-formed programs:

$$[\![\cdot]\!]_A : A \rightarrow \mathcal{P}((Id \rightarrow \mathbb{N}) \times \mathbb{N})$$

$$[\![e_n / e_m]\!]_A \equiv \{(\sigma, k) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\} \wedge v_m * k = v_n\}$$

$$[\![n]\!]_A \equiv \{(\sigma, n)\}$$

$$[\![x]\!]_A \equiv \{(\sigma, \sigma(x))\}$$

$$[\![e_n + e_m]\!]_A \equiv \{(\sigma, v_n +_{\mathbb{N}} v_m) \mid (\sigma, v_n) \in [\![e_n]\!]_A \wedge (\sigma, v_m) \in [\![e_m]\!]_A\}$$

Reasoning

- Denotational semantics have several nice properties which make them nice to reason with:
  - They are **compositional**: the denotation of a term is a function of the denotations of its subterms:

    $$\text{If } [\![e_n]\!]_A = [\![e_o]\!]_A, \text{ then } [\![e_n + e_m]\!]_A = [\![e_o + e_m]\!]_A$$

  - They inherit properties of the semantic domain:

    $$[\![e_n + e_m]\!]_A = [\![e_m + e_n]\!]_A$$

Reasoning

- Two programs are **semantically equivalent** if they have the same denotation
- Can show that if $[\![e_n]\!]_A$ is semantically equivalent to $[\![e_o]\!]_A$, then $[\![e_n+e_m]\!]_A$ and $[\![e_o+e_m]\!]_A$ are also semantically equivalent:

$$
\begin{aligned}
[\![e_n+e_m]\!]_A &= \{(\sigma, v_1 +_{\mathbb{N}} v_2) \mid (\sigma, v_1) \in [\![e_n]\!]_A \land (\sigma, v_2) \in [\![e_m]\!]_A\} \\
&= \{(\sigma, v_1 +_{\mathbb{N}} v_2) \mid (\sigma, v_1) \in [\![e_o]\!]_A \land (\sigma, v_2) \in [\![e_m]\!]_A\} \\
&= [\![e_o+e_m]\!]_A
\end{aligned}
$$

Recap

- <u>Key Idea</u>: define semantics via translation to a well-understood **semantic domain:**
  - Using sets, we can model partial and total functions on state
  - Can also represent nondeterministic semantics
- Can relate different kinds of semantics
- Denotational semantics are designed to be **compositional**
- Denotational semantics are useful for reasoning about program equivalence

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$[\![\cdot]\!]_C$:

$[\![\texttt{skip}]\!]_C \equiv$

$[\![\texttt{x:=a}]\!]_C \equiv$

$[\![\texttt{c}_1\texttt{;c}_2]\!]_C \equiv$

$[\![\textbf{if}\ \texttt{b}\ \textbf{then}\ \texttt{c}_t\ \textbf{else}\ \texttt{c}_e]\!]_C \equiv$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$[\![\cdot]\!]_C : \mathtt{C} \to \mathcal{P}((\mathtt{Id} \to \mathbb{N}) \times (\mathtt{Id} \to \mathbb{N}))$

$[\![\mathtt{skip}]\!]_C \equiv \{(\sigma, \sigma)\}$

$[\![\mathtt{x:=a}]\!]_C \equiv$

$[\![\mathtt{c}_1\mathtt{;c}_2]\!]_C \equiv$

$[\![\mathbf{if}\ \mathtt{b}\ \mathbf{then}\ \mathtt{c}_t\ \mathbf{else}\ \mathtt{c}_e]\!]_C \equiv$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$[\![\cdot]\!]_C : \mathtt{C} \to \mathcal{P}((\mathtt{Id} \to \mathbb{N}) \times (\mathtt{Id} \to \mathbb{N}))$

$[\![\mathtt{skip}]\!]_C \equiv \{(\sigma, \sigma)\}$

$[\![\mathtt{x:=a}]\!]_C \equiv \{(\sigma, [\mathtt{x} \mapsto \mathtt{v}]\sigma) \mid (\sigma, \mathtt{v}) \in [\![\mathtt{a}]\!]_A\}$

$[\![\mathtt{c}_1\mathtt{;c}_2]\!]_C \equiv$

$[\![\mathbf{if}\ \mathtt{b}\ \mathbf{then}\ \mathtt{c}_t\ \mathbf{else}\ \mathtt{c}_e]\!]_C \equiv$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$[\![\cdot]\!]_C : C \rightarrow \mathcal{P}((Id \rightarrow \mathbb{N}) \times (Id \rightarrow \mathbb{N}))$

$[\![skip]\!]_C \equiv \{(\sigma, \sigma)\}$

$[\![x := a]\!]_C \equiv \{(\sigma, [x \mapsto v]\sigma) \mid (\sigma, v) \in [\![a]\!]_A\}$

$[\![c_1 ; c_2]\!]_C \equiv \{(\sigma_1, \sigma_3) \mid \exists \sigma_2.\ (\sigma_1, \sigma_2) \in [\![c_1]\!]_C \wedge (\sigma_2, \sigma_3) \in [\![c_2]\!]_C\}$

$[\![\textbf{if}\ b\ \textbf{then}\ c_t\ \textbf{else}\ c_e]\!]_C \equiv$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$$[\![\cdot]\!]_C : C \to \mathcal{P}((Id \to \mathbb{N}) \times (Id \to \mathbb{N}))$$

$$[\![skip]\!]_C \equiv \{(\sigma, \sigma)\}$$

$$[\![x:=a]\!]_C \equiv \{(\sigma, [x \mapsto v]\sigma) \mid (\sigma, v) \in [\![a]\!]_A \}$$

$$[\![c_1;c_2]\!]_C \equiv \{(\sigma_1, \sigma_3) \mid \exists \sigma_2.\ (\sigma_1, \sigma_2) \in [\![c_1]\!]_C \wedge (\sigma_2, \sigma_3) \in [\![c_2]\!]_C\}$$

$$[\![\textbf{if } b \textbf{ then } c_t \textbf{ else } c_e]\!]_C \equiv$$

$$\{(\sigma_1, \sigma_2) \mid (\sigma_1, true) \in [\![e_B]\!]_B \wedge (\sigma_1, \sigma_2) \in [\![c_t]\!]_C \}$$
$$\cup \{(\sigma_1, \sigma_2) \mid (\sigma_1, false) \in [\![e_B]\!]_B \wedge (\sigma_1, \sigma_2) \in [\![c_e]\!]_C\}$$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$[\![\cdot]\!]_C : \mathtt{C} \to \mathcal{P}((\mathtt{Id} \to \mathbb{N}) \times (\mathtt{Id} \to \mathbb{N}))$

$[\![\mathtt{while\ b\ do\ c\ end}]\!]_C \equiv$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$$[\![\cdot]\!]_C : C \to \mathcal{P}((\mathrm{Id} \to \mathbb{N}) \times (\mathrm{Id} \to \mathbb{N}))$$

$$[\![\texttt{while b do c end}]\!]_C \equiv$$

$$\{(\sigma, \sigma) \mid (\sigma, \texttt{false}) \in [\![e_B]\!]_B\}$$
$$\cup \{(\sigma_1, \sigma_3) \mid (\sigma_1, \texttt{true}) \in [\![e_B]\!]_B \wedge \exists \sigma_2.\ (\sigma_1, \sigma_2) \in [\![c]\!]_C$$
$$\wedge\ (\sigma_2, \sigma_3) \in [\![\texttt{while b do c end}]\!]_C\}$$

?

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$$[\![\cdot]\!]_C : C \rightarrow \mathcal{P}((\mathtt{Id} \rightarrow \mathbb{N}) \times (\mathtt{Id} \rightarrow \mathbb{N}))$$

$$[\![\mathtt{while\ b\ do\ c\ end}]\!]_C =$$

$$\{(\sigma, \sigma) \mid (\sigma, \mathtt{false}) \in [\![e_B]\!]_B\}$$
$$\cup \{(\sigma_1, \sigma_3) \mid (\sigma_1, \mathtt{true}) \in [\![e_B]\!]_B \wedge \exists \sigma_2.\ (\sigma_1, \sigma_2) \in [\![c]\!]_C$$
$$\wedge\ (\sigma_2, \sigma_3) \in [\![\mathtt{while\ b\ do\ c\ end}]\!]_C\}$$

- The meaning of while is defined in terms of the meaning of while
- This is not a *definition*, it is a *recursive equation*
- Goal: find a **set** that satisfies this equation

Recursive Equations

Is there a function f that satisfies these constraints:

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$

Is there a function g that satisfies these constraints:

$$g(x) = g(x) + 1$$

Recursive Equations

Can build up an approximation of the solution iteratively

$$f_0 = \varnothing$$

$$f_1 = \begin{array}{l}\{(0,0)\} \cup \\ \{(x, m + 2x - 1) \mid (x - 1, m) \in f_0\}\end{array} = \{(0,0)\}$$

$$f_2 = \begin{array}{l}\{(0,0)\} \cup \\ \{(x, m + 2x - 1) \mid (x - 1, m) \in f_1\}\end{array} = \{(0,0), (1,1)\}$$

$$f_3 = \begin{array}{l}\{(0,0)\} \cup \\ \{(x, m + 2x - 1) \mid (x - 1, m) \in f_2\}\end{array} = \begin{array}{l}\{(0,0), (1,1), \\ (2,4)\}\end{array}$$

Fixpoints

- A **fixpoint** is solution Fix_F to a recursive equation of the form:

$$Fix_F = F\,(Fix_F)$$

$$\text{where} \quad F : \mathcal{P}A \rightarrow \mathcal{P}A$$

- A fixpoint is also a solution to this sequence:

$$Fix_F = F^0(\emptyset) \cup F^1(\emptyset) \cup F^2(\emptyset) \cup F^3(\emptyset) \cup \ldots$$

Denotational Semantics

Key idea: represent all terminating iterations of the loop as a fixpoint using the denotation of its body:

$$[\![\cdot]\!]_C : C \rightarrow \mathcal{P}((Id \rightarrow \mathbb{N}) \times (Id \rightarrow \mathbb{N}))$$

$$[\![\texttt{while b do c end}]\!]_C \equiv \texttt{Fix}_F$$

where

$$\begin{aligned} F(rec) = \{(\sigma, \sigma) \mid (\sigma, false) \in [\![b]\!]_B\} \\ \cup \{(\sigma_1, \sigma_3) \mid \exists \sigma_1.(\sigma_1, true) \in [\![b]\!]_B \\ \wedge (\sigma_1, \sigma_2) \in [\![c]\!]_c \\ \wedge (\sigma_2, \sigma_3) \in rec\} \end{aligned}$$

How do we know that a fixpoint exists for any function or loop?