Principles of Programming Languages
Lecture 5: Semantics

Andrei Arusoaie[1]

[1] Department of Computer Science

October 26, 2021

Outline

Semantics: introduction

An evaluator for IMP

Semantics: motivation

C

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 0
```

Java

```
-$ cat File.java
public class File {
  ... void main(...) {
    int x = 0;
    println((x=1) + (x=2));
  }
}
-$ javac File.java
-$ java File
 3
```

Semantics: motivation

C

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 0
```

Java

```
-$ cat File.java
public class File {
  ... void main(...) {
    int x = 0;
    println((x=1) + (x=2));
  }
}
-$ javac File.java
-$ java File
 3
```

Semantics: motivation

C

```
~$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
 0
```

Java

```
~$ cat File.java
public class File {
  ... void main(...) {
    int x = 0;
    println((x=1) + (x=2));
   }
}
~$ javac File.java
~$ java File
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

GCC: Apple clang 13.0.0

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
~$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
  4
```

GCC: Apple clang 13.0.0

```
~$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
  3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

GCC: Apple clang 13.0.0

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

GCC: Apple clang 13.0.0

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 3
```

Semantics

- Semantics is concerned with the meaning of language constructs
- Semantics must be unambiguous
- Semantics must be flexible

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- "Trust the programmer"
- "Don't prevent the programmer from doing what needs to be done"
- "Keep the language small and simple"
- "Provide only one way to do an operation"
- "Make it fast, even if it is not guaranteed to be portable"

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- "Trust the programmer"
- "Don't prevent the programmer from doing what needs to be done"
- "Keep the language small and simple"
- "Provide only one way to do an operation"
- "Make it fast, even if it is not guaranteed to be portable"

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- "Trust the programmer"
- "Don't prevent the programmer from doing what needs to be done"
- "Keep the language small and simple"
- "Provide only one way to do an operation"
- "Make it fast, even if it is not guaranteed to be portable"

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- "Trust the programmer"
- "Don't prevent the programmer from doing what needs to be done"
- "Keep the language small and simple"
- "Provide only one way to do an operation"
- "Make it fast, even if it is not guaranteed to be portable"

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- "Trust the programmer"
- "Don't prevent the programmer from doing what needs to be done"
- "Keep the language small and simple"
- "Provide only one way to do an operation"
- "Make it fast, even if it is not guaranteed to be portable"

Inexact! – could lead to undefined behavior in programs

Formal Semantics

Some (formal) semantics styles:

- operational
- denotational
- axiomatic

We will focus more on operational semantics styles: Small-step SOS, Big-Step SOS

Formal Semantics

Some (formal) semantics styles:

- operational
- denotational
- axiomatic

We will focus more on operational semantics styles: Small-step SOS, Big-Step SOS

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
     sum ::= sum +’ i;;
     i ::= i +’ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
     sum ::= sum +′ i;;
     i ::= i +′ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
     sum ::= sum +′ i;;
     i ::= i +′ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
     sum ::= sum +′ i;;
     i ::= i +′ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
      sum ::= sum +′ i;;
      i ::= i +′ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
    sum ::= sum +' i;;
    i ::= i +' 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
     sum ::= sum +′ i;;
     i ::= i +′ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
    sum ::= sum +′ i;;
    i ::= i +′ 1
end
```

Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

A simple imperative language

Program:

```
n ::= 10;;
i ::= 1;;
sum ::= 0;;
while (i «= n) do
     sum ::= sum +′ i;;
     i ::= i +′ 1
end
```

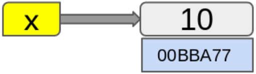
Features:

- variables
- arithmetic expressions
- boolean expressions
- assignment statements
- loop statements
- decisional statements

How can we define the semantics of this language?

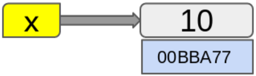
variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- $var\_name \mapsto value$
- Environment + memory



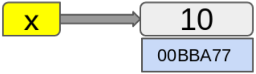
variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- *var_name* $\mapsto$ *value*
- Environment + memory



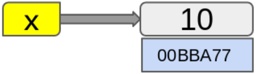
variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- $var\_name \mapsto value$
- Environment + memory



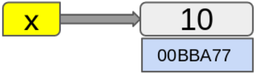
variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- *var_name* $\mapsto$ *value*
- Environment + memory



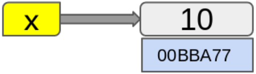
variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- $var\_name \mapsto value$
- Environment + memory



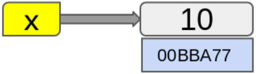
variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- $var\_name \mapsto value$
- Environment + memory



variables

- variable = *storage location* at *address* + *symbolic name*
- variables can be *accessed* or *modified*
- depending on the language, they may only be able to store a specified datatype (e.g., integer, string, etc.)
- *Scope*: global, local
- Lifetime (*extent*)
- $var\_name \mapsto value$
- Environment + memory



Environment

- Environment: variables mapped to values

```
Definition Env := string -> nat.
Definition env (string : string) :=
if (string_dec string "n")
                then 10
                else 0.
Compute (env n).
= 10
: nat
Compute (env i).
= 0
: nat
Check env.
env
     : string -> nat
```

- The environment is a function!

Environment update

- Update = a new function s.t. the value for `x` is updated to `v`:

```
Definition update (x : string)(v : nat)(env : Env) : Env :=
  fun y => if (string_dec x y)
             then v
             else env y.
```

Arithmetic expressions

- Syntax:

  BNF:

AExp	::=	$\mathbb{N}$
	|	AExp '+' AExp
	|	AExp '*' AExp

  ```
  Inductive AExp :=
  | anum : nat -> AExp
  | aplus : AExp -> AExp -> AExp
  | amul : AExp -> AExp -> AExp.
  ```

- Semantics:

  ```
  Fixpoint aeval (a : AExp) : nat :=
  match a with
  | anum v => v
  | aplus a1 a2 => (aeval a1) + (aeval a2)
  | amul a1 a2 => (aeval a1) * (aeval a2)
  end.
  ```

- What about expressions in a PL: `i ::= i +' 1;;` ?

Arithmetic expressions

- Syntax:

  BNF:

AExp	::=	$\mathbb{N}$
	|	AExp ‘+’ AExp
	|	AExp ‘*’ AExp

  ```
  Inductive AExp :=
  | anum : nat -> AExp
  | aplus : AExp -> AExp -> AExp
  | amul : AExp -> AExp -> AExp.
  ```

- Semantics:

  ```
  Fixpoint aeval (a : AExp) : nat :=
  match a with
  | anum v => v
  | aplus a1 a2 => (aeval a1) + (aeval a2)
  | amul a1 a2 => (aeval a1) * (aeval a2)
  end.
  ```

- What about expressions in a PL: `i ::= i +' 1;;` ?

Arithmetic expressions

- Syntax:

BNF:

$$\begin{array}{lcl} \text{AExp} & ::= & \mathbb{N} \\ & | & \text{AExp '+' AExp} \\ & | & \text{AExp '*' AExp} \end{array}$$

```
Inductive AExp :=
| anum : nat -> AExp
| aplus : AExp -> AExp -> AExp
| amul : AExp -> AExp -> AExp.
```

- Semantics:

```
Fixpoint aeval (a : AExp) : nat :=
match a with
| anum v => v
| aplus a1 a2 => (aeval a1) + (aeval a2)
| amul a1 a2 => (aeval a1) * (aeval a2)
end.
```

- What about expressions in a PL: `i ::= i +' 1;;` ?

Arithmetic expressions

- Syntax:

  BNF:

  AExp ::= $\mathbb{N}$
  | AExp '+' AExp
  | AExp '*' AExp

  ```
  Inductive AExp :=
  | anum : nat -> AExp
  | aplus : AExp -> AExp -> AExp
  | amul : AExp -> AExp -> AExp.
  ```

- Semantics:

  ```
  Fixpoint aeval (a : AExp) : nat :=
  match a with
  | anum v => v
  | aplus a1 a2 => (aeval a1) + (aeval a2)
  | amul a1 a2 => (aeval a1) * (aeval a2)
  end.
  ```

- What about expressions in a PL: `i ::= i +' 1;;` ?

Arithmetic expressions with variables

- Syntax:

BNF:

AExp	::=	string
	|	$\mathbb{N}$
	|	AExp '+' AExp
	|	AExp '*' AExp

```
Inductive AExp :=
| var : string -> AExp
| anum : nat -> AExp
| aplus : AExp -> AExp -> AExp
| amul : AExp -> AExp -> AExp.
```

- Semantics:

```
Fixpoint aeval (a : AExp) : nat :=
match a with
| var v => ???
| anum v => v
| aplus a1 a2 => (aeval a1) + (aeval a2)
| amul a1 a2 => (aeval a1) * (aeval a2)
end.
```

- We need an environment to evaluate expressions with variables!

Arithmetic expressions with variables

- Syntax:

BNF:

AExp	::=	string
	|	$\mathbb{N}$
	|	AExp '+' AExp
	|	AExp '*' AExp

```
Inductive AExp :=
| var : string -> AExp
| anum : nat -> AExp
| aplus : AExp -> AExp -> AExp
| amul : AExp -> AExp -> AExp.
```

- Semantics:

```
Fixpoint aeval (a : AExp) : nat :=
match a with
| var v => ???
| anum v => v
| aplus a1 a2 => (aeval a1) + (aeval a2)
| amul a1 a2 => (aeval a1) * (aeval a2)
end.
```

- We need an environment to evaluate expressions with variables!

Arithmetic expressions with variables

- Syntax:

BNF:

AExp	::=	string
	|	$\mathbb{N}$
	|	AExp '+' AExp
	|	AExp '*' AExp

```
Inductive AExp :=
| var : string -> AExp
| anum : nat -> AExp
| aplus : AExp -> AExp -> AExp
| amul : AExp -> AExp -> AExp.
```

- Semantics:

```
Fixpoint aeval (a : AExp) : nat :=
match a with
| var v => ???
| anum v => v
| aplus a1 a2 => (aeval a1) + (aeval a2)
| amul a1 a2 => (aeval a1) * (aeval a2)
end.
```

- We need an environment to evaluate expressions with variables!

Arithmetic expressions with variables

- Syntax:

BNF:

AExp	::=	string
	|	$\mathbb{N}$
	|	AExp '+' AExp
	|	AExp '*' AExp

```
Inductive AExp :=
| var : string -> AExp
| anum : nat -> AExp
| aplus : AExp -> AExp -> AExp
| amul : AExp -> AExp -> AExp.
```

- Semantics:

```
Fixpoint aeval (a : AExp) : nat :=
match a with
| var v => ???
| anum v => v
| aplus a1 a2 => (aeval a1) + (aeval a2)
| amul a1 a2 => (aeval a1) * (aeval a2)
end.
```

- We need an environment to evaluate expressions with variables!

Evaluate expression with variables

- Semantics:

```
Fixpoint aeval (a : AExp) (env : Env) : nat :=
match a with
| var v => env v
| anum v => v
| aplus a1 a2 => (aeval a1 env) + (aeval a2 env)
| amul a1 a2 => (aeval a1 env) * (aeval a2 env)
end.
```

- Notations and testing:

```
Coercion var : string >-> AExp.
Coercion anum : nat >-> AExp.
Notation "A +' B" := (aplus A B) (at level 49).
Notation "A *' B" := (amul A B) (at level 48).
Compute aeval (2 +' 3 *' 4) env.
= 14
: nat
Compute env n.
= 10
: nat
Compute aeval (2 +' 3 *' n) env.
= 32
: nat
```

Boolean expressions

- Syntax:

```
Inductive BExp :=
| btrue : BExp
| bfalse : BExp
| blessthan : AExp -> AExp -> BExp
| band : BExp -> BExp -> BExp
| bnot : BExp -> BExp.
```

- Semantics:

```
Fixpoint beval (b : BExp) (env : Env) : bool :=
match b with
| btrue => true
| bfalse => false
| blessthan a1 a2 => Nat.leb (aeval a1 env) (aeval a2 env)
| band b1 b2 => andb (beval b1 env) (beval b2 env)
| bnot b' => negb (beval b' env)
end.
```

Boolean expressions

- Syntax:

```
Inductive BExp :=
| btrue : BExp
| bfalse : BExp
| blessthan : AExp -> AExp -> BExp
| band : BExp -> BExp -> BExp
| bnot : BExp -> BExp.
```

- Semantics:

```
Fixpoint beval (b : BExp) (env : Env) : bool :=
match b with
| btrue => true
| bfalse => false
| blessthan a1 a2 => Nat.leb (aeval a1 env) (aeval a2 env)
| band b1 b2 => andb (beval b1 env) (beval b2 env)
| bnot b' => negb (beval b' env)
end.
```

Assignments

- Syntax:

```
Inductive Stmt :=
| assignment : string -> AExp -> Stmt.
Notation "A ::= B" := (assignment A B) (at level 54).
Check n ::= 100 .
n ::= 100
     : Stmt
```

- Assignments modify the environment
- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
| assignment x a => update x (aeval a env) env
end.
```

- `(eval (n ::= 100) env)` is of type Env!

Assignments

- Syntax:

```
Inductive Stmt :=
| assignment : string -> AExp -> Stmt.
Notation "A ::= B" := (assignment A B) (at level 54).
Check n ::= 100 .
n ::= 100
      : Stmt
```

- Assignments modify the environment
- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
| assignment x a => update x (aeval a env) env
end.
```

- `(eval (n ::= 100) env)` is of type Env!

Assignments

- Syntax:

```
Inductive Stmt :=
| assignment : string -> AExp -> Stmt.
Notation "A ::= B" := (assignment A B) (at level 54).
Check n ::= 100 .
n ::= 100
      : Stmt
```

- Assignments modify the environment
- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
| assignment x a => update x (aeval a env) env
end.
```

- `(eval (n ::= 100) env)` is of type Env!

Assignments

- Syntax:

```
Inductive Stmt :=
| assignment : string -> AExp -> Stmt.
Notation "A ::= B" := (assignment A B) (at level 54).
Check n ::= 100 .
n ::= 100
      : Stmt
```

- Assignments modify the environment
- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
| assignment x a => update x (aeval a env) env
end.
```

- `(eval (n ::= 100) env)` is of type Env!

Assignments

- Syntax:

```
Inductive Stmt :=
| assignment : string -> AExp -> Stmt.
Notation "A ::= B" := (assignment A B) (at level 54).
Check n ::= 100 .
n ::= 100
      : Stmt
```

- Assignments modify the environment
- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
| assignment x a => update x (aeval a env) env
end.
```

- `(eval (n ::= 100) env)` is of type Env!

Programs: sequence of statements

- Syntax:

```
Inductive Stmt :=
...
| seq s1 s2 => eval s2 (eval s1 env)
...
Notation "S S'" := (assignment S S') (at level 54).
Check n ::= 100 ;; i ::= 7 .
n ::= 100 ;; i ::= 7
      : Stmt
```

- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
...
| seq s1 s2 => eval s2 (eval s1 env)
end.
```

Programs: sequence of statements

- Syntax:

```
Inductive Stmt :=
...
| seq s1 s2 => eval s2 (eval s1 env)
...
Notation "S S'" := (assignment S S') (at level 54).
Check n ::= 100 ;; i ::= 7 .
n ::= 100 ;; i ::= 7
      : Stmt
```

- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
...
| seq s1 s2 => eval s2 (eval s1 env)
end.
```

Programs: sequence of statements

- Syntax:

```
Inductive Stmt :=
...
| seq s1 s2 => eval s2 (eval s1 env)
...
Notation "S S'" := (assignment S S') (at level 54).
Check n ::= 100 ;; i ::= 7 .
n ::= 100 ;; i ::= 7
      : Stmt
```

- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
...
| seq s1 s2 => eval s2 (eval s1 env)
end.
```

Loops

- Syntax:

```
Inductive Stmt :=
...
| while : BExp -> Stmt -> Stmt.
...
```

- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
...
| while b s' => if (beval b env)
                then (eval (seq s' (while b s')) env)
                else env
end.
```

- ERROR:

```
Error: Cannot guess decreasing argument of fix.
```

- Solution: define `eval` as a *relation*!

Loops

- Syntax:

```
Inductive Stmt :=
...
| while : BExp -> Stmt -> Stmt.
...
```

- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
...
| while b s' => if (beval b env)
                then (eval (seq s' (while b s')) env)
                else env
end.
```

- ERROR:

```
Error: Cannot guess decreasing argument of fix.
```

- Solution: define `eval` as a *relation*!

Loops

- Syntax:

```
Inductive Stmt :=
...
| while : BExp -> Stmt -> Stmt.
...
```

- Semantics:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
...
| while b s' => if (beval b env)
                then (eval (seq s' (while b s')) env)
                else env
end.
```

- ERROR:

```
Error: Cannot guess decreasing argument of fix.
```

- Solution: define `eval` as a *relation*!

Evaluation as a relation

- Assignment: $\dfrac{\texttt{aeval a E} = \texttt{v}}{(\texttt{eval}\ (\texttt{x} ::= \texttt{a})\ \texttt{E}\ (\texttt{x} \mapsto \texttt{v}\ ;\ \texttt{E}))}$

- Sequence: $\dfrac{\texttt{eval}\ \texttt{s}_1\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ \texttt{s}_2\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{seq}\ \texttt{s}_1\ \texttt{s}_2)\ \texttt{E}_1\ \texttt{E}_2)}$

- Loop (true case):

$$\frac{\texttt{beval b E}_1 = \texttt{true} \quad \texttt{eval s E}_1\ \texttt{E}' \quad \texttt{eval}\ (\texttt{while b s})\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E}_1\ \texttt{E}_2)}$$

- Loop (false case): $\dfrac{\texttt{beval b E} = \texttt{false}}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E}\ \texttt{E})}$

Evaluation as a relation

- Assignment: $\dfrac{\texttt{aeval a E} = \texttt{v}}{(\texttt{eval}\ (\texttt{x} ::= \texttt{a})\ \texttt{E}\ (\texttt{x} \mapsto \texttt{v}\ ;\ \texttt{E}))}$
- Sequence: $\dfrac{\texttt{eval}\ \texttt{s}_1\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ \texttt{s}_2\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{seq}\ \texttt{s}_1\ \texttt{s}_2)\ \texttt{E}_1\ \texttt{E}_2)}$
- Loop (true case):

$$\frac{\texttt{beval b}\ \texttt{E}_1 = \texttt{true} \quad \texttt{eval s}\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ (\texttt{while b s})\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E}_1\ \texttt{E}_2)}$$

- Loop (false case): $\dfrac{\texttt{beval b E} = \texttt{false}}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E E})}$

Evaluation as a relation

- Assignment: $$\frac{\texttt{aeval a E} = \texttt{v}}{(\texttt{eval}\ (\texttt{x} ::= \texttt{a})\ \texttt{E}\ (\texttt{x} \mapsto \texttt{v}\ ;\ \texttt{E}))}$$
- Sequence: $$\frac{\texttt{eval}\ \texttt{s}_1\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ \texttt{s}_2\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{seq}\ \texttt{s}_1\ \texttt{s}_2)\ \texttt{E}_1\ \texttt{E}_2)}$$
- Loop (true case):

$$\frac{\texttt{beval b}\ \texttt{E}_1 = \texttt{true} \quad \texttt{eval s}\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ (\texttt{while b s})\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E}_1\ \texttt{E}_2)}$$

- Loop (false case): $$\frac{\texttt{beval b E} = \texttt{false}}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E E})}$$

Evaluation as a relation

- Assignment: $\dfrac{\texttt{aeval a E} = \texttt{v}}{(\texttt{eval}\ (\texttt{x} ::= \texttt{a})\ \texttt{E}\ (\texttt{x} \mapsto \texttt{v}\ ;\ \texttt{E}))}$
- Sequence: $\dfrac{\texttt{eval}\ \texttt{s}_1\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ \texttt{s}_2\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{seq}\ \texttt{s}_1\ \texttt{s}_2)\ \texttt{E}_1\ \texttt{E}_2)}$
- Loop (true case):

$$\frac{\texttt{beval b}\ \texttt{E}_1 = \texttt{true} \quad \texttt{eval s}\ \texttt{E}_1\ \texttt{E}' \quad \texttt{eval}\ (\texttt{while b s})\ \texttt{E}'\ \texttt{E}_2}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E}_1\ \texttt{E}_2)}$$

- Loop (false case): $\dfrac{\texttt{beval b E} = \texttt{false}}{(\texttt{eval}\ (\texttt{while b s})\ \texttt{E E})}$

Bibliography

- Chapter Simple Imperative Programs in Software Foundations - Volume 1, Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey https://softwarefoundations.cis.upenn.edu/lf-current/Imp.html