Crash Course on Notation in Programming Language Theory

Jeremy G. Siek

Indiana University, Bloomington

LambdaConf
HOP Workshop
June 2018

Outline

- Sets, Tuples, Relations, and Functions
- Language Syntax and Grammars
- Operational Semantics
- Type Systems

Sets

A **set** is a collection of objects.

Examples:

$$\emptyset \qquad \{0, 1, 2\}$$

Order and duplication doesn't matter:

$$\{0, 1, 2\} = \{2, 0, 1\} = \{1, 2, 1, 0, 1\}$$

Set operations:

$$1 \in \{0, 1, 2\} \qquad 3 \notin \{0, 1, 2\}$$
$$\{0, 1\} \cup \{1, 3\} = \{0, 1, 3\} \qquad \{0, 1\} \cap \{1, 3\} = \{1\}$$
$$\{0, 1\} - \{1, 3\} = \{0\}$$

Sets can be infinite:

$$\mathbb{N} = \{0, 1, 2, \ldots\} \qquad \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

Question: if $X = \{0\} \cup Y$, is it true that $0 \notin Y$?

Tuples

A **tuple** is a sequence of objects.
A **pair** is a tuple of two objects.

Example:

$$(0, 1, 2) \qquad \text{or} \qquad \langle 0, 1, 2 \rangle$$

Order and duplication matters:

$$(0, 1, 2) \neq (2, 0, 1)$$
$$(0, 1) \neq (0, 0, 1) \neq (0, 0, 1, 1)$$

Subscript with index to access *n*th object of the tuple:

$$(a, b, c)_0 = a$$
$$(a, b, c)_1 = b$$
$$(a, b, c)_2 = c$$

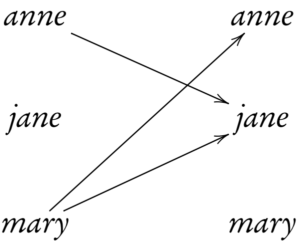
Relations

A **relation** is a set of pairs.

Example:

$$\{(anne, jane), (mary, anne), (mary, jane)\}$$

Represents associations between entities:



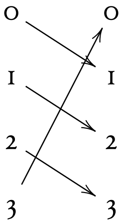
Functions

A **function** is a relation that associates to each entity at most one other entity.

Example:

$$F = \{(0, 1), (1, 2), (2, 3), (3, 0)\}$$

Represents input/output:



$$F(0) = 1$$
$$F(1) = 2$$
$$F(2) = 3$$
$$F(3) = 0$$

Some functions are infinite:

$$Inc = \{(0, 1), (1, 2), (2, 3), (3, 4), \ldots\}$$

Definition by Rules

We can define an infinite set by a collection of rules.

Example: *Inc* is the set that contains only those elements specified by the following rules:

1. $(0, 1) \in \mathit{Inc}$.
2. For any n and m, **if** $(n, m) \in \mathit{Inc}$, **then** $(n + 1, m + 1) \in \mathit{Inc}$.

It's OK for the rules to be recursive, for example, $(n, m) \in \mathit{Inc}$ in rule 2.

aka. inductively defined set

Nonsensical Rules

Some collections of rules are nonsensical.

1. $(0, 1) \in R$.
2. $(0, 1) \notin R$.

a. **If** $(n, m) \in S$, **then** $(n, m + 1) \in S$.
b. **If** $(n, m) \in S$, **then** $(n + 1, m) \in S$.

Nonsensical Rules

Some collections of rules are nonsensical.

1. $(0, 1) \in R$.
2. $(0, 1) \notin R$.

a. **If** $(n, m) \in S$, **then** $(n, m + 1) \in S$.
b. **If** $(n, m) \in S$, **then** $(n + 1, m) \in S$.

Avoid using negation.
Include at least one non-recursive rule.

Those Horizontal Lines

Just a notation for if-then rules.
Premises go on top, conclusion on the bottom.

Recall *Inc*:

1. $(0, 1) \in Inc$.
2. For any n and m, **if** $(n, m) \in Inc$, **then** $(n+1, m+1) \in Inc$.

Definition of *Inc* via horizontal lines:

$$\frac{}{(0, 1) \in Inc} \qquad \frac{(n, m) \in Inc}{(n+1, m+1) \in Inc}$$

Derivations Justify Membership

Recall *Inc*:

$$\text{Rule 1}\ \frac{}{(0, 1) \in Inc} \qquad \text{Rule 2}\ \frac{(n, m) \in Inc}{(n+1, m+1) \in Inc}$$

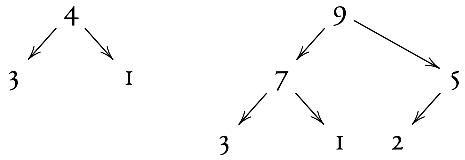
Is $(2, 3) \in Inc$?

Yes, and here's why:

$$\text{(Rule 2)}\ \dfrac{\text{(Rule 2)}\ \dfrac{\text{(Rule 1)}\ \dfrac{}{(0, 1) \in Inc}}{(1, 2) \in Inc}}{(2, 3) \in Inc}$$

Exercises

- Define the even natural numbers $Even = \{0, 2, 4, 6, \ldots\}$ using definition-by-rules.
- Give the derivation for $4 \in Even$.
- Define the set of all binary trees that satisfy the max-heap property (child's label is less-or-equal to parent's) using definition-by-rules. Examples:



Outline

- Sets, Tuples, Relations, and Functions
- **Language Syntax and Grammars**
- Operational Semantics
- Type Systems

Language Syntax and Grammars

Let's define a language of integer arithmetic, call it *Arith*.

A **language** is a set of programs, usually an infinite set.

Example:

$$
Arith = \left\{ \begin{array}{l} 3, \\ 7, \\ 3 + 7, \\ -(3 + 7), \\ 4, \\ -(3 + 7) + 4, \\ \vdots \end{array} \right\}
$$

Syntax via Definition by Rules

Definition by rules to the rescue!

Arith is the set containing only those programs justified by the following rules:

- For any $n \in \mathbb{Z}$, $n \in Arith$.
- For any e, **if** $e \in Arith$, **then** `-`$e \in Arith$.
- For any e_1 and e_2, **if** $e_1 \in Arith$ and $e_2 \in Arith$, **then** e_1 `+` $e_2 \in Arith$.
- For any e, **if** $e \in Arith$, **then** `(`e`)` $\in Arith$.

Backus-Nauer Form (BNF)

BNF is a notation for definition-by-rules that is specialized to programming languages.

A collection of BNF rules is called a **grammar**.

```
Arith ::= integer
Arith ::= "-" Arith
Arith ::= Arith "+" Arith
Arith ::= "(" Arith ")"
```

or equivalently

```
Arith ::= integer | "-" Arith | Arith "+" Arith
        | "(" Arith ")"
```

derivation = parse tree

Syntax Conventions in PL Theory

Instead of BNF:

```
Arith ::= integer | "-" Arith | Arith "+" Arith
        | "(" Arith ")"
```

We select some variables to range over elements of the sets, e.g., $n \in \mathbb{Z}, m \in \mathbb{Z}$ and $e \in Arith$,

then replace the set names with the variables:

$$e ::= n \mid \text{-}\, e \mid e + e$$

We omit the rule for parentheses; they are always allowed.

Exercise

Give the PL-theory style syntax definition for a language *ArithPair* that includes integer arithmetic and pairs.
Example programs:

$$-(2+3,-4)_0 \qquad ((1+3,(5,2+4))_1)_0$$

Outline

- Sets, Tuples, Relations, and Functions
- Language Syntax and Grammars
- **Operational Semantics**
- Type Systems

Operational Semantics

Define the meaning of programs by
saying what happens when you run them.

Many styles of operational semantics:

- big-step semantics
- small-step semantics

Big-step Semantics

A relation named *Eval* between programs and result values.

Notation:

$$e \Downarrow n \quad \equiv \quad (e, n) \in \mathit{Eval}$$

We define *Eval* as the set containing only those program-result pairs justified by the following rules.

$$\mathrm{I}\,\frac{}{n \Downarrow n} \qquad \mathrm{N}\,\frac{e \Downarrow n}{\texttt{-}e \Downarrow -n} \qquad \mathrm{P}\,\frac{e_1 \Downarrow n \quad e_2 \Downarrow m}{e_1 \texttt{+} e_2 \Downarrow n + m}$$

Example:

$$-(3 + 7) + 4 \Downarrow -6$$

aka. natural semantics

Derivation of a big-step

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{3 \Downarrow 3}\,\mathrm{I}
      \qquad
      \dfrac{}{7 \Downarrow 7}\,\mathrm{I}
    }{3 + 7 \Downarrow 10}\,\mathrm{P}
  }{-(3 + 7) \Downarrow -10}\,\mathrm{N}
  \qquad
  \dfrac{}{4 \Downarrow 4}\,\mathrm{I}
}{-(3 + 7) + 4 \Downarrow -6}\,\mathrm{P}
$$

Small-step Semantics

A relation *Step* on programs that does just one computation.

Notation:

$$e \longrightarrow e' \quad \equiv \quad (e, e') \in \mathit{Step}$$

We define *Step* as the set containing only those program-program pairs justified by the following rules.

$$\mathrm{N}\,\frac{}{\texttt{-}\,n \longrightarrow -n} \qquad \mathrm{P}\,\frac{}{n \texttt{+} m \longrightarrow n + m}$$

$$\mathrm{N}_1\,\frac{e \longrightarrow e'}{\texttt{-}\,e \longrightarrow \texttt{-}\,e'} \qquad \mathrm{P}_1\,\frac{e_1 \longrightarrow e_1'}{e_1 \texttt{+} e_2 \longrightarrow e_1' \texttt{+} e_2} \qquad \mathrm{P}_2\,\frac{e_2 \longrightarrow e_2'}{n \texttt{+} e_2 \longrightarrow n \texttt{+} e_2'}$$

Example:

$$\texttt{-}(3 \texttt{+} 7) \texttt{+} 4 \longrightarrow \texttt{-}(10) \texttt{+} 4 \longrightarrow -10 \texttt{+} 4 \longrightarrow -6$$

Derivation of a small-step

$$\mathrm{P}_1\,\dfrac{\mathrm{N}_1\,\dfrac{\mathrm{P}\,\dfrac{}{3+7 \longrightarrow 10}}{-(3+7) \longrightarrow -(10)}}{-(3+7)+4 \longrightarrow -(10)+4}$$

Exercises

- Define the set of values for the language *ArithPair*.
- Extend the big-step semantics to handle the language *ArithPair*.
- Extend the small-step semantics to handle the language *ArithPair*.

Outline

- Sets, Tuples, Relations, and Functions
- Language Syntax and Grammars
- Operational Semantics
- **Type Systems**

Type Systems

Consider a language *ArithBool* of integers, Booleans, and conditionals:

$$
\begin{array}{rcl}
e & ::= & n \mid \texttt{-}\, e \mid e \,\texttt{+}\, e \\
 & \mid & \mathit{true} \mid \mathit{false} \mid \neg e \mid e \vee e \\
 & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } e
\end{array}
$$

A **type** classifies values, it is a set of values.

$$
\begin{aligned}
\texttt{Int} &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
\texttt{Bool} &= \{\mathit{true}, \mathit{false}\}
\end{aligned}
$$

- A type system **predicts** the type of the result value.
  $(-(3 + 7) + 4)$ produces an `Int`,
  $(\mathit{true} \vee \neg \mathit{false})$ produces a `Bool`.
- A type system **enforces** that the arguments of an operation make sense.
  $(3 + \mathit{false})$ is ill-typed

Programs can "go wrong"

Examples:

$$\nexists n.\ 3 + \mathit{false} \Downarrow n$$
$$\nexists n.\ \mathbf{if}\ 0 + 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \Downarrow n$$

In a small-step semantics, the reductions get "stuck":

$$3 + \mathit{false} \not\longrightarrow$$
$$\mathbf{if}\ 0 + 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \longrightarrow \mathbf{if}\ 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \not\longrightarrow$$

Type System for *ArithBool*

WellTyped is a relation between programs and types.

Notation: let T range over types (`Int` and `Bool`).

$$\vdash e : T \quad \equiv \quad (e, T) \in \mathit{WellTyped}$$

Type System: (definition-by-rules yet again!)

$$\frac{}{\vdash n : \texttt{Int}} \qquad \frac{\vdash e : \texttt{Int}}{\vdash \texttt{-}e : \texttt{Int}} \qquad \frac{\vdash e_1 : \texttt{Int} \quad \vdash e_2 : \texttt{Int}}{\vdash e_1 \texttt{+} e_2 : \texttt{Int}}$$

$$\frac{}{\vdash \mathit{true} : \texttt{Bool}} \qquad \frac{}{\vdash \mathit{false} : \texttt{Bool}} \qquad \frac{\vdash e : \texttt{Bool}}{\vdash \neg e : \texttt{Bool}}$$

$$\frac{\begin{array}{c}\vdash e_1 : \texttt{Bool} \\ \vdash e_2 : \texttt{Bool}\end{array}}{\vdash e_1 \vee e_2 : \texttt{Bool}} \qquad \frac{\vdash e_1 : \texttt{Bool} \quad \vdash e_2 : T \quad \vdash e_3 : T}{\vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : T}$$

Type Safety

Well-typed programs cannot "go wrong".
—Robin Milner (1978)

Let v range over values (integers and Booleans).

Theorem (Type Safety)

If $\vdash e : T$, *then* $e \Downarrow v$ *and* $\vdash v : T$ *for some* v.

Rule Induction

Suppose set S is defined by a collection of rules such as

$$\frac{}{a_1 \in S} \qquad \frac{a_2 \in S}{f(a_2) \in S} \qquad \frac{a_3 \in S \quad a_4 \in S}{g(a_3, a_4) \in S}$$

You want to prove $\forall x \in S.\ R(x)$.
It is sufficient to prove:

- $R(a_1)$,
- if $R(a_2)$, then $R(f(a_2))$, and
- if $R(a_3)$ and $R(a_4)$, then $R(g(a_3, a_4))$.

Proof of Type Safety

By induction on the program e. The premise is that $\vdash e : T$.

- Case $\boxed{e = n}$: $n \Downarrow n$ and $\vdash n : \texttt{Int}$.
- Case $\boxed{e = \texttt{-}\, e'}$: By the induction hypothesis, $e' \Downarrow v$ and $\vdash v : \texttt{Int}$. So v is an integer. Thus, $-e' \Downarrow -v$ and $\vdash -v : \texttt{Int}$.
- Case $\boxed{e = \neg e'}$: By the induction hypothesis, $e' \Downarrow v$ and $\vdash v : \texttt{Bool}$. So v is *true* or *false*. Thus, $\neg e' \Downarrow \neg v$ and $\vdash \neg v : \texttt{Bool}$.

$\vdots$

Exercises

- Devise a type system for the *ArithPair* language.
- Prove the Type Safety theorem for your type system.

Suggested Reading

- Types and Programming Languages
  by Benjamin C. Pierce.
- Semantic Engineering with PLT Redex
  by Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt.
- My blog: `http://siek.blogspot.com/`
- Papers in the annual ACM International Conference on Functional Programming (ICFP).
- Papers in the annual ACM Symposium on Principles of Programming Languages (POPL).

Conclusion

- Infinite sets can be defined via definition-by-rules (aka. inductively defined sets).
- Use definition-by-rules to define:
  - language syntax,
  - operational semantics, and
  - type systems!
- Type Safety is the property that all well-typed programs cannot "go wrong" and the result value is of the expected type.

Recall *Inc*:

$$\text{Rule 1}\ \frac{}{(0, 1) \in \mathit{Inc}} \qquad \text{Rule 2}\ \frac{(n, m) \in \mathit{Inc}}{(n+1, m+1) \in \mathit{Inc}}$$

We capture the notion of applying the rules just once, starting from an arbitrary set X, with the following function.

$$F(X) = \{(0, 1)\} \cup \{(n+1, m+1) \mid (n, m) \in X\}$$

A set Y is **closed under** F if applying F to Y is already in Y:

$$F(Y) \subseteq Y$$

The intersection of all such Y's is the set *Inc*:

$$\mathit{Inc} = \bigcap\{Y \mid F(Y) \subseteq Y\}$$

Theorem (Knaster-Tarski)

Suppose $X = \bigcap\{Y \mid F(Y) \subseteq Y\}$ *and* F *is a monotone function. Then* X *is a least fixed point of* F.

Proof. First we show that X is a fixed point of F.

$$\begin{aligned}
F(X) &= \bigcap\{F(Y) \mid F(Y) \subseteq Y\} && && (1)\\
F(X) &\subseteq X && && (2)\\
F(F(X)) &\subseteq F(X) && F \text{ monotone} && (3)\\
X &\subseteq F(X) && \text{def. of } X && (4)\\
X &= F(X) && \text{from (2), (4)} && (5)
\end{aligned}$$

Next we show that X is the least fixed point.
Suppose X' is another fixed point.

$$\begin{aligned}
X' &= F(X') && \\
F(X') &\subseteq X' && \\
X &\subseteq X' && \text{def. of } X
\end{aligned}$$

So by the Knaster-Tarski Theorem, *Inc* is the least fixed point of F, that is,

$$
\begin{array}{l}
Inc = F(Inc) \\
\text{for any } X',\ \text{if } X' = F(X'), \text{then } Inc \subseteq X'
\end{array}
$$

Definitional Interpreter

A recursive procedure that performs the computation.

Example interpreter for *Arith*:

$$\begin{aligned} eval &: Arith \to \mathbb{Z} \\ eval(n) &= n \\ eval(\texttt{-}\, e) &= -eval(e) \\ eval(e_1 \texttt{+} e_2) &= eval(e_1) + eval(e_2) \end{aligned}$$

Example run:

$$eval(-(3 + 7) + 4) = -6$$

Definitional interpreter for *ArithBool*

$$eval : \mathit{ArithBool} \to \mathbb{Z}$$

$$\vdots$$

$$\begin{aligned}
eval(true) &= true \\
eval(false) &= false \\
eval(\neg e) &= \neg eval(e) \\
eval(e_1 \vee e_2) &= eval(e_1) \vee eval(e_2) \\
eval(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) &= \begin{cases} eval(e_2) & \text{if } eval(e_1) = true \\ eval(e_3) & \text{if } eval(e_1) = false \end{cases}
\end{aligned}$$

eval is a partial function.