Lecture Notes

Contents

Chapter 1

Lecture 2

Shiven Sinha

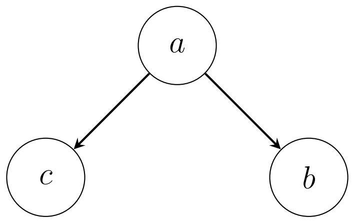
1.0.1 Abstract Reduction System

An abstract reduction system is a binary relation $\rightarrow$ over a set of elements A.

1.0.2 Other Relations defined on $\overrightarrow{A}$

Identity	$\overrightarrow{A^0} = \{(a,a) : a \in A\}$
Reflexive Closure	$\overrightarrow{A^=} = \overrightarrow{A} \cup \overrightarrow{A^0}$
Inverse	$\overrightarrow{A^{-1}} = \{(b,a) : a \underset{A}{\rightarrow} b\}$
Transitive Closure	$\overrightarrow{A^+} = \bigcup_{i>0} \overrightarrow{A^i}$
Reflexive Transitive Closure	$\overrightarrow{A^*} = \overrightarrow{A^+} \cup \overrightarrow{A^0}$
Symmetric closure	$\overrightarrow{A^{\leftrightarrow}} = \overrightarrow{A} \cup \overrightarrow{A^{-1}}$

Note: Pay attention to the order of terms in the name of the relations. For example, the transitive symmetric closure isn't necessarily the same as the symmetric transitive closure. Consider the following relation graph to demonstrate this.



$$(A^{\leftrightarrow})^* = \{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c),(c,a),(c,b),(c,c)\}$$
$$(A^*)^{\leftrightarrow} = \{(a,b),(a,c),(b,a),(c,a)\}$$

1.0.3 Terminology

Describing the elements and their relations

- x is reducible if $\exists\, x'$ such that $x \underset{A}{\rightarrow} x'$.
- $\theta : F \rightarrow E$ is called an **invariant** if $\forall x \in A, \theta(x) = \theta(F(x))$. For example, for the run $x_0 \underset{F}{\rightarrow} x_1 \underset{F}{\rightarrow} x_2 \underset{F}{\rightarrow} \cdots$, it must hold that $\theta(x_0) = \theta(x_1) = \theta(x_2) = \cdots$.
- x is in **normal form** if it is not reducible.
- x **simplifies to** x' in A iff $x \underset{A^*}{\longrightarrow} x'$.
- x' is a normal form of x in A if:
  - x' is in normal form
  - x simplifies to x'
- x has a normal form in A if $\exists\, x' \in A$ such that x' is a normal form of x.
- x' is an **immediate successor** of x if $x \underset{A}{\rightarrow} x'$.
- x' is a **proper successor** of x if $x \underset{A^+}{\longrightarrow} x'$.
- x' is a **successor** of x if $x \underset{A^*}{\longrightarrow} x'$.
- Two elements a and b in A are **joinable** if $\exists\, c \in A$ such that $a \underset{A^*}{\longrightarrow} c$ and $b \underset{A^*}{\longrightarrow} c$. This is denoted as $a \downarrow b$.
- a and b are **connected** in A if $a \xrightarrow[(A^{\leftrightarrow})^*]{} b$

Describing the system as a whole

- $\overrightarrow{A}$ is terminating if there is no infinite run: $a_0 \rightarrow a_1 \rightarrow \cdots$.

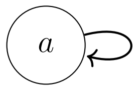


Figure 1.1: A non-terminating relation graph

- $\overrightarrow{A}$ is normalising if every element has a normal form.

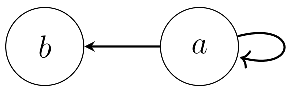


Figure 1.2: A normalising, non-terminating relation graph

- $\overrightarrow{A}$ is confluent if $\forall a, b, c \in A$ such that $a \underset{A^*}{\longrightarrow} b, a \underset{A^*}{\longrightarrow} c$, it must hold that $b \downarrow c$.

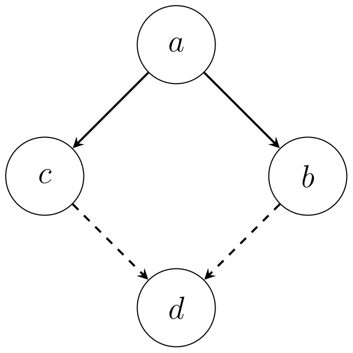


Figure 1.3: A visual representation of confluence

Chapter 2

Lecture 3

Dheeraja Rajreddygari

2.0.1 Properties of abstract reduction systems

In this lecture, we explore some properties of abstract reduction systems and look at the relationships between them.

Definition 2.0.1. An abstract reduction system $(A, \underset{A}{\rightarrow})$ is said to be **Church-Rosser** if

$$\forall x, y \in A, \quad x \overset{*}{\underset{A}{\leftrightarrow}} y \implies x \downarrow_A y$$

Consider abstract reduction systems A and B as shown in the figure below. System A is not Church-Rosser, since $a \overset{*}{\underset{A}{\leftrightarrow}} b$ but a and b are not joinable. System B is a simple example of a Church-Rosser system.

Definition 2.0.2. An abstract reduction system $(A, \underset{A}{\rightarrow})$ is said to be **Confluent** if

$$\forall a, b, c \in A, \quad a \overset{*}{\underset{A}{\rightarrow}} b \text{ and } a \overset{*}{\underset{A}{\rightarrow}} c \implies b \downarrow_A c$$

Definition 2.0.3. An abstract reduction system $(A, \underset{A}{\rightarrow})$ is said to be **Semi-confluent** if

$$\forall a, b, c \in A, \quad a \underset{A}{\rightarrow} b \text{ and } a \overset{*}{\underset{A}{\rightarrow}} c \implies b \downarrow_A c$$

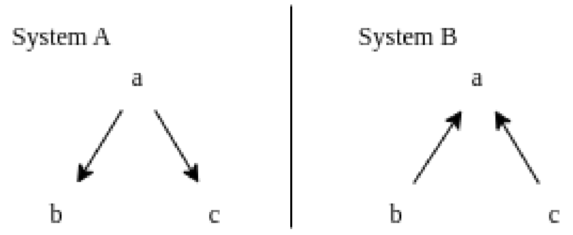


Figure 2.1: Examples of abstract reduction systems

Theorem 2.0.4. For an abstract reduction system, **Church-Rosser, Confluence and Semi-confluence are equivalent**.

Proof: We will prove this theorem in the following three stages. Clearly, the three of them combined result in the theorem stated above.

1. Church-Rosser $\implies$ Confluence
2. Confluence $\implies$ Semi-confluence
3. Semi-confluence $\implies$ Church-Rosser

First, we will prove that **if a system is Church-Rosser, it is confluent**. Let $(A, \underset{A}{\rightarrow})$ be an abstract reduction system that is Church-Rosser.

Let $a, b, c \in A : a \overset{*}{\underset{A}{\rightarrow}} b$ and $a \overset{*}{\underset{A}{\rightarrow}} c$

By definition, $b \overset{*}{\underset{A}{\leftrightarrow}} c$

Since A is Church-Rosser, $b \overset{*}{\underset{A}{\leftrightarrow}} c \implies b \downarrow_A c$

i.e. $a \overset{*}{\underset{A}{\rightarrow}} b$ and $a \overset{*}{\underset{A}{\rightarrow}} c \implies b \downarrow_A c$, proving that A is confluent

Next, we will prove that **if a system is Confluent, it is also semi-confluent**. Let $(A, \underset{A}{\rightarrow})$ be a confluent abstract reduction system.

Let $a, b, c \in A : a \underset{A}{\rightarrow} b$ and $a \overset{*}{\underset{A}{\rightarrow}} c$

Since $A \subseteq A^*$, $a \underset{A}{\rightarrow} b \implies a \overset{*}{\underset{A}{\rightarrow}} b$

Since A is confluent, $a \overset{*}{\underset{A}{\rightarrow}} b$ and $a \overset{*}{\underset{A}{\rightarrow}} c \implies b \downarrow_A c$

i.e. $a \underset{A}{\rightarrow} b$ and $a \overset{*}{\underset{A}{\rightarrow}} c \implies b \downarrow_A c$, proving that A is semi-confluent

Finally, we will prove that **if a system is semi-confluent, it is also**

Church-Rosser. Let $(A, \underset{A}{\rightarrow})$ be a semi-confluent abstract reduction system.
Let $a, b \in A : a \overset{*}{\underset{A}{\leftrightarrow}} b$
Let p be the shortest path connecting a and b in $A^{\leftrightarrow^*}$. We will use induction on $|p|$ to prove that a and b are joinable.
Base case: For $p = 0$, we have $a = b$ which makes them trivially joinable.
Induction step: Let it be true that if the shortest path connecting a and b in $A^{\leftrightarrow^*}$ is $|p|$, then a and b are joinable in A. We will prove that this is also true for $|p| + 1$.

Let $a, b' \in A$ such that the shortest path connecting them in $A^{\leftrightarrow^*}$ is of length $|p| + 1$. Then, $\exists b \in A$: the shortest path connecting a and b in $A^{\leftrightarrow^*}$ is $|p|$ and $b \underset{A}{\leftrightarrow} b'$. Since our induction hypothesis holds true for $|p|$, a and b are joinable (they both reduce to some $c \in A$).

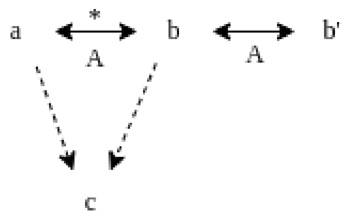


Figure 2.2: Abstract reduction system representing the induction step

We now have 2 cases.

Case 1: $b \underset{A}{\leftarrow} b'$

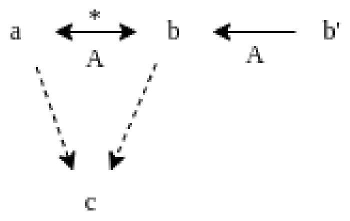


Figure 2.3: Abstract reduction system in case 1

$b' \underset{A}{\rightarrow} b \overset{*}{\underset{A}{\rightarrow}} c$. Therefore, $a \downarrow_A b'$.

Case 2: $b \underset{A}{\rightarrow} b'$

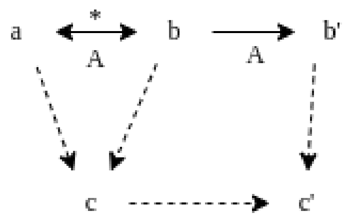

$$\begin{tikzcd} a \arrow[r, leftrightarrow, "*", "A"'] \arrow[dr, dashed] & b \arrow[r, "A"'] \arrow[d, dashed] & b' \arrow[d, dashed] \\ & c \arrow[r, dashed] & c' \end{tikzcd}$$

Figure 2.4: Abstract reduction system in case 2

Here we have $b \underset{A}{\rightarrow} b'$ and $b \overset{*}{\underset{A}{\leftrightarrow}} c$. Since A is semi-conflient, b' and c must be joinable. That is, $\exists c' \in A : b' \overset{*}{\underset{A}{\rightarrow}} c'$ and $c \overset{*}{\underset{A}{\rightarrow}} c'$. Since $a \overset{*}{\underset{A}{\rightarrow}} c \overset{*}{\underset{A}{\rightarrow}} c'$, we have $a \overset{*}{\underset{A}{\rightarrow}} c'$. Hence, $a \downarrow_A b'$

In either case, we have shown that $a \downarrow_A b'$, which completes our induction. We have proven that $a \overset{*}{\underset{A}{\leftrightarrow}} b \implies a \downarrow_A b$, which means A is also Church-Rosser.

This completes our proof that Church-Rosser, Confluence and Semi-confluence are equivalent properties of an abstract reduction system. While it may seem redundant to have multiple terms to refer to the same thing, they each give us a different perspective of looking at the same property which can prove to be helpful.

2.0.2 Address space

Data structures provide a way to organize and address data. For a data structure, a valid set of addresses form its address space. This is **not** a formal definition of address spaces and is only meant to give a broad idea. We will look at an example below to illustrate one way of addressing a binary tree. Let us consider the following addressing of a binary tree: each edge is labelled 1 or 2 depending on whether it leads to the left or right descendant of a node; the address of each node is obrained by appending the label of the edge leading into it to the address of its parent, with the root being ϵ. Look at the figure below to bettter understand this.

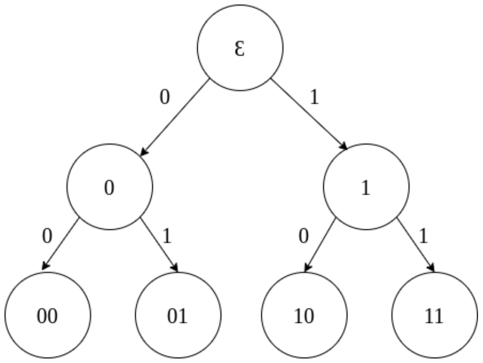


Figure 2.5: Addressing a binary tree

The address space for this binary tree would be the set $S = \{\epsilon, 0, 00, 01, 1, 10, 11\}$. The set $S_1 = \{\epsilon, 0, 00, 01, 1\}$ would also be a valid address space for some binary tree but the set $S_2 = \{0, 00, 1\}$ would not, since it does not contain ϵ, the address of the root.

Chapter 3

Lecture 4

Kriti Gupta

3.0.1 N-Arity

$\bar{Z_n} = [1...n]$

An n-ary address space A is a subset of $\bar{Z}_n^*$ that is prefix closed if $q : A$ and p is a prefix of q, then $p : A$.

Example:

$A = \epsilon, 1, 2, 11, 21$

Let A be an n-ary address space and let $p : A$.

$$A@p = \{q : \bar{Z}_n^* \mid pq : A\} \rightarrow \text{Address space of A relative to p} \tag{3.1}$$

e.g. $A@2 = \{\epsilon, 1\}$

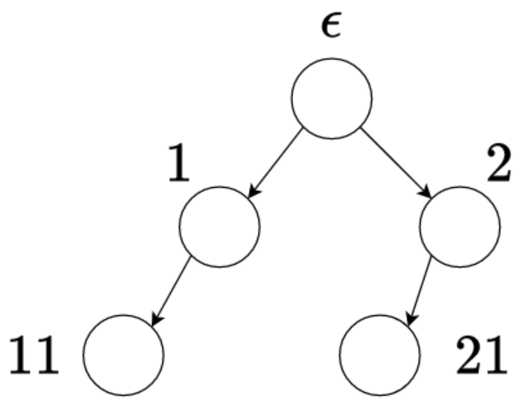


Figure 3.1:

Context at p (excluding p)

$$C_{ex}(A,p) = A \backslash pA@p \tag{3.2}$$
$$\rightarrow \text{concatenating p with each address (in subtree) relative to p} \tag{3.3}$$

Context at p (including p)

$$C_{in}(A,p) = C_{ex}(A,p) \cup \{p\} \tag{3.4}$$

3.0.2 Terms, Subtrees, Tree Contex

Let A be an address space. A term is a map $t : A \rightarrow v$ where V is a set of values.

Example:

$$V = \mathcal{N}, A = \{\epsilon, 1, 2\} \tag{3.5}$$
$$t : \{\epsilon \rightarrow 5, 1 \rightarrow 7, 2 \rightarrow 7\} \tag{3.6}$$

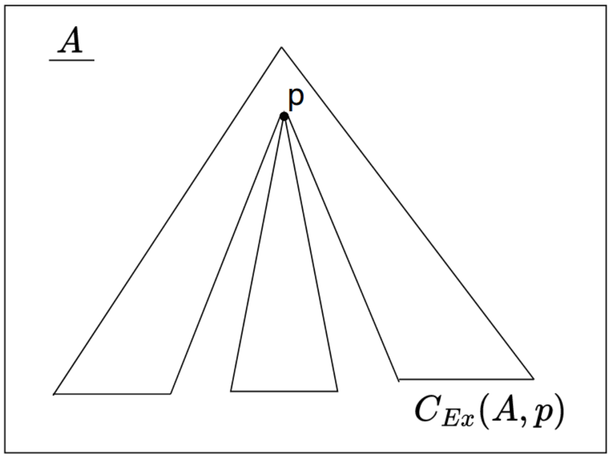


Figure 3.2:

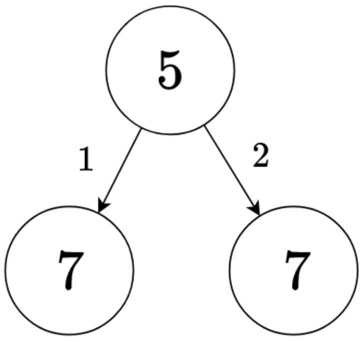


Figure 3.3:

Let $t : A \xrightarrow{v}$ be a tree and let $p : A$

$$C_{ex}(t,p) = df = t|_{C_{ex}(A,p} \tag{3.7}$$

Let Σ_s be a set of symbols.
Let $\alpha : \Sigma_s \xrightarrow{\mathcal{N}}$ denote the arity map of Σ_s.

Example,

$$\Sigma_s = \{f, a, b\}\alpha_s : \{f \to 2, a \to 0, b \to 0\} \tag{3.8}$$
$$\implies f(a,b)\checkmark \quad a\checkmark \quad b\checkmark \quad f(a)\times \tag{3.9}$$

A term is $(A, \Sigma_s, \alpha, t : A \to \Sigma_s)$
s.t.

1. A is a prefix-closed language over $\mathcal{N}_+$
2. $\forall p : A$, if $t_p = S$ and $\alpha(S) = n$
then $p1, ..., p\alpha(S) : A \qquad \forall i : 1 \leq i \leq \alpha(S), pi : A, \forall i : \mathcal{N}_+ \backslash [1, ..., \alpha(S)] pi \notin A$

Chapter 4

Lecture 5

Bharat Sahlot

4.0.1 Terms

ARS Abstract Reduction System

Terminating ARS An ARS is terminating *iff.* it has no infinite runs.

4.0.2 Well Founded Induction(WFI)

This is property of an abstract reduction system. A system having this property implies that,

$$\forall x \in A. \left(\forall y \in A.\ x \xrightarrow{+} y \implies P(y)\right) \implies P(x)$$

in other words, $P(x)$ is satisfied if $\forall y.x \xrightarrow{+} y \implies P(y)$ is satisfied.

Theorem 4.0.1. Let, $(A, \longrightarrow)$ be an ARS, then A satisfies WFI iff. $(A, \longrightarrow)$ is terminating.

Proof.
1. If $\longrightarrow$ terminates, then A satisfies WFI.

Proof by contraposition. Assume that WFI does not hold.
That implies that $\neg P(a_0)$ for some $a_0 \in A$. Since we assumed that WFI does not hold, $\exists a_1$, such that, $a_0 \xrightarrow{+} a_1$ and $\neg P(a_1)$. Using the same arguement, $\exists a_2$, such that, $a_1 \xrightarrow{+} a_2$ and $\neg P(a_2)$. Hence there is an infinite chain $a_0 \xrightarrow{+} a_1 \xrightarrow{+} a_2 \xrightarrow{+} ...$, i.e. $\longrightarrow$ does not terminate.

2. If A satisfies WFI, then $\longrightarrow$ terminates.
Proof by WFI. Let,

$$P(x) := \text{there is no infinite chain starting from x.}$$

Clearly, if there is no infinite chain starting from any successor of x, then there is no infinite chain starting from x. Hence, WFI holds and we can conclude that $P(x)$ holds for all x, i.e., $\longrightarrow$ terminates. □

4.0.3 Confluence

Order of evaluation does not matter.

Joinable x and y are joinable, denoted by $\downarrow$ *iff.* they have the same normal form.

Local Confluence An element $x \in A$ is said to be locally confluent if $\forall y, z \in A, z \xleftarrow{+} x \xrightarrow{+} y \; \exists w : y \xrightarrow{*} w \xleftarrow{*} z$, in other words, $y \downarrow z$.

$\rightarrow$ If a system is terminating and has local confluence for all vertices, then the system has **confluence**.

4.0.4 Interative Evaluation

TODO

Chapter 5

Lecture 7

Kommireddy Bhargav Srinivas

5.1 Languages in general

For any language in general we want to define the following.

1. Define the syntax, which consists of:

   (a) Abstract syntax (eg: AST)

   (b) Concrete Syntax

   (c) A Parser, which is a map from concrete syntax to abstract syntax.

$$Parser : ConcreteSyntax \rightarrow AbstractSyntax$$

2. Define the Semantics, which consists of:

   (a) Semantic domains

   (b) An evaluator, which is a map from abstract syntax to the domain of answers.

$$Evaluator : AbstractSyntax \rightarrow Answers$$

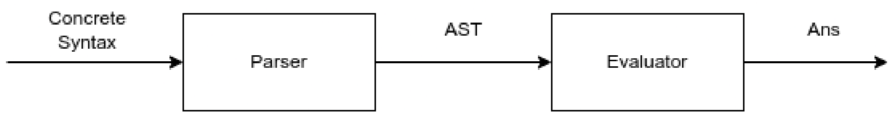


Figure 5.1: Parser Evaluator flow

5.1.1 Language 0: ARITHMETIC

This is a simple language, which works on the integer domain and has the addition, subtraction and multiplication operators defined.

The syntax of the expressions in this language can be defined as follows:

$$
\begin{aligned}
expr ::= & \; Num && (5.1)\\
| & \; expr + expr && (5.2)\\
| & \; expr * expr && (5.3)\\
| & \; expr - expr && (5.4)
\end{aligned}
$$

where Num represents any Integer. Any expression that matches this pattern is a syntactically valid expression in this language.

The semantics of Arithmetic were covered in previous lectures. It will consist of the evaluation semantics (eval-ast : AST to Answer)

The following is the racket implementation of the eval-ast function.

```
(define (eval-ast e)
  (match e
        [(number?) a]
        [(list + e1 e2)
                (let ([a1 (eval-ast e1)]
                      [a2 (eval-ast e2)])
                  (+ a1 a2))]))
;; similarly there would be cases for -, *
```

5.2 Language 1: IF+DIV

This language is syntactically a super set of ARITHMETIC. We introduce the if-then-else operator and also division. Division introduces new problems like dividing by zero. We also need to enforce that the condition given to if operator is a boolean.

These situations cannot be predicted before the program is run for every program. They need to be caught during run-time. In these situations we would like to raise (or throw) errors (or exceptions).

Now, the result of an expression is not just a number, it could be be a number or an error. There are two ways of dealing with errors, either they are treated specially and the whole execution of the program stops (In racket, you can achieve this using "invoke") and gives the error as the answer to the program. The other way is the treat errors as values, which are returned by the expression that produced the error.

Errors like non-boolean if-condition (test) or incorrect type being provided to operators (eg: +, -) are called Type Errors

The syntax of the expressions in this language can be defined as follows:

$$e ::= Num \tag{5.5}$$
$$\mid \quad e \quad + \quad e \tag{5.6}$$
$$\mid \quad e \quad * \quad e \tag{5.7}$$
$$\mid \quad e \quad - \quad e \tag{5.8}$$
$$\mid \quad e \quad / \quad e \tag{5.9}$$
$$\mid \quad if \quad e_{test} \quad e_{then} \quad e_{else} \tag{5.10}$$

The evaluation semantics have an extra rule for division (DIV) similar to addition and multiplication in ARITHMETIC. Along with that we need to define rules to evaluate if-then-else

$$\frac{e_{test} \implies \#t, \quad e_{then} \implies v}{if \quad e_{test} \quad e_{then} \quad e_{else} \implies v}\text{IF-TRUE}$$

$$\frac{e_{test} \implies \#f, \quad e_{else} \implies v}{if \quad e_{test} \quad e_{then} \quad e_{else} \implies v}\text{IF-FALSE}$$

5.3 Language-2: GLOBAL

A language doesn't need variables (eg: iptables) but programmers do. To make use of variables we need to introduce new constructs called identifiers and environment. Identifiers act as a proxy to values and the environment defines what value a specific identifier takes. (Read later: Combinatory Theory).

The values that identifiers can take are called denotable values. The values that expressions can take are called expressible values. For the current language both are the same, but languages like C have them different (One cannot store a function in a variable in C).

The syntax looks almost the same, except the fact that identifiers by themselves are also expressions.

An environment is a map from identifiers to denotable values.

$$Env : Identifiers \rightarrow Denotablevalues$$

In this language, the denotable values is a disjoint union of Numbers and Booleans.

$$Denotablevalues \in Num + Bool$$

The evaluation semantics change to include the environment as well.

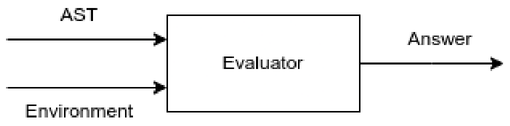


Figure 5.2: Global lang evaluator flow

When there is a mapping of an identifier in the environment, the identifier is said to be bound to that value. Introduction of identifiers creates a new type of error - Unbound identifier error. This happens when an identifier is used, but it is not bound to any value (It is a free variable.).

While evaluating the AST if an identifier is encountered, a simple look up is performed in the environment and the corresponding value is taken. The identifier could be bound to an expression. In that case, the expression is evaluated and the corresponding value returned is substituted for the identifier.

Example Expression is (+ x (* y 2)) and the environment is

$$\Gamma = \{x \mapsto 5, y \mapsto 3, z \mapsto 4\}$$

We annotate the AST from bottom up get the final value. So we would first annotate x to 5, then y to 3, then get the result of multiplication as 6 and then the result of addition to be 11.

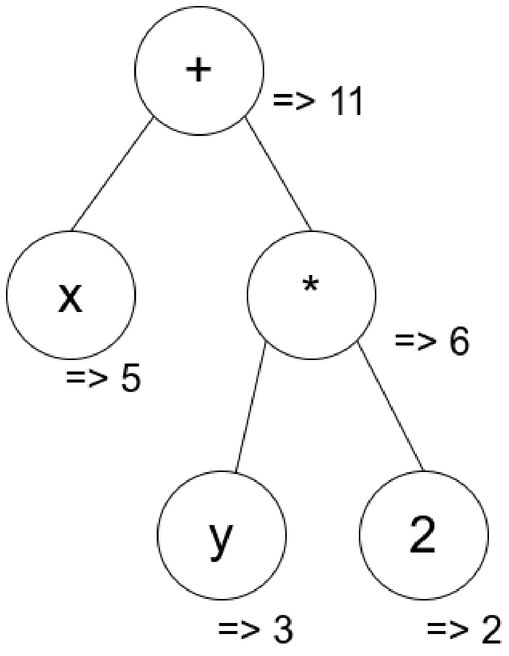


Figure 5.3: Global lang evaluator flow

Chapter 6

Lecture 9

PRANAV SUBRAMANIAM

6.1 Overview of Programming Languages

6.1.1 Mathematical Foundation

1. Before lambda-calculus (”Pre-history” of computing)
    - (a) Set Theory
    - (a) First-Order Logic
    - (a) Combinatory Logic
2. Predecessor to Programming Languages
   Provided mathematical framework for programming languages and computing
    - (a) Category Theory
    - (b) λ Calculus $\equiv$ Turing Machines (1930s)
        - i. λ Calculus
            - A. Framework for programming language design
            - B. Basis for functional programming

ii. Turing Machines

A. Computability, complexity, determinism

B. Related work- Von Neumann architecture, Kurt Gödel's incompleteness theorem

6.1.2 History

S No.	**Functional Languages**	**Imperative Languages**	**OOP**	**PROLOG (Logic Programming)**	**Concurrent Distributed Programing Languages**
1	Lisp (1959)	FORTRAN (1957)	Simula-Simulation-oriented, (Dahl and Nygaard - 1996)	Prolog (Colmerauer, 1971)	CSP (Communicating Sequential Processes), (Hoare, 1978)
2	Landin-SECD Machine (1964)	ALGOL (1968)	CLOS (1988)		Π Calculus, (Milner, 1990s)
3	Scheme (1977)	Pascal (1970)	C++ (1985)		
4	ML (1973) (feature-types)	C (1972)	Java (1985)		
5	Haskell (1987) (feature-types) Purely function language		C# (2000)		
6	Racket				

6.2 Closures: Functions as values

Higher-order functions can take functions as arguments and can return functions.

```
> (define add
      (lambda (x y)
          (+ x y)))
```

Add function can be rewritten as-

```
> (define add
      (lambda (x) ;;  lambda (x) takes x, returns a function lambda (y)
          (lambda (y)
              (+ x y)))
> (add 3)
#(procedure>
> ((add 3) 4)
7
```

Examples of higher-order functions include computing derivatives- ($\frac{d}{dx}$ takes f, returns f') and function composition ($h = f \circ g$)

Higher-order functions require the domain for expressible values to contain functions-

$$EXPVAL = NUM \quad | \quad BOOL \quad | \quad FUNCTION$$

6.2.1 Functional Language

Recall the ojective to build a program to compute expressions of numbers. We add the Functional Language to the langauge previously consisting of Arithmetic, IF+DIV, Global and Lexical Scope components.

Abstract Syntax

$$
\begin{array}{rll}
 & \overline{n} & \text{NUM} \\
 & \overline{b} & \text{BOOL} \\
e ::= & ifte & e\ e\ e \\
 & \lambda & \overline{x}\ ...\ e \\
 & @ & e\ e\ ...
\end{array}
$$

Abstract syntax form- (number, boolean, if-then-else, procedure, application) Notice Arithmetic operators and LET are replaced by λ and @.

Concrete Syntax

$$
(let\ ([x\ e]\ ...)\ body) = ((\lambda\ (x\ ...) \\ body) \\ e\ ...)
$$

6.2.2 Evaluation Semantics

Expressible values -

$$
EXPVAL = NUM \oplus BOOL \oplus FUNCTION \\
DENVAL = EXPVAL
$$

<u>NOTE</u>: Evaluation semantics of a good language design requires precision to ensure bug-free logic and language. Ideally, EXPVAL should be partioned completely and thus should be a disjoint union.
Example: integer datatypes in C allows pointer values. Better design-

$$
EXPVAL = NUM \oplus BOOL \oplus POINTER
$$

Rules: For grammar Γ, expression e, EXPVAL v-

$$
\Gamma \vdash e \Rightarrow v
$$

($\vdash$: Turnstile operator, expression reads "Under Γ, e evaluates to v)

1. $\dfrac{}{\Gamma \vdash \overline{n} \Rightarrow n}$ NUM

2. $\dfrac{}{\Gamma \vdash \overline{b} \Rightarrow b}$ BOOL

3. $\dfrac{\Gamma(x) = v}{\Gamma \vdash x \Rightarrow v}$ ID

4. $\dfrac{\Gamma \vdash e_1 \Rightarrow \#t \quad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \mathit{ifte}\ e_1\ e_2\ e_3 \Rightarrow v_2}$ IFTE-TRUE

5. $\dfrac{\Gamma \vdash e_1 \Rightarrow \#f \quad \Gamma \vdash e_3 \Rightarrow v_3}{\Gamma \vdash \mathit{ifte}\ e_1\ e_2\ e_3 \Rightarrow v_3}$ IFTE-FALSE

Building an application: $\dfrac{}{\Gamma \vdash (\lambda\ (x\ ...\ )\ e) \Rightarrow v}$

6.2.3 Lexical Scope vs Dynamic Scope

Racket code demonstrating lexical scope

```
> (let ([x 5])
    (let ([f (lambda (y)   ;; lexical scope
               (+ x y))])
      (let ([x 0])         ;; dynamic scope
        (f 3))))
8
```

Lexical scope gives returns 8, dynamic scope returns 3.