Operational Semantics
Abstract Machines
and
Correctness

Roy L. Crole
University of Leicester, UK

Introduction

By the end of this introduction, you should be able to
- briefly explain the meaning of syntax and semantics;
- give a snap-shot overview of the course;
- explain what inductively defined sets are; and
- do simple rule inductions.

What’s Next? Background

- What is a Programming Language?
- What is Syntax?
- What is Semantics?

Syntax refers to particular arrangements of “words and letters” eg David hit the ball or
if t > 2 then H = Off.

- A grammar is a set of rules which can be used to specify how syntax is created.
- Examples can be seen in automata theory, or programming manuals.
- Theories of syntax and grammars can be developed—ideas are used in compiler construction.

Semantic descriptions are often informal. Consider

while (expression) command ;

adapted from Kernighan and Ritchie 1978/1988, p 224:

The command is executed repeatedly so long as the value of the expression remains unequal to 0: the expression must have arithmetic or pointer type. The execution of the (test) expression, including all side effects, occurs before each execution of the command.

We want to be more precise, more succinct.

Some Answers

- Programming Languages are formal languages used to “communicate” with a “computer”.
- Programming languages may be “low level”. They give direct instructions to the processor (instruction set architecture).
- Or “high level”. The instructions are indirect—being (eg) compiled for the processor—but much closer to concepts understood by the user (Java, C++, . . . ).

- Semantics is the study of “meaning”.
- In particular, syntax can be given meaning. The word run can mean
  - execution of a computer program,
  - spread of ink on paper, . . .
- Programming language syntax can be given a semantics—at least in theory! We need this to write meaningful programs . . .

Top Level view of Course

- Define syntax for programs $P$ and types $\sigma$.
- (define type assignments $P :: \sigma$);
- define operational semantics looking like

\[
(P, s) \Downarrow (V, s') \quad P \Downarrow V
\]

and compile $P$ and $V$ to abstract machine instructions

\[
P \mapsto [\{P\}] \quad V \mapsto \{V\}
\]

- Then prove correctness: $P \Downarrow V$ iff $[\{P\}] \mapsto^* \{V\}$
What’s Next? Inductively Defined Sets

- Specify inductively defined sets; programs, types etc will be defined this way. BNF grammars are a form of inductive definition; abstract syntax trees are also defined inductively.
- Define Rule Induction; properties of programs will be proved using this. It is important.

Inductively Defined Sets in General

- Given a set of rules, a deduction is a finite tree such that
  - each leaf node label c occurs as a base rule
    \((\varnothing, c) \in R\)
  - for any non-leaf node label c, if \(H\) is the set of children of c then \((H, c) \in R\) is an inductive rule.
- The set \(L\) inductively defined by \(R\) consists of those elements \(e\) which have a deduction with root node c. One may prove \(\forall e \in L \phi(e)\) for a property \(\phi(e)\) by rule induction. See the notes . . .

Chapter 1

By the end of this chapter, you should be able to

- describe the programs (syntax) of a simple imperative language called IMP;
- give a type system to IMP and derive types;
- explain the idea of evaluation relations;
- derive example evaluations.

Program Expressions and Types for IMP

The program expressions are given (inductively) by

\[
P ::= \mathcal{L} \quad \text{constant}
| l \quad \text{memory location}
| P \text{ iop } P' \quad \text{integer operator}
| P \text{ bop } P' \quad \text{boolean operator}
| l := P' \quad \text{assignment}
| P ; P' \quad \text{sequencing}
| \text{if } P \text{ then } P' \text{ else } P'' \quad \text{conditional}
| \text{while } P \text{ do } P' \quad \text{while loop}
\]

Each proposition is created by a deduction . . .

Example Inductive Definition

Let \(\text{Var}\) be a set of propositional variables. Then the set \(\text{Prop}\) of propositions of propositional logic is inductively defined by the rules

\[
\frac{\phi \in \text{Var}}{\phi} \quad (A) \\
\frac{\phi \land \psi \in \text{Prop}}{\phi \land \psi} \quad (\land) \\
\frac{\phi \lor \psi \in \text{Prop}}{\phi \lor \psi} \quad (\lor) \\
\frac{\phi \rightarrow \psi \in \text{Prop}}{\phi \rightarrow \psi} \quad (\rightarrow) \\
\frac{\phi \neg \psi \in \text{Prop}}{\phi \neg \psi} \quad (\neg) \\
\]

Example of Rule Induction

Consider the set of trees \(T\) defined inductively by

\[
\frac{T_1 \lor T_2 \in T}{n \in \mathbb{Z} \quad T_1, T_2} \\
\]

Let \(L(T)\) be the number of leaves in \(T\), and \(N(T)\) be the number of +−-nodes of \(T\). We prove (see board)

\[
\forall T \in T. \quad L(T) = N(T) + 1
\]

where the functions \(L, N : T \rightarrow \mathbb{N}\) are defined recursively by

- \(L(n) = 1\) and \(L(+(T_1, T_2)) = L(T_1) + L(T_2)\)
- \(N(n) = 0\) and \(N(+(T_1, T_2)) = N(T_1) + N(T_2) + 1\)

What’s Next? Types and Expressions

- We define the types and expressions of IMP.
- We give an inductive definition of a formal type system.

The types of the language \(\text{IMP}\) are given by the grammar

\[
\sigma ::= \text{int} \mid \text{bool} \mid \text{cmd}
\]

A location environment \(\mathcal{L}\) is a finite set of (location, type) pairs, with type being just int or bool:

\[
\mathcal{L} = l_1 : \text{int}, \ldots, l_n : \text{int}, l_{n+1} : \text{bool}, \ldots, l_m : \text{bool}
\]

Given \(\mathcal{L}\), then any \(P\) whose locations all appear in \(\mathcal{L}\) can (sometimes) be assigned a type; we write \(P : \sigma\) to indicate this, and define such type assignments inductively.
We define a notion of evaluation relation

\[ \langle \rangle \]

For example, updated evaluation relationship

\[ \langle \rangle \]

We derive deductions for

\[ \langle \rangle \]

We say that state \( s \) is updated at \( l \) by \( c \).
Chapter 2

By the end of this chapter you should be able to

- describe the “compiled” CSS machine, which executes compiled IMP programs;
- show how to compile to CSS instruction sequences;
- give some example executions.

Re-Write Rules (Abstract Machine)

\[ R_0 ightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow \ldots \rightarrow V \]

Defining the CSS Machine

- A CSS code \( C \) is a list:
  \[ C ::= \text{ins} : C \]

- The objects \( \text{ins} \) are CSS instructions. We will overload \( : \) to denote append; and write \( \xi \) for \( \xi : - \) (ditto below).

- A stack \( S \) is produced by the grammar
  \[ S ::= - | S : \xi \]

Motivating the CSS Machine

An operational semantics gives a useful model of IMP—we seek a more direct, “computational” method for evaluating configurations. If \( P \Downarrow^e V \), how do we “mechanically produce” \( V \) from \( P \)?

\[ P \equiv P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \equiv V \]

“Mechanically produce” can be made precise using a relation \( P \longrightarrow P' \) defined by rules with no hypotheses.

\[ n + m \longrightarrow m + n \]

An Example

Let \( s(l) = 6 \). Execute \( 10 - l \) on the CSS machine.
First, compile the program.

\[ [10 - l] = \text{FETCH}(l) : \text{PUSH}(10) : \text{OP}(\text{OP}) \]

Then

\[ \text{FETCH}(l) : \text{PUSH}(10) : \text{OP}(\text{OP}) \]
\[ \text{PUSH}(10) : \text{OP}(\text{OP}) \]
\[ \text{OP}(\text{OP}) \]
\[ 10 : 6 \]
\[ 6 \]
\[ 4 \]

- A CSS configuration is a triple \( (C,S,s) \).
- A CSS re-write takes the form
  \[ (C_1,S_1,s_1) \longrightarrow (C_2,S_2,s_2) \]
  and re-writes are specified inductively by rules with no hypotheses (such rules are often called axioms)

\[ (C_1,S_1,s_1) \longrightarrow (C_2,S_2,s_2) \]

- Note that the CSS re-writes are deterministic.

Midlands Graduate School, University of Birmingham, April 2008
Chapter 3

By the end of this chapter you should be able to

- describe the “interpreted” CSS machine, which executes IMP programs;
- explain the outline of a proof of correctness;
- explain some of the results required for establishing correctness, and the proofs of these results.

| g : C | S s | → | C g : S s |
| P₁ op P₂ : C | S s | → | P₂ : P₁ : op : C | S s |


Architecture of the Machine

- A CSS code C is a list of instructions which is produced by the following grammars:

  \( C ::= - | \text{ins : C} \quad \text{ins ::= P | op | STO(l) | BR(P₁, P₂)} \)

We will overload : to denote append; and write \( \xi ; \) (ditto below).

- A stack S is produced by the grammar

  \( S ::= - | \xi ; S \)


A Correctness Theorem

For all \( n \in \mathbb{Z}, b \in \mathbb{B}, P₁ ::= \text{int}, P₂ ::= \text{bool}, P₃ ::= \text{cmd} \) and \( s, s₁, s₂ \in \text{States} \) we have

\[
(P₁, s) \downarrow (n, s) \iff P₁ \rightarrow s₁ \rightarrow s₂ \rightarrow s
\]

\[
(P₂, s) \downarrow (b, s) \iff P₂ \rightarrow s₁ \rightarrow s₂ \rightarrow s
\]

\[
(P₃, s₁) \downarrow (\text{skip}, s₂) \iff P₃ \rightarrow s₁ \rightarrow s₂ \rightarrow s
\]

where \( \rightarrow \) denotes the transitive closure of \( \rightarrow \).


Proof Method

- \( \equiv \text{only if} \) by Rule Induction for \( \downarrow \).
- \( \iff \text{by Mathematical Induction on } k. \) Recall \( k \rightarrow k' \)

  \( \exists k \geq 1 (k \rightarrow k', k) \), where for \( k \geq 1, k \rightarrow k' \) \( k' \) means that

  \( (\forall 1 \leq i \leq k)(\exists k_i)(k \rightarrow k_i \rightarrow \ldots \rightarrow k_0 = k') \)

Then note if the \( \Box \) are configurations with \( \xi \) parameters

\[
(\forall \xi)(\Box \rightarrow k \Box) \implies (\Box \rightarrow \Box)
\]

\[
(\forall k)(\forall \xi)(\Box \rightarrow k \Box) \implies (\Box \rightarrow \Box)
\]

\[
(\Box \rightarrow k \Box)
\]


Code and Stack Extension

For all \( k \in \mathbb{N} \), and for all appropriate codes, stacks and states,

\[
\begin{align*}
C₁ & : S₁ & s₁ & \rightarrow^k & C₂ & : S₂ & s₂ \\
C₁ & : S₂ & s₂ & \rightarrow^k & C₂ & : S₁ & s₁
\end{align*}
\]

implies

\[
\begin{align*}
C₁ : C₂ & : S₁ & s₁ & \rightarrow^k & C₂ : C₁ & : S₂ & s₂ \\
C₂ & : C₁ & : S₁ & s₁ & \rightarrow^k & C₁ & : C₂ & : S₂ & s₂
\end{align*}
\]

where \( \rightarrow \) is reflexive closure of \( \rightarrow \).


Typing and Termination Yields Values

For all \( k \in \mathbb{N}, \) and for all appropriate codes, stacks, states,

\[
P ::= \text{int} \quad \text{and} \quad P : S s \rightarrow^k : S' s' \implies
\]

\[
\begin{align*}
s &= s' \quad \text{and} \quad S' = n : S \text{ some } n \in \mathbb{Z}
\end{align*}
\]

and similarly for Booleans.

Code Splitting

For all \( k \in \mathbb{N} \), and for all appropriate codes, stacks and states,

\[
P ::= \text{int} \quad \text{and} \quad P : S s \rightarrow^k : S' s' \implies
\]

\[
\begin{align*}
s &= s' \quad \text{and} \quad S' = n : S \text{ some } n \in \mathbb{Z}
\end{align*}
\]
Chapter 4

By the end of this chapter you should be able to

- describe the expressions and type system of a language with higher order functions;
- explain how to write simple programs;
- specify an eager evaluation relation;
- prove properties such as determinism.

Examples of FUN Declarations

```haskell
ghci> g :: Int -> Int -> Int
g x y = x*y

ghci> l1 :: [Int]
l1 = 5:6:8:4:Nil

ghci> h :: Int
h = hd (5:6:8:4:Nil)

ghci> length :: [Bool] -> Int
length l = if elist(l) then 0 else (1 + length t)
```

What's Next? Expressions and Types for FUN

- Define the expression syntax and type system.

FUN Types

- The types of FUN are
  
  \[ \sigma ::= \text{int} \mid \text{bool} \mid \sigma \to \sigma \mid [\sigma] \]

- We shall write
  
  \[ \sigma_1 \to \sigma_2 \to \sigma_3 \to \cdots \to \sigma_n \to \sigma \]

  for
  
  \[ \sigma_1 \to (\sigma_2 \to (\sigma_3 \to \cdots \to (\sigma_n \to \sigma)\cdots)). \]

  Thus for example \( \sigma_1 \to \sigma_2 \to \sigma_3 \) means \( \sigma_1 \to (\sigma_2 \to \sigma_3) \).
FUN Expressions

The expressions are

\[ E ::= x \quad \text{variables} \]
\[ c \quad \text{constants} \]
\[ K \quad \text{constant identifier} \]
\[ F \quad \text{function identifier} \]
\[ E_1 E_2 \quad \text{function application} \]
\[ tE \quad \text{tail of list} \]
\[ E_1 : E_2 \quad \text{cons for lists} \]
\[ \text{elist}(E) \quad \text{Boolean test for empty list} \]

Bracketing conventions apply . . .

What's Next? A Formal FUN Type System

- Show how to declare the types of variables and identifiers.
- Give some examples.
- Define a type assignment system.

Contexts (Variable Environments)

- When we write a FUN program, we shall declare the types of variables, for example
  \[ x :: \text{int}, y :: \text{bool}, z :: \text{bool} \]
- A context, variables assumed distinct, takes the form
  \[ \Gamma = \{ x_1 :: \sigma_1, \ldots, x_n :: \sigma_n \} \]

Identifier Environments

- When we write a FUN program, we want to declare the types of constants and functions.
- A simple example of an identifier environment is
  \[ K :: \text{bool}, \quad \text{map} :: (\text{int} \to \text{int}) \to ([\text{int}] \to [\text{int}]) \to \text{int} \to \text{int} \]
- An identifier type looks like \( \sigma_1 \to \sigma_2 \to \cdots \to \sigma_a \to \sigma \) where \( a \geq 0 \) and \( \sigma \) is NOT a function type.
- An identifier environment looks like
  \[ I = \{ I_1 :: \iota_1, \ldots, I_m :: \iota_m \} \]

Example Type Assignments

- With the previous identifier environment
  \[ x :: \text{int}, y :: \text{int}, z :: \text{bool} \]
  \[ \text{int} :: \text{map} \text{ suc} (x :: y :: z :: \text{nil}_{\text{int}}) :: \text{int} \]
- We have
  \[ \emptyset \vdash \text{if} \ I \ \text{then} \ \text{hd}(z :: \text{nil}_{\text{int}}) \ \text{else} \ \text{hd}(d :: e :: \text{nil}_{\text{int}}) :: \text{int} \]

Inductively Defining Type Assignments

Start with an identifier environment \( I \) and a context \( \Gamma \).

- Then

  \[ \Gamma \vdash x :: \sigma \quad \text{(where } x :: \sigma \in \Gamma) \]
  \[ \Gamma \vdash y :: \text{int} \]
  \[ \Gamma \vdash E_1 :: \text{int} \]
  \[ \Gamma \vdash E_2 :: \text{int} \]
  \[ \Gamma \vdash E_1 \text{ op} E_2 :: \text{int} \]

What's Next? Function Declarations and Programs

- Show how to code up functions.
- Define what makes up a FUN program.
- Give some examples.
Introducing Function Declarations

- To declare plus can write \( x + y = x + y \).
- To declare fac
  \[
  \text{fac} \ x = \begin{cases} 1 & \text{if } x = 1 \\ \text{else} \ x \ \text{fac} (x - 1) & \end{cases}
  \]
- And to declare that true denotes \( T \) we write \( \text{true} = T \).
- In \( \text{FUN}^c \), can specify (recursive) declarations
  \[
  K = E \quad F x = E' \quad G x y = E'' \ldots
  \]

An Example Program

Let \( I = F :: \text{int} \rightarrow \text{int} \rightarrow \text{int}, K :: \text{int} \). Then an identifier declaration \( dec \) is

\[
F x y = x + y
\]

\[
K = 10
\]

An example of a program is \( \text{dec} \) \( \in \text{FUN} \). Note that

\[
\emptyset \vdash E \ \leq \ K :: \text{bool}
\]

\[
x :: \text{int}, y :: \text{int} \vdash x + y :: \text{int}
\]

What's Next? Values and the Evaluation Relation

- Look at the notion of evaluation order.
- Define values, which are the results of eager program executions.
- Define an eager evaluation semantics: \( P \Downarrow^e V \).
- Give some examples.

Defining Programs

A program in \( \text{FUN}^c \) is a judgement of the form

\[
\text{dec} \ \in \ P
\]

where \( \text{dec} \) is a given identifier declaration and the program expression \( P \) satisfies a type assignment of the form

\[
\emptyset \vdash P :: \sigma \quad (\text{written } P :: \sigma)
\]

and

\[
\forall \ E F :: \sigma_F \in \text{dec}
\]

\[
\Gamma F \vdash E F :: \sigma_F
\]

Evaluation Orders

- The operational semantics of \( \text{FUN}^c \) says when a program \( P \) evaluates to a value \( V \). It is like the IMP evaluation semantics.
- Write this in general as \( P \Downarrow^e V \), and examples are
  \[
  3 + 4 + 10 \Downarrow^e 17
  \]
  \[
  \text{hd}(\text{nil}) \Downarrow^e 2
  \]

Defining and Explaining (Eager) Values

- Let \( dec \) be an identifier declaration, with typical typing
  \[
  F :: \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \ldots \rightarrow \sigma_a \rightarrow \sigma
  \]

  Informally \( a \) is the maximum number of inputs taken by \( F \). A value expression is any expression \( V \) produced by

  \[
  V ::= \epsilon | \text{nil} | F \bar{V} | V : V
  \]

  where \( \bar{V} \) abbreviates \( V_1, V_2, \ldots, V_{a-1}, V_1 \) and \( 0 \leq k < a \).
- Note also that \( k \) is strictly less than \( a \), and that if \( a = 1 \) then \( F \bar{V} \) denotes \( F \).
- A value is any value expression for which $\text{dec}_1$ in $V$ is a valid $\text{FUN}^\epsilon$ program.
- Suppose that $F : \text{int} \rightarrow \text{int} \rightarrow \text{int}$ and that $P_1 \Downarrow F$ and $P_2 \Downarrow \epsilon$ with $P_1$ not values. Then

\[
\begin{array}{c|c|c|c}
F & P & V \\
\hline
P_1 & F & F \Downarrow 1 \\
F P_1 & F & F \Downarrow \epsilon \\
F . P_2 & F & F \Downarrow \epsilon \\
\end{array}
\]

where either $P_1$ or $P_2$ is a value $\Downarrow V$.

\[
\begin{array}{c}
\text{AP} \Downarrow E_f[V_1, \ldots, V_n, s_1, \ldots, s_m] \Downarrow V_f \Downarrow V \\
\text{FID} \Downarrow \epsilon_f \Downarrow V_f \Downarrow V \\
\text{CID} \Downarrow K \Downarrow \epsilon_f \Downarrow V_f \Downarrow V \\
\end{array}
\]

Examples of Evaluations

Suppose that $\text{dec}_1$ is

\[
G x = x \times 2 \\
K = \frac{3}{2}
\]

\[
\begin{array}{c}
\text{VAL} \Downarrow \frac{1}{2} \Downarrow \frac{3}{2} \\
\text{CID} \Downarrow \frac{1}{2} \Downarrow \frac{3}{2} \\
\text{FID} \Downarrow \frac{1}{2} \Downarrow \frac{3}{2} \\
\end{array}
\]

We can prove that

\[
F \Downarrow \frac{2}{2} (4 + 1) \Downarrow 10
\]

where $FXYZ = XYZ + Z$ as follows:

\[
\begin{array}{c}
\Downarrow \text{VAL} \Downarrow \frac{4}{2} \Downarrow \frac{1}{2} \Downarrow 1 \\
\Downarrow \text{FID} \Downarrow \frac{2}{2} \Downarrow \frac{3}{2} \Downarrow 10
\end{array}
\]

What's Next? $\text{FUN}^\epsilon$ Properties of Eager Evaluation

- Explain and define determinism.
- Explain and define subject reduction, that is, preservation of types during program execution.
• The evaluation relation for $\text{FUN}$ is deterministic. More precisely, for all $P$, $V_1$ and $V_2$, if

$$P \Downarrow V_1 \quad \text{and} \quad P \Downarrow V_2$$

then $V_1 = V_2$. (Thus $\Downarrow$ is a partial function.)

• Evaluating a program $\text{dec}_1$ in $P$ does not alter its type. More precisely,

$$(\varnothing \Downarrow P :: \sigma \text{ and } P \Downarrow V) \implies \varnothing \Downarrow V :: \sigma$$

for any $P$, $V$, $\sigma$ and $\text{dec}_1$. The conservation of type during program evaluation is called subject reduction.

### Properties of $\text{FUN}$

#### Chapter 5

By the end of this chapter you should be able to

• describe the SECD machine, which executes compiled $\text{FUN}$ programs; here the expressions $\text{Exp}$ are defined by $E ::= x | n | F | E E$;

• show how to compile to SECD instruction sequences;

• write down example executions.

#### Architecture of the Machine

• The SECD machine consists of rules for transforming SECD configurations $(S,E,C,D)$.

• The non-empty stack $S$ is generated by

$$S ::= \frac{n}{\uparrow} \mid \text{clo}_F S_1 \ldots S_l$$

• Each node occurs at a level $\geq 1$.

• A stack $S$ has a height the maximum level of any $\text{clo}_F$, or 0 otherwise.

Given any other stack $S_{l+1}$ there is a stack $S''$

$$S_{l+1} S_l \ldots S_1 \quad \text{clo}_F \quad \uparrow$$

• Write $S_{l+1} \uplus S$ for $S$ with $S'$ replaced by $S''$.

• A SECD code $C$ is a list which is produced by the following grammars:

$$\text{ins ::= } x | n | F | \text{APP}$$

$$C ::= - | \text{ins} : C$$

• A typical dump looks like

$$(S_1,E_1,C_1,(S_2,E_2,C_2,\ldots,(S_n,E_n,C_n,\ldots))$$

• We will overload $\uplus$ to denote append; and write $\xi$ for $\uplus$.

#### The environment $E$ takes the form

$$x_1 = ?S_1 : \ldots : x_n = ?S_n.$$  

The value of each $e$ is determined by the form of an $S_i$.

• If $S_i$ is $\frac{n}{\uparrow}$ then $e = 0$; if $S_i$ is $\text{clo}_F$ then $e = 1$; in any other case, $e$ is $\perp 1$.

We define a compilation function $[\cdot]: \text{Exp} \to \text{SECDcodes}$ which takes an SECD expression and turns it into code.

$$[x] \overset{\text{def}}{=} x$$  

$$[n] \overset{\text{def}}{=} n$$  

$$[F] \overset{\text{def}}{=} F$$  

$$[E_1 E_2] \overset{\text{def}}{=} [E_1] : [E_2] : \text{APP}$$
There is a representation of program values as stacks, given by

- $(\alpha \delta) \overset{\text{def}}{=} \alpha \delta$

- $(F V_1 \ldots V_k) \overset{\text{def}}{=} e_{\text{prog}} (\alpha \delta)^k = (\alpha \delta)^0 \oplus (\alpha \delta)^1 \oplus \ldots \oplus (\alpha \delta)^k$

Recall $k < a$ with $a$ the arity of $F$.

### The Re-writes

A number is pushed onto the stack (the initial stack can be of any status):

<table>
<thead>
<tr>
<th>S</th>
<th>$\alpha \delta \oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>$\mathbf{x} : C$</td>
</tr>
<tr>
<td>D</td>
<td>$\mathbf{D}$</td>
</tr>
</tbody>
</table>

A variable’s value is pushed onto the stack, provided that the environment $E$ contains $\mathbf{x} = \mathbf{\tau} \equiv [\alpha \delta \mathbf{T}]$ (where $\delta$ is 0 or 1). Note that by definition, the status of $\mathbf{T}$ determines the status of the re-written stack:

<table>
<thead>
<tr>
<th>S</th>
<th>$\alpha \delta \oplus \mathbf{T} \oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>$\mathbf{x} : C$</td>
</tr>
<tr>
<td>D</td>
<td>$\mathbf{D}$</td>
</tr>
</tbody>
</table>

An APP command creates an application value, type 0:

<table>
<thead>
<tr>
<th>S</th>
<th>$\mathbf{S}_1 \ldots \mathbf{S}_n$</th>
<th>$\alpha \delta \oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\mathbf{APP} : C$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\mathbf{D}$</td>
<td></td>
</tr>
</tbody>
</table>

An APP command creates an application value, type 1:

<table>
<thead>
<tr>
<th>S</th>
<th>$\mathbf{S}_1 \ldots \mathbf{S}_n$</th>
<th>$\alpha \delta \oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\mathbf{APP} : C$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\mathbf{D}$</td>
<td></td>
</tr>
</tbody>
</table>

An APP command produces an application value from an application value:

<table>
<thead>
<tr>
<th>S</th>
<th>$\mathbf{S}_1 \ldots \mathbf{S}_n$</th>
<th>$\alpha \delta \oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\mathbf{APP} : C$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\mathbf{D}$</td>
<td></td>
</tr>
</tbody>
</table>

An APP command calls a function, type 0:

<table>
<thead>
<tr>
<th>S</th>
<th>$\mathbf{S}_1 \ldots \mathbf{S}_n$</th>
<th>$\alpha \delta \oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$\mathbf{E}$</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\mathbf{APP} : C$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\mathbf{D}$</td>
<td>$(\alpha - 1) \mathbf{S}, \mathbf{E}, \mathbf{C}, \mathbf{D}$</td>
</tr>
</tbody>
</table>
An APP command calls a function, type 1:

<table>
<thead>
<tr>
<th>S</th>
<th>$\alpha$</th>
<th>$S_{\alpha-1} \cdots S_1$</th>
<th>cloy</th>
<th>$\oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$\alpha = 5$</td>
<td>$S_5$ \ldots $S_1$</td>
<td>$\oplus S$</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\alpha = 2$</td>
<td>$S_3$ \ldots $S_1$</td>
<td>$\oplus S$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\alpha = 2$</td>
<td>$S_2$ \ldots $S_1$</td>
<td>$\oplus S$</td>
<td></td>
</tr>
</tbody>
</table>

Restore, where the final status is determined by the initial status:

<table>
<thead>
<tr>
<th>S</th>
<th>$[\alpha] \beta = T$</th>
<th>$\oplus S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$\alpha = 5$</td>
<td>$T$</td>
</tr>
<tr>
<td>C</td>
<td>$\alpha = 2$</td>
<td>$T$</td>
</tr>
<tr>
<td>D</td>
<td>$\alpha = 2$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Suppose that $K$, $N$ and $MN$ are functions which are also values, and that:

$F \equiv x = x$  
$I a b = b$  
$H \equiv L (M N) z$

Then:

$[(F (H 4)) (I 2 K)] = (1. \text{def } F) : H : 4 : \text{APP} : I : 2 : \text{APP} : K : \text{APP} : (\text{APP def } 1.)$

and:

$[L (M N) z] \equiv 7, \text{def } L : M : N : \text{APP} : z : \text{APP def } 1.$

Chapter 6

By the end of this chapter you should be able to:

- explain the outline of a proof of correctness;
- explain some of the results required for establishing correctness, and the proofs of these results.
A Correctness Theorem

For all programs $\text{dec}_i$ in $P$ for which $\emptyset \vdash P :: \sigma$ we have

\[
P \models^* V \iff (V)
\]

- Need to prove “lemma plus”: if $D = (S', E', C', D')$ we can also similarly arbitrarily extend any of the stacks and codes in $D$ (say to $D'$).
- We use induction on $k$. Suppose lemma plus is true for all $k < k_0$. Must prove we can extend any rewrite $M \rightarrow^{k_0+1} M'$ to $\overline{M} \rightarrow^{k_0+1} \overline{M'}$. By determinism, we have $M \rightarrow^1 M'' \rightarrow^{k_0} M'$.
- If no function call during $M \rightarrow^1 M''$, trivial to extend to get $\overline{M} \rightarrow^1 \overline{M''}$. And by induction, $\overline{M''} \rightarrow^{k_0} \overline{M'}$.

By induction, we have

\[
M' \overset{\text{def}}{=} (S, E, \sigma, C, D)
\]

It is easy to see that $\overline{M} \rightarrow^1 \overline{M''}$, and obviously

Code Splitting

For any stacks, environments, codes, and dumps, if $C_1$ and $C_2$ are non-empty then

\[
\begin{array}{c|c|c|c}
S & S & S' \\
E & E & E' \\
C & C' & C'' \\
D & D & D \\
\end{array}
\]

implies that

If there is a function call, there are $k_1$ and $k_2$ such that

\[
M' \overset{\text{def}}{=} (S, E', \sigma', C', D')
\]

If $k_2 = 0$ then we are done.

If $k_2 \geq 1$ then we can similarly extend the stack and code of the final $k_2 \geq 1$ transitions by induction

\[
\begin{array}{c|c|c|c}
S & S'' & S' \\
E & E' & E'' \\
C & C' & C'' \\
D & D & D \\
\end{array}
\]

where $k = k_1 + k_2$. 
Program Code Factors Through Value Code

For any well typed \textsc{fun} program \textit{dec} \textsubscript{1} in \textit{P} where \textit{P} :: \sigma and \textit{P} \not\equiv \sigma \textit{V},

\[
\begin{array}{c|c|c|c|c|c|c}
S & S & S & S & S & S & S \\
E & E & E & E & E & E & E \\
C & C & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

\[\text{implies } (\exists \overline{k} \leq k) \]

with equality only if \textit{P} is a value (and hence equal to \textit{V}).

Suppose that

\[
\begin{array}{c|c|c|c|c|c|c}
S & S & S & S' & S' & S' & S' \\
E & E & E & E & E & E & E \\
D & D & D & D & D & D & D \\
\end{array}
\]

Then appealing to splitting and the induction hypothesis, we get

\[
\begin{array}{c|c|c|c|c|c|c}
S & S & S & S' & S' & S' & S' \\
E & E & E & E & E & E & E \\
C & [P_1] & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

Appealing to splitting again, and by induction,

\[
\begin{array}{c|c|c|c|c|c|c}
S & (V_2) \oplus (F \overline{V}) \oplus S & S & (V_2) \oplus (F \overline{V}) \oplus S & S & (F \overline{V}) \oplus S & S \\
E & E & E & E & E & E & E \\
C & [P_2] & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

and

\[
\begin{array}{c|c|c|c|c|c|c}
S & (V_2) \oplus (F \overline{V}) \oplus S & S & S' & S' & S' & S' \\
E & E & E & E & E & E & E \\
C & APP & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

where \textit{P}_2 \not\equiv V_2.

Proving the Theorem

\[\text{\textit{(}\iff\textit{)}}: \text{We shall prove that if } \textit{P} :: \sigma \text{ then}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
S & S & S & S' & S' & S' & S' \\
E & E & E & E & E & E & E \\
C & [P_1] & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

\[\text{implies } (\exists V) \quad S' = ([V] \oplus S \text{ and } \textit{P} \not\equiv V)\]

from which the required result follows. Induction on \textit{k}. If \textit{P} is a number or a function the result is trivial. Else \textit{P} has the form \textit{P}_1 \textit{P}_2.

\[
\begin{array}{c|c|c|c|c|c|c}
S & (F \overline{V}) \oplus S & S & (F \overline{V}) \oplus S & S & S' & S' \\
E & E & E & E & E & E & E \\
C & [P_2] : \text{APP} & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

and

where \textit{P}_1 \not\equiv F \overline{V}.

By factorization on \textit{P}_1 and \textit{P}_2, and extension we have (check!)

\[
\begin{array}{c|c|c|c|c|c|c}
S & S & S & S & S & S & S \\
E & E & E & E & E & E & E \\
C & [V_2] & C & C & C & C & C \\
D & D & D & D & D & D & D \\
\end{array}
\]

and so if \textit{P}_1 \textit{P}_2 is not a value then

\[k_1 + k_2 + k_3 < k_0 + 1\]

and by induction \textit{S'} = ([V] \oplus S) for some \textit{V} where \textit{F} \overline{V} \textit{V}_2 \not\equiv V.

Hence \textit{P}_1 \textit{P}_2 \not\equiv V as required.

If \textit{P}_1 \textit{P}_2 is a value, refer to part (\iff_{\text{num}}) of the proof, case \not\equiv \textit{val}.