# 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?

## 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++, . . . ).

- Syntax refers to particular arrangements of "words and letters" eg David hit the ball or

$$\text{if } t > 2 \text{ then } H = \text{Off}.$$

- 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.

- 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 . . .

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.**

# Top Level view of Course

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

$$\boxed{(P, s) \Downarrow (V, s')} \qquad \boxed{P \Downarrow V};$$

- and compile $P$ and $V$ to abstract machine instructions $\boxed{P \mapsto [\![P]\!]}$ and $\boxed{V \mapsto (\!|V|\!)}$
- Then prove correctness: $P \Downarrow V$ *iff* $[\![P]\!] \longmapsto^{t} (\!|V|\!)$

## 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.

# Example Inductive Definition

Let *Var* be a set of propositional variables. Then the set *Prpn* of propositions of propositional logic is inductively defined by the rules

$$\frac{}{P}\,[P \in \mathit{Var}] \;\; (A) \qquad \frac{\phi \quad \psi}{\phi \wedge \psi}\,(\wedge)$$

$$\frac{\phi \quad \psi}{\phi \vee \psi}\,(\vee) \qquad \frac{\phi \quad \psi}{\phi \rightarrow \psi}\,(\rightarrow) \qquad \frac{\phi}{\neg\phi}\,(\neg)$$

Each proposition is created by a deduction . . .

# 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 $(\varnothing, c) \in \mathcal{R}$
  - for any non-leaf node label $c$, if $H$ is the set of children of $c$ then $(H, c) \in \mathcal{R}$ is an inductive rule.
- The set $I$ inductively defined by $\mathcal{R}$ consists of those elements $e$ which have a deduction with root node $e$. One may prove $\forall e \in I. \boxed{\phi(e)}$ for a property $\phi(e)$ by rule induction. See the notes ...

# Example of Rule Induction

Consider the set of trees $\mathcal{T}$ defined inductively by

$$\frac{}{n}\ [n \in \mathbb{Z}] \qquad\qquad \frac{T_1 \quad T_2}{+(T_1, T_2)}$$

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

$$\forall T \in \mathcal{T}. \quad \boxed{L(T) = N(T) + 1}$$

where the functions $L, N \colon \mathcal{T} \to \mathbb{N}$ are defined recursively by

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

# 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.

## What's Next? Types and Expressions

- We define the types and expressions of $\mathbb{IMP}$.
- We give an inductive definition of a formal type system.

# Program Expressions and Types for IMP

The program expressions are given (inductively) by

| | | |
|---|---|---|
| $P$ | $::= \ \underline{c}$ | constant |
| | $\mid \ l$ | memory location |
| | $\mid \ P \ iop \ P'$ | integer operator |
| | $\mid \ P \ bop \ P'$ | boolean operator |
| | $\mid \ l := P'$ | assignment |
| | $\mid \ P \, ; P'$ | sequencing |
| | $\mid \ \mathsf{if}\ P\ \mathsf{then}\ P'\ \mathsf{else}\ P''$ | conditional |
| | $\mid \ \mathsf{while}\ P\ \mathsf{do}\ P'$ | while loop |

- The types of the language $\mathbb{IMP}$ are given by the grammar

$$\sigma \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \mathsf{cmd}$$

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

$$\mathcal{L} = l_1 :: \mathsf{int}, \dots, l_n :: \mathsf{int},\ l_{n+1} :: \mathsf{bool}, \dots, l_m :: \mathsf{bool}$$

- 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.

$$\frac{}{\underline{n} :: \mathsf{int}}\ [\text{any } n \in \mathbb{Z}] \qquad \frac{}{\underline{T} :: \mathsf{bool}} \qquad \frac{}{\underline{F} :: \mathsf{bool}}$$

$$\frac{}{l :: \mathsf{int}}\ [l :: \mathsf{int} \in \mathcal{L}] \qquad \frac{P_1 :: \mathsf{int} \quad P_2 :: \mathsf{int}}{P_1\ bop\ P_2 :: \mathsf{bool}}\ [\ bop \in BOpr]$$

$$\frac{}{\mathsf{skip} :: \mathsf{cmd}} \qquad \frac{l :: \sigma \quad P :: \sigma}{l := P :: \mathsf{cmd}}$$

$$\frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd} \quad P_3 :: \mathsf{cmd}}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 :: \mathsf{cmd}} \qquad \frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd}}{\mathsf{while}\ P_1\ \mathsf{do}\ P_2 :: \mathsf{cmd}}$$

# Example: Deduction of a Type Assignment

$$\dfrac{\dfrac{\dfrac{}{l :: \mathsf{int}} \qquad \dfrac{}{\underline{5} :: \mathsf{int}}}{l \,\underline{\geq}\, \underline{5} :: \mathsf{bool}} \qquad \mathcal{D}2 \qquad \dfrac{\mathcal{D}3 \quad \mathcal{D}4}{l := l - 1 \,;\, l' := l' * l :: \mathsf{cmd}}}{\mathsf{if}\ l \,\underline{\geq}\, 5\ \mathsf{then}\ l' := \underline{1}\ \mathsf{else}\ (l := l + 1 \,;\, l' := l' * l) :: \mathsf{cmd}}$$

## What's Next? An Evaluation Relation

- We define a notion of state.
- We define an evaluation relation for $\mathbb{IMP}$.
- We look at an example.

# States

- A state $s$ is a finite partial function $Loc \rightarrow \mathbb{Z} \cup \mathbb{B}$.
- For example $s = \langle l_1 \mapsto 4, l_2 \mapsto T, l_3 \mapsto 21 \rangle$
- There is a state denoted by $s\{l \mapsto c\} : Loc \rightarrow \mathbb{Z} \cup \mathbb{B}$ which is the partial function

$$(s\{l \mapsto c\})(l') \stackrel{\text{def}}{=} \begin{cases} c & \text{if } l' = l \\ s(l') & \text{otherwise} \end{cases}$$

- We say that state $s$ is updated at $l$ by $c$.

# An Evaluation Relation

Consider the following evaluation relationship

$$( \ l' := \underline{T} \, ; l := \underline{4} + \underline{1} \ , \ \langle\rangle \ ) \Downarrow ( \ \mathsf{skip} \ , \ \langle l' \mapsto T, l \mapsto 5 \rangle \ )$$

The idea is

$$\textit{Starting program} \Downarrow \textit{final result}$$

We describe an operational semantics which has assertions which look like

$$(P\,,s) \Downarrow (\underline{c}\,,s) \qquad \text{and} \qquad (P\,,s_1) \Downarrow (\mathsf{skip}\,,s_2)$$

$$\frac{}{(l\,,\,s) \Downarrow (\underline{s(l)}\,,\,s)}\ [\text{ provided } l \in \text{ domain of } s] \Downarrow\text{LOC}$$

$$\frac{(P_1\,,\,s) \Downarrow (\underline{n_1}\,,\,s) \qquad (P_2\,,\,s) \Downarrow (\underline{n_2}\,,\,s)}{(P_1\ op\ P_2\,,\,s) \Downarrow (\underline{n_1\ op\ n_2}\,,\,s)} \Downarrow\text{OP}$$

$$\frac{(P\,,\,s) \Downarrow (\underline{c}\,,\,s)}{(l := P\,,\,s) \Downarrow (\mathsf{skip}\,,\,s\{l \mapsto c\})} \Downarrow\text{ASS}$$

$$\frac{(P_1\,,\,s_1) \Downarrow (\mathsf{skip}\,,\,s_2) \qquad (P_2\,,\,s_2) \Downarrow (\mathsf{skip}\,,\,s_3)}{(P_1\,;\,P_2\,,\,s_1) \Downarrow (\mathsf{skip}\,,\,s_3)} \Downarrow\text{SEQ}$$

$$\frac{(P\,,s_1)\Downarrow(\underline{F}\,,s_1)\qquad(P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}{(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}\ \Downarrow\mathrm{COND}_2$$

$$\frac{(P_1\,,s_1)\Downarrow(\underline{T}\,,s_1)\qquad(P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)\qquad(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s_2)\Downarrow(\mathsf{skip}\,,s_3)}{(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_3)}$$

$$\frac{(P_1\,,s)\Downarrow(\underline{F}\,,s)}{(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s)\Downarrow(\mathsf{skip}\,,s)}\ \Downarrow\mathrm{LOOP}_2$$

# Example Evaluations

We derive deductions for

$$((\underline{3}+\underline{2})*\underline{6}\,,\,s) \Downarrow (\underline{30}\,,\,s)$$

and

$$(\mathsf{while}\; l = \underline{1}\; \mathsf{do}\; l := l - \underline{1}\,,\, \langle l \mapsto 1\rangle) \Downarrow (\mathsf{skip}\,,\, \langle l \mapsto 0\rangle)$$

# 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.

# Motivating the CSS Machine

An operational semantics gives a useful model of $\mathbb{IMP}$—we seek a more direct, "computational" method for evaluating configurations. If $P \Downarrow^e V$, how do we "mechanically produce" $V$ from $P$?

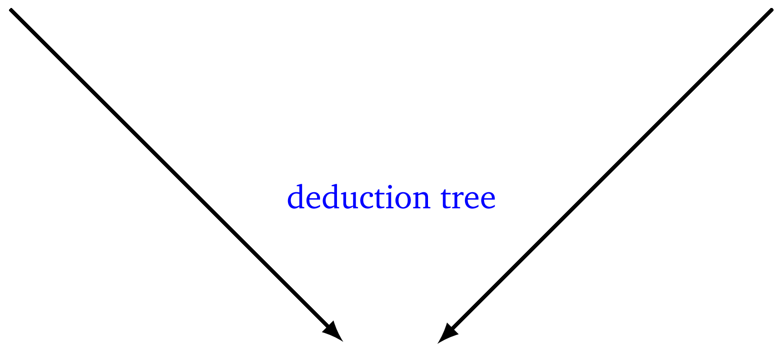
$$P \equiv P_0 \mapsto P_1 \mapsto P_2 \mapsto \ldots \mapsto P_n \equiv V$$

"Mechanically produce" can be made precise using a relation $P \longmapsto P'$ defined by rules with no hypotheses.

$$\frac{}{\underline{n} + \underline{m} \longmapsto \underline{m + n}}$$

$$P_0 \longmapsto P_1 \longmapsto P_2 \longmapsto P_3 \longmapsto P_4 \ldots \longmapsto V$$

Re-Write Rules (Abstract Machine)




$$P \quad \Downarrow^e \quad V$$

Evaluation Semantics

# An Example

Let $s(l) = 6$. Execute $\underline{10} - l$ on the CSS machine.

First, compile the program.

$$[\![\underline{10} - l]\!] \quad = \quad \mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-)$$

Then

$$\begin{array}{ll} & \boxed{\mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-) \parallel - \parallel s} \\ \longmapsto & \boxed{\mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-) \parallel \underline{6} \parallel s} \\ \longmapsto & \boxed{\mathsf{OP}(-) \parallel \underline{10} : \underline{6} \parallel s} \\ \longmapsto & \boxed{- \parallel \underline{4} \parallel s} \end{array}$$

# Defining the CSS Machine

- A CSS code $C$ is a list:

$$C \quad ::= \quad - \mid ins : C$$

$$ins \quad ::= \quad \mathsf{PUSH}(\underline{c}) \mid \mathsf{FETCH}(l) \mid \mathsf{OP}(op) \mid \mathsf{SKIP}$$

$$\mid \mathsf{STO}(l) \mid \mathsf{BR}(C,C) \mid \mathsf{LOOP}(C,C)$$

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

- A stack $S$ is produced by the grammar

$$S ::= - \mid \underline{c} : S$$

- A CSS configuration is a triple $(C, S, s)$.
- A CSS re-write takes the form

$$(C_1, S_1, s_1) \longmapsto (C_2, S_2, s_2)$$

and re-writes are specified inductively by rules with no hypotheses (such rules are often called axioms)

$$\frac{}{(C_1, S_1, s_1) \longmapsto (C_2, S_2, s_2)} \; R$$

- Note that the CSS re-writes are deterministic.

$$\mathsf{PUSH}(\underline{c}) : C \;\|\; S \;\|\; s \quad\longmapsto\quad C \;\|\; \underline{c} : S \;\|\; s$$

$$\mathsf{FETCH}(l) : C \;\|\; S \;\|\; s \quad\longmapsto\quad C \;\|\; \underline{s(l)} : S \;\|\; s$$

$$\mathsf{OP}(\, op \,) : C \;\|\; \underline{n_1} : \underline{n_2} : S \;\|\; s \quad\longmapsto\quad C \;\|\; \underline{n_1 \; op \; n_2} : S \;\|\; s$$

$$\mathsf{STO}(l) : C \;\|\; \underline{c} : S \;\|\; s \quad\longmapsto\quad C \;\|\; S \;\|\; s\{l \mapsto c\}$$

$$\mathsf{BR}(C_1, C_2) : C \;\|\; \underline{F} : S \;\|\; s \quad\longmapsto\quad C_2 : C \;\|\; S \;\|\; s$$

$$\mathsf{LOOP}(C_1, C_2) : C \;\|\; S \;\|\; s \quad\longmapsto$$

$$C_1 : \mathsf{BR}(C_2 : \mathsf{LOOP}(C_1, C_2), \mathsf{SKIP}) : C \;\|\; S \;\|\; s$$

$$[\![\underline{c}]\!] \stackrel{\text{def}}{=} \mathsf{PUSH}(\underline{c})$$

$$[\![l]\!] \stackrel{\text{def}}{=} \mathsf{FETCH}(l)$$

$$[\![P_1\ op\ P_2]\!] \stackrel{\text{def}}{=} [\![P_2]\!] : [\![P_1]\!] : \mathsf{OP}(op)$$

$$[\![l := P]\!] \stackrel{\text{def}}{=} [\![P]\!] : \mathsf{STO}(l)$$

$$[\![\mathsf{skip}]\!] \stackrel{\text{def}}{=} \mathsf{SKIP}$$

$$[\![P_1\ ;\ P_2]\!] \stackrel{\text{def}}{=} [\![P_1]\!] : [\![P_2]\!]$$

$$[\![\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2]\!] \stackrel{\text{def}}{=} [\![P]\!] : \mathsf{BR}([\![P_1]\!], [\![P_2]\!])$$

$$[\![\mathsf{while}\ P_1\ \mathsf{do}\ P_2]\!] \stackrel{\text{def}}{=} \mathsf{LOOP}([\![P_1]\!], [\![P_2]\!])$$

# 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.

# Architecture of the Machine

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

$$C ::= - \mid ins : C \qquad ins ::= P \mid \; op \; \mid \mathsf{STO}(l) \mid \mathsf{BR}(P_1, P_2)$$

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

- A stack $S$ is produced by the grammar

$$S ::= - \mid \underline{c} : S$$

$$\underline{n} : C \parallel S \parallel s \longmapsto C \parallel \underline{n} : S \parallel s$$

$$P_1 \; op \; P_2 : C \parallel S \parallel s \longmapsto P_2 : P_1 : op : C \parallel S \parallel s$$

$$op : C \parallel \underline{n_1} : \underline{n_2} : S \parallel s \longmapsto C \parallel \underline{n_1 \; op \; n_2} : S \parallel s$$

$$l := P : C \parallel S \parallel s \longmapsto P : \mathsf{STO}(l) : C \parallel S \parallel s$$

$$\mathsf{STO}(l) : C \parallel \underline{n} : S \parallel s \longmapsto C \parallel S \parallel s\{l \mapsto n\}$$

$$\mathsf{while}\; P_1 \;\mathsf{do}\; P_2 : C \parallel S \parallel s \longmapsto$$

$$P_1 : \mathsf{BR}((P_2 \,;\, \mathsf{while}\; P_1 \;\mathsf{do}\; P_2), \mathsf{skip}) : C \parallel S \parallel s$$

# A Correctness Theorem

For all $n \in \mathbb{Z}$, $b \in \mathbb{B}$, $P_1 :: \mathsf{int}$, $P_2 :: \mathsf{bool}$, $P_3 :: \mathsf{cmd}$ and $s, s_1, s_2 \in \mathit{States}$ we have

$$(P_1\,,\,s) \Downarrow (\underline{n}\,,\,s) \quad \text{iff} \quad \boxed{P_1}\,\boxed{-}\,\boxed{s} \overset{t}{\longmapsto} \boxed{-}\,\boxed{\underline{n}}\,\boxed{s}$$

$$(P_2\,,\,s) \Downarrow (\underline{b}\,,\,s) \quad \text{iff} \quad \boxed{P_2}\,\boxed{-}\,\boxed{s} \overset{t}{\longmapsto} \boxed{-}\,\boxed{\underline{b}}\,\boxed{s}$$

$$(P_3\,,\,s_1) \Downarrow (\mathsf{skip}\,,\,s_2) \quad \text{iff} \quad \boxed{P_3}\,\boxed{-}\,\boxed{s_1} \overset{t}{\longmapsto} \boxed{-}\,\boxed{-}\,\boxed{s_2}$$

where $\overset{t}{\longmapsto}$ denotes the transitive closure of $\longmapsto$.

# Proof Method

- $\Longrightarrow_{onlyif}$ by Rule Induction for $\Downarrow$.

- $\Longleftarrow_{if}$ by Mathematical Induction on $k$. Recall $\kappa \longmapsto^t \kappa'$ iff $(\exists\, k \geq 1)(\kappa \longmapsto^k \kappa')$, where for $k \geq 1$, $\kappa \longmapsto^k \kappa'$ means that

$$(\forall 1 \leq i \leq k)(\exists \kappa_i)(\kappa \longmapsto \kappa_1 \longmapsto \ldots \longmapsto \kappa_k = \kappa')$$

Then note if the $\square$ are configurations with $\xi$ parameters

$$(\forall \xi)(\ (\exists k)(\square \longmapsto^k \square)\ \mathit{implies}\ \square \Downarrow \square\ )$$

$\equiv$

$$(\forall k)\underbrace{(\forall \xi)\ \boxed{(\square \longmapsto^k \square\ \mathit{implies}\ \square \Downarrow \square)}}_{\phi(k)}$$

# Code and Stack Extension

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

$$\boxed{C_1} \| \boxed{S_1} \| \boxed{s_1} \longmapsto^k \boxed{C_2} \| \boxed{S_2} \| \boxed{s_2}$$

*implies*

$$\boxed{C_1 : C_3} \| \boxed{S_1 : S_3} \| \boxed{s_1} \longmapsto^k \boxed{C_2 : C_3} \| \boxed{S_2 : S_3} \| \boxed{s_2}$$

where $\longmapsto^0$ is reflexive closure of $\longmapsto$.

# Code Splitting

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

$$\boxed{C_1 : C_2 \parallel S \parallel s} \longmapsto^{k} \boxed{- \parallel S'' \parallel s''}$$

then there is a stack and state $S'$ and $s'$, and $k_1, k_2 \in \mathbb{N}$ for which

$$\boxed{C_1 \parallel S \parallel s} \longmapsto^{k_1} \boxed{- \parallel S' \parallel s'}$$

$$\boxed{C_2 \parallel S' \parallel s'} \longmapsto^{k_2} \boxed{- \parallel S'' \parallel s''}$$

where $k_1 + k_2 = k$.

# Typing and Termination Yields Values

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

$$P :: \mathsf{int} \quad \textit{and} \quad \boxed{P \parallel S \parallel s} \longmapsto^{k} \boxed{- \parallel S' \parallel s'} \quad \textit{implies}$$

$$s = s' \quad \textit{and} \quad S' = \underline{n} : S \text{ some } n \in \mathbb{Z}$$

$$\textit{and} \quad \boxed{P \parallel - \parallel s} \longmapsto^{k} \boxed{- \parallel \underline{n} \parallel s}$$

and similarly for Booleans.

# Proving the Theorem

$(\Longrightarrow_{onlyif})$: Rule Induction for $\Downarrow$

$(Case \Downarrow \mathrm{OP}_1)$: The inductive hypotheses are

$$\boxed{P_1 \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{n_1} \parallel s} \quad \boxed{P_2 \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{n_2} \parallel s}$$

Then

$$\boxed{P_1\ op\ P_2 \parallel - \parallel s} \longmapsto \boxed{P_2 : P_1 : op \parallel - \parallel s}$$

$$\longmapsto^t \boxed{P_1 : op \parallel \underline{n_2} \parallel s} \equiv \boxed{P_1 : op \parallel \underline{n_2} \parallel s}$$

$$\longmapsto^t \boxed{op \parallel \underline{n_1} : \underline{n_2} \parallel s}$$

$$\longmapsto \boxed{- \parallel \underline{n_1\ op\ n_2} \parallel s}$$

$(\Longleftarrow_{if})$: We prove by induction for all $k$, for all $P :: \mathsf{int}, n, s,$

$$\underbrace{\boxed{P \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, \underline{n} \,\|\, s} \quad \textit{implies} \quad (P\,,\,s) \Downarrow (\underline{n}\,,\,s)}_{\phi(k)}$$

(*Proof of* $\forall k_0 \in \mathbb{N}$, $\phi(k)_{k \leq k_0}$ *implies* $\phi(k_0+1)$): Suppose that for some arbitrary $k_0$, $P :: \mathsf{int}$, $n$ and $s$

$$\boxed{P \,\|\, - \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, \underline{n} \,\|\, s} \qquad (*)$$

and then we prove $(P\,,\,s) \Downarrow (\underline{n}\,,\,s)$ by considering cases on $P$.

(*Case* $P$ is $P_1\ op\ P_2$): Suppose that

$$\begin{array}{|c||c||c|}\hline P_1\ op\ P_2 & - & s \\ \hline\end{array} \longmapsto^{k_0+1} \begin{array}{|c||c||c|}\hline - & \underline{n} & s \\ \hline\end{array}$$

and so

$$\begin{array}{|c||c||c|}\hline P_2 : P_1 : op & - & s \\ \hline\end{array} \longmapsto^{k_0} \begin{array}{|c||c||c|}\hline - & \underline{n} & s \\ \hline\end{array}.$$

Using splitting and termination we have, noting $P_2 :: \mathsf{int}$, that

$$\begin{array}{|c||c||c|}\hline P_2 & - & s \\ \hline\end{array} \longmapsto^{k_1} \begin{array}{|c||c||c|}\hline - & \underline{n_2} & s \\ \hline\end{array}$$

$$\begin{array}{|c||c||c|}\hline P_1 : op & \underline{n_2} & s \\ \hline\end{array} \longmapsto^{k_2} \begin{array}{|c||c||c|}\hline - & \underline{n} & s \\ \hline\end{array}$$

where $k_1 + k_2 = k_0$,

and repeating for the latter re-write we get

$$\begin{array}{|c||c||c|}\hline P_1 & \underline{n_2} & s \\ \hline\end{array} \longmapsto^{k_{21}} \begin{array}{|c||c||c|}\hline - & \underline{n_1} : \underline{n_2} & s \\ \hline\end{array}$$

$$\begin{array}{|c||c||c|}\hline op & \underline{n_1} : \underline{n_2} & s \\ \hline\end{array} \longmapsto^{k_{22}} \begin{array}{|c||c||c|}\hline - & \underline{n} & s \\ \hline\end{array} \qquad (1)$$

where $k_{21} + k_{22} = k_2$. So as $k_1 \leq k_0$, by induction we deduce that $(P_2\,,s) \Downarrow (\underline{n_2}\,,s)$, and from termination that

$$\begin{array}{|c||c||c|}\hline P_1 & - & s \\ \hline\end{array} \longmapsto^{k_{21}} \begin{array}{|c||c||c|}\hline - & \underline{n_1} & s \\ \hline\end{array}.$$

Also, as $k_{21} \leq k_0$, we have inductively that $(P_1\,,s) \Downarrow (\underline{n_1}\,,s)$ and hence

$$(P_1 \; op \; P_2\,,s) \Downarrow (\underline{n_1 \; op \; n_2}\,,s).$$

But from determinism and (1) we see that $\underline{n_1 \; op \; n_2} = \underline{n}$ and we are done.

# 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.

## What's Next? Expressions and Types for FUN

- Define the expression syntax and type system.

# Examples of FUN Declarations

```
g :: Int -> Int -> Int
g x y = x+y

l1 :: [Int]
l1 = 5:(6:(8:(4:(nil))))

h :: Int
h  = hd (5:6:8:4:nil)

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

# FUN Types

- The types of $\mathbb{FUN}^e$ are

$$\sigma \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid [\sigma]$$

- We shall write

$$\sigma_1 \to \sigma_2 \to \sigma_3 \to \ \dots \to \sigma_n \to \sigma$$

for

$$\sigma_1 \to (\sigma_2 \to (\sigma_3 \to (\dots \to (\sigma_n \to \sigma)\dots))).$$

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

| | | | |
|---|---|---|---|
| $E$ | $::=$ | $x$ | variables |
| | $\mid$ | $\underline{c}$ | constants |
| | $\mid$ | $K$ | constant identifier |
| | $\mid$ | $F$ | function identifier |
| | $\mid$ | $E_1\ E_2$ | function application |
| | $\mid$ | $\mathsf{tl}(E)$ | tail of list |
| | $\mid$ | $E_1 : E_2$ | cons for lists |
| | $\mid$ | $\mathsf{elist}(E)$ | 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

$$x :: \mathsf{int}, y :: \mathsf{bool}, z :: \mathsf{bool}$$

- 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

$$\mathsf{K} :: \mathsf{bool},\ \ \mathsf{map} :: (\mathsf{int} \to \mathsf{int}) \to [\mathsf{int}] \to [\mathsf{int}],\ \ \mathsf{suc} :: \mathsf{int} \to \mathsf{int}$$

- An identifier type looks like $\sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_a \to \sigma$ 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

$$x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash \mathsf{map}\,\mathsf{suc}\,(x : y : z : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

- We have

$$\varnothing \vdash \mathsf{if}\ \underline{T}\ \mathsf{then}\ \mathsf{hd}(\underline{2} : \mathsf{nil}_{\mathsf{int}})\ \mathsf{else}\ \mathsf{hd}(\underline{4} : \underline{6} : \mathsf{nil}_{\mathsf{int}}) :: \mathsf{int}$$

# Inductively Defining Type Assignments

Start with an identifier environment $I$ and a context $\Gamma$. Then

$$\frac{}{\Gamma \vdash x :: \sigma} \; (\text{ where } x :: \sigma \in \Gamma) \quad :: \text{VAR} \qquad \frac{}{\Gamma \vdash \underline{n} :: \mathsf{int}} \; :: \text{INT}$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1 \; iop \; E_2 :: \mathsf{int}} \; :: \text{OP}_1$$

$$\frac{\Gamma \vdash E_1 :: \sigma_2 \to \sigma_1 \quad \Gamma \vdash E_2 :: \sigma_2}{\Gamma \vdash E_1\, E_2 :: \sigma_1} :: \text{AP}$$

$$\frac{}{\Gamma \vdash I :: \iota} \;(\text{ where } I :: \iota \in \mathcal{I}) \quad :: \text{IDR}$$

$$\frac{}{\Gamma \vdash \mathsf{nil}_\sigma :: [\sigma]} :: \text{NIL} \qquad \frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2 :: [\sigma]}{\Gamma \vdash E_1 : E_2 :: [\sigma]} :: \text{CONS}$$

## 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 $\mathsf{plus}$ can write $\mathsf{plus}\,x\,y = x + y$.
- To declare $\mathsf{fac}$

$$\mathsf{fac}\,x = \mathsf{if}\ x == \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{fac}(x - \underline{1})$$

- And to declare that $\mathsf{true}$ denotes $\underline{T}$ we write $\mathsf{true} = \underline{T}$.
- In $\mathbb{FUN}^e$, can specify (recursive) declarations

$$K = E \qquad Fx = E' \qquad G\,x\,y = E'' \ldots$$

# An Example Declaration

Let $I = I_1 :: [\mathsf{int}] \rightarrow \mathsf{int} \rightarrow \mathsf{int}, I_2 :: \mathsf{int} \rightarrow \mathsf{int}, I_3 :: \mathsf{bool}$. Then an example of an identifier declaration $dec_I$ is

$$
\begin{array}{rcll}
I_1\ l\ y & = & \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(l))) + I_2\ y & \stackrel{\text{def}}{=} \quad E_{I_1} \\
I_2 x & = & x * x & \stackrel{\text{def}}{=} \quad E_{I_2} \\
I_3 & = & \underline{T} & \stackrel{\text{def}}{=} \quad E_{I_3} \\
I_4\ u\ v\ w & = & u + v + w & \stackrel{\text{def}}{=} \quad E_{I_4}
\end{array}
$$

# An Example Program

Let $I = F :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}, K :: \mathsf{int}$. Then an identifier declaration $dec_I$ is

$$\begin{array}{rcl} F\,x\,y & = & x + \underline{7} - y \quad \stackrel{\text{def}}{=} \quad E_F \\ K & = & \underline{10} \end{array}$$

An example of a program is $\boxed{dec_I \quad in \;\; F\,\underline{8}\,\underline{1} \leq K}$. Note that

$$\varnothing \vdash F\,\underline{8}\,\underline{1} \leq K :: \mathsf{bool}$$

and

$$\underbrace{x :: \mathsf{int}, y :: \mathsf{int}}_{\Gamma_F} \vdash x + \underline{7} - y :: \underbrace{\mathsf{int}}_{\sigma_F} \quad \text{and} \quad \varnothing \vdash K :: \mathsf{int}$$

# Defining Programs

A program in $\mathbb{FUN}^e$ is a judgement of the form

$$dec_I \quad in \;\; P$$

where $dec_I$ is a given identifier declaration and the program expression $P$ satisfies a type assignment of the form

$$\varnothing \vdash P :: \sigma \qquad (\text{written} \quad P :: \sigma)$$

and $\quad \forall \;\; F\vec{x} = E_F \;\; \in dec_I$

$$\Gamma_F \vdash E_F \; :: \; \sigma_F$$

## 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.

# Evaluation Orders

- The operational semantics of $\mathbb{FUN}^e$ 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

$$\underline{3} + \underline{4} + \underline{10} \Downarrow^e \underline{17} \qquad \mathsf{hd}(\underline{2} : \mathsf{nil}_{\mathsf{int}}) \Downarrow^e \underline{2}$$

- Let $F\,x\,y = x + y$. We would expect $F\,(\underline{2} * \underline{3})\,(\underline{4} * \underline{5}) \Downarrow^{e} \underline{26}$.
- We could
  - evaluate $\underline{2} * \underline{3}$ to get value $\underline{6}$ yielding $F\,\underline{6}\,(\underline{4} * \underline{5})$,
  - then evaluate $\underline{4} * \underline{5}$ to get value $\underline{20}$ yielding $F\,\underline{6}\,\underline{20}$.
- We then call the function to get $\underline{6} + \underline{20}$, which evaluates to $\underline{26}$. This is call-by-value or eager evaluation.
- Or the function could be called first yielding $(\underline{2} * \underline{3}) + (\underline{4} * \underline{5})$ and then we continue to get $\underline{6} + (\underline{4} * \underline{5})$ and $\underline{6} + \underline{20}$ and $\underline{26}$. This is called call-by-name or lazy evaluation.

# Defining and Explaining (Eager) Values

- Let $dec_I$ be an identifier declaration, with typical typing

$$F :: \sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_a \to \sigma$$

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

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid F\,\vec{V} \mid V : V$$

where $\vec{V}$ abbreviates $V_1\ V_2\ \ldots\ V_{k-1}\ V_k$ and $0 \leq k < a$.

- Note also that $k$ is strictly less than $a$, and that if $a = 1$ then $F\,\vec{V}$ denotes $F$.

- A value is any value expression for which $dec_I \quad in \; V$ is a valid $\mathbb{FUN}^e$ program.

- Suppose that $F :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and that $P_1 \Downarrow^e \underline{2}$ and $P_2 \Downarrow^e \underline{5}$ and $P_3 \Downarrow^e \underline{7}$ with $P_i$ not values. Then

| $P$ | $V$ |
|---|---|
| | $F$ |
| $F\,P_1$ | $F\,\underline{2}$ |
| $F\;\underline{2}\;P_2$ | $F\;\underline{2}\;\underline{5}$ |

| $P$ | $V$ |
|---|---|
| $F\;\underline{2}\;\underline{5}\;P_3$ | |
| $F\;\underline{2}\;\underline{5}\;\underline{7}$ | $\underline{14}$ |
| $F\;P_1\;P_2\;P_3$ | $\underline{14}$ |

# The Evaluation Relation

$$\frac{}{V \Downarrow^e V} \Downarrow^e{}_{\mathrm{VAL}} \qquad \frac{P_1 \Downarrow^e \underline{m} \quad P_2 \Downarrow^e \underline{n}}{P_1 \; op \; P_2 \Downarrow^e \underline{m \; op \; n}} \Downarrow^e{}_{\mathrm{OP}}$$

$$\frac{P_1 \Downarrow^e \underline{T} \quad P_2 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e{}_{\mathrm{COND1}}$$

$$\frac{\left\{\begin{array}{l} P_1 \Downarrow^e F\,\vec{V} \quad P_2 \Downarrow^e V_2 \quad F\,\vec{V}\,V_2 \Downarrow^e V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{array}\right.}{P_1\;P_2 \Downarrow^e V} \Downarrow^e\mathrm{AP}$$

$$\frac{E_F[V_1,\ldots,V_a/x_1,\ldots,x_a] \Downarrow^e V}{FV_1\ldots V_a \Downarrow^e V}\,[F\vec{x} = E_F \text{ declared in } dec_I]\ \Downarrow^e\mathrm{FID}$$

$$\frac{E_K \Downarrow^e V}{K \Downarrow^e V}\,[K = E_K \text{ declared in } dec_I]\ \Downarrow^e\mathrm{CID}$$

$$\frac{P \Downarrow^e V : V'}{\mathsf{hd}(P) \Downarrow^e V} \Downarrow^e\text{HD} \qquad \frac{P \Downarrow^e V : V'}{\mathsf{tl}(P) \Downarrow^e V'} \Downarrow^e\text{TL}$$

$$\frac{P_1 \Downarrow^e V \quad P_2 \Downarrow^e V'}{P_1 : P_2 \Downarrow^e V : V'} \Downarrow^e\text{CONS}$$

$$\frac{P \Downarrow^e \mathsf{nil}_\sigma}{\mathsf{elist}(P) \Downarrow^e \underline{T}} \Downarrow^e\text{ELIST}_1 \qquad \frac{P \Downarrow^e V : V'}{\mathsf{elist}(P) \Downarrow^e \underline{F}} \Downarrow^e\text{ELIST}_2$$

# Examples of Evaluations

Suppose that $dec_I$ is

$$\begin{aligned} G\,x &= x * \underline{2} \\ K &= \underline{3} \end{aligned}$$

$$\dfrac{\dfrac{}{G \Downarrow^e G}\ \mathrm{VAL} \qquad \dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\ \mathrm{VAL}}{K \Downarrow^e \underline{3}}\ \mathrm{CID} \qquad \dfrac{\dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\ \mathrm{VAL} \qquad \dfrac{}{\underline{2} \Downarrow^e \underline{2}}\ \mathrm{VAL}}{(x * \underline{2})[\underline{3}/x] = \underline{3} * \underline{2} \Downarrow^e \underline{6}}\ \mathrm{OP}}{G\,\underline{3} \Downarrow^e \underline{6}}\ \mathrm{FID}}{G\,K \Downarrow^e \underline{6}}\ \mathrm{AP}$$

We can prove that

$$F\ \underline{2}\ \underline{3}\ (\underline{4}+\underline{1}) \Downarrow^e \underline{10}$$

where $F\,x\,y\,z = x + y + z$ as follows:

$$\dfrac{\dfrac{}{F\ \underline{2}\ \underline{3} \Downarrow^e F\ \underline{2}\ \underline{3}}\ \Downarrow^e \text{VAL} \qquad \dfrac{\dfrac{}{\underline{4} \Downarrow^e \underline{4}} \qquad \dfrac{}{\underline{1} \Downarrow^e \underline{1}}}{\underline{4}+\underline{1} \Downarrow^e \underline{5}} \qquad T}{F\ \underline{2}\ \underline{3}\ (\underline{4}+\underline{1}) \Downarrow^e \underline{10}}\ \Downarrow^e \text{AP}$$

where $T$ is the tree

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{2} \Downarrow^e \underline{2}} \qquad \dfrac{}{\underline{3} \Downarrow^e \underline{3}}}{\underline{2} + \underline{3} \Downarrow^e \underline{5}} \qquad \dfrac{}{\underline{5} \Downarrow^e \underline{5}}}{\underline{2} + \underline{3} + \underline{5} \Downarrow^e \underline{10}}}{\dfrac{(x+y+z)[\underline{2}, \underline{3}, \underline{5}/x, y, z] \Downarrow^e \underline{10}}{F\ \underline{2}\ \underline{3}\ \underline{5} \Downarrow^e \underline{10}} \ \Downarrow^e \mathrm{FID}}
$$

## What's Next? FUN Properties of Eager Evaluation

- Explain and define determinism.
- Explain and define subject reduction, that is, preservation of types during program execution.

# Properties of FUN

- The evaluation relation for $\mathbb{FUN}^e$ is deterministic. More precisely, for all $P$, $V_1$ and $V_2$, if

$$P \Downarrow^e V_1 \qquad \textit{and} \qquad P \Downarrow^e V_2$$

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

- Evaluating a program $dec_I \;\; in \;\; P$ does not alter its type. More precisely,

$$(\varnothing \vdash P :: \sigma \textit{ and } P \Downarrow^e V) \quad \textit{implies} \quad \varnothing \vdash V :: \sigma$$

for any $P$, $V$, $\sigma$ and $dec_I$. The conservation of type during program evaluation is called subject reduction.

# Chapter 5

By the end of this chapter you should be able to

- describe the SECD machine, which executes compiled $\mathbb{FUN}^e$ programs; here the expressions $Exp$ are defined by $E ::= x \mid \underline{n} \mid F \mid E\ E$;
- 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

$$S ::= \begin{array}{c} \underline{n} \\ \uparrow \end{array} \quad | \quad \begin{array}{c} S_l \ldots S_1 \\ clo_F \\ \uparrow \end{array}$$

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

- A stack $S$ has a height the maximum level of any $clo_F$, or 0 otherwize.

- If the (unique) left-most closure node $clo_F$ at level $\alpha$ exists, call it the $\alpha$-prescribed node, and write $\alpha \quad S$.

- For any stack $\alpha \quad S$ of height $\geq 1$ there is a sub-stack $S'$ of shape

$$S_l \ \ldots \ S_1$$

$$\centerdot clo_F$$

$$\uparrow$$

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

$$S_{l+1} \; S_l \; \ldots \; S_1$$

$$\centerdot clo_F$$

$$\uparrow$$

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

- The environment $E$ takes the form

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

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

- If $S_i$ is $\underset{\uparrow}{\underline{n}}$ then ? is 0; if $S_i$ is $\underset{\uparrow}{clo_F}$ then ? is 1; in any other case, ? is ${}_{\mathsf{Av}}1$.

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

$$ins \quad ::= \quad x \mid \underline{n} \mid F \mid \mathsf{APP}$$

$$C \quad ::= \quad - \mid ins : C$$

- 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 $:$ to denote append; and write $\xi$ for $\xi : -$.

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

- $[\![x]\!] \stackrel{\text{def}}{=} x$
- $[\![\underline{n}]\!] \stackrel{\text{def}}{=} \underline{n}$
- $[\![F]\!] \stackrel{\text{def}}{=} F$
- $[\![E_1\ E_2]\!] \stackrel{\text{def}}{=} [\![E_1]\!] : [\![E_2]\!] : \mathsf{APP}$

There is a representation of program values as stacks, given by

- $(\!|\underline{n}|\!) \stackrel{\text{def}}{=} \begin{array}{c}\underline{n}\\ \uparrow\end{array}$
-

$$(\!|F\ V_1 \ldots V_k|\!) \stackrel{\text{def}}{=} \begin{array}{c}(\!|V_k|\!)\ \ \ldots\ \ (\!|V_1|\!)\\ clo_F\\ \uparrow\end{array} = (\!|V_k|\!) \oplus \ldots \oplus (\!|V_1|\!) \oplus \begin{array}{c}clo_F\\ \uparrow\end{array}$$

- 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):

| | |
|---|---|
| **S** | $[\,_{\mathsf{Av}}\,]\alpha \quad S$ |
| **E** | $E$ |
| **C** | $\underline{n} : C$ |
| **D** | $D$ |

$\overset{\text{num}}{\longmapsto}$

| | |
|---|---|
| **S** | $\alpha \quad \underset{\uparrow}{\underline{n}} \quad \oplus S$ |
| **E** | $E$ |
| **C** | $C$ |
| **D** | $D$ |

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

| | |
|---|---|
| **S** | $[\mathsf{Av}]\alpha \quad S$ |
| **E** | $E$ |
| **C** | $F : C$ |
| **D** | $D$ |

$\overset{\text{fn}}{\longmapsto}$

| | |
|---|---|
| **S** | $\alpha + 1 \quad \underset{\uparrow}{\blacksquare clo_F} \quad \oplus S$ |
| **E** | $E$ |
| **C** | $C$ |
| **D** | $D$ |

A variable's value is pushed onto the stack, provided that the environment $E$ contains $x = ?T \equiv [\mathsf{Av}]\delta \quad T$ (where $\delta$ is 0 or 1).

Note that by definition, the status of $T$ determines the status of the re-written stack:

$$
\begin{array}{|c||c|}
\hline
\mathbf{S} & [\mathsf{Av}]\alpha \quad S \\
\hline
\mathbf{E} & E \\
\hline
\mathbf{C} & x : C \\
\hline
\mathbf{D} & D \\
\hline
\end{array}
\quad \xmapsto{\text{var}} \quad
\begin{array}{|c||c|}
\hline
\mathbf{S} & [\mathsf{Av}]\delta + \alpha \quad T \oplus S \\
\hline
\mathbf{E} & E \\
\hline
\mathbf{C} & C \\
\hline
\mathbf{D} & D \\
\hline
\end{array}
$$

# An APP command creates an application value, type 0:

| | |
|---|---|
| **S** | $S_k \ldots S_1$ <br> $\alpha \quad \blacksquare clo_F \quad \oplus S$ <br> $\uparrow$ |
| **E** | $E$ |
| **C** | APP : $C$ |
| **D** | $D$ |

$$\overset{\text{cav0}}{\longmapsto}$$

| | |
|---|---|
| **S** | $S_k \ldots S_1$ <br> Av $\alpha \quad \blacksquare clo_F \quad \oplus S$ <br> $\uparrow$ |
| **E** | $E$ |
| **C** | $C$ |
| **D** | $D$ |

## An APP command creates an application value, type 1:

| | |
|---|---|
| **S** | $\alpha$ $\blacksquare clo_H$ ↑ $clo_F$ ↑ $S_{k-1} \ldots S_1$ $\oplus S$ |
| **E** | $E$ |
| **C** | APP : $C$ |
| **D** | $D$ |

$\overset{\text{cav1}}{\longmapsto}$

| | |
|---|---|
| **S** | $_{\mathsf{Av}}\,\alpha - 1$ $clo_H$ ↑ $\blacksquare clo_F$ ↑ $S_{k-1} \ldots S_1$ $\oplus S$ |
| **E** | $E$ |
| **C** | $C$ |
| **D** | $D$ |

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

| | |
|---|---|
| **S** | $S_k \ldots S_1$ <br> $\bullet clo_F \quad S'_{k'-1} \ldots S'_1$ <br> $\mathsf{Av}\ \alpha \quad \uparrow \quad \oplus S$ <br> $clo_G$ <br> $\uparrow$ |
| **E** | $E$ |
| **C** | $\mathsf{APP} : C$ |
| **D** | $D$ |

$\overset{\text{avtav}}{\longmapsto}$

| | |
|---|---|
| **S** | $S_k \ldots S_1$ <br> $clo_F \quad S'_{k'-1} \ldots S'_1$ <br> $\mathsf{Av}\ \alpha - 1 \quad \uparrow \quad \oplus S$ <br> $\bullet clo_G$ <br> $\uparrow$ |
| **E** | $E$ |
| **C** | $C$ |
| **D** | $D$ |

# An APP command calls a function, type 0:

| | |
|---|---|
| **S** | $\alpha \quad \underset{\uparrow}{\overset{S_a \ldots S_1}{\centerdot clo_F}} \quad \oplus S$ |
| **E** | $E$ |
| **C** | $\mathsf{APP} : C$ |
| **D** | $D$ |

$\overset{\text{call0}}{\longmapsto}$

| | |
|---|---|
| **S** | $-$ |
| **E** | $x_a = ?S_a : \ldots : x_1 = ?S_1 : E$ |
| **C** | $[\![E_F]\!]$ |
| **D** | $(\alpha - 1 \quad S, E, C, D)$ |

# An APP command calls a function, type 1:

| | |
|---|---|
| **S** | $\alpha \quad \blacksquare clo_H \uparrow \quad S_{a-1}\ldots S_1 \quad clo_F \uparrow \quad \oplus S$ |
| **E** | $E$ |
| **C** | APP : $C$ |
| **D** | $D$ |

$\overset{\text{call1}}{\longmapsto}$

| | |
|---|---|
| **S** | $-$ |
| **E** | $x_a = ?S_a : \ldots : x_1 = ?S_1 : E$ |
| **C** | $[\![E_F]\!]$ |
| **D** | $(\alpha - 2 \quad S, E, C, D)$ |

# An APP command calls a function, type 2:

| | |
|---|---|
| **S** | $\mathrm{Av}\,\alpha$ $S_k \ldots S_1$ $\mathbf{.}clo_F$ $S'_{a-1} \ldots S'_1$ $\uparrow$ $clo_G$ $\uparrow$ $\oplus S$ |
| **E** | $E$ |
| **C** | $\mathsf{APP} : C$ |
| **D** | $D$ |

$\overset{\text{call2}}{\longmapsto}$

| | |
|---|---|
| **S** | $-$ |
| **E** | $x_a = ?S'_a : \ldots : x_1 = ?S'_1 : E$ |
| **C** | $[\![E_G]\!]$ |
| **D** | $(\alpha - 2 \quad S, E, C, D)$ |

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

| | |
|---|---|
| **S** | $[\mathsf{Av}]\beta \quad T$ |
| **E** | $E'$ |
| **C** | $-$ |
| **D** | $(\alpha \quad S, E, C, D)$ |

$\overset{\text{res}}{\longmapsto}$

| | |
|---|---|
| **S** | $[\mathsf{Av}]\alpha + \beta \quad T \oplus S$ |
| **E** | $E$ |
| **C** | $C$ |
| **D** | $D$ |

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

$$F\,x\,y = x \qquad I\,a\,b = b$$
$$L\,u\,v = u \qquad H\,z = L\,(M\,N)\,z$$

Then $(F\,(H\,\underline{4}))\,(I\,\underline{2}\,K) \Downarrow^e M\,N$. Note that

$$[\![ (F\,(H\,\underline{4}))\,(I\,\underline{2}\,K) ]\!] =$$

$$(11. \stackrel{\text{def}}{=} F) : H : \underline{4} : \mathsf{APP} : \mathsf{APP} : I : \underline{2} : \mathsf{APP} : K : \mathsf{APP} : (\mathsf{APP} \stackrel{\text{def}}{=} 1.)$$

and

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

| | |
|---|---|
| **S** | 0 — |
| **E** | — |
| **C** | 11. |
| **D** | — |

$\stackrel{\text{num/fn}^3}{\longmapsto}$

| | |
|---|---|
| **S** | 2 $\underline{4}$ $\uparrow$ $\blacksquare clo_H$ $\uparrow$ $clo_F$ $\uparrow$ |
| **E** | — |
| **C** | 8. $\equiv$ APP : 7. |
| **D** | — |

$\overset{\text{call0}}{\longmapsto}$

| | |
|---|---|
| **S** | $0 \quad -$ |
| **E** | $E' \stackrel{\text{def}}{=} z = 0 \quad \underline{4}\ \uparrow$ |
| **C** | $[\![L\ (M\ N)\ z]\!]$ |
| **D** | $\xi \stackrel{\text{def}}{=} (1\ \ clo_F \uparrow\ , -, 7., -)$ |

$\overset{\text{fn}}{\longmapsto} 3$

| | |
|---|---|
| **S** | $3 \quad clo_L \uparrow clo_M \uparrow clo_N\ \blacksquare$ |
| **E** | $E'$ |
| **C** | $4.$ |
| **D** | $\xi$ |

$\stackrel{\text{cav1}}{\longmapsto}$

| | |
|---|---|
| **S** | $\mathsf{Av}\ 2$ $\quad clo_N \uparrow {\scriptstyle\blacksquare} clo_M \uparrow clo_L \uparrow$ |
| **E** | $E'$ |
| **C** | 3. |
| **D** | $\xi$ |

$\stackrel{\text{avtav}}{\longmapsto}$

| | |
|---|---|
| **S** | $\mathsf{Av}\ 1$ $\quad clo_N \uparrow clo_M \uparrow {\scriptstyle\blacksquare} clo_L \uparrow$ |
| **E** | $E'$ |
| **C** | 2. |
| **D** | $\xi$ |

# 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 $dec_I \quad in \;\; P$ for which $\varnothing \vdash P :: \sigma$ we have

$$P \Downarrow^e V \qquad iff \qquad \begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^t \begin{array}{|c||c|}\hline \mathbf{S} & (\!|V|\!) \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

# Code and Stack Extension

For any stacks, environments, codes, and dumps, if $C_1$ is non-empty

$$M \stackrel{\text{def}}{=} \begin{array}{|c||c|}\hline \mathbf{S} & S_1 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & S_2 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \stackrel{\text{def}}{=} M'$$

*implies*

$$\overline{M} \stackrel{\text{def}}{=} \begin{array}{|c||c|}\hline \mathbf{S} & S_1 \oplus S_3 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 : C_3 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & S_2 \oplus S_3 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_2 : C_3 \\ \hline \mathbf{D} & D \\ \hline \end{array} \stackrel{\text{def}}{=} \overline{M'}$$

- Need to prove "lemma plus": if $D \equiv (S', E', C', D')$ we can also similarly arbitrarily extend any of the stacks and codes in $D$ (say to $\overline{D}$).

- We use induction on $k$. Suppose lemma plus is true $\forall k \leq k_0$. Must prove we can extend any re-write $M \longmapsto^{k_0+1} M'$ to $\overline{M} \longmapsto^{k_0+1} \overline{M'}$. By determinism, we have $M \longmapsto^{1} M'' \longmapsto^{k_0} M'$.

- If no function call during $M \longmapsto^{1} M''$, trivial to extend to get $\overline{M} \longmapsto^{1} \overline{M''}$. And by induction, $\overline{M''} \longmapsto^{k_0} \overline{M'}$.

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

$$M \stackrel{\text{def}}{=} \begin{array}{|c||c|}\hline \mathbf{S} & T \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & \mathsf{APP}:C \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{1} \begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & [\![E_F]\!] \\ \hline \mathbf{D} & (S,E,C,D) \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & C'' \\ \hline \mathbf{D} & (S,E,C,D) \\ \hline \end{array}$$

$$\overset{\text{res}}{\longmapsto}{}^{1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_2} M'$$

where there are no function calls in the $k_2$ re-writes.

By induction, we have

$\overline{M''} \stackrel{\text{def}}{=}$

| | |
|---|---|
| **S** | $-$ |
| **E** | $E'$ |
| **C** | $[\![E_F]\!]$ |
| **D** | $(S \oplus S_3, E, C : C_3, \overline{D})$ |

$\longmapsto^{k_1}$

| | |
|---|---|
| **S** | $S''$ |
| **E** | $E''$ |
| **C** | $C''$ |
| **D** | $(S \oplus S_3, E, C : C_3, \overline{D})$ |

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

| | |
|---|---|
| **S** | $S''$ |
| **E** | $E''$ |
| **C** | $C''$ |
| **D** | $(S \oplus S_3, E, C : C_3, \overline{D})$ |

$\longmapsto^{1}$

| | |
|---|---|
| **S** | $S'' \oplus S \oplus S_3$ |
| **E** | $E$ |
| **C** | $C : C_3$ |
| **D** | $\overline{D}$ |

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

| | |
|---|---|
| **S** | $S'' \oplus S \oplus S_3$ |
| **E** | $E$ |
| **C** | $C : C_3$ |
| **D** | $\overline{D}$ |

$$\longmapsto^1 \overline{M'}.$$

and we are also done.

# Code Splitting

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

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 : C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

implies that

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array} \qquad \textit{and} \qquad \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_2} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

where $k = k_1 + k_2$.

# Program Code Factors Through Value Code

For any well typed $\mathbb{FUN}^e$ program $dec_I \quad in \;\; P$ where $P :: \sigma$ and $P \Downarrow^e V$,

| | |
|---|---|
| **S** | $S$ |
| **E** | $E$ |
| **C** | $[\![P]\!]$ |
| **D** | $D$ |

$\longmapsto^{k}$

| | |
|---|---|
| **S** | $\hat{S}$ |
| **E** | $\hat{E}$ |
| **C** | $-$ |
| **D** | $\hat{D}$ |

*implies* $(\exists \overline{k} \leq k)$

| | |
|---|---|
| **S** | $S$ |
| **E** | $E$ |
| **C** | $[\![V]\!]$ |
| **D** | $D$ |

$\longmapsto^{\overline{k}}$

| | |
|---|---|
| **S** | $\hat{S}$ |
| **E** | $\hat{E}$ |
| **C** | $-$ |
| **D** | $\hat{D}$ |

with equality only if $P$ is a value (and hence equal to $V$).

# Proving the Theorem

($\Longleftarrow_{if}$): We shall prove that if $P :: \sigma$ then

| | | | | | | |
|---|---|---|---|---|---|---|
| **S** | $S$ | | **S** | $S'$ | | |
| **E** | $-$ | | **E** | $-$ | | |
| **C** | $[\![P]\!]$ | $\longmapsto^k$ | **C** | $-$ | *implies* | $(\exists V) \quad S' = (\!|V|\!) \oplus S \;\; and \;\; P \Downarrow^e V$ |
| **D** | $-$ | | **D** | $-$ | | |

from which the required result follows. Induction on $k$. If $P$ is a number or a function the result is trivial. Else $P$ has the form $P_1 P_2$.

Suppose that

| | | | | | |
|---|---|---|---|---|---|
| **S** | $S$ | | | **S** | $S'$ |
| **E** | $-$ | | | **E** | $-$ |
| **C** | $[\![P_1]\!] : [\![P_2]\!] :$ APP | $\longmapsto^{k_0+1}$ | | **C** | $-$ |
| **D** | $-$ | | | **D** | $-$ |

Then appealing to splitting and the induction hypothesis, we get

| | |
|---|---|
| **S** | $S$ |
| **E** | $-$ |
| **C** | $[\![P_1]\!]$ |
| **D** | $-$ |

$$\longmapsto^{k_1}$$

| | |
|---|---|
| **S** | $(\!\lvert F\ \vec{V}\rvert\!) \oplus S$ |
| **E** | $-$ |
| **C** | $-$ |
| **D** | $-$ |

and

| | |
|---|---|
| **S** | $(\!\lvert F\ \vec{V}\rvert\!) \oplus S$ |
| **E** | $-$ |
| **C** | $[\![P_2]\!] : \mathsf{APP}$ |
| **D** | $-$ |

$$\longmapsto^{k_2}$$

| | |
|---|---|
| **S** | $S'$ |
| **E** | $-$ |
| **C** | $-$ |
| **D** | $-$ |

where $P_1 \Downarrow^e F\ \vec{V}$.

Appealing to splitting again, and by induction,

| | |
|---|---|
| **S** | $(\lvert F\ \vec{V} \rvert) \oplus S$ |
| **E** | $-$ |
| **C** | $[\![P_2]\!]$ |
| **D** | $-$ |

$\longmapsto^{k_{21}}$

| | |
|---|---|
| **S** | $(\lvert V_2 \rvert) \oplus (\lvert F\ \vec{V} \rvert) \oplus S$ |
| **E** | $-$ |
| **C** | $-$ |
| **D** | $-$ |

and

| | |
|---|---|
| **S** | $(\lvert V_2 \rvert) \oplus (\lvert F\ \vec{V} \rvert) \oplus S$ |
| **E** | $-$ |
| **C** | APP |
| **D** | $-$ |

$\longmapsto^{k_{22}}$

| | |
|---|---|
| **S** | $S'$ |
| **E** | $-$ |
| **C** | $-$ |
| **D** | $-$ |

where $P_2 \Downarrow^e V_2$.

By factorization on $P_1$ and $P_2$, and extension we have (check!)

| | |
|---|---|
| **S** | $S$ |
| **E** | $-$ |
| **C** | $[\![F\ \vec{V}\ V_2]\!]$ |
| **D** | $-$ |

$$\longmapsto^{\overline{k_1}+\overline{k_{21}}}$$

| | |
|---|---|
| **S** | $(\|V_2\|) \oplus (\|F\ \vec{V}\|) \oplus S$ |
| **E** | $-$ |
| **C** | APP |
| **D** | $-$ |

$$\longmapsto^{k_{22}}$$

| | |
|---|---|
| **S** | $S'$ |
| **E** | $-$ |
| **C** | $-$ |
| **D** | $-$ |

and so if $P_1\ P_2$ is not a value then

$$\overline{k_1} + \overline{k_{21}} + k_{22} < k_0 + 1$$

and by induction $S' = (|V|) \oplus S$ for some $V$ where $F\ \vec{V}\ V_2 \Downarrow^e V$.
Hence $P_1\ P_2 \Downarrow^e V$ as required.

If $P_1\ P_2$ is a value, refer to part $(\Longrightarrow_{onlyif})$ of the proof, case $\Downarrow^e$ VAL