

What is a Computer?

State of the machine.

- Internal registers, memory, *etc.*
- Initial and final state.

1

What is a Computer?

Instructions for transforming the state.

- Must be **effective**, usually **simple**.
- Rules for determining order of execution.

Execution = instruction by instruction state transformation.

2

What is a Computer?

A computer is a **transition system**.

- Set of states S .
- Initial states $I \subseteq S$.
- Final states $F \subseteq S$.
- Relation $\mapsto \subseteq S \times S$.

Execution = transition sequence.

3

The \mathcal{L} Machines

Can view $\mathcal{L}\{\text{num str}\}$, $\mathcal{L}\{\text{nat} \rightarrow\}$, *etc.* as implicitly defining abstract machines, which we might call the \mathcal{L} Machines.

States: closed (simple arithmetic, PCF, *etc.*) expressions.

- Initial: any well-typed, closed expression.
- Final: closed values.

Transitions: \mapsto is given by structural semantics rules.

Evaluation: $e \mapsto^* v$.

4

The \mathcal{L} Machines

The \mathcal{L} Machines are very high-level, in two senses:

- **Control**. Complex search rules specify order of execution. Rely on a “metastack” to manage search.
- **Data**. Parameter passing is by substitution, which is complex and generates “new” code on the fly.

5

Managing Data

We've already considered alternatives to the \mathcal{L} Machines. At least one was aimed at supporting more realistic treatment of data:

- Environment semantics: combine hypothetical judgements and evaluation semantics for a more realistic treatment of variable binding

6

Managing Control

The \mathcal{L} Machines "cheat" by relying on implicit storage management.

- Search rules have one or more premises that must be applied recursively.
- Interpreter is not tail recursive (iterative).

"Real" machines cannot rely on implicit storage management!

7

Managing Control

For example,

$$\frac{e_1 \mapsto e'_1}{\text{ap}(e_1; e_2) \mapsto \text{ap}(e'_1; e_2)}$$

To apply this rule, we must (as the contextual semantics made somewhat more explicit):

1. Save the current state: $\text{ap}(o; e_2)$.
2. Execute a step in e_1 , obtaining e'_1 .
3. Restore the state: $\text{ap}(e'_1; e_2)$.

8

Managing Control

The $\mathcal{K}\{\text{nat} \rightarrow\}$ abstract machine for the language $\mathcal{L}\{\text{nat} \rightarrow\}$ makes the flow of control **explicit** in the **state** of the machine.

- Explicit **control stack**, or **continuation**, that manages the flow of control.
- Transition rules will have **no premises** — fully iterative (tail recursive) implementation.

9

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

The **state** of the \mathcal{K} Machine is a pair (k, e) , where

- k is a control stack, or continuation;
- e is a closed expression.

The $\mathcal{K}\{\text{nat} \rightarrow\}$ **transition relation** \mapsto is defined inductively by a set of rules.

10

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

The **state** of $\mathcal{K}\{\text{nat} \rightarrow\}$ takes one of two forms:

- an **evaluation** state $k \triangleright e$ corresponds to evaluating closed expression e relative to control stack k
- a **return** state $k \triangleleft e$, where e val, corresponds to evaluating stack k relative to closed value e

The separator **points to** the focal entity of the state.

11

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine: Control Stacks

A **control stack**, or **continuation**, represents the context of evaluation, into which the value of the current expression is to be returned. Formally, the control stack is a list of **stack frames**:

$$\frac{}{\varepsilon \text{ stack}} \quad \frac{f \text{ frame} \quad k \text{ stack}}{k; f \text{ stack}}$$

Think of pushing a frame onto the control stack.

12

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

States: as just described

- Initial: $\varepsilon \triangleright e$
- Final: $\varepsilon \triangleleft e$ with $e \text{ val}$

Transitions: \mapsto is given by structural semantics rules.

Evaluation: $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e$ with $e \text{ val}$.

13

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine: Stack Frames

A **stack frame** records one pending computation. For $\mathcal{K}\{\text{nat} \rightarrow\}$ the appropriate stack frames are inductively defined as follows:

$$\frac{}{\overline{s(-) \text{ frame}}} \quad \frac{}{\overline{\text{ifz}(-; e_1; x.e_2) \text{ frame}}} \quad \frac{}{\overline{\text{ap}(-; e_2) \text{ frame}}}$$

Think of a frame as an abstract return address; the “-” marks the return point.

Frames correspond to rules with transition premises (search rules) of $\mathcal{L}\{\text{nat} \rightarrow\}$: an explicit record of pending computations replaces reliance on structure of transition derivation to record context.

14

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine: Natural Numbers

To evaluate z we simply return it:

$$\overline{k \triangleright z \mapsto k \triangleleft z}$$

To evaluate $s(e)$ we first evaluate the argument:

$$\overline{k \triangleright s(e) \mapsto k; s(-) \triangleright e}$$

... and when that completes we return its successor to the stack:

$$\overline{k; s(-) \triangleleft e \mapsto k \triangleleft s(e)}$$

15

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine: Case Analysis

First we evaluate the test:

$$\overline{k \triangleright \text{ifz}(e; e_1; x.e_2) \mapsto k; \text{ifz}(-; e_1; x.e_2) \triangleright e}$$

... and then decide how to proceed:

$$\overline{k; \text{ifz}(-; e_1; x.e_2) \triangleleft z \mapsto k \triangleright e_1}$$

$$\overline{k; \text{ifz}(-; e_1; x.e_2) \triangleleft s(e) \mapsto k \triangleright [e/x]e_2}$$

16

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine: Functions

To evaluate $\text{lam}[\tau](x.e)$ we simply return it:

$$\overline{k \triangleright \text{lam}[\tau](x.e) \mapsto k \triangleleft \text{lam}[\tau](x.e)}$$

To evaluate $\text{ap}(e_1; e_2)$ we first evaluate the function:

$$\overline{k \triangleright \text{ap}(e_1; e_2) \mapsto k; \text{ap}(-; e_2) \triangleright e_1}$$

... and then perform the application:

$$\overline{k; \text{ap}(-; e_2) \triangleleft \text{lam}[\tau](x.e) \mapsto k \triangleright [e_2/x]e}$$

17

The $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine: General Recursion

$$\overline{k \triangleright \text{fix}[\tau](x.e) \mapsto k \triangleright [\text{fix}[\tau](x.e)/x]e}$$

Note that evaluation of general recursion requires no stack space!

18

Safety for the $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

Based on a new typing judgement, $k : \tau$, meaning stack k is well-formed and expects a value of type τ .

A stack is well-formed if, when passed a value of the appropriate type, it safely transforms that value into another value.

A stack represents the work remaining to complete a computation, so $k : \tau$ means stack k transforms a value of type τ into a value of type τ'

This judgement is inductively defined by:

$$\frac{\overline{\varepsilon : \tau}}{k : \tau' \quad f : \tau \Rightarrow \tau' \quad k; f : \tau}$$

19

Frame Transformation Judgement

The auxiliary judgement $f : \tau \Rightarrow \tau'$, stating that f transforms a value of type τ to a value of type τ' is inductively defined by:

$$\frac{}{\overline{s(-) : \text{nat} \Rightarrow \text{nat}}}$$

$$\frac{e_1 : \tau \quad x : \text{nat} \vdash e_2 : \tau}{\text{ifz}(-; e_1; x.e_2) : \text{nat} \Rightarrow \tau}$$

$$\frac{e_2 : \tau_2}{\text{ap}(-; e_2) : \text{parr}(\tau_2; \tau) \Rightarrow \tau}$$

20

Well-Formed States for the $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

The two forms of state for the $\mathcal{K}\{\text{nat} \rightarrow\}$ Machine are well-formed provided that their stack and expression components match.

This judgement is inductively defined by:

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \text{ ok}}$$

$$\frac{k : \tau \quad e : \tau \quad e \text{ val}}{k \triangleleft e \text{ ok}}$$

21

Type Safety

Theorem 1 (Preservation)

If s ok and $s \mapsto s'$, then s' ok.

Theorem 2 (Progress)

If s ok then either

1. s final, or
2. there exists s' such that $s \mapsto s'$.

22

Correctness of the $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

Does the $\mathcal{K}\{\text{nat} \rightarrow\}$ abstract machine correctly implement the PCF \mathcal{L} Machine?

Given a PCF expression (program) e , do we get the same result from evaluating e with the $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine and with the PCF \mathcal{L} Machine and vice versa?

We want the following relationship between the $\mathcal{K}\{\text{nat} \rightarrow\}$ and the PCF \mathcal{L} Machines:

1. **[Completeness]** If $e \mapsto^* e'$, where e' val, then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$.
2. **[Soundness]** If $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$, then $e \mapsto^* e'$ with e' val.

23

Proving Completeness

Proceed by induction on the definition of $e \mapsto^* e'$.

This reduces to two cases:

1. If e val, then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e$.
2. If $e \mapsto e'$, then for every v val, if $\varepsilon \triangleright e' \mapsto^* \varepsilon \triangleleft v$, then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft v$.

First of these is easily proven by induction on structure of e .

24

Proving Completeness

Proving the second requires inductive analysis of the derivation of $e \mapsto e'$.

But there is a problem:

- Can't just consider empty stacks, since for example if e is $\text{ap}(e_1; e_2)$ the first step of the $\mathcal{K}\{\text{nat} \rightarrow\}$ abstract machine is $\varepsilon \triangleright \text{ap}(e_1; e_2) \mapsto \varepsilon; \text{ap}(-; e_2) \triangleright e_1$.

So more generally we need to prove:

If $e \mapsto e'$, then for every v val, if $k \triangleright e' \mapsto^* k \triangleleft v$ then $k \triangleright e \mapsto^* k \triangleleft v$.

25

Proving Completeness

Returning to our example where e is $\text{ap}(e_1; e_2)$ and e' is $\text{ap}(e'_1; e_2)$ with $e_1 \mapsto e'_1$:

- We are given that $k \triangleright \text{ap}(e'_1; e_2) \mapsto^* k \triangleleft v$ and need to show that $k \triangleright \text{ap}(e_1; e_2) \mapsto^* k \triangleleft v$.
- It is easy to show that the first step of the former transition is $k \triangleright \text{ap}(e'_1; e_2) \mapsto k; \text{ap}(-; e_2) \triangleright e'_1$.
- But this leaves us needing to apply induction to the derivation of $e_1 \mapsto e'_1$, which in turn requires having a v_1 such that $e'_1 \mapsto^* v_1$, which structural semantics doesn't give us.

26

Proving Completeness

Idea: if $e \mapsto e'$, then any complete execution from e' in any context determines a corresponding complete execution from e in that context.

Evaluation semantics gives us the connection to the ultimate value v of each sub-expression that we require.

Theorem 3

If $e \Downarrow v$, then for every k stack, $k \triangleright e \mapsto^* k \triangleleft v$.

27

Proving Completeness

Proof is by induction on the evaluation semantics for the PCF \mathcal{L} Machine.

Consider, for example, the rule:

$$\frac{e_1 \Downarrow \text{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\text{ap}(e_1; e_2) \Downarrow v}$$

For arbitrary control stack k we are to show that $k \triangleright \text{ap}(e_1; e_2) \mapsto^* k \triangleleft v$.

28

Proving Completeness

Using each of the three inductive hypotheses in succession, interleaved with steps of the abstract machine, we calculate:

$$\begin{aligned} k \triangleright \text{ap}(e_1; e_2) &\mapsto k; \text{ap}(-; e_2) \triangleright e_1 \\ &\mapsto^* k; \text{ap}(-; e_2) \triangleleft \text{lam}[\tau_2](x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v \end{aligned}$$

The other cases are handled similarly.

29

Proving Soundness

We would like to proceed by induction on the multistep transition $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft v$.

But this is awkward; intervening steps may alternate between evaluation and return states.

Instead, we will view each $\mathcal{K}\{\mathbf{nat} \rightarrow\}$ abstract machine state as encoding a PCF expression and show that $\mathcal{K}\{\mathbf{nat} \rightarrow\}$ abstract machine transitions are simulated by PCF \mathcal{L} Machine transitions under this encoding.

30

Proving Soundness

We call the encoding **unravel** and write $s \triangleright e$, meaning state s unravels to expression e .

For initial states $s = \varepsilon \triangleright e$ and final states $s = \varepsilon \triangleleft e$, $s \triangleright e$.

Then we show that if $s \mapsto^* s'$, where s' final, $s \triangleright e$, and $s' \triangleright e'$, then e' val and $e \mapsto^* e'$.

31

Proving Soundness

To do this, we show the following two facts:

1. If $s \triangleright e$ and s final, then e val.
2. If $s \mapsto s'$, $s \triangleright e$, $s' \triangleright e'$ and $e' \mapsto^* v$ where v val, then $e \mapsto^* v$.

First of these is obvious, since unravelling of a final state is a value. Second requires proving:

If $s \mapsto s'$, $s \triangleright e$, and $s' \triangleright e'$, then $e \mapsto^* e'$.

32

Proving Soundness

The judgement $s \triangleright e'$, where s is either $k \triangleright e$ or $k \triangleleft e$ is defined in terms of the auxiliary judgement $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \triangleright e'}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \triangleright e'}$$

That is, to unravel a state we **wrap** the stack around the expression.

33

Proving Soundness

The **wrap** relation is inductively defined as follows:

$$\overline{\varepsilon \bowtie e = e}$$

$$\frac{k \bowtie s(e) = e'}{k; s(-) \bowtie e = e'}$$

$$\frac{k \bowtie \mathbf{ifz}(e_1; e_2; x.e_3) = e'}{k; \mathbf{ifz}(-; e_2; x.e_3) \bowtie e_1 = e'}$$

$$\frac{k \bowtie \mathbf{ap}(e_1; e_2) = e}{k; \mathbf{ap}(-; e_2) \bowtie e_1 = e}$$

34

Proving Soundness

Lemma 4

Judgement $s \triangleright e$ has mode $(\forall, \exists!)$ and judgement $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$

That is, each state unravels to a unique expression and the result of wrapping a stack around an expression is uniquely determined.

We are therefore justified in writing $k \bowtie e$ for the unique e' such that $k \bowtie e = e'$.

35

Proving Soundness

The next lemma states that unravelling preserves the transition relation.

Lemma 5

If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.

Proof is by induction on the transition $e \mapsto e'$. When the transition is a search rule (i.e., has a premise), the proof is an easy induction. When the transition is an axiom, the proof is by an inductive analysis of the stack k .

36

Proving Soundness

Search rule case:

For example, suppose $e = \mathbf{ap}(e_1; e_2)$, $e' = \mathbf{ap}(e'_1; e_2)$ and $e_1 \mapsto e'_1$.

We have $k \bowtie e = d$ and $k \bowtie e' = d'$.

By the inductive definition of **wrap**, $k; \mathbf{ap}(-; e_2) \bowtie e_1 = d$ and $k; \mathbf{ap}(-; e_2) \bowtie e'_1 = d'$.

So by induction, $d \mapsto d'$ as desired.

37

Proving Soundness

Axiom case:

For example, suppose $e = \mathbf{ap}(\mathbf{lam}[\tau_2](x.e); e_2)$, $e' = [e_2/x]e$ with $e \mapsto e'$ directly.

Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$.

We proceed by inner induction on the structure of k .

If $k = \varepsilon$, the result is immediate.

38

Proving Soundness

Consider, for instance, the stack $k = k'; \mathbf{ap}(-; c_2)$.

By the inductive definition rules of **wrap**, $k' \bowtie \mathbf{ap}(e; c_2) = d$ and $k' \bowtie \mathbf{ap}(e'; c_2) = d'$.

But by the dynamic semantics rules $\mathbf{ap}(e; c_2) \mapsto \mathbf{ap}(e'; c_2)$.

So by the inner inductive hypothesis we have $d \mapsto d'$ as desired.

39

Proving Soundness

We are finally in position to prove:

Theorem 6

If $s \mapsto s'$, $s \rightsquigarrow e$, and $s' \rightsquigarrow e'$, then $e \mapsto^* e'$.

Proof is by case analysis on the transition of $\mathcal{K}\{\mathbf{nat} \rightarrow\}$. In each case, after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\mathbf{nat} \rightarrow\}$.

Suppose that $s = k \triangleright s(e)$ and $s' = k; s(-) \triangleright e$. Note that $k \bowtie s(e) = e'$ iff $k; s(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k; \mathbf{ap}(\mathbf{lam}[\tau](x.e_1); -) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k; \mathbf{ap}(\mathbf{lam}[\tau](x.e_1); -) \bowtie e_2 = e'$ and e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathbf{ap}(\mathbf{lam}[\tau](x.e_1); e_2) = e'$. The result follows from Lemma 5.

40

Summary

Abstract machines come in all shapes and sizes. They differ in how many details of execution are made explicit.

The $\mathcal{K}\{\mathbf{nat} \rightarrow\}$ abstract machine manages control explicitly.

41