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.

The \( \mathcal{L} \) Machines

Can view \( \mathcal{L}[\text{sum asts}] \), \( \mathcal{L}[\text{nat \to}] \), 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: \( \to \) is given by structural semantics rules.

Evaluation: \( e \to^* e' \).

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 \( \to \subseteq S \times S \).

Execution = transition sequence.

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.
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

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!

Managing Control
For example,
\[
\begin{align*}
\epsilon_1 & \Rightarrow \epsilon'_1 \\
\text{ap}(\epsilon_1; \epsilon_2) & \Rightarrow \text{ap}(\epsilon'_1; \epsilon_2)
\end{align*}
\]

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

1. Save the current state: \( \text{ap}(\epsilon_1; \epsilon_2) \).
2. Execute a step in \( \epsilon_1 \), obtaining \( \epsilon'_1 \).
3. Restore the state: \( \text{ap}(\epsilon'_1; \epsilon_2) \).

Managing Control
The \( \mathcal{K}(\text{sat} \rightarrow) \) abstract machine for the language \( \mathcal{L}(\text{sat} \rightarrow) \) 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.

The \( \mathcal{K}(\text{sat} \rightarrow) \) 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 \( \mathcal{K}(\text{sat} \rightarrow) \) transition relation \( \Rightarrow \) is defined inductively by a set of rules.
The $\mathcal{K}$-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:

\[
\begin{array}{c|c|c}
\text{f stack} & \text{k stack} & \text{k, f stack} \\
\end{array}
\]

Think of pushing a frame onto the control stack.

The $\mathcal{K}$-Abstract Machine: Stack Frames

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

\[
\begin{array}{c}
\text{z(−) frame} \\
\text{ifz(−, e1, e2) frame} \\
\text{ap(−, e2) frame} \\
\end{array}
\]

Think of a frame as an abstract return address; the "−" marks the return point.

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

The $\mathcal{K}$-Abstract Machine: Case Analysis

First we evaluate the test:

\[
k \triangleright \text{ifz}(e, e1, e2) \Rightarrow k \triangleright \text{ifz}(e, e1, e2) \triangleright e
\]

... and then decide how to proceed:

\[
k \triangleright \text{ifz}(e, e1, e2) \triangleright e \Rightarrow k \triangleright e
\]

\[
k \triangleright \text{ifz}(e, e1, e2) \triangleright e \Rightarrow k \triangleright e
\]

The $\mathcal{K}$-Abstract Machine: Functions

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

\[
k \triangleright \text{lam}[\tau](x, e) \Rightarrow k \triangleright \text{lam}[\tau](x, e)
\]

To evaluate $\text{ap}(e1, e2)$ we first evaluate the function:

\[
k \triangleright \text{ap}(e1, e2) \Rightarrow k \triangleright \text{ap}(e1, e2) \triangleright e1
\]

... and then perform the application:

\[
k \triangleright \text{ap}(e1, e2) \triangleright \text{lam}[\tau](x, e) \Rightarrow k \triangleright [e2/x]e
\]
The $\texttt{K[nat \rightarrow]}$ Abstract Machine: General Recursion

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

Note that evaluation of general recursion requires no stack space.

Safety for the $\texttt{K[nat \rightarrow]}$ Abstract Machine

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

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 $\tau$ into a value of type $\tau'$.

This judgement is inductively defined by:

\[
\frac{k : \tau' \quad f : \tau = \tau'}{k[f] : \tau = \tau'}
\]

Well-Formed States for the $\texttt{K[nat \rightarrow]}$ Abstract Machine

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

This judgement is inductively defined by:

\[
\frac{k : \tau \quad e : \tau \quad e \text{ val}}{k[e] : \tau \quad e \text{ ok}}
\]

Correctness of the $\texttt{K[nat \rightarrow]}$ Abstract Machine

Does the $\texttt{K[nat \rightarrow]}$ 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 $\texttt{K[nat \rightarrow]}$ Abstract Machine and with the PCF $\mathcal{L}$ Machine and vice versa?

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

1. **[Completeness]** If $e \mapsto e'$, where $e' \text{ val}$, then $e \mapsto^* e \circ e'$.
2. **[Soundness]** If $e \mapsto^* e'$, then $e \mapsto^* e'$ with $e' \text{ val}$.
Proving Completeness

Proceed by induction on the definition of \( e \mapsto e' \).

This reduces to two cases:

1. If \( e \) val, then \( e \mapsto e \).

2. If \( e \mapsto e' \), then for every \( v \) val, if \( e \mapsto e \), then \( e' \mapsto v \).

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

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 \( k \mapsto \text{ap}(e'_1; e_2) \mapsto^* k \triangleleft v \) and need to show that \( k \mapsto \text{ap}(e_1; e_2) \mapsto^* k \triangleleft v \).

- It is easy to show that the first step of the former transition is \( k \mapsto \text{ap}(e'_1; e_2) \mapsto k \mapsto \text{ap}(e_1; e_2) \mapsto e'_1 \).

- 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 does not give us.

Proving Completeness

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

Consider, for example, the rule:

\[
\begin{align*}
  e_1 & \Downarrow \lambda e_2 (e_1) \quad e_2 / x \Downarrow v \\
  \text{ap}(e_1; e_2) & \Downarrow v
\end{align*}
\]

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

Proving Completeness

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

\[
\begin{align*}
  k \mapsto \text{ap}(e_1; e_2) & \mapsto k \mapsto \text{ap}(e_1; e_2) \mapsto e_1 \\
  & \mapsto^* k \mapsto \text{ap}(e_1; e_2) \triangleleft \lambda e_2 [(x e)] \\
  & \mapsto k \mapsto [e_2 / x] e \\
  & \mapsto^* k \triangleleft v
\end{align*}
\]

The other cases are handled similarly.
Proving Soundness

We would like to proceed by induction on the multistep transition
\( e \rightarrow^* v \).

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

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

Proving Soundness

To do this, we show the following two facts:

1. If \( s \not\triangleright e \) and \( s \) final, then \( e \) val.

2. If \( s \rightarrow s', s \not\triangleright e', s' \not\triangleright e' \) and \( e' \rightarrow^* v \) where \( v \) val, then \( e \rightarrow^* v \).

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

If \( s \rightarrow s', s \not\triangleright e', \) and \( s' \not\triangleright e' \), then \( e \rightarrow^* e' \).

Proving Soundness

The \( \text{wrap} \) relation is inductively defined as follows:

\[
\begin{align*}
K \triangleright a & \Rightarrow e' \\
K; s(\cdot) & \triangleright e \\
K \triangleright \text{if}e[1, e_1, e_2] & \Rightarrow e' \\
K; [s_1, s_2, s_3] & \triangleright e \\
K \triangleright \text{ap}(e_1, e_2) & \Rightarrow e \\
K; [s_1, s_2] & \triangleright e
\end{align*}
\]

That is, to unravel a state we \textbf{wrap} the stack around the ex-
pression.

Lemma 4

Judgement \( s \not\triangleright e \) has mode \( \langle \gamma, \beta \rangle \) and judgement \( K \triangleright e = e' \) has
mode \( \langle \gamma, \nu, \beta \rangle \)

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 \( K \triangleright e \) for the unique \( e' \) such that \( K \triangleright e = e' \).
Proving Soundness

The next lemma states that unravelling preserves the transition relation.

Lemma 5
If $s \rightarrow s'$, $k \triangleright e = d$, $k \triangleright e' = d'$, then $d \rightarrow d'$.

Proof is by induction on the transition $s \rightarrow s'$. 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 $k$.

Axiom case:

For example, suppose $e = \text{ap}(\lambda a_1[a_2](x); e_2)$, $e' = [v_2/x]e$ with $e \rightarrow e'$ directly.

Assume that $k \triangleright e = d$ and $k \triangleright e' = d'$, we are to show that $d \rightarrow d'$.

We proceed by inner induction on the structure of $k$.

If $k = e$, the result is immediate.

Consider, for instance, the stack $k = k'; \text{ap}(-; e_2)$.

By the inductive definition of wrap, $k' \triangleright \text{ap}(e; e_2) = d$ and $k' \triangleright \text{ap}(e'; e_2) = d'$.

But by the dynamic semantics rules $\text{ap}(e; e_2) \rightarrow \text{ap}(e'; e_2)$.

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

We are finally in position to prove:

Theorem 6
If $s \rightarrow s'$, $s \triangleright e$, and $s' \triangleright e'$, then $e \rightarrow^* e'$.

Proof is by case analysis on the transition of $k'[\text{sat} \rightarrow]$. In each case, after unravelling the transition will correspond to zero or one transitions of $k'[\text{sat} \rightarrow]$.

Suppose that $s = k \triangleright e$ and $s' = k; \triangleright e$. Note that $k \triangleright \text{ap}(e) = e'$ if $k; \triangleright e = e'$, from which the result follows immediately.

Suppose that $s = k; \triangleright \text{ap}(\lambda a_1[a_2](x); e_2)$ and $s' = k; \triangleright [v_2/x]e_2$. Let $e'$ be such that $k; \triangleright \text{ap}(\lambda a_1[a_2](x); e_2) \triangleright e'$. Then $e''$ be such that $k \triangleright [v_2/x]e_2 = e''$. Observe that $k \triangleright \text{ap}(\lambda a_1[a_2](x); e_2) = e'$. The result follows from Lemma 5.

Summary

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

The $k[\text{sat} \rightarrow]$ abstract machine manages control explicitly.