Semantics

1. Denotational Semantics
2. Big Step Semantics
3. Small Step Semantics aka Contextual Semantics
4. Operational Semantics aka Abstract Machine Semantics
5. Extracting Return Values
6. Flattening Terms

1. Denotational Semantics

One major way to explain the meanings of programs is to give a denotational semantics to them. Typically this means providing a translation from the language of interest to a metalanguage. Let's start with a simple pair calculus that has a natural number type as well:

$$\begin{array}{lll} \text{Type } A,B ::= & \mathsf{Nat} & \text{(natural numbers)} \\ \mid & A \otimes B & \text{(pairs)} \end{array}$$

$$\begin{array}{lll} \text{Term } M,N,P ::= & \mathsf{zero} & \text{(zero)} \\ \mid & \mathsf{suc}\ M & \text{(successor)} \\ \mid & M \oplus N & \text{(addition)} \\ \mid & \langle M,N \rangle & \text{(pair)} \\ \mid & \mathsf{fst}\ P & \text{(first)} \\ \mid & \mathsf{snd}\ P & \text{(second)} \end{array}$$

$\boxed{M : A}$ Term M has type A.

$$\frac{}{\mathsf{zero} : \mathsf{Nat}}\ \mathsf{Nat\text{-}I\text{-}1$$

$$\frac{M : \mathsf{Nat} \qquad N : \mathsf{Nat}}{M \oplus N : \mathsf{Nat}}\ \mathsf{Nat\text{-}E$$

$$\mathsf{zero} \oplus N \mapsto_\beta N$$

$$\mathsf{suc}\ M \oplus N \mapsto_\beta \mathsf{suc}(M \oplus N)$$

$$\frac{M : \mathsf{Nat}}{\mathsf{suc}\ M : \mathsf{Nat}}\ \mathsf{Nat\text{-}I\text{-}2$$

$$\frac{M : A \qquad N : B}{\langle M,N \rangle : A \otimes B}\ \otimes\text{I}$$

$$\frac{P : A \otimes B}{\mathsf{fst}\ P : A}\ \otimes\text{E-1} \qquad \mathsf{fst}\ \langle M,N \rangle \mapsto_\beta M$$

$$\frac{P : A \otimes B}{\mathsf{snd}\ P : B}\ \otimes\text{E-2} \qquad \mathsf{snd}\ \langle M,N \rangle \mapsto_\beta N$$

1. Denotational Semantics (cont.)

The language has both types and terms, so we must give denotations (i.e. translations) to each. Let's use mathematics as the language we translate into. The notation $[\![X]\!]$ is used to mean "the denotation/translation of X"; the $[\![\ ]\!]$ are called Scott Brackets, after their inventor Dana Scott. For this system, we'll write $[\![A]\!]_{ty}$ to mean the denotation of type A, and $[\![M]\!]_{tm}$ to mean the denotation of term M. They are given as follows:

$$[\![\mathsf{Nat}]\!]_{ty} = \mathbb{N}$$
$$[\![A \otimes B]\!]_{ty} = [\![A]\!]_{ty} \times [\![B]\!]_{ty}$$

$$[\![\mathsf{zero}]\!]_{tm} = \emptyset$$
$$[\![\mathsf{suc}\ M]\!]_{tm} = 1 + [\![M]\!]_{tm}$$
$$[\![M \oplus N]\!]_{tm} = [\![M]\!]_{tm} + [\![N]\!]_{tm}$$
$$[\![\langle M, N \rangle]\!]_{tm} = ([\![M]\!]_{tm}, [\![N]\!]_{tm})$$
$$[\![\mathsf{fst}\ P]\!]_{tm} = u \quad \text{where} \quad (u, v) = [\![P]\!]_{tm}$$
$$[\![\mathsf{snd}\ P]\!]_{tm} = v \quad \text{where} \quad (u, v) = [\![P]\!]_{tm}$$

This translation can be further elaborated on by proving a soundness property:

If $M : A$, then $[\![M]\!]_{tm} \in [\![A]\!]_{ty}$.

1. Denotational Semantics

The proof proceeds by cases on the proof of $M : A$:

Suppose $\begin{array}{c}\vdots \mathcal{D} \\ M : A\end{array}$ is $\dfrac{}{\text{zero} : \mathbb{N}at}\ \text{Nat-I-1}$. Then we must show that $[\![\text{zero}]\!]_{tm} \in [\![\text{Nat}]\!]_{ty}$, i.e. $\mathcal{O} \in \mathbb{N}$, which is trivially true.

Suppose $\begin{array}{c}\vdots \mathcal{D} \\ M : A\end{array}$ is $\dfrac{\begin{array}{c}\vdots \mathcal{D}' \\ M : \text{Nat}\end{array}}{\text{suc } M : \text{Nat}}\ \text{Nat-I-2}$. Then we must show that $[\![\text{suc } M]\!]_{tm} \in [\![\text{Nat}]\!]_{ty}$, i.e. $1 + [\![M]\!]_{tm} \in \mathbb{N}$. By induction on $\mathcal{D}'$, we know $[\![M]\!]_{tm} \in [\![\text{Nat}]\!]_{ty}$, i.e. $[\![M]\!]_{tm} \in \mathbb{N}$, so $1 + [\![M]\!]_{tm} \in \mathbb{N}$.

EXERCISE

Show the remaining 4 cases.

We could give a different semantics to this system if we wanted. For example, we can give Haskell programs as the denotations of terms, and Haskell types as denotations of types. There are in fact multiple such translations into Haskell! Here is one called a Church Encoding, after its inventor Alonzo Church:

1. Denotational Semantics (cont.)

$$[\![\mathrm{Nat}]\!]_{ty} = \forall a.\ a \to (a \to a) \to a$$
$$[\![A \otimes B]\!]_{ty} = \forall a.\ ([\![A]\!]_{ty} \to [\![B]\!]_{ty} \to a) \to a$$

$$[\![\mathrm{zero}]\!]_{tm} = \lambda z \to \lambda s \to z$$
$$[\![\mathrm{suc}\ m]\!]_{tm} = \lambda z \to \lambda s \to s\ ([\![m]\!]_{tm}\ z\ s)$$
$$[\![m \oplus n]\!]_{tm} = [\![m]\!]_{tm}\ [\![n]\!]\ (\lambda n \to \lambda z \to \lambda s \to s\ (n\ z\ s))$$
$$[\![\langle m, n \rangle]\!]_{tm} = \lambda p \to p\ [\![m]\!]_{tm}\ [\![n]\!]_{tm}$$
$$[\![\mathrm{fst}\ p]\!]_{tm} = [\![p]\!]_{tm}\ (\lambda x \to \lambda y \to x)$$
$$[\![\mathrm{snd}\ p]\!]_{tm} = [\![p]\!]_{tm}\ (\lambda x \to \lambda y \to y)$$

Denotational semanticses are usually not unique.

One major benefit of having a denotational semantics is that we can use it as a reference point for optimizations. Suppose two programs m and m' are supposed to be related by an optimization, so that m' is the "same" as m, but faster. What does "same" mean here? One answer is that they have the same denotation:

$$[\![m]\!] = [\![m']\!]$$

This kind of transformation from m to m' would then be called a "meaning-preserving transformation."

2. Big Step Semantics

Another important kind of semantics is called Big Step Semantics. In many ways, it can be seen as a relational variant of denotational semantics. One common distinction, tho, is that the "values" of big step semantics tend to be a subset of the terms of the language, rather than some arbitrary thing. In this setting, we call such a subset "normal forms".

It's common to find that subset of values specified by a judgment. Here is one for the Nat-⊗ system:

$\boxed{M \text{ val}}$ A term $\underline{M}$ is in value form.

$$\frac{}{\text{zero val}} \qquad \frac{M \text{ val}}{\text{suc } M \text{ val}}$$

$$\frac{M \text{ val} \quad N \text{ val}}{\langle M, N \rangle \text{ val}}$$

Another common way to specify values is via a grammar:

$$\begin{aligned} \text{Value } U, V ::= \ & \text{zero} \\ \mid \ & \text{suc } V \\ \mid \ & \langle U, V \rangle \end{aligned}$$

2. Big Step Semantics

The task of a big step semantics is now to provide a relation/judgment on terms $M \Downarrow N$ (alternatively $M \Downarrow V$) which holds just when N (or V) is the value form for M. Here it is for Nat-$\oplus$:

$$\frac{}{\mathsf{zero} \Downarrow \mathsf{zero}} \qquad \frac{M \Downarrow \mathsf{zero} \quad N \Downarrow V}{M \oplus N \Downarrow V}$$

$$\frac{M \Downarrow V}{\mathsf{suc}\, M \Downarrow \mathsf{suc}\, V} \qquad \frac{M \Downarrow \mathsf{suc}\, U \quad U \oplus N \Downarrow V}{M \oplus N \Downarrow \mathsf{suc}\, V}$$

$$\frac{M \Downarrow U \quad N \Downarrow V}{\langle M, N \rangle \Downarrow \langle U, V \rangle} \qquad \frac{P \Downarrow \langle U, V \rangle}{\mathsf{fst}\, P \Downarrow U}$$

$$\frac{P \Downarrow \langle U, V \rangle}{\mathsf{snd}\, P \Downarrow V}$$

We typically must also show that if $M : A$ and $M \Downarrow V$, then $V : A$. This property is called subject reduction. Let's show this for the above big step semantics:

2. Big Step Semantics

Suppose M is $\langle M', N' \rangle$ and A is $B \otimes C$. We must show that if we have a proof $\dfrac{\begin{array}{cc} \vdots \mathcal{D} & \vdots \mathcal{E} \\ M' : B & N' : C \end{array}}{\langle M', N' \rangle : B \otimes C} \otimes I$ and a proof $\dfrac{\begin{array}{cc} \vdots \mathcal{F} & \vdots \mathcal{G} \\ M' \Downarrow U & N' \Downarrow V \end{array}}{\langle M', N' \rangle \Downarrow \langle U, V \rangle}$, then it's provable that $\langle U, V \rangle : B \otimes C$.

So, by induction, $\begin{array}{c} \vdots \mathcal{D} \\ M' : B \end{array}$ and $\begin{array}{c} \vdots \mathcal{F} \\ M' \Downarrow U \end{array}$ tells us $U : B$,

and $\begin{array}{c} \vdots \mathcal{E} \\ N' : C \end{array}$ and $\begin{array}{c} \vdots \mathcal{G} \\ N' \Downarrow V \end{array}$ tells us $V : C$. So: $\dfrac{U : B \quad V : C}{\langle U, V \rangle : B \otimes C} \otimes I$

EXERCISE

Show subject reduction for the rest of Nat-$\otimes$.

3. Small Step Semantics aka Contextual Semantics

Another kind of semantics we can give to a programming language is a small-step semantics, aka a contextual semantics. This kind captures the kinds of reasoning about program execution that one finds being done in manual proofs, where β-reductions are performed at the leftmost position only.

As the names suggest, this kind of semantics involves steps that are small(er than big step), and involve some kind of context for the steps. Let's start by considering the contexts.

Contexts, but for what? The leftmost redex! That is, the leftmost place that a β-reduction can happen. How do we express this idea? By reference to values, since a value form has no redexes:

Reduction Context K	$::= \{\}$	(the position of the redex)
	$\mid \mathsf{suc}\ K$	(the redex is inside a successor)
	$\mid K \oplus N$	(the redex is in the first arg of $\oplus$)
	$\mid U \oplus K$	(the redex is in the second arg of $\oplus$)
	$\mid \langle K, N \rangle$	(redex in first part of a pair)
	$\mid \langle U, K \rangle$	(redex in second part of a pair)
	$\mid \mathsf{fst}\ K$	(redex is in arg of fst)
	$\mid \mathsf{snd}\ K$	(redex is in arg of snd)

Note that, for example, if the context has the form $U \oplus K$, we know that the redex can't be in U since U is a value and has no redexes. Thus, if the redex is in the right arg of $\oplus$, it's leftmost in $U \oplus$ whatever.

3. Small Step Semantics aka Contextual Semantics (cont.)

In the Nat-⊗ term $\langle \text{zero} \otimes \text{zero}, \text{snd}\langle \text{zero}, \text{zero}\rangle\rangle$, the leftmost redex is $\text{zero} \otimes \text{zero}$. There's a second redex, namely $\text{snd}\langle \text{zero}, \text{zero}\rangle$, but that one is further right.

Therefore the reduction context for this term is $\langle \{\ \}, \text{snd}\langle \text{zero}, \text{zero}\rangle\rangle$. If this context is called K, we write $K\{\text{zero} \otimes \text{zero}\}$ for the term above.

Ok so that's reduction contexts, now how do we use them? By defining a stepping relation over terms, via contextual closure of β-reduction:

$\boxed{M \mapsto M'}$ Term M steps to/simplifies to term M'.

$$\frac{M \mapsto_\beta M'}{K\{M\} \mapsto K\{M'\}}$$

And then by defining its reflexive transitive closure:

$\boxed{M \mapsto^* M'}$ Term M steps some number of times to M'

$$\frac{}{M \mapsto^* M}\ \text{refl} \qquad \frac{M \mapsto^* M' \quad M' \mapsto M''}{M \mapsto^* M''}\ \text{trans}$$

3. Small Step Semantics aka Contextual Semantics (cont.)

Here is a full proof of $\langle \text{zero} \oplus \text{zero}, \text{snd}\langle \text{zero}, \text{zero}\rangle\rangle \mapsto^* \langle \text{zero}, \text{zero}\rangle$:

$$\dfrac{\dfrac{}{\langle \text{zero}\oplus\text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \mapsto^* \langle \text{zero}\oplus\text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle}\ \text{refl} \qquad \dfrac{\text{choose } K = \langle \{\,\}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \qquad \text{zero}\oplus\text{zero} \mapsto_\beta \text{zero}}{\langle \text{zero}\oplus\text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \mapsto \langle \text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle}}{\langle \text{zero}\oplus\text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \mapsto^* \langle \text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle}$$

Call this $\mathcal{D}$

$$\dfrac{\mathcal{D} \qquad \dfrac{\text{choose } K = \langle \text{zero}, \{\,\}\rangle \qquad \text{snd}\langle \text{zero},\text{zero}\rangle \mapsto_\beta \text{zero}}{\langle \text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \mapsto \langle \text{zero},\text{zero}\rangle}}{\langle \text{zero}\oplus\text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \mapsto^* \langle \text{zero},\text{zero}\rangle}$$

The fact that all of this is transitive closure over contextual closure of β-reduction means we can just write the simple blackboard proof instead:

$$\begin{aligned} & \langle \text{zero}\oplus\text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \\ \mapsto\ & \langle \text{zero}, \text{snd}\langle \text{zero},\text{zero}\rangle\rangle \\ \mapsto\ & \langle \text{zero},\text{zero}\rangle \end{aligned}$$

It's good form to check subject reduction for $M \mapsto^* M'$ as well. Additionally, it's important to verify that for every M, there is some V such that $M \mapsto^* V$. Moreover, if $M \not\mapsto V$, then $M \mapsto^* V$, and vice versa.

EXERCISE

Do all of this for Nat-⊗.

4. Operational Semantics aka Abstract Machine Semantics

The last kind of semantics we're going to look at is operational semantics, or abstract machine semantics, so called because it explains the meanings of programs in terms of how some abstract computing machine operates.

The high level view is that we must define some notion of machine state, usually referenced with a metavar like M, and a state transition relation/judgment $M \mapsto^* M'$. In most opsems, $M \mapsto^* M'$ is just the reflexive transitive closure of a relation $M \mapsto M'$, which is what we really care about defining. Then typically we define some translation of programs or terms into machine states in such a way that if m translates to M, V translates to M' and $m \Downarrow V$, then $M \mapsto^* M'$.

For any given programming language/type theory, there is likely going to be many different valid abstract machines that correspond to it. Despite this, one stands out as "obvious": the CK machine.

A CK machine (C for control, K for continuation) is a machine that very closely matches the structure of the big step semantics.

The machine states for the CK machine for Nat-⊗ are

$$\begin{array}{lll} \text{Machine State } M ::= & K \triangleright m & \text{(starting work on sub-term } m\text{)} \\ \mid & K \triangleleft V & \text{(finished work on a subterm with value } V\text{)} \\ \mid & \text{halt } V & \text{(finished completely with value } V\text{)} \end{array}$$

4. Operational Semantics aka Abstract Machine Semantics (cont.)

We haven't defined K yet, which is the "continuation", but it's simple: just a list of "frames":

$$\begin{array}{lll} \text{Continuation } K ::= \varepsilon & & \text{(empty continuation)} \\ \quad\quad\quad\quad\quad\quad \mid K, F & & \text{(non-empty continuation)} \end{array}$$

We still must define what "F" is, but up to here, almost every part of this definition is language agnostic! We can also state the generic embedding of terms into machine states:

M embeds to $\varepsilon \triangleright M$

V embeds to $\text{halt}\, V$

if $M \Downarrow V$, then $\varepsilon \triangleright M \longmapsto^* \text{halt}\, V$ must be provable

Now let's define F, which is language dependent. For Nat-⊕, this is

$$\begin{array}{lll} \text{Continuation Frame } F ::= \text{suc}\ \_ & & \text{(inside successor)} \\ \quad\quad\quad\quad\quad\quad \mid \_ \oplus M & & \text{(inside left arg of } \oplus\text{)} \\ \quad\quad\quad\quad\quad\quad \mid V \oplus \_ & & \text{(inside right arg of } \oplus\text{)} \\ \quad\quad\quad\quad\quad\quad \mid \langle \_, M \rangle & & \text{(inside first part of pair)} \\ \quad\quad\quad\quad\quad\quad \mid \langle V, \_ \rangle & & \text{(inside second part of pair)} \\ \quad\quad\quad\quad\quad\quad \mid \text{fst}\ \_ & & \text{(inside scrutinee of fst)} \\ \quad\quad\quad\quad\quad\quad \mid \text{snd}\ \_ & & \text{(inside scrutinee of snd)} \end{array}$$

4. Operational Semantics aka Abstract Machine Semantics (cont.)

The continuations for Nat-⊗ are very similar to the contexts in a contextual semantics, but flattened to a list and upside down, exposing the deepest

The stepping relation for Nat-⊗'s CK machine is as follows:

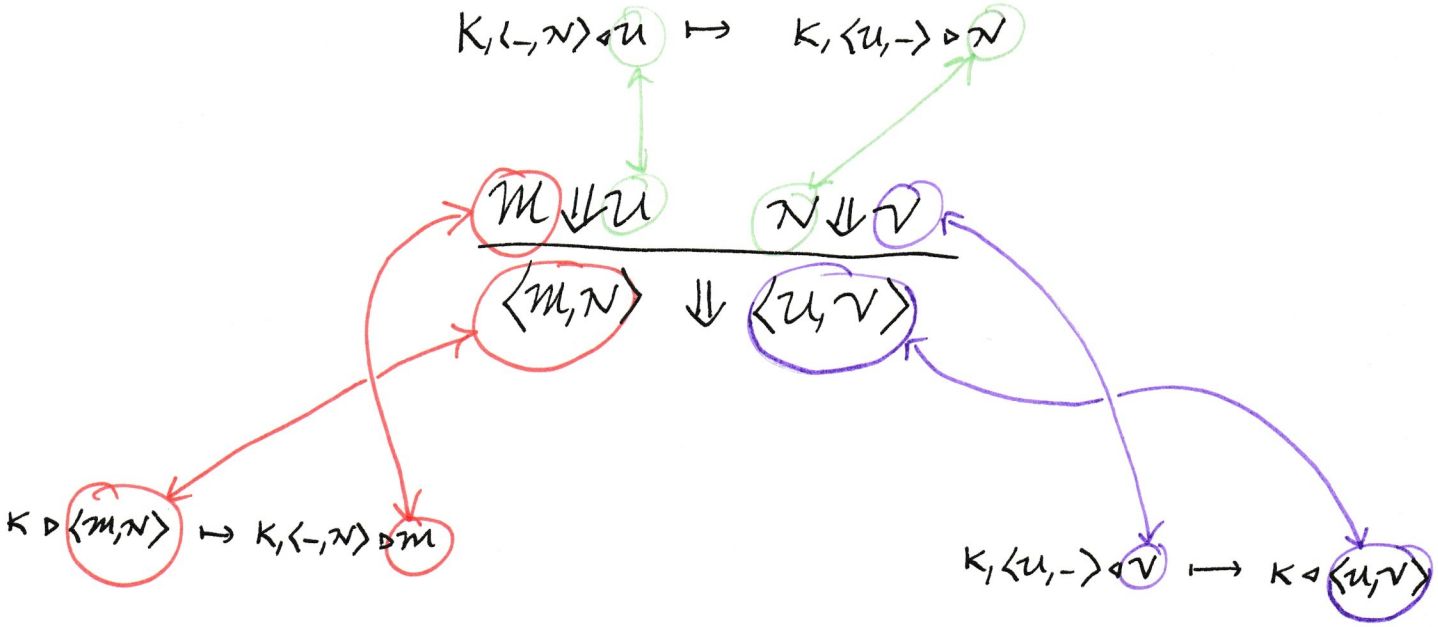
$$
\begin{array}{lcl}
K \triangleright \mathsf{zero} & \mapsto & K \triangleleft \mathsf{zero} \\
K \triangleright \mathsf{suc}\, M & \mapsto & K, \mathsf{suc}\_ \triangleright M \\
K, \mathsf{suc}\_ \triangleleft V & \mapsto & K \triangleleft \mathsf{suc}\, V \\
K \triangleright M \oplus N & \mapsto & K, \_ \oplus N \triangleright M \\
K, \_ \oplus N \triangleleft U & \mapsto & K, U \oplus \_ \triangleright N \\
K, \mathsf{zero} \oplus \_ \triangleleft V & \mapsto & K \triangleleft V \\
K, \mathsf{suc}\, U \oplus \_ \triangleleft V & \mapsto & K, U \oplus \_ \triangleleft \mathsf{suc}\, V \\
K \triangleright \langle M, N \rangle & \mapsto & K, \langle \_, N \rangle \triangleright M \\
K, \langle \_, N \rangle \triangleleft U & \mapsto & K, \langle U, \_ \rangle \triangleright N \\
K, \langle U, \_ \rangle \triangleleft V & \mapsto & K \triangleleft \langle U, V \rangle \\
K \triangleright \mathsf{fst}\, P & \mapsto & K, \mathsf{fst}\_ \triangleright P \\
K, \mathsf{fst}\_ \triangleleft \langle U, V \rangle & \mapsto & K \triangleleft U \\
K \triangleright \mathsf{snd}\, P & \mapsto & K, \mathsf{snd}\_ \triangleright P \\
K, \mathsf{snd}\_ \triangleleft \langle U, V \rangle & \mapsto & K \triangleleft V
\end{array}
$$

This is a bewildering array of rules! Where does it all come from? Well, that's actually quite nice: It comes from the definition of $M \Downarrow V$! Let's see how by looking

4. Operational Semantics aka Abstract Machine Semantics (cont.)

at the rule for pairs: $\dfrac{M \Downarrow U \quad N \Downarrow V}{\langle M, N\rangle \Downarrow \langle U, V\rangle}$

Each part of this rule corresponds to parts of the three stepping relation rules for pairs:



Additionally, the continuation K corresponds to the rest of the term that contained the pair $\langle M, N\rangle$ in the proof of big step reduction. Indeed, if we "perform" the big step reduction left to right, the additions to the continuation — $\langle -, N\rangle$ and $\langle U, -\rangle$ — correspond to precisely the information we need to remember to finish evaluating the pair while working on the sub-parts M and N, respectively.

4. Operational Semantics aka Abstract Machine Semantics (cont.)

Let's look now at how the big step semantics relates to the operational semantics. We said before that the general relationship is that if $M \Downarrow V$, then embedding $M = \varepsilon \triangleright M$ and embedding $M' = \text{halt } V$ step as $M \mapsto^* M'$ i.e. $\varepsilon \triangleright M \mapsto^* \text{halt } V$. How do we show this? As is common, we try induction, on M in this case. For pairs:

> If $M = \langle M', N' \rangle$, then there are U, V such that $M' \Downarrow U$, $N' \Downarrow V$, and $\langle M', N' \rangle \Downarrow \langle U, V \rangle$.
> The embedding of $\langle M', N' \rangle$ is $M = \varepsilon \triangleright \langle M', N' \rangle$, and of $\langle U, V \rangle$ is $M' = \text{halt } \langle U, V \rangle$. We must show $\varepsilon \triangleright \langle M', N' \rangle \mapsto^* \text{halt } \langle U, V \rangle$. By induction, since $M' \Downarrow U$, it must be that $\varepsilon \triangleright M \mapsto^* \text{halt } U$, and likewise since $N' \Downarrow V$, it must be that $\varepsilon \triangleright N \mapsto^* \text{halt } V$. But at this point we've stuck. There's no way to combine these proofs.

Ok, so induction like this is the wrong approach. What's the <u>right</u> one? We <u>strengthen</u> the claim, then prove it inductively!

4. Operational Semantics aka Abstract Machine Semantics (cont.)

The claim we actually need to prove is this much stronger one:

For all K, $K \triangleright M \longmapsto^* K \triangleleft V$ whenever $M \Downarrow V$.

Now if we do induction on M, we'll succeed:

If $M = \langle M', N' \rangle$, then $M' \Downarrow U$, $N' \Downarrow V$, and $\langle M', N' \rangle \Downarrow \langle U, V \rangle$ for some U and V. Inductively, we know that for all K, $K \triangleright M' \longmapsto^* K \triangleleft U$, also for all K, $K \triangleright N' \longmapsto^* K \triangleleft V$. So:

for all K, $K \triangleright \langle M', N' \rangle \longmapsto K, \langle -, N' \rangle \triangleright M'$ by the definition of $\longmapsto$, and $K, \langle -, N' \rangle \triangleright M' \longmapsto^* K, \langle -, N' \rangle \triangleleft U$ by induction, choosing the context to be $K, \langle -, N' \rangle$. Then by the definition of $\longmapsto$, $K, \langle -, N' \rangle \triangleleft U \longmapsto K, \langle U, - \rangle \triangleright N'$, and then by induction, $K, \langle U, - \rangle \triangleright N' \longmapsto^* K, \langle U, - \rangle \triangleleft V$. Finally by definition of $\longmapsto$, $K, \langle U, - \rangle \triangleleft V \longmapsto K \triangleleft \langle U, V \rangle$. So, $K \triangleright \langle M', N' \rangle \longmapsto^* K \triangleleft \langle U, V \rangle$

4. Operational Semantics aka Abstract Machine Semantics (cont.)

Once we have the stronger claim, we can use it to prove the original goal that if $M \Downarrow V$, then $\varepsilon \triangleright M \mapsto^* \mathrm{halt}\, V$.

Namely, pick $K = \varepsilon$. Then $\varepsilon \triangleright M \mapsto^* \varepsilon \triangleleft V$ by the stronger theorem. Then trivially, the definition of $\mapsto$ says $\varepsilon \triangleleft V \mapsto \mathrm{halt}\, V$. Easy!

5. Extracting Return Values

In the canonical CK machine, when a value is returned to a continuation with state $K \triangleleft V$, it's common to keep that value around for later use by storing it in a continuation frame, such as with the rule

$$K, \langle -, N \rangle \triangleleft U \longmapsto K, \langle U, - \rangle \triangleright N$$

A useful alternative is to extract all those values into a separate return stack ℓ like so:

$$K ; \ell \triangleright \langle M, N \rangle \longrightarrow K, \langle -, N \rangle ; \ell \triangleright M$$

$$K, \langle -, N \rangle ; \ell \triangleleft \longmapsto K, \langle -, - \rangle ; \ell \triangleright N$$

$$K, \langle -, - \rangle ; \ell, U, V \triangleleft \longmapsto K ; \ell, \langle U, V \rangle \triangleleft$$

Notice that we no longer return a value with $\triangleleft$, instead, they go onto the return stack ℓ, and we also <u>expect</u> that ℓ will have the values needed on it as well.

Here's the full CKℓ machine for Nat $-\otimes$:

5. Extracting Return Values (cont.)

Machine State $M ::= K ; \rho \triangleright m$ (starting on m)
$\mid K ; \rho \triangleleft$ (finishing)
$\mid \text{halt } V$ (halted)

Return Stack $\rho ::= \varepsilon$ (empty)
$\mid \rho, V$ (non-empty)

Continuation $K ::= \varepsilon$ (empty)
$\mid K, F$ (non-empty)

Continuation Frame $F ::= \text{suc}\_$ (in suc)
$\mid \_ \oplus N$ (in $\oplus$'s first arg)
$\mid \_ \oplus \_$ (in $\oplus$'s second arg)
$\mid \langle \_, N \rangle$ (in pair's first component)
$\mid \langle \_, \_ \rangle$ (in pair's second component)
$\mid \text{fst}\_$ (in fst)
$\mid \text{snd}\_$ (in snd)

$K ; \rho \triangleright \text{zero} \mapsto K ; \rho, \text{zero} \triangleleft$

$K ; \rho \triangleright \text{suc } m \mapsto K, \text{suc}\_ ; \rho \triangleright m$

$K, \text{suc}\_ ; \rho, V \triangleleft \mapsto K ; \rho, \text{suc } V \triangleleft$

$K ; \rho \triangleright m \oplus N \mapsto K, \_ \oplus N ; \rho \triangleright m$

$K, \_ \oplus N ; \rho \triangleleft \mapsto K, \_ \oplus \_ ; \rho \triangleright N$

$K, \_ \oplus \_ ; \rho, \text{zero}, V \triangleleft \mapsto K ; \rho, V \triangleleft$

$K, \_ \oplus \_ ; \rho, \text{suc } U, V \triangleleft \mapsto K, \_ \oplus \_ ; \rho, U, \text{suc } V$

$K ; \rho \triangleright \langle m, N \rangle \mapsto K, \langle \_, N \rangle ; \rho \triangleright m$

$K, \langle \_, N \rangle ; \rho \triangleleft \mapsto K, \langle \_, \_ \rangle ; \rho \triangleright N$

$K, \langle \_, \_ \rangle ; \rho, U, V \mapsto K ; \rho, \langle U, V \rangle \triangleleft$

$K ; \rho \triangleright \text{fst } p \mapsto K, \text{fst}\_ ; \rho \triangleright p$

$K, \text{fst}\_ ; \rho, \langle U, V \rangle \triangleleft \mapsto K ; \rho, U \triangleleft$

$K ; \rho \triangleright \text{snd } p \mapsto K, \text{snd}\_ ; \rho \triangleright p$

$K, \text{snd}\_ ; \rho, \langle U, V \rangle \triangleleft \mapsto K ; \rho, V \triangleleft$

6. Flattening Terms

One last move we can make that's especially interesting is to move from hierachical terms to flattened terms. We observe that in the CKe machine, the continuation K is where much of the future computation is remembered, the rest being to the right of ▷, and that the actual interesting computations happen when the continuation frames have no sub-terms (i.e. when they've be computed already).

So the part of K that really "conveys" the computational core of the term is a sequence of term former names, things like $\langle -, - \rangle$, or $- \oplus -$, or snd.

Further reflection shows that we reach those in a specific order relative to the original term: a post-order traversal!

The term $\mathsf{fst}\langle \mathsf{suc}\ \mathsf{zero}, \mathsf{zero} \rangle$ will visit its corresponding sub-term-free frames in the order $\mathsf{suc}-$, $\langle -, - \rangle$, then $\mathsf{fst}-$. Moreover, the subterms zero and zero (the other one), which don't have frames themselves, do change e just like $\mathsf{suc}-$, $\langle -, - \rangle$, and $\mathsf{fst}-$ do, and they do this also in post-order traversal fashion. The full post-order traversal is $\mathsf{zero}, \mathsf{suc}-, \mathsf{zero}, \langle -, - \rangle, \mathsf{fst}-$. You might recognize this as Reverse Polish Notation for these operations.

6. Flattening Terms (cont.)

Flattening to RPN gives us a very useful, simple abstract machine. Let's first define the flattening process:

```
Operation  O ::= ZERO
               | SUC
               | PLUS
               | PAIR
               | FST
               | SND

Program    P ::= DONE
               | O; P
```

```
flatten :: Term → Program
flatten Zero      = ZERO
flatten (Suc m)   = flatten M ; SUC
flatten (m ⊕ n)   = flatten M ; flatten N ; PLUS
flatten ⟨M,N⟩     = flatten M ; flatten N ; PAIR
flatten (fst P)   = flatten P ; FST
flatten (snd P)   = flatten P ; SND
```

The machines for the RPN programs are simple:

```
Machine State  M ::= ρ ▷ P
                   | halt V
```

with the state transitions

$$\begin{aligned}
\ell \triangleright \text{ZERO}; P &\longmapsto \ell, \text{zero} \triangleright P \\
\ell, V \triangleright \text{SUC}; P &\longmapsto \ell, \text{suc}\, V \triangleright P \\
\ell, \text{zero}, V \triangleright \text{PLUS}; P &\longmapsto \ell, V \triangleright P \\
\ell, \text{suc}, V \triangleright \text{PLUS}; P &\longmapsto \ell, U, \text{suc}\, V \triangleright \text{PLUS}; P \\
\ell, U, V \triangleright \text{PAIR}; P &\longmapsto \ell, \langle U, V \rangle \triangleright P \\
\ell, \langle U, V \rangle \triangleright \text{FST}; P &\longmapsto \ell\, U \triangleright P \\
\ell, \langle U, V \rangle \triangleright \text{SND}; P &\longmapsto \ell\, V \triangleright P \\
\varepsilon, V \triangleright \text{DONE} &\longmapsto \text{halt}\, V
\end{aligned}$$

This RPN machine is in a very real sense one of simplest sort-of implemental machines for Nat-⊛. How values are translated to concrete machines isn't yet clear, but there are very well established methods for turning these operations and the transition relation into actual chip designs. Some minor details are all that stand between us and an actual chip and compiler.