6. Automata-Theoretic Models of Higher-Order Computation

based on "Recursive Schemes, Krivine Machines, and Collapsible Pushdown Automata",

Sylvain Salvati & Igor Walukiewicz, 2013.

Goal: Introduce Krivine machines and collapsible pushdown automata, operational models capturing higher-order computation.

Idea: Krivine machines $\to$ Operational understanding of reduction

Collapsible pushdown automata $\to$ Pushdown view to functional programming.

6.1 Krivine Machines

Idea: Compute head normal form.

Technically: Use explicit substitutions called environments.

Environments assign closures to free variables,

very much like valuations σ in $\mathcal{M}[\![F]\!]\sigma$ in logic.

Closures are pairs consisting of a λY-term and again an environment.

Definition:

- Closures C and environments ρ are defined by mutual recursion:

$$C ::= (M, \rho)$$

$$\rho ::= \emptyset \mid \rho[x \mapsto C].$$

Here, M is a λY-term, $\emptyset$ stands for the empty environment, and $\rho[x \mapsto C]$ coincides with ρ on all variables except x, which it maps to C.

- We will require that in a closure (M, ρ), the environment ρ is defined for every free variable of M. Intuitively, the closure denotes the λY-term obtained by substituting in M every free variable x by the λY-term denoted by the closure $\rho(x)$. // Recursion.

For example,

$$(\lambda y.\, x\, y\, ,\, [x \mapsto (a\, z\, .\, [z \mapsto b])])$$

denotes

$$\lambda y. (a\, b)\, y .$$

- Krivine machines are interpreters for λY-terms. As such, they do not have a syntax. Their state at runtime (configuration) is solely determined by the input.

Definition:

- A configuration of a Krivine machine is a triple

$$(M, \rho, S),$$

where
  - M is a λY-term,
  - ρ is an environment, and
  - $S \in C^*$ is a stack of closures, with the topmost stack element written to the left.

Computations of Krivine machines are defined by means of a <u>transition relation</u> among configurations. It is defined by the following rules:

(1) $(\lambda x. M,\ \rho,\ (N, \rho').S) \longrightarrow (M, \rho[x \mapsto (N,\rho')], S)$

(2) $(MN,\ \rho,\ S) \longrightarrow (M,\ \rho,\ (N,\rho).S)$

(3) $(Y \lambda x. M, \rho, S) \longrightarrow (M, \rho[x \mapsto (Y\lambda x. M, \rho)], S)$

(4) $(x, \rho, S) \longrightarrow (M, \rho', S)$, where $\rho(x) = (M, \rho')$.

<u>Note</u>:

- The machine is deterministic.
- It is actually nonsense to refer to Krivine machines in plural, there is precisely one.

<u>Intuition</u>:

Rule (1): To evaluate $\lambda x. M$, look for the argument at the top of the stack and bind it to x. Then calculate the value of M.

Rule (2): To evaluate an application MN, put N onto the stack together with the current environment. This allows us to evaluate N when needed. Continue with the evaluation of M.

Rule (3): This combines the previous two rules.

Rule (4): Take the value of the variable from the environment and evaluate it.
The value is not just a term but also an environment, giving the right meaning of free variables in a term.

Example:

Consider $(\lambda x y z.\ x\ y\ z)\ a\ b\ c$

which (with more brackets) is

$$(((\lambda x.\ \lambda y.\ \lambda z.\ (x\ y)\ z)\ a)\ b)\ c.$$

Here,

$$x, a : \sigma \to \sigma \to \sigma$$
$$y, z, b, c : \sigma.$$

The Krivine machine has the following transitions:

$$(((\lambda x.\ \lambda y.\ \lambda z.\ (xy)\ z)\ a)\ b)\ c,\ \emptyset,\ \varepsilon)$$
$$\xrightarrow{(2)} (((\lambda x.\ \lambda y.\ \lambda z.\ (xy)\ z)\ a)\ b,\ \emptyset,\ (c, \emptyset))$$
$$\xrightarrow{(2)} (\ (\lambda x.\ \lambda y.\ \lambda z.\ (x\ y) z)\ a,\ \emptyset,\ (b, \emptyset).(c, \emptyset)\ )$$
$$\xrightarrow{(2)} (\ \lambda x.\ \lambda y.\ \lambda z.\ (x\ y)\ z,\ \emptyset,\ (a, \emptyset).(b, \emptyset).(c, \emptyset)\ )$$
$$\xrightarrow{(1)} (\ \lambda y.\ \lambda z.\ (x\ y)\ z,\ [x \mapsto (a, \emptyset)],\ (b, \emptyset).(c, \emptyset)\ )$$
$$\xrightarrow{(1)} (\ \lambda z.\ (x\ y)\ z,\ [x \mapsto (a, \emptyset),\ y \mapsto (b, \emptyset)],\ (c, \emptyset)\ )$$
$$\xrightarrow{(1)} (\ (x\ y)\ z,\ \underbrace{[x \mapsto (a, \emptyset),\ y \mapsto (b, \emptyset),\ z \mapsto (c, \emptyset)]}_{=:\rho},\ \varepsilon)$$

$\xrightarrow{(2)} (x\,y, \rho, (z,\rho))$

$\xrightarrow{(2)} (x, \rho, (y,\rho).(z,\rho))$

$\xrightarrow{(4)} (a, \emptyset, (y,\rho).(z,\rho))$.

There are no more transitions left.

Indeed, the head normal form is

$$(a\,b)\,c.$$

We will be interested in configurations reachable from $(M, \emptyset, \varepsilon)$.

Every such configuration satisfies strong invariants, summarized by the next lemma.

<u>Lemma</u>:

Let M be a λY-term of type o and (N, ρ, S) a configuration reachable from $(M, \emptyset, \varepsilon)$.

Then

(1) N is a subterm of M (hence typable and from a finite set).

(2) Environment ρ associates to a free variable x^α a closure (K, ρ') so that K also has type α.
We also say the closure is of type α.
Moreover, K is a subterm of M.

(3) The number of elements in S is determined by the type of N.
There are k elements when the type of N is $\alpha_1 \to \dots \to \alpha_k \to o$.

We explain how Krivine machines compute Böhm trees.

Definition:

Consider configuration (M, ρ, ε).

- If the computation of the Krivine machine from (M, ρ, ε) does not terminate, we set
$$KTree(M, \rho, \varepsilon) := \Omega .$$
- If the computation terminates, then
$$(M, \rho, \varepsilon) \longrightarrow^{*} (b, \rho', (N_1, \rho_1) \dots (N_k, \rho_k)),$$
for some $b \neq Y$ and $b \neq \Omega$.
In this situation,
$$KTree(M, \rho, \varepsilon) := \begin{array}{ccc} & b & \\ \swarrow & \cdots & \searrow \\ KTree(N_1, \rho_1, \varepsilon) & & KTree(N_k, \rho_k, \varepsilon) \end{array} .$$
By the lemma above, k is the arity of b.
If we are working over tree signatures, $k = 0$ or $k = 2$.
- We now define
$$KTree(M) := KTree(M, \emptyset, \varepsilon),$$
for M closed and of type o.

The <u>Krivine tree</u> we just defined is exactly the Böhm tree.

Lemma:

Let M be closed and of type o.
Then $KTree(M) = BT(M)$.

6.2 Collapsible Pushdown Automata

Goal: · Define higher-order stacks: stacks of stacks of stacks of ...

· Define collapsible pushdown automata that work with such higher-order stacks.

· What is special is a collapse operation that, intuitively, restores an environment. This relates to Rule (4) in Krivine machines.

Intuitively, an order-n stack is a stack of order-$(n-1)$ stacks. The stack characters are annotated by collapse links that point to a position in the stack. This can be understood as the context in which the character was created.

Definition:

· Let Σ be a stack alphabet (alphabets are always finite) together with a partition function

$$\lambda : \Sigma \longrightarrow [1, n], \quad n \geq 1.$$

// The partition assumption is not standard but often used.

· An order-0 stack (with up to order-n collapse links) is an annotated stack symbol $a^i \in \Sigma \times \mathbb{N}$.

· An order-k stack (with up to order-n collapse links), $k \geq 1$, is a non-empty sequence $w = [w_1 \ldots w_\ell]_k$ (with $\ell > 0$) such that each w_i is an order-$(k-1)$ stack (with up to order-n collapse links).

- By $Stacks_n$ we denote the set of order-n stacks (with up to order-n collapse links).

The top-of-stack is drawn to the left.

We use the following operations on stacks.

<u>Definition</u>:

Given an order-k stack with up to order-n collapse links,

- $top_{k'}$ returns the topmost element of the topmost order-k' stack.

  The definition is by induction on the order:

  Let $w = [w_1 \ldots w_\ell]_k$.

  Then
  $$top_k(w) := w_1$$
  $$top_{k'}(w) := top_{k'}(w_1), \text{ where } k' < k.$$

- The operation bot_k^i removes all but the last i elements from the topmost order-k stack.

  It does not change the order ($top_{k'}$ returns an element of order-$(k'-1)$) and requires $i \in [1, \ell]$ (with $w = [w_1 \ldots w_\ell]_k$).

  We have
  $$bot_k^i(w) := [w_{\ell-i+1} \ldots w_\ell]_k$$
  $$bot_{k'}^i(w) := [bot_{k'}^i(w_1).w_2 \ldots w_\ell]_k, \text{ where } k' < k.$$

- For technical convenience, $top_{n+1}(w) := w$.

- The idea on a collapse link i on $a \in \Sigma$ with $\lambda(a) = k$ is to collapse a stack w down to $bot_k^i(w)$, provided this is defined.

When $i=0$, the link is considered <u>null</u>.

We omit irrelevant collapse links to improve readability.

- To give an easy definition of (higher-order) push operations, we introduce another auxiliary operation.

<u>Definition</u>:

Let u be a $(k-1)$-stack and $v = [v_1 \dots v_\ell]_n$ be an n-stack, with $k \in [1, n]$.

We define $u :_k v$ as the stack obtained by adding u on top of the topmost order-k stack in v.

Formally,

$$u :_k v \;:=\; [u . v_1 \dots v_\ell]_n \;, \quad \text{if } k = n$$

$$u :_k v \;:=\; [(u :_k v_1)\, v_2 \dots v_\ell]_n \;, \quad \text{if } k < n.$$

<u>Example</u>:

Let $\lambda(a) = 3$ and $\lambda(b) = 2$.

Let $w = [[[a^1 b^1]_1\ [b^1]_1]_2\ [[b^0]_1]_2]_3$ be an order-3 stack.

Then $top_1(w) = a^1$.

The destination of this link is

$$bot_3^1(w) = [[[b^0]_1]_2]_3 .$$

Furthermore,

$$bot_2^1(w) = [[[b^1]_1]_2\ [[b^0]_1]_2]_3 .$$

We have

$$top_2(w) = [a^1 b^1]_1 .$$

Now

$$top_2(w) :_2 bot_2^1(w) = w .$$

Using the above auxiliary operations,

collapsible pushdown automata modify order-n stacks as follows.

<u>Definition</u>:

We define the set

$$\mathrm{Ops}_n := \{\mathrm{push}_2, \dots, \mathrm{push}_n\} \cup \{\mathrm{push}_a, \mathrm{rew}_a \mid a \in \Sigma\} \cup \{\mathrm{pop}_1, \dots, \mathrm{pop}_n\} \cup \{\mathrm{collapse}\}$$

of operations on order-n stacks used by collapsible pushdown automata.

The definition of the operations is as follows, with $w \in \mathrm{Stacks}_n$:

$\mathrm{push}_h(w) := \mathrm{top}_h(w) :_h w,$

$\mathrm{push}_a(w) := a^{l-1} :_1 w,$ where $\mathrm{top}_{h+1}(w) = [w_1 \dots w_l]_h$ with $h = \lambda(a)$ the link order,

$\mathrm{pop}_h(w) := v,$ if $w = u :_h v,$

$\mathrm{collapse}(w) := \mathrm{bot}_h^i(w),$ if $\mathrm{top}_1(w) = a^i$ and $\lambda(a) = h,$

$\mathrm{rew}_b(w) := b^i :_1 v,$ if $w = a^i :_1 v$ and $\lambda(a) = \lambda(b).$

<u>Note</u>:

Our definition of stacks does not permit the empty stack.

This means pop_h is undefined if the resulting v is empty.

Similarly, collapse is undefined if $i=0$.

Thus, the empty stack cannot be reached by the above operations.

Instead, the offending operations will be unavailable.

The same holds for a rewrite operation that would change the order of a link.

Explanation:

- Operation $push_k$ of order $k > 1$ copies the topmost order-k stack.
- Order-1 $push_a$ pushes a onto the topmost order-1 stack, annotated with an order-$\lambda(a)$ collapse link. When executed on a stack w, the link destination is $pop_{\lambda(a)}(w)$.
- pop_k removes the topmost element from the topmost order-k stack.
- The rewrite operation rew_a modifies the topmost character while maintaining the link.
- Collapse, when executed on a^i with $\lambda(a) = k$, pops the topmost order-k stack down to the last i elements.

Example:

Let $\lambda(a) = 3$, $\lambda(b) = 2$, and $w = [[[a^1 b^1]_1 [b^1]_1]_2 [[b^0]_1]_2]_3$.

Given the order-3 link 1 of the topmost a, we have

$$collapse(w) = u \quad \text{with} \quad u = [[[b^0]_1]_2]_3.$$

Now

$$push_3(u) = [[[b^0]_1]_2 [[b^0]_1]_2]_3.$$

A $push_a$ on this stack results in

$$v = [[[a^1 b^0]_1]_2 [[b^0]_1]_2]_3.$$

We have

$$pop_3(v) = u = collapse(v).$$

Note:

There is a subtlety in the interplay of collapse links and higher-order pushes.

For a push$_k$, links pointing outside of $u = top_k(w)$ have the same destination in both copies of u.

Links pointing within u point to different sub-stacks.

Besides the order-n stack, collapsible pushdown automata have a finite control.

Definition:

- The order-n collapsible pushdown automaton over (Σ, λ) is a tuple

$$C = (P, R, p_{init}, a_{init})$$

with

↳ P a finite set of control states with initial state $p_{init} \in P$,

↳ $R \subseteq P \times \Sigma \times Ops_n \times P$ a finite set of transitions (or rules),

↳ $a_{init} \in \Sigma$ an initial stack character.

- A configuration of C is a pair

$$c = (p, w) \in P \times Stacks_n.$$

The transition relation among configurations is defined by

$(p, w) \to (p', w')$, if there is $(p, a, o, p') \in R$ with $a = top_1(w)$ and $w' = o(w)$.

The initial configuration is (p_{init}, w_{init}), $w_{init} = [\ldots[a_{init}]_1\ldots]_n$.

To begin from another stack, one can adjust the rules
and build up the stack as required.

- A <u>computation</u> is a (potentially infinite) sequence of configurations, starting in the initial one,

$c_0, c_1, \ldots$ with $c_0 = c_{init}$ and $c_i \to c_{i+1}$ for all $i \in \mathbb{N}$.

Recall that transitions do not empty the stack
nor change the order of links.

<u>Note</u>:

- It is standard to assign priorities to states
and consider infinite computations.
- It is also common to assume an ownership partitioning of the states
and study parity games over collapsible pushdown automata.
- Also the relationship with Böhm trees can be made explicit.