Linear Reduction

Ari Feiglin and Noam Kaplinski

In this paper we will define the concept of linear reduction in the context of syntax parsing. We will progress through more and more complicated examples, beginning from the programming of a simple calculator until we ultimately have created an extensible programming language.

Table of Contents

0 Notation

$\mathbb{N}$ denotes the set of natural numbers, including 0.
$\overline{\mathbb{N}}$ is defined to be $\mathbb{N} \cup \{\infty\}$.
$f\colon A \longrightarrow B$ means that f is a partial function from A to B.
If X is a set and x is some symbol, then $\mathsf{X}_{\mathsf{x}} = \mathsf{X}^{\mathsf{x}} = \mathsf{X} \cup \{\mathsf{x}\}$.

1 Theoretical Background

1.1 Stateless Reduction

The idea of linear reduction is simple: given a string ξ the first character looks if it can bind with the second character to produce a new character, and the process repeats itself. There is of course, nuance. This nuance hides in the statement "if it can bind": we must define the rules for binding.

Let us define an *reducer* to be a tuple (Σ, β, π) where Σ is an alphabet; $\beta\colon \overline{\Sigma}\times\overline{\Sigma} \longrightarrow \overline{\Sigma}$ is a partial function called the *reduction function* where $\overline{\Sigma} = \Sigma \times \overline{\mathbb{N}}$; and π is the *initial priority function.* A *program* over an reducer is a string over $\overline{\Sigma}$. We write a program like $\sigma^1_{i_1}\cdots\sigma^n_{i_n}$ instead of as pairs $(\sigma^1, i_1)\dots(\sigma^n, i_n)$. In the character σ_i, we call i the *priority* of σ.

Then the rules of reduction are as follows, meaning we define $\beta(\xi)$ for a program: We do so in cases:

(1) If $\xi = \sigma_i$ then $\beta(\xi) = \sigma_0$.

(2) If $\xi = \sigma^1_i\sigma^2_j\xi'$ where $i \geq j$ and $\beta(\sigma^1_i, \sigma^2_j) = \sigma^3_k$ is defined then $\beta(\xi) = \sigma^3_k\xi'$.

(3) Otherwise, for $\xi = \sigma^1_i\sigma^2_j\xi'$, $\beta(\xi) = \sigma^1_i\beta(\sigma^2_j\xi')$.

A string ξ such that $\beta(\xi) = \xi$ is called *irreducible.* Notice that it is possible for a string of length more than 1 to be irreducible: for example if $\beta(\sigma^1, \sigma^2)$ is not defined then $\sigma^1_i\sigma^2_j$ is irreducible.

$$\beta(\sigma_1\tau_2) \xrightarrow{(3)} \sigma_1\beta(\tau_2) \xrightarrow{(1)} \sigma_1\tau_2$$

But such strings are not desired, since in the end we'd like a string to give us a value. So an irreducible string which is not a single character is called *ill-written*, and a string which is not ill-written is *well-written.*

Now the initial priority function is $\pi\colon \Sigma \longrightarrow \overline{\mathbb{N}}$ which gives characters their initial priority. We can then canonically extend this to a function $\pi\colon \Sigma^* \longrightarrow (\Sigma\times\overline{\mathbb{N}})^*$ defined by $\pi(\sigma^1\cdots\sigma^n) = \sigma^1_{\pi(\sigma^1)}\cdots\sigma^n_{\pi(\sigma^n)}$. Then a β-reduction of a string $\xi \in \Sigma^*$ is taken to mean a β-reduction of $\pi(\xi)$.

Notice that once again we require that π only be a partial function. This is since that we don't always need every character in Σ to have an initial priority; some symbols are only given their priority through the β-reduction of another pair of symbols. So we now provide a new definition of a *program*, which is a string $\xi = \sigma^1\cdots\sigma^n \in \Sigma^*$ such that $\pi(\sigma^i)$ exists for all $1 \leq i \leq n$. We can only of course discuss the reductions of programs, as $\pi(\xi)$ is only defined if ξ is a program.

Example: let $\Sigma = \mathbb{N} \cup \{+, \cdot\} \cup \{(n+), (n\cdot) \mid n \in \mathbb{N}\}$. β as follows:

σ^1_i, σ^2_j	$\beta(\sigma^1_i, \sigma^2_j)$
$n, +$	$(n+)$
$n, \cdot$	$(n\cdot)$
$(n+), m$	$n+m$
$(n\cdot), m$	$n\cdot m$
$(n\cdot), (m+)$	$(n\cdot m, +)$
$(n+), (m+)$	$(n+m, +)$
$(n\cdot), (m\cdot)$	$(n\cdot m, \cdot)$

Where n, m range over all values in $\mathbb{N}$. Here $\beta(\sigma_i, \sigma_j)$'s priority is j. We define the initial priorities

$$\pi(n) = \infty, \quad \pi(+) = 1, \quad \pi(\cdot) = 2$$

Now let us look at the string $1 + 2 \cdot 3 + 4$;. Here,

$$\begin{aligned} 1_\infty +_1 2_\infty \cdot_2 3_\infty +_1 4_\infty &\longrightarrow (1+)_1 2_\infty \cdot_2 3_\infty +_1 4_\infty \\ &\longrightarrow (1+)_1(2\cdot)_2 3_\infty +_1 4_\infty \\ &\longrightarrow (1+)_1(2\cdot)_2(3+)_1 4_\infty \\ &\longrightarrow (1+)_1(6+)_1 4_\infty \\ &\longrightarrow (7+)_1 4_\infty \\ &\longrightarrow (7+)_1 4_0 \\ &\longrightarrow (11)_0 \end{aligned}$$

So the rules for β we supplied seem to be sufficient for computing arithmetic expressions following the order of operations. ◊

Example: We can also expand our language to include parentheses. So our alphabet becomes $\Sigma = \mathbb{N} \cup \{+, \cdot, (,)\} \cup \left\{ \underline{n+}, \underline{n\cdot}, \underline{n)} \;\middle|\; n \in \mathbb{N} \right\}$. We distinguish between parentheses and bold parentheses for readability. We extend β as follows:

σ_i^1, σ_j^2	$\beta(\sigma_i^1, \sigma_j^2)$
$n, +$	$\underline{n+}_j$
$n, \cdot$	$\underline{n\cdot}_j$
$\underline{n+}, m$	$(n+m)_j$
$\underline{n\cdot}, m$	$(n \cdot m)_j$
$\underline{n\cdot}, \underline{m+}$	$\underline{n \cdot m, +}_j$
$\underline{n+}, \underline{m+}$	$\underline{n + m, +}_j$
$\underline{n\cdot}, \underline{m\cdot}$	$\underline{n \cdot m, \cdot}_j$
$n,)$	$\underline{n)}_j$
$\underline{n+}, \underline{m)}$	$\underline{n+m)}_j$
$\underline{n\cdot}, \underline{m)}$	$\underline{n \cdot m)}_j$
$(, \underline{n)}$	n_i

$(n+m)_j$ means $n+m$ with a priority of j, not $\underline{n+m}_j$. And we define the initial priorities

$$\pi(n) = \infty, \quad \pi(+) = 1, \quad \pi(\cdot) = 2, \quad \pi(() = \infty, \quad \pi()) = 0$$

So for example reducing $2 \cdot ((1+2) \cdot 2) + 1$,

$$\begin{aligned}
2_\infty *_2 (_\infty (_\infty 1_\infty +_1 2_\infty)_0 *_2 2_\infty)_0 +_1 1_\infty &\longrightarrow \underline{2*}_2 (_\infty (_\infty 1_\infty +_1 2_\infty)_0 *_2 2_\infty)_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty (_\infty \underline{1+}_1 2_\infty)_0 *_2 2_\infty)_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty (_\infty \underline{1+}_1 \underline{2)}_0 *_2 2_\infty)_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty (_\infty \underline{3)}_0 *_2 2_\infty)_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty 3_\infty *_2 2_\infty)_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty \underline{3*}_2 2_\infty)_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty \underline{3*}_2 \underline{2)}_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 (_\infty \underline{6)}_0 +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 6_\infty +_1 1_\infty \\
&\longrightarrow \underline{2*}_2 \underline{6+}_1 1_\infty \\
&\longrightarrow \underline{12+}_1 1_\infty \\
&\longrightarrow \underline{12+}_1 1_0 \\
&\longrightarrow 13_0
\end{aligned}$$

$\diamondsuit$

1.2 Stateful Reduction

Suppose we'd like to reduce a program with variables in it. Then we cannot just use the previous definitions, as the actions of σ (which is to be understood as the function $\beta(\sigma, \bullet)$) are determined before any reduction occurs. We need a way to store the value of variables, a state.

This leads us to the following definition: let Σ_P and Σ_A be two disjoint sets of symbols: Σ_P the set of *printable symbols* and Σ_A the set of *abstract symbols*. Σ_P will generally be a set consisting of the string representations of abstract symbols, be it operators like $+$ and $\cdot$ or variable names. Σ_A are the actual objects which can "execute something". Let us further define $\Sigma = \Sigma_P \cup \Sigma_A$.

Now a state is a mapping from printable symbols to strings. So for example, if x is a printable symbol a line like $\mathsf{let} x = 1$ should change the state so that x maps to the abstract symbol representing 1.

A *point state* is a partial function $s \colon \Sigma_P \longrightarrow \Sigma_A$. If s_1, s_2 are point states, define their composition to be a point state $s_1 s_2$ such that

$$s_1 s_2(\sigma) = \begin{cases} s_2(\sigma) & \sigma \in \operatorname{dom}(s_2) \\ s_1(\sigma) & \sigma \in \operatorname{dom}(s_1) \end{cases}$$

A *state* is a sequence of point states: $\overline{s} = (s_1, \dots, s_n)$. Let us define

$$\mathsf{State} = \{\Sigma_P \longrightarrow \Sigma_A\}^+$$

the set of all states.

Let $\overline{s} = (s_1 \cdots s_n) \in \mathsf{State}$ be a state, then define

- for $\sigma \in \Sigma_P$ we define $s(\sigma) = s_1 \cdots s_n(\sigma)$ (the composition of states),
- define *pop* $\overline{s} = (s_1, \ldots, s_{n-1})$,
- define *push* $\overline{s} = (s_1, \ldots, s_n, \varnothing)$ ($\varnothing$ is the empty state),
- if s is a point state, $\overline{s}s = (s_1, \ldots, s_{n-1}, s_n s)$,
- if s is a point state, $\overline{s} + s = (s_1, \ldots, s_n, s)$ (so *push* $\overline{s} = \overline{s} + \varnothing$).

So if we'd like to revert to a previous state, we simply pop from the current state. And substituting the current state only alters the current (topmost) point state.

Now we begin with an initial β function which is a partial function

$$\beta\colon \overline{\Sigma}_A \times (\overline{\Sigma} \cup \{\varepsilon\}) \times \mathsf{State} \longrightarrow \overline{\Sigma}^* \times \mathsf{State}$$

Recall that $\overline{\Sigma}$ is $\Sigma \times \mathbb{N}$. We will denote tuples in $X \times \mathsf{State}$ by $\langle x, \mid s\rangle$ for $x \in X$ and $s \in \mathsf{State}$ for the sake of readability. So we now wish to extend to a β function

$$\beta\colon \overline{\Sigma}^* \times \mathsf{State} \longrightarrow \overline{\Sigma}^* \times \mathsf{State}$$

We do this as follows: given $\xi \in (\Sigma \times \overline{\mathbb{N}})^*$ and $s \in \mathsf{State}$ we define $\beta\langle \xi \mid s\rangle$ as follows:

(1) if $\xi = \sigma_i \xi'$ for $\sigma \in \Sigma_P$ then $\beta\langle \xi \mid s\rangle = \langle s(\sigma)_i \xi' \mid s\rangle$,

(2) if $\xi = \sigma_i \xi'$ such that $\beta\langle \sigma_i \varepsilon \mid s\rangle = \langle \xi'' \mid s'\rangle$ is defined then $\beta\langle \xi \mid s\rangle = \langle \xi''\xi' \mid s'\rangle$,

(3) if $\xi = \sigma^1_i \sigma^2_j \xi'$ for $\sigma^1 \in \Sigma_A$, $i \geq j$, such that $\beta\big\langle \sigma^1_i \sigma^2_j \bigm| s\big\rangle = \langle \xi'' \mid s'\rangle$ is defined, then $\beta\langle \xi \mid s\rangle = \langle \xi''\xi' \mid s'\rangle$,

(4) otherwise for $\xi = \sigma^1_i \sigma^2_j \xi'$, if $\beta\big\langle \sigma^2_j \xi' \bigm| s\big\rangle = \langle \xi'' \mid s'\rangle$ then $\beta\langle \xi \mid s\rangle = \big\langle \sigma^1_i \xi'' \bigm| s'\big\rangle$.

Notice that (**2**) cares not about the priority of σ, and neither if $\beta(\sigma_i, \tau_j)$ is defined for some $\tau \neq \varepsilon$.

We also define the *initial priority function* to be a map $\pi\colon \Sigma_P \longrightarrow \overline{\mathbb{N}}$ (this is not a partial function: every printable symbol must be given a priority). This is once again canonically extended to a function $\pi\colon \Sigma_P^* \longrightarrow (\Sigma_P \times \overline{\mathbb{N}})^*$. And an *initial state* s_0 which is a point state. The quintuple $(\Sigma_P, \Sigma_A, \beta, \pi, s_0)$ is called an *reducer*. The reduction of a string $\xi \in S$ is the process of iteratively applying β to $\langle \pi(\xi) \mid s_0\rangle$.

Example: let

$$\begin{aligned}
\Sigma_P &= \mathbb{N} \cup \{+, \cdot, =, ;\} \cup \{\texttt{let}\} \cup \big\{x^i \bigm| i \in \mathbb{N}\big\},\\
\Sigma_A &= \mathbb{N} \cup \{(n+), (n\cdot) \mid n \in \mathbb{N}\} \cup \{\texttt{let}\} \cup \big\{(\texttt{let}x^i), (\texttt{let}x^i =) \bigm| i \in \mathbb{N}\big\}
\end{aligned}$$

where the natural numbers in Σ_A are not the same as the natural numbers in Σ_P since they must be disjoint, same for `let`. But they both essentially represent the same thing: s_0 maps $n \mapsto n$ for $n \in \mathbb{N}$ (the left-hand n is in Σ_p, the right-hand n is in Σ_A) and $\texttt{let} \mapsto \texttt{let}$. All other printable symbols are mapped to ε.

And similar to the previous example we define $\pi(n) = \infty$, $\pi(+) = 1$, and $\pi(\cdot) = 2$. We extend this to $\pi(;) = 0$, $\pi(=) = 0$, $\pi(\texttt{let}) = \infty$, and $\pi(x^i) = \infty$.

Let us take the same transitions as the example in the previous section for $n, (n+), (n\cdot)$ (we have to add the condition that the state doesnt change). We further add the transitions

$\big\langle \sigma^1_i \sigma^2_j \bigm| s\big\rangle$	$\beta\big\langle \sigma^1 \sigma^2 \bigm| s\big\rangle$
$\langle \sigma; \mid s\rangle$	$\langle \sigma_j \mid s\rangle$
$\big\langle \mathsf{let}x^i \bigm| s\big\rangle$	$\big\langle (\mathsf{let}x^i)_j \bigm| s\big\rangle$
$\big\langle (\mathsf{let}x^i) = \bigm| s\big\rangle$	$\big\langle (\mathsf{let}x^i =)_j \bigm| s\big\rangle$
$\big\langle (\mathsf{let}x^i =)\sigma \bigm| s\big\rangle$	$\big\langle \varepsilon \bigm| s[x^i \mapsto \sigma]\big\rangle$

In the final transition, $n \in \Sigma_A$. Then for example (we will be skipping trivial reductions):

$$\begin{aligned}
\mathsf{let}x^1 = 1 + 2;\ \mathsf{let}x^2 = 2;\ x^1 \cdot x^2; &\longrightarrow \texttt{let}_\infty x^1_\infty =_0 1_\infty +_1 2_\infty ;_0\ \texttt{let}_\infty x^2_\infty =_0 2_\infty ;_0\ x^1_\infty \cdot_2 x^2_\infty ;_0\\
s_0 \quad &\longrightarrow (\texttt{let}x^1 =)_0 1_\infty +_1 2_\infty ;_0\ \texttt{let}_\infty x^2_\infty =_0 2_\infty ;_0\ x^1_\infty \cdot_2 x^2_\infty ;_0\\
s_0 \quad &\longrightarrow (\texttt{let}x^1 =)_0 3_0\ \texttt{let}_\infty x^2_\infty =_0 2_\infty ;_0\ x^1_\infty \cdot_2 x^2_\infty ;_0\\
s_0[x^1 \mapsto 3] \quad &\longrightarrow \texttt{let}_\infty x^2_\infty =_0 2_\infty ;_0\ x^1_\infty \cdot_2 x^2_\infty ;_0\\
s_0[x^1 \mapsto 3,\ x^2 \mapsto 2] \quad &\longrightarrow x^1_\infty \cdot_2 x^2_\infty ;_0\\
s_0[x^1 \mapsto 3,\ x^2 \mapsto 2] \quad &\longrightarrow 3_\infty \cdot_2 x^2_\infty ;_0\\
s_0[x^1 \mapsto 3,\ x^2 \mapsto 2] \quad &\longrightarrow (3\cdot)_2 x^2_\infty ;_0\\
s_0[x^1 \mapsto 3,\ x^2 \mapsto 2] \quad &\longrightarrow (3\cdot)_2 2_\infty ;_0\\
s_0[x^1 \mapsto 3,\ x^2 \mapsto 2] \quad &\longrightarrow 6_0
\end{aligned}$$

$\lozenge$

1.3 Valued Reduction

Notice that in the previous examples, many symbols had values, be it a number, a list, etc. This leads us to the next variation of linear reduction: *valued* linear reduction. Here we start with the sets: $\mathcal{U}$ the universe of values, Σ_P the set of printable types, and Σ_A the set of abstract types.

Here $\mathcal{U}$ is some arbitrary set, it will contain of all possible values (numbers, lists, strings, functions, etc.) in our program. Σ_P and Σ_A are sets of symbols which we call *types*.

Let us further define $\Pi_A := \Sigma_A \times \mathcal{U}$ to be the set of *abstract values*, $\Sigma := \Sigma_P \cup \Sigma_A$ the set of types, and $\Pi = \Sigma_P \cup \Pi_A$ the set of *values* (printable types are also values). Elements of $\overline{\Pi}$ will be written like $\sigma_n(v)$ where σ is the type, n the priority, and v the value (nothing for printable types).

Let us define a point-state, similar to stateful reductions, as partial functions from Σ_P to Π_A. Then a state is defined as a sequence of point-states similar to before.

In valued reduction, we abstract away some inputs to the initial beta-reducer in order to allow for easier implementation. An initial beta-reducer is a partial function

$$\hat{\beta}\colon \Sigma_A \times \Sigma_\varepsilon \times \text{State} \longrightarrow \Sigma_A^\varepsilon \times (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}} \to \overline{\mathbb{Z}}) \times (\mathcal{U} \times \mathcal{U} \rightharpoonup \mathcal{U} \times \Sigma_P^* \times \text{State})$$

We extend this to a derived β-reducer,

$$\beta\colon \overline{\Pi}^* \times \text{State} \longrightarrow \overline{\Pi}^* \times \text{State}$$

with the following rules: given an input $\langle \xi \mid s\rangle$ its image is

(1) If $\xi = \sigma_n \xi'$ for $\sigma \in \Sigma_p$ then
$$\beta\langle \xi \mid s\rangle = \langle s(\sigma)_n \xi' \mid s\rangle.$$

(2) If $\xi = \sigma_i(v)\xi'$ and $\hat{\beta}(\sigma, \varepsilon, s) = (\alpha, \rho, f)$ is defined, then if $f(v) = (w, \omega, s')$ and $\rho(i) = k$ then
$$\beta\langle \xi \mid s\rangle = \langle \alpha_k(w)\pi(\omega)\xi' \mid s'\rangle.$$

(3) If $\xi = \sigma_i(v)\tau_j(u)\xi'$ and $i \geq j$ and $\hat{\beta}(\sigma, \tau, s) = (\alpha, \rho, f)$ is defined, then if $f(v, u) = (w, \omega, s')$ and $\rho(i, j) = k$ then
$$\beta\langle \xi \mid s\rangle = \langle \alpha_k(w)\pi(\omega)\xi' \mid s\rangle.$$

(4) Otherwise, if $\xi = \sigma_i(v)\xi'$ and $\beta\langle \xi' \mid s\rangle = \langle \xi'' \mid s'\rangle$,
$$\beta\langle \xi \mid s\rangle = \langle \sigma_i(v)\xi'' \mid s'\rangle.$$

1.3.1 States

Similar to before, we define point-states as partial maps $\Sigma_P \longrightarrow \Pi_A$. And if s_1, s_2 are two point-states and $\sigma \in \Sigma_P$ then
$$s_1 s_2(\sigma) = \begin{cases} s_2(\sigma) & \sigma \in \operatorname{dom} s_2 \\ s_1(\sigma) & \sigma \in \operatorname{dom} s_1 \end{cases}$$

We will denote finite point states as $[\sigma_1 \mapsto \varkappa_1, \ldots, \sigma_n \mapsto \varkappa_n]$, and this denotes the point-state which maps σ_i to $\varkappa_i$. Then we define a state to be a sequence of point-states. For a state $\bar{s} = (s_1, \ldots, s_n)$, let us define

(1) $\bar{s} + s = (s_1, \ldots, s_n, s)$

(2) *pop* $\bar{s} = (s_1, \ldots, s_{n-1})$

(3) $\bar{s}s = (s_1, \ldots, s_n s)$

(4) $\bar{s}(\sigma) = s_1 \cdots s_n(\sigma)$ for $\sigma \in \Sigma_P$

Furthermore, if $\sigma \in \Sigma_P$ and $\varkappa \in \Pi_A$ let us define $\bar{s}\{\sigma \mapsto \varkappa\}$ as $(s_1, \ldots, s_i[\sigma \mapsto \varkappa], \ldots, s_n)$ where i is the maximum index such that $\sigma \in \operatorname{dom} s_i$.

1.3.2 The Initial Beta Reducer

We will now construct the initial β-reducer for our programming language. By convention, abstract types will be colored red.

Matching: We add abstract types $\mathsf{gobble}, \mathsf{end}$ and the "compound" abstract type $\sigma\mathsf{match}$ with the initial reduction rule

(1) $\sigma\mathsf{match}\ \sigma\ s \longrightarrow \varepsilon\mathsf{fst}\ (\to \varepsilon, \varepsilon, s)$

(2) $\mathsf{gobble}\ \sigma\ s \longrightarrow \varepsilon\ \mathsf{fst}\ (\to \varepsilon, \varepsilon, s)$

(3) $\sigma\ \mathsf{end}\ s \longrightarrow \sigma\ \mathsf{minfty}\ (u \to u)$

Where $\mathsf{fst}, \mathsf{snd}$ are the first and second projection maps respectively, and minfty is the constant map equal to $-\infty$.

Arithmetic: For each abstract type σ we define the "compound" abstract type $\sigma\mathsf{op}$ with rules:

- $\sigma\ \mathsf{op}\ s \longrightarrow \sigma\mathsf{op}\ \mathsf{snd}\ \big(v, f \to (v, f), \varepsilon, s\big)$
- $\sigma\mathsf{op}\ \sigma\mathsf{op}\ s \longrightarrow \sigma\mathsf{op}\ \mathsf{snd}\ \big((v, f), (u, g) \to (f(v, u), g), \varepsilon, s\big)$
- $\sigma\mathsf{op}\ \sigma\ s \longrightarrow \sigma\ \mathsf{snd}\ \big((v, f), u \longrightarrow f(v, u), \varepsilon, s\big)$

Where zero is the constant zero map. Notice that here things of the form (n, f) etc. are *values* (which are just arbitrary). We further define the abstract types num and str. These abstract types we have just defined allow for basic arithmetic:

$$\begin{aligned}
\mathsf{num}_\infty(1)\mathsf{op}_1(+)\mathsf{num}_\infty(2)\mathsf{op}_2(\cdot)\mathsf{num}_\infty(3)\mathsf{end}_{-infty} &\longrightarrow \mathsf{numop}_1(1,+)\mathsf{num}_\infty(2)\mathsf{op}_2(\cdot)\mathsf{num}_\infty(3)\mathsf{end}_{-infty}\\
&\longrightarrow \mathsf{numop}_1(1,+)\mathsf{numop}_2(2,\cdot)\mathsf{num}_\infty(3)\mathsf{end}_{-infty}\\
&\longrightarrow \mathsf{numop}_1(1,+)\mathsf{numop}_2(2,\cdot)\mathsf{num}_{-\infty}(3)\\
&\longrightarrow \mathsf{numop}_1(1,+)\mathsf{num}_{-\infty}(6)\\
&\longrightarrow \mathsf{num}_{-\infty}(7)
\end{aligned}$$

We further define lparen and rparen, as well as the compound type $\sigma\mathsf{rparen}$ as follows:

- $\sigma\ \mathsf{rparen}\ s \longrightarrow \sigma\mathsf{rparen}\ \mathsf{snd}\ \big(n \to n, \varepsilon, s\big)$
- $\sigma\mathsf{op}\ \sigma\mathsf{rparen}\ s \longrightarrow \sigma\mathsf{rparen}\ \mathsf{snd}\ \big((f, n), m \to f(n, m), \varepsilon, s\big)$
- $\mathsf{lparen}\ \sigma\mathsf{rparen}\ s \longrightarrow \sigma\ \mathsf{fst}\ \big(n \to n, \varepsilon, s\big)$

Lists: We add lbrack, rbrack, the compound type $\sigma\mathsf{lbrack}$, the compound type $\sigma\mathsf{list}$, period, and index:

- $\mathsf{lbrack}\ \sigma\ s \longrightarrow \sigma\mathsf{lbrack}\ \mathsf{fst}\ \big(u \to (u), \varepsilon, s\big)$ for $\sigma \neq \mathsf{lbrack}, \mathsf{rbrack}$
- $\sigma\mathsf{lbrack}\ \sigma\ s \longrightarrow \sigma\mathsf{lbrack}\ \mathsf{fst}\ \big(\ell, u \to (\ell, \sigma(u)), \varepsilon, s\big)$ for $\sigma \neq \mathsf{lbrack}, \mathsf{rbrack}$
- $\sigma\mathsf{lbrack}\ \mathsf{rbrack}\ s \longrightarrow \sigma\mathsf{list}\ \mathsf{infty}\ \big(\ell \to \ell, \varepsilon, s\big)$
- $\mathsf{period}\ \mathsf{num}\ s \longrightarrow \mathsf{index}\ \mathsf{zero}\ \big(n \to n, \varepsilon, s\big)$
- $\sigma\mathsf{list}\ \mathsf{index}\ s \longrightarrow \sigma\ \mathsf{fst}\ \big((v_0, \ldots, v_n), i \to v_i, \varepsilon, s\big)$

So for example $[1+2; 2;]$ will be converted to (really this is misleading, since the printable tokens aren't converted into abstract tokens before parsing, but rather during it. The following example is just to give some intuition)

$$\begin{aligned}
&\mathsf{lbrack}_0\mathsf{num}_\infty(1)\mathsf{op}_1(+)\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{lbrack}_0\mathsf{numop}_1(1,+)\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{lbrack}_0\mathsf{numop}_1(1,+)\mathsf{num}_{-\infty}(2)\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{lbrack}_0\mathsf{num}_{-\infty}(3)\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{numlbrack}_0(3)\mathsf{num}_\infty(2)\mathsf{end}_{-infty}\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{numlbrack}_0(3)\mathsf{num}_{-\infty}(2)\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{numlbrack}_0(3,2)\mathsf{rbrack}_0\\
&\qquad\longrightarrow \mathsf{numlist}_\infty(3,2)
\end{aligned}$$

And

$$\begin{aligned}
\mathsf{numlist}_\infty(0,1,2,3,4)\mathsf{period}_\infty\mathsf{num}_\infty(2) &\longrightarrow \mathsf{numlist}_\infty(0,1,2,3,4)\mathsf{index}_0(2)\\
&\longrightarrow \mathsf{num}_\infty(2)
\end{aligned}$$

Variables: We add let, leteq, set, seteq, and equal,

- $\mathsf{let}\ x\ s \longrightarrow \mathsf{letvar}\ \mathsf{snd}\ \big(\to (x, \varnothing), \varepsilon, s\big)$

- $\mathsf{letvar}\ \mathsf{index}\ s \longrightarrow \mathsf{letvar}\ \mathsf{fst}\ \big((x,\ell), n \to (x,(\ell,n)), \varepsilon, s\big)$
- $\mathsf{letvar}\ \mathsf{equal}\ s \longrightarrow \mathsf{leteq}\ \mathsf{minfty}\ \big((x,\ell) \to (x,\ell), \varepsilon, s\big)$
- $\mathsf{leteq}\ \sigma\ s \longrightarrow \varepsilon\ \mathsf{fst}\ \big((x,\ell), v \to \varepsilon, \varepsilon, s'\big)$ where s' is $s[x \mapsto \sigma(v)]$ if $\ell = \varnothing$ otherwise if $\ell = (i_1, \dots, i_n)$ then let $\varkappa$ be $s(x)$ where $s(x).i_1 \dots i_n$ is set to be v, then $s' = s[x \mapsto \varkappa]$.
- $\mathsf{set}\ x\ s \longrightarrow \mathsf{setvar}\ \mathsf{snd}\ \big(\to (x,\varnothing), \varepsilon, s\big)$
- $\mathsf{setvar}\ \mathsf{index}\ s \longrightarrow \mathsf{setvar}\ \mathsf{fst}\ \big((x,\ell), n \to (x,(\ell,n)), \varepsilon, s\big)$
- $\mathsf{setvar}\ \mathsf{equal}\ s \longrightarrow \mathsf{seteq}\ \mathsf{minfty}\ \big((x,\ell) \to (x,\ell), \varepsilon, s\big)$
- $\mathsf{seteq}\ \sigma\ s \longrightarrow \varepsilon\ \mathsf{fst}\ \big((x,\ell), v \to \varepsilon, \varepsilon, s'\big)$ where s' is $s[x \mapsto v]$ if ℓ is empty and otherwise if $\ell = (i_1, \dots, i_n)$ then let $\varkappa$ be $s(x)$ where $s(x).i_1.i_2 \dots i_n$ is swapped with v, and s' is $s\{x \mapsto \varkappa\}$.

For example

$$\begin{aligned}
\mathsf{let}_\infty x_\infty \mathsf{equal}_{-\infty} \mathsf{num}_\infty(2)\mathsf{end}_{-infty} &\longrightarrow \mathsf{letvar}_\infty(x)\mathsf{equal}_{-\infty}\mathsf{num}_\infty(2)\mathsf{end}_{-infty} \\
&\longrightarrow \mathsf{leteq}_{-\infty}(x)\mathsf{num}_\infty(2)\mathsf{end}_{-infty} \\
&\longrightarrow \mathsf{leteq}_{-\infty}(x)\mathsf{num}_{-\infty}(2) \\
&\longrightarrow \varepsilon
\end{aligned}$$

and the state will have changed to $s[x \mapsto \mathsf{num}(2)]$. And if s is a state where $x \mapsto \mathsf{numlist}(0,1)$ then

$$\begin{aligned}
\mathsf{let}_\infty x_\infty \mathsf{period}_\infty \mathsf{num}_\infty(1)\mathsf{equal}_{-\infty}\mathsf{num}_\infty(2) &\longrightarrow \mathsf{letvar}_\infty(x)\mathsf{period}_\infty\mathsf{num}_\infty(1)\mathsf{equal}_{-\infty}\mathsf{num}_\infty(2)\mathsf{end}_{-infty} \\
&\longrightarrow \mathsf{letvar}_\infty(x)\mathsf{index}_0(1)\mathsf{equal}_{-\infty}\mathsf{num}_\infty(2)\mathsf{end}_{-infty} \\
&\longrightarrow \mathsf{letvar}_\infty(x,(1))\mathsf{equal}_{-\infty}\mathsf{num}_\infty(2)\mathsf{end}_{-infty} \\
&\longrightarrow \mathsf{leteq}_{-\infty}(x,(1))\mathsf{num}_\infty(2)\mathsf{end}_{-infty} \\
&\longrightarrow \mathsf{leteq}_{-\infty}(x,(1))\mathsf{num}_{-\infty}(2) \\
&\longrightarrow \varepsilon
\end{aligned}$$

and the state will have changed to $s[x \mapsto \mathsf{numlist}(0,2)]$.
Scoping: We add lbrace and rbrace:

- $\mathsf{lbrace}\ \varepsilon\ s \longrightarrow \varepsilon\ \varnothing\ \big(\to \varepsilon, \varepsilon, s + \varnothing\big)$
- $\mathsf{rbrace}\ \varepsilon\ s \longrightarrow \varepsilon\ \varnothing\ \big(\to \varepsilon, \varepsilon, pop\ s\big)$

This highlights another difference between let and set: let creates a new variable in the current scope, and set alters an existing variable in the scope for which it is defined. This is like doing `int x = 0` vs `x = 0` in C.
Products We add the compound type $\Omega\mathsf{product}, \Omega\mathsf{comma}, \Omega\mathsf{rparen}$ where Ω is a list of abstract types.

- $\sigma\ \mathsf{comma}\ s \longrightarrow [\sigma]\mathsf{comma}\ \mathsf{snd}\ (u \to [u], \varepsilon, s)$
- $\sigma\mathsf{op}\ [\sigma]\mathsf{comma}\ s \longrightarrow [\sigma]\mathsf{comma}\ \mathsf{snd}\ ((f,u),[v] \to f(u,v), \varepsilon, s)$
- $\Omega\mathsf{comma}\ [\sigma]\mathsf{comma}\ s \longrightarrow \Omega@[\sigma]\mathsf{comma}\ \mathsf{snd}(\ell, \ell' \to \ell@\ell', \varepsilon, s)$
- $\Omega\mathsf{comma}\ \sigma\mathsf{rparen}\ s \longrightarrow \Omega@[\sigma]\mathsf{rparen}\ \mathsf{snd}(\ell, v \to \ell@[v], \varepsilon, s)$
- $\mathsf{lparen}\ \Omega\mathsf{rparen}\ s \longrightarrow \Omega\mathsf{product}\ \mathsf{infty}\ (\ell \to \ell, \varepsilon, s)$

Functions: First we need a way of capturing the input variable list for a function. To do so, we add "alternative" the alternative abstract type $\mathsf{lparen}^{\mathsf{a}}$ as well as $\mathsf{fun}, \mathsf{fun}^{\mathsf{a}}$ and plist (parameter list).

- $\mathsf{fun}\ x\ s \longrightarrow \mathsf{fun}^{\mathsf{a}}\ \mathsf{fst}\ \big(\to x, \varepsilon, s + [\{\mapsto \mathsf{lbrace}^{\mathsf{a}}, \} \mapsto \mathsf{rbrace}^{\mathsf{a}}, (\mapsto \mathsf{lparen}^{\mathsf{a}},) \mapsto \mathsf{rparen}^{\mathsf{a}}]\big)$
- $\mathsf{lparen}^{\mathsf{a}}\ x\ s \longrightarrow \mathsf{lparen}^{\mathsf{a}}\ \mathsf{fst}\ \big(\ell \to (\ell, x), \varepsilon, s\big)$ for $x \neq)$
- $\mathsf{lparen}^{\mathsf{a}}\ )\ s \longrightarrow \mathsf{plist}\ \mathsf{fst}\ \big(\ell \to \ell, \varepsilon, s\big)$
- $\mathsf{fun}^{\mathsf{a}}\ \mathsf{plist}\ s \longrightarrow \mathsf{fun}^{\mathsf{a}}\ \mathsf{fst}\ \big(x, \ell \to (x,\ell), \varepsilon, s\big)$

Next we need to capture the code for the function. Again we add an alternative type $\mathsf{lbrace}^{\mathsf{a}}$ and code.

- $\mathsf{lbrace}^{\mathsf{a}}\ x\ s \longrightarrow \mathsf{lbrace}^{\mathsf{a}}\ \mathsf{fst}\ \big(\text{“}\xi\text{”} \to \text{“}\xi x\text{”}, \varepsilon, s\big)$ for $x \neq \{, \}$
- $\mathsf{lbrace}^{\mathsf{a}}\ \}\ s \longrightarrow \mathsf{code}\ \mathsf{fst}\ \big(\text{“}\xi\text{”} \to \text{“}\{\xi\}\text{”}, \varepsilon, s\big)$
- $\mathsf{lbrace}^{\mathsf{a}}\ \mathsf{code}\ s \longrightarrow \mathsf{lbrace}^{\mathsf{a}}\ \mathsf{fst}\ \big(\text{“}\xi\text{”}, \text{“}\xi'\text{”} \to \text{“}\xi\xi'\text{”}, \varepsilon, s\big)$

Now we convert $\mathsf{fun}^{\mathsf{a}}$ and code to a $\mathsf{closure}$

- $\mathsf{fun}^{\mathsf{a}}\ \mathsf{code}\ s \longrightarrow \mathsf{closure}\ \mathsf{fst}\ \big((x,\ell), \text{“}\xi\text{”} \to C = \langle \ell, \text{“}\xi\text{”}, pop\ s[x \mapsto \mathsf{closure}(C)]\rangle, pop\ s[x \mapsto \mathsf{closure}(C)]\big)$

And now we add rules for calling a function

- $\mathsf{closure}\ \mathsf{list}\ s \longrightarrow \varepsilon\ \varnothing\ \big(\langle \ell, \text{“}\{\xi\}\text{”}, s\rangle, \ell' \to \varepsilon, \xi\}, s + [\ell \mapsto \ell']\big)$