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, which we call L-Lang.

Table of Contents

0 Notation

(1) $\mathbb{N}$ denotes the set of natural numbers, including 0.

(2) $\overline{\mathbb{N}}$ is defined to be $\mathbb{N} \cup \{\infty\}$.

(3) $\mathbb{Z}$ denotes the set of integers.

(4) $\overline{\mathbb{Z}}$ is defined to be $\mathbb{Z} \cup \{\pm\infty\}$.

(5) $f\colon A \rightharpoonup B$ means that f is a partial function from A to B.

(6) If X is a set and x is some symbol, then $\mathsf{X}_{\mathsf{x}} = \mathsf{X}^{\mathsf{x}} = \mathsf{X} \cup \{\mathsf{x}\}$.

(7) ε is the empty string, it and $\varnothing$ are also used to denote "nothing" in whatever context that may be.

(8) $(x_1, \ldots, x_n)$ denotes a list.

(9) If ℓ_1, ℓ_2 are lists, $\ell_1 @ \ell_2$ is their concatenation.

(10) $t{::}\ell$ is the list whose first element is t and whose tail is ℓ.

1 The Algorithm

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. $\Diamond$

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 $2 \cdot ((1+2) \cdot 2) + 1$,

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

$\Diamond$

1.2 Stateful Reduction

We define the following four base sets:

(1) $\mathcal{U}$ the universe of *values*, these are all the internal values an object may have.

(2) $\mathcal{T}_\mathcal{P}$ the set of *printable terms*, these are the tokens which a programmer may pass to the reducer.

(3) $\mathcal{T}_\Sigma$ the set of *type terms.*

(4) $\mathcal{T}_\mathcal{A}$ the set of *abstract terms.*

The sets $\mathcal{T}_\mathcal{P}, \mathcal{T}_\Sigma, \mathcal{T}_\mathcal{A}$ are all disjoint, we place no such restriction on $\mathcal{U}$ as the purpose it serves is different. Let $\mathcal{A}$ be a set of *atomic abstract terms*, then the construction of abstract terms is

$$\mathcal{T}_\mathcal{A} ::= \mathcal{A} \mid \mathcal{A}\mathcal{T}_\Sigma$$

And let Σ be a set of *atomic types*, each with an associated arity, which may be ∞. Let Σ^n be the set of atomic types of arity n, then the construction of type terms is

$$\mathcal{T}_\Sigma ::= \Sigma^0 \mid \Sigma^n \mathcal{T}_\Sigma^1 \cdots \mathcal{T}_\Sigma^n \mid \Sigma^\infty \mathcal{T}_\Sigma^1 \cdots \mathcal{T}_\Sigma^n$$

as n ranges over all $\mathbb{N}_{>0}$.

Define

(1) $\mathcal{T} := \mathcal{T}_\mathcal{P} \cup \mathcal{T}_\Sigma \cup \mathcal{T}_\mathcal{A}$ the set of *basic terms.*

(2) $\mathcal{T}_\mathcal{I} := \mathcal{T}_\Sigma \cup \mathcal{T}_\mathcal{A}$ the set of *internal terms.*

(3) $\Pi_\mathcal{I} := \mathcal{T}_\mathcal{I} \times \mathcal{U}$ the set of *termed values.*

(4) $\Pi := \Pi_\mathcal{I} \cup \mathcal{T}_\mathcal{P}$ the set of *atomic expressions.*

Elements of $\overline{\Pi}$ will be written like $\sigma_n(v)$ where σ is the term, n the priority, and v the value (nothing for printable terms).

We define the *initial priority function* as a function $\pi\colon \mathcal{T}_\mathcal{P} \longrightarrow \overline{\mathbb{Z}}$. This can be extended canonically to a function $\pi\colon \mathcal{T}_\mathcal{P}^* \longrightarrow \overline{\Pi}^*$.

In stateful 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

$$\widehat{\beta}\colon \mathcal{T}_\mathcal{I} \times \mathcal{T}^\varepsilon \rightharpoonup \mathcal{T}_\mathcal{I}^\varepsilon \times (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}} \to \overline{\mathbb{Z}}) \times (\mathcal{U} \times \mathcal{U} \times \text{State} \rightharpoonup \mathcal{U} \times \mathcal{T}_\mathcal{P}^* \times \text{State})$$

We extend this to a derived β-reducer,

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

We also define β^* where given an input $\langle \xi \mid s\rangle$, it runs β on it iteratively until convergence (of ξ). β is defined with the following rules: given an input $\langle \xi \mid s\rangle$ its image is

(1) If $\xi = \sigma_n \xi'$ for $\sigma \in \mathcal{T}_\mathcal{P}$ then

$$\beta\langle \xi \mid s\rangle = \langle s(\sigma)_n \xi' \mid s\rangle.$$

(2) If $\xi = \sigma_i(v)\xi'$ and $\widehat{\beta}(\sigma, \varepsilon) = (\alpha, \rho, f)$ is defined, then if $f(v, \_, s) = (w, \zeta, s')$ and $\rho(i) = k$ and $\beta^*\langle \pi\zeta \mid s'\rangle = \langle \zeta' \mid s''\rangle$ then

$$\beta\langle \xi \mid s\rangle = \langle \alpha_k(w)\zeta'\xi' \mid s''\rangle.$$

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

$$\beta\langle \xi \mid s\rangle = \langle \alpha_k(w)\zeta'\xi' \mid s''\rangle.$$

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

$$\beta\langle \xi \mid s\rangle = \langle \sigma_i(v)\xi'' \mid s'\rangle.$$

1.2.1 States

Similar to before, we define point-states as partial maps $\mathcal{T}_\mathcal{P} \rightharpoonup \Pi_\mathcal{I}$. And if s_1, s_2 are two point-states and $\sigma \in \mathcal{T}_\mathcal{P}$ then

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

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 $\varkappa_i$.

A state will now have two fields: a sequence of point-states, as well as a sequence of indexes. For a state $\bar{s} = \big[(s_1, \ldots, s_n), I = (i_1, \ldots, i_k)\big]$, let us define

(1) $\bar{s} + s = \big[(s_1, \ldots, s_n, s), I\big]$

(2) $\bar{s} +_c s = \big[(s_1, \ldots, s_n, s), (i_1, \ldots, i_k, n+1)\big]$

(3) *pop* $\bar{s} = \big[(s_1, \ldots, s_{n-1}), I\big]$ if $i_k < n$ otherwise, $\big[(s_1, \ldots, s_{n-1}), (i_1, \ldots, i_{k-1})\big]$

(4) $\bar{s}s = \big[(s_1, \ldots, s_n s), I\big]$

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

(6) $\bar{s}_c = s_{i_k} \cdots s_n$

Furthermore, if $\sigma \in \mathcal{T}_\mathcal{P}$ and $\varkappa \in \Pi_\mathcal{I}$ 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 $\sigma \in \text{dom} s_i$.

2 The Grammar

In this section we discuss the grammar of the language of L-Lang. This is not naturally imposed by the parser, but it will properly parse programs of this form.

Identifiers

$$\begin{aligned}
\mathit{str} &::= (a \dots z \mid A \dots Z \mid _) \\
\mathit{digit} &::= (0 \dots 9) \\
\mathit{ident} &::= \mathit{str}\ (\mathit{str} \mid \mathit{digit})^*
\end{aligned}$$

Constant Expressions

$$\begin{aligned}
\mathit{const} &::= (\mathit{number} \mid \mathit{product} \mid \mathit{list}) \\
\mathit{number} &::= (\mathit{digit})^*[.(\mathit{digit})^*] \\
\mathit{product} &:= (\mathit{production}\ (, \mathit{production})^*) \\
\mathit{production} &:= (\mathit{expr} \mid \mathit{product})
\end{aligned}$$

Expressions

$$\begin{aligned}
\mathit{op} &::= + \mid * \mid / \\
\mathit{pop} &::= - \\
\mathit{expr} &::= \ \mathit{ident} \\
&\mid \mathit{const} \\
&\mid \mathit{expr}; \\
&\mid \mathit{expr}\ \mathit{expr} \\
&\mid \mathit{primexpr} \\
&\mid (\mathit{expr}) \\
&\mid \mathit{expr}\ (\mathit{op} \mid \mathit{pop})\ \mathit{expr} \\
&\mid \mathit{pop}\ \mathit{expr} \\
&\mid \mathit{expr}.\mathit{expr} \\
&\mid \textbf{if}\ (\mathit{expr})\ \{\mathit{expr}\}\{\mathit{expr}\} \\
&\mid \textbf{fun}\ \mathit{ident}\ (\mathit{pattern})\ \{\mathit{expr}\} \\
\mathit{primexpr} &::= _\textbf{prim}_\mathit{ident}
\end{aligned}$$

3 Initializing the Algorithm

3.1 The Initial Beta Reducer

We now describe the initial beta reducer of L-Lang according to stateful reduction. By convention, atomic abstract terms will be red, type terms will be green, internal terms will be blue.

End:

- σ end $\longrightarrow$ σ minfty $(u, \_, s \to u, \varepsilon, s)$

Arithmetic:

- σ op $\longrightarrow$ opσ snd $(u, f, s \to (u, f), \varepsilon, s)$
- opσ opσ $\longrightarrow$ opσ snd $\big((u, f), (v, g), s \to (f(u, v), g), \varepsilon, s\big)$
- opσ σ $\longrightarrow$ σ snd $\big((u, f), v, s \to f(u, v), \varepsilon, s\big)$
- pop σ $\longrightarrow$ σ snd $\big((f, g), u, s \to f(u), \varepsilon, s\big)$
- σ pop $\longrightarrow$ opσ one $\big(u, (f, g), s \to (u, g), \varepsilon, s\big)$
- σ rparen $\longrightarrow$ rparenσ snd $(u, \_, s \to u, \varepsilon, s)$
- opσ rparenσ $\longrightarrow$ rparenσ snd $\big((f, u), v, s \to f(u, v), \varepsilon, s\big)$
- pop rparenσ $\longrightarrow$ rparenσ snd $\big((f, g), u, s \to f(u), \varepsilon, s\big)$
- lparen rparenσ $\longrightarrow$ σ fst $(\_, u, s \to u, \varepsilon, s)$

Lists:

- lbrack σ $\longrightarrow$ lbrackσ fst $(\_, u, s \to (u), \varepsilon, s)$
- lbrackσ σ $\longrightarrow$ lbrackσ fst $(\ell, u, s \to (\ell, u), \varepsilon, s)$
- lbrackσ rbrack $\longrightarrow$ listσ infty $(\ell, \_, s \to \ell, \varepsilon, s)$
- period num $\longrightarrow$ index zero $(\_, n, s \to n, \varepsilon, s)$
- listσ index $\longrightarrow$ σ fst $(\ell, i, s \to \ell_i, \varepsilon, s)$

Variables:

- let x $\longrightarrow$ letvar snd $(\_, \_, s \to (x, \varnothing), \varepsilon, s)$
- letvar index $\longrightarrow$ letvar fst $\big((x, \ell), n, s \to (x, (\ell, n)), \varepsilon, s\big)$
- letvar equal $\longrightarrow$ leteq minfty $\big((x, \ell), \_, s \to (x, \ell), \varepsilon, s\big)$
- leteq σ $\longrightarrow$ ε $\varnothing$ $\big((x, \ell), v, s \to \varepsilon, \varepsilon, s'\big)$ where s' is $s[x \mapsto \sigma(v)]$ if $\ell = \varnothing$ and otherwise let t be the result of setting $s(x).\ell_1.\dots.\ell_n$ to v, then $s' = s[x \mapsto t]$.

Scoping:

- lbrace ε $\longrightarrow$ ε $\varnothing$ $(\_, \_, s \to \varepsilon, \varepsilon, s + \varnothing)$
- rbrace ε $\longrightarrow$ ε $\varnothing$ $(\_, \_, s \to \varepsilon, \varepsilon, pop\ s)$

Products:

- σ comma $\longrightarrow$ comma(σ) snd $(u, \_, s \to (u), \varepsilon, s)$
- opσ comma(σ) $\longrightarrow$ comma(σ) snd $((f, u), (v) \to (f(u, v)), \varepsilon, s)$
- pop comma(σ) $\longrightarrow$ comma(σ) snd $((f, g), (u) \to (f(u), \varepsilon, s))$
- commaΩ comma(σ) $\longrightarrow$ comma(Ω, σ) snd $(\ell, \ell', s \to (\ell, \ell'), \varepsilon, s)$
- commaΩ rparenσ $\longrightarrow$ listrparen(Ω, σ) snd $(\ell, v \to (\ell, v), \varepsilon, s)$
- lparen listrparenΩ $\longrightarrow$ productΩ infty $(\_, \ell, s \to \ell, \varepsilon, s)$

Primitives:

- primitive σ $\longrightarrow$ ε $\varnothing$ $(f, v, s \to \varepsilon, w, s)$ where $f(\sigma, v) = (w, s')$ (the purpose is for f to have a side effect)

Code Capture

- $\mathsf{lbrace}^a\ x \longrightarrow \mathsf{lbrace}^a\ \mathsf{infty}\ (\xi, \_, s \to \xi x, \varepsilon, s)$ if $x \neq \{,\}$
- $\mathsf{lbrace}^a\ x \longrightarrow \mathsf{code}\ \mathsf{infty}\ (\xi, \_, s \to \xi, \varepsilon, s)$
- $\mathsf{lbrace}^a\ \mathsf{code} \longrightarrow \mathsf{lbrace}^a\ \mathsf{infty}\ (\xi, \xi', s \to \xi\{\xi'\}, \varepsilon, s)$

Parameter Capture

- $\mathsf{lparen}^a\ x \longrightarrow \mathsf{lparen}^a\ \mathsf{fst}\ (\ell, \_, s \to \ell@(x), \varepsilon, s)$ for $x \neq (,)$
- $\mathsf{lparen}^a\ ) \longrightarrow \mathsf{plist}\ \mathsf{fst}\ (\ell, \_, s \to \ell, \varepsilon, s)$
- $\mathsf{lparen}^a\ \mathsf{plist} \longrightarrow \mathsf{lparen}^a\ \mathsf{fst}\ (\ell, \ell', s \to (\ell@(\ell')), \varepsilon, s)$

Function Definitions

- $\mathsf{fun}\ x \longrightarrow \mathsf{funname}\ \mathsf{infty}\ (\_, \_, s \to (x, \varepsilon), \varepsilon, s + [\{\mapsto \mathsf{lbrace}^a, \} \mapsto \mathsf{rbrace}^a, (\mapsto \mathsf{lparen}^a,) \mapsto \mathsf{rparen}^a])$
- $\mathsf{funname}\ \mathsf{plist} \longrightarrow \mathsf{funvars}\ \mathsf{infty}\ \big((x, \varepsilon), u, s \to (x, u), \varepsilon, s\big)$
- $\mathsf{funvars}\ \mathsf{code} \longrightarrow \mathsf{closure}\ \mathsf{fst}\ \big((x, \ell), \xi, s \to C = \langle \ell, \xi, s'[x \mapsto \mathsf{closure}(C)]\rangle, \varepsilon, pop\ s[x \mapsto \mathsf{closure}(C)]\big)$ where $s' = (pop\ s)_c$.

Function Calls

- $\mathsf{closure}\ \sigma \longrightarrow \varepsilon\ \varnothing\ \big(\langle \ell, \xi, ps\rangle, u, s \mapsto \varepsilon, \xi\}, s +_c ps[\ell \mapsto \sigma(u)]\big)$ where $\ell \mapsto \sigma(u)$ means that if $\ell = (x)$ then $x \mapsto \sigma(u)$. Otherwise $\ell = (x_1, \ldots, x_n)$, $\sigma = \mathsf{product}\sigma_1 \cdots \sigma_n$, and $u = (u_1, \ldots, u_n)$ and $x_i \mapsto \sigma_i(u_i)$ (recursively).

If Statements

- $\mathsf{if}\ \sigma \longrightarrow \mathsf{ifbool}\ \mathsf{fst}\ (\_, n, s \to n, \varepsilon, s + [\{\mapsto \mathsf{lbrace}^a, (\mapsto \mathsf{lparen}^a)]$
- $\mathsf{ifbool}\ \mathsf{code} \longrightarrow \mathsf{ifthen}\ \mathsf{fst}\ (n, \xi, s \to (n, \xi), \varepsilon, s)$
- $\mathsf{ifthen}\ \mathsf{code} \longrightarrow \varepsilon\ \_\ ((n, \xi_1), \xi_2, s \to \varnothing, (n = 0?\ \xi_2 : \xi_1), pop\ s)$

Types

- $\mathsf{type}\sigma\ \mathsf{type}\tau \longrightarrow \mathsf{type}\sigma(\tau)\ \mathsf{snd}\ (\_, \_, s \to \sigma(\tau), \varepsilon, s)$
- $\mathsf{type}\sigma\ \mathsf{product}(\mathsf{type}\tau_1, \ldots, \mathsf{type}\tau_n) \longrightarrow \mathsf{type}\sigma(\tau_1, \ldots, \tau_n)\ \mathsf{snd}\ (\_, \_, s \to \sigma(\tau_1, \ldots, \tau_n), \varepsilon, s)$
- $\mathsf{colon}\ \mathsf{type}\sigma \longrightarrow \mathsf{typer}\sigma\ \mathsf{snd}\ (\_, u, s \to u, \varepsilon, s)$
- $\sigma\ \mathsf{typer}\tau \longrightarrow \tau\ \mathsf{snd}\ (u, \_, s \to u, \varepsilon, s)$

3.2 The Initial State

The initial state is a partial state, defined as follows:

End

- $; \mapsto (\mathsf{end}, \varnothing)$

Arithmetic

- $(\mapsto (\mathsf{lparen}, \varnothing)$
- $) \mapsto (\mathsf{rparen}, \varnothing)$
- $+ \mapsto (\mathsf{op}, (n, m \to n + m))$
- $+ \mapsto (\mathsf{op}, (n, m \to n + m))$
- $* \mapsto (\mathsf{op}, (n, m \to n * m))$
- $/ \mapsto (\mathsf{op}, (n, m \to n/m))$
- $- \mapsto (\mathsf{pop}, (n \to -n), (n, m \to n - m))$
- $@ \mapsto (\mathsf{op}, (\ell_1, \ell_2 \to \ell_1 @ \ell_2))$
- $!= \ \mapsto (\mathsf{op}, (u, v \to u \neq v))$
- $<= \ \mapsto (\mathsf{op}, (n, m \to n \leq m))$
- $>= \ \mapsto (\mathsf{op}, (n, m \to n \geq m))$
- $== \ \mapsto (\mathsf{op}, (u, v \to u = v))$
- $< \ \mapsto (\mathsf{op}, (n, m \to n < m))$
- $> \ \mapsto (\mathsf{op}, (n, m \to n > m))$

Lists

- $[\mapsto (\mathsf{lbrack}, [])$
- $] \mapsto (\mathsf{rbrack}, \varnothing)$
- $. \mapsto (\mathsf{period}, \varnothing)$

Variables

- $\text{let} \mapsto (\mathsf{let}, \varnothing)$
- $= \ \mapsto (\mathsf{equal}, \varnothing)$

Scoping

- $\{ \mapsto (\mathsf{lbrace}, \varnothing)$
- $\} \mapsto (\mathsf{rbrace}, \varnothing)$

Products

- $, \mapsto (\mathsf{comma}, \varnothing)$

Primitives

- $\_\mathrm{prim\_print} \mapsto$ $(\textsf{primitive}, (a, v \to \mathrm{print}(v); (\varnothing, \varnothing)))$
- $\_\mathrm{prim\_len} \mapsto (\textsf{primitive}, (a, \ell \to \textsf{num}, |\ell|))$
- $\_\mathrm{prim\_tail} \mapsto (\textsf{primitive}, (\sigma, t :: \ell \to \sigma, \ell))$
- $\_\mathrm{prim\_type} \mapsto (\textsf{primitive}, (\sigma, \_ \to \textsf{type}\sigma, \sigma))$

Keywords

- $\mathrm{fun} \mapsto (\textsf{fun}, \varnothing)$
- $\mathrm{if} \mapsto (\textsf{if}, \varnothing)$

Types

- $: \ \mapsto (:, \varnothing)$
- $\mathrm{Num} \mapsto (\textsf{type num}, \textsf{num})$
- $\mathrm{List} \mapsto (\textsf{type list}, \textsf{list})$
- $\mathrm{Closure} \mapsto (\textsf{type closure}, \textsf{closure})$
- $\mathrm{Product} \mapsto (\textsf{type product}, \textsf{product})$
- $\mathrm{Primitive} \mapsto (\textsf{type primitive}, \textsf{primitive})$
- $\mathrm{Type} \mapsto (\textsf{type type}, \textsf{type})$

3.3 The Initial Priorities

The initial priority function, π, is defined as follows:

End

- $; \ \mapsto -\infty$

Arithmetic

- $(\ \mapsto \infty$
- $) \mapsto 0$
- $+ \mapsto 1$
- $* \mapsto 2$
- $- \mapsto 1$
- $/ \mapsto 2$
- $@ \mapsto 1$
- $== \ \mapsto 0$
- $!= \ \mapsto 0$
- $<= \ \mapsto 0$
- $>= \ \mapsto 0$
- $< \ \mapsto 0$
- $> \ \mapsto 0$

Lists

- $[\mapsto 0$
- $] \mapsto 0$
- $. \mapsto 0$

Variables

- $= \ \mapsto -\infty$

Scoping

- $\{ \mapsto 0$
- $\} \mapsto 0$

Products

- $, \ \mapsto 0$

Everything else is mapped to ∞

4 Proving Equivalence

In this section we will prove that our algorithm interprets according to the proper order of operations. Specifically we will define a natural method of parsing and interpreting a numerical expression, and show that given a valid expression our algorithm gives the same result.

The problem can be formulated as follows: we are given the following:

(1) A set X, which is our *universe*.

(2) A set S of *operator symbols*.

(3) For every operator symbol $\mathsf{s} \in \mathsf{S}$, a function $f_{\mathsf{s}} \colon \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{X}$.

(4) For every operator symbol $\mathsf{s} \in \mathsf{S}$, a *priority* $\pi(\mathsf{s}) \in \mathbb{N}$.

We define the following grammar of *expressions*:

$$\mathit{expr} ::= \mathsf{L}\ \mathit{expr}\ \mathsf{R} \mid \mathit{expr}\ \mathsf{S}\ \mathit{expr} \mid \mathsf{X}$$

All operators are left-associative. L, R represent left and right parentheses respectively. We define $\textsc{unparen}(\xi, i)$ to mean that the ith character in ξ is not within parentheses (this can be implemented easily: iterate backwards and count how many opening and closing parentheses there are. If there are more opening than closing, than it is inside parentheses.) We now define the evaluator for expressions:

function $\textsc{eval}(\xi)$
    if $(\xi = \mathsf{x} \in \mathsf{X})$
        return x
    else if $(\xi[0] = \mathsf{L})$
        ξ is of the form $\mathsf{L}\rho\mathsf{R}\xi'$
        return $\textsc{eval}(\textsc{eval}(\rho)\xi')$
    else
        $\triangleright$ *Find the greatest index with the smallest operator.*
        Note that $\max_i \operatorname{argmin}_i f(i)$ *is the maximal i which minimizes f.*
        $i := \max_i \operatorname{argmin}_i \{\pi(\xi[i]) \mid \xi[i] \in \mathsf{S}, \textsc{unparen}(\xi, i)\}$
        $\mathsf{s} := \xi[i]$
        $\triangleright\ \xi' = \xi[: i-1],\ \xi'' = \xi[i+1 :]$
        ξ is of the form $\xi'\mathsf{s}\xi''$
        return $f_{\mathsf{s}}(\textsc{eval}(\xi'), \textsc{eval}(\xi''))$
    end if
end function

For our beta reducer, our type term will be X, the atomic abstract terms will be $\mathsf{S}, \mathsf{L}, \mathsf{R}$ (L for left parentheses, R for right), and printable terms will be elements of $\mathsf{X} \cup \mathsf{S}$. We will define the following initial beta reducer, we ignore states and printable term outputs because they are held constant:

- $\mathsf{X}\ \mathsf{S} \longrightarrow \mathsf{SX}\ \mathsf{snd}\ (u, f \to (u, f))$
- $\mathsf{SX}\ \mathsf{SX} \longrightarrow \mathsf{SX}\ \mathsf{snd}\ ((u, f), (v, g) \to (f(u, v), g))$
- $\mathsf{SX}\ \mathsf{X} \longrightarrow \mathsf{X}\ \mathsf{snd}\ ((u, f), v \to f(u, v))$
- $\mathsf{X}\ \mathsf{R} \longrightarrow \mathsf{RX}\ \mathsf{snd}\ (u, \_ \to u)$
- $\mathsf{SX}\ \mathsf{RX} \longrightarrow \mathsf{RX}\ \mathsf{snd}\ ((u, f), v \to f(u, v))$
- $\mathsf{L}\ \mathsf{RX} \longrightarrow \mathsf{X}\ \mathsf{fst}\ (\_, u \to u)$

The initial state is self-explanatory. The initial priorities map $\mathsf{x} \in \mathsf{X}$ to ∞, $\mathsf{s} \in \mathsf{S}$ to $\pi(\mathsf{s})$, (to ∞, and) to 0. Since the state never changes, we can assume that all tokens are given their value in the initial state from the outset.

Let β^* be the total β-reducer: it iteratively applies β until convergence We also define β_0 and β_0^* which include the rule that if ξ is a single character $\sigma_n(u)$ then $\beta_0\xi = \sigma_0(u)$. We write $\xi \to^* \xi'$ to mean that successive applications of β_0 lead from ξ to ξ'.

4.1 Expressions without Parentheses

First we will prove this result for expressions without parentheses, i.e. those of the grammar

$$\mathit{expr} ::= \mathsf{X}_\infty \mid \mathsf{X}_\infty\ \mathsf{S}_n\ \mathit{expr}$$

Let us define the following set of *reduced expressions*, as a grammar including priorities:

$$rexpr ::= \mathsf{X}_n \mid \mathsf{SX}_n\ rexpr \mid \mathsf{X}_n\ \mathsf{S}_m\ rexpr\ (n \geq m)$$

Note that $expr \subseteq rexpr$. Say that a string of internal terms S is *closed under β-reductions* if for every string $\xi \in S$ with arbitrary priorities, $\beta\langle\xi\rangle \in S$.

Lemma 4.1.1: *rexpr is closed under β-reductions. And under β_0-reductions, every reduced expression converges to an element in* X.

Proof: We induct on the length of the string ξ. We split into cases

(1) A string of the form X_n retains its value under a β-reduction.

(2) For string of the form $\mathsf{SX}_n\xi$, we have the following cases:

- **(i)** $\xi = \mathsf{X}_m$, if $n \geq m$ this becomes $\mathsf{X}_m \in rexpr$. Otherwise, a β-reduction gives $\mathsf{SX}_n\beta\langle\mathsf{X}_m\rangle = \mathsf{SX}_n\mathsf{X}_m \in rexpr$.
- **(ii)** $\xi = \mathsf{SX}_m\xi'$, if $n \geq m$ then this becomes $\mathsf{SX}_m\xi' \in rexpr$. Otherwise, a β-reduction gives $\mathsf{SX}_n\beta\langle\xi\rangle$, and $\beta\langle\xi\rangle \in rexpr$ inductively.
- **(iii)** $\xi = \mathsf{X}_m\ \mathsf{S}_k\xi'$, if $n \geq m$ then this becomes $\mathsf{X}_m\ \mathsf{S}_k\xi' \in rexpr$. Otherwise, a β-reduction gives $\mathsf{SX}_n\beta\langle\xi\rangle \in rexpr$ inductively.

(3) For a string of the form $\mathsf{X}_n\mathsf{S}_m\xi$, if $n \geq m$ then a β-reduction gives $\mathsf{SX}_m\xi \in rexpr$. Otherwise we get $\mathsf{X}_n\beta\langle\mathsf{S}_m\xi\rangle$, but there are no reduction rules in which S is the first term, so this is just $\mathsf{X}_n\mathsf{S}_m\langle\xi\rangle \in rexpr$ inductively.

For the second claim, we inductively show that if $\xi \neq \mathsf{X}$ has a final priority of 0 then a β_0 reduction will decrease its length by 1, and if its final priority is > 0 then it either decreases its length by 1 or sets its final priority to 0.

(1) If $\xi = \mathsf{SX}_n\xi'$, then a β_0-reduction either compounds SX_n with another token, or gives $\mathsf{SX}_n\beta_0\langle\xi'\rangle$, which inductively will decrease ξ''s length by 1 if ξ''s final priority is zero, or will decrease it by a token or set its final priority to zero otherwise.

(2) If $\xi = \mathsf{X}_n\ \mathsf{S}_m\xi'$, since $n \geq m$ then this reduces to $\mathsf{SX}_m\xi'$, which is one token less.

So this means that every two β_0-reductions (in fact every single β_0-reduction, since we can never increase a priority: once a token's priority is 0, it will never increase) will decrease the length of the string. This means that eventually the string will converge to a single token. Since *rexpr* is closed under β_0-reductions, this string must be in X. ■

Since $expr \subseteq rexpr$, when β-reducing an expression, we will remain inside *rexpr*.

Notice that if $\xi \in rexpr$ then $\beta^*\langle\xi\rangle$ cannot contain a substring of the form $\mathsf{X}_n\ \mathsf{S}_m$ since this can be reduced. And since by the above lemma, $\beta^*\langle\xi\rangle \in rexpr$, we have $\beta^*\langle\xi\rangle = \mathsf{SX}_{n_1}\mathsf{SX}_{n_2}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$ with $n_1 < \cdots < n_k < n_{k+1}$.

Instead of writing

- $\mathsf{X}_n(\mathsf{x})$, we write x_n.
- $\mathsf{S}_n(f_\mathsf{s})$, we write s_n.
- $\mathsf{SX}_n(\mathsf{x}, f_\mathsf{s})$, we write $\mathsf{s}_n[\mathsf{x}]$.

We begin with the case that ξ does not include any parentheses. Let us define $\pi(\xi) = \{\pi(\xi[i])\}_{0\leq i<|\xi|}$.

Lemma 4.1.2: *Let $\xi \in rexpr$ such that $\pi(\xi) \geq n$, then $\beta^*\langle\xi\mathsf{s}_n\rangle = \mathsf{s}_n[\beta_0^*\langle\xi\rangle]$.*

Proof: we know that $\xi\mathsf{s}_n$ is reduced to, at some point, $\beta^*\langle\xi\rangle\mathsf{s}_n$, and as noted this is equal to

$$\mathsf{s}^1_{n_1}[\mathsf{x}^1]\cdots\mathsf{s}^k_{n_k}[\mathsf{x}^k]\mathsf{x}^{k+1}_{n_{k+1}}\mathsf{s}_n, \qquad n \leq n_1 < \cdots < n_k < n_{k+1}$$

Then a β-reduction gives

$$\mathsf{s}^1_{n_1}[\mathsf{x}^1]\cdots\mathsf{s}^k_{n_k}[\mathsf{x}^k]\mathsf{s}_n[\mathsf{x}^{k+1}]$$

and further β-reductions give (instead f_{s_i} we'll write f_i):

$$\mathsf{s}^1_{n_1}[\mathsf{x}^1]\cdots\mathsf{s}_n[f_k(\mathsf{x}^k,\mathsf{x}^{k+1})] \to \cdots \to \mathsf{s}_n\big[f_1(\mathsf{x}^1, f_2(\mathsf{x}^2,\cdots))\big]$$

And we see that β_0-reducing ξ gives precisely $f_1(\mathsf{x}^1, f_2(\mathsf{x}^2, \cdots))$, as required. ■

Lemma 4.1.3: *If $\xi \in$ rexpr and $n < \pi(\xi)$ then $\beta_0^*\langle \mathsf{s}_n[\mathsf{x}]\xi\rangle = f_{\mathsf{s}}(\mathsf{x}, \beta_0^*\langle\xi\rangle)$.*

Proof: since $n < \pi(\xi)$, $\mathsf{s}_n[\mathsf{x}]$ cannot reduce with the first element of ξ, so $\beta_0\langle \mathsf{s}_n[\mathsf{x}]\xi\rangle = \mathsf{s}_n[\mathsf{x}]\beta_0\langle\xi\rangle$. If ξ can be β-reduced, then we simply induct on the new string, which is one token less. Otherwise $\xi = \mathsf{s}_{n_1}^1[\mathsf{x}^1]\cdots\mathsf{s}_{n_k}^k[\mathsf{x}^k]\mathsf{x}_{n_{k+1}}^{k+1}$ with $n_1 < \cdots < n_{k+1}$, as remarked before. Then we continue with a similar process as before. ■

Let $\xi \in$ *expr*, define sym(ξ) be the operator symbols in ξ. By *inducting on the levels of priority*: we mean induction on $\pi\text{sym}(\xi)$.

Theorem 4.1.4: *Let $\xi \in$ expr, then $\beta_0^*\langle\xi\rangle = \text{EVAL}(\xi)$.*

Proof: we induct on the levels of priority in ξ. If $\xi = \mathsf{x}_n$ then this is trivial. Otherwise let n be the lowest priority of operator in ξ, so ξ is of the form

$$\xi = \xi_1\mathsf{s}_n^1\xi_2\mathsf{s}_n^2\cdots\xi_k\mathsf{s}_n^k\xi_{k+1}$$

Now notice that

$$\xi \to^* \beta^*\big\langle\xi_1\mathsf{s}_n^1\xi_2\cdots\xi_k\mathsf{s}_n^k\big\rangle\xi_{k+1}$$

by lemma 4.0.2 this is equal to

$$\xi \to^* \mathsf{s}_n^k[\beta_0^*\big\langle\xi_1\mathsf{s}_n^1\cdots\mathsf{s}_n^{k-1}\xi_k\big\rangle]\xi_{k+1}$$

and by lemma 4.0.3 reducing this gives

$$f_k\big(\beta_0^*\big\langle\xi_1\mathsf{s}_n^1\cdots\mathsf{s}_n^k\xi_k\big\rangle, \beta_0^*\langle\xi_{k+1}\rangle\big)$$

Continuing inductively on k, we get

$$f_k\big(\cdots f_2(f_1(\beta_0^*\langle\xi_1\rangle, \beta_0^*\langle\xi_2\rangle), \beta_0^*\langle\xi_3\rangle)\cdots, \beta_0^*\langle\xi_{k+1}\rangle\big)$$

By induction, this is just

$$f_k\big(\cdots f_2(f_1(\text{EVAL}(\xi_1), \text{EVAL}(\xi_2)), \text{EVAL}(\xi_3))\cdots, \text{EVAL}(\xi_{k+1})\big) \tag{1}$$

Now notice that

$$\text{EVAL}(\xi) = f_k(\text{EVAL}(\xi_1\mathsf{s}_n^1\cdots\mathsf{s}_n^{k-1}\xi_k), \text{EVAL}(\xi_{k+1}))$$

And continuing we get that this is equal to (1), as required. ■

4.2 Expressions with Parentheses

Our expressions will now be of the grammar

$$expr_P ::= \mathsf{X}_\infty \mid \mathsf{X}_\infty\ \mathsf{S}_n\ expr_P \mid \mathsf{L}_\infty\ expr_P\ \mathsf{R}_0$$

and *reduced expressions* will be

$$rexpr_P ::= \mathsf{X}_n \mid \mathsf{SX}_n\ rexpr_P \mid \mathsf{X}_n\ \mathsf{S}_m\ rexpr_P\ (n \geq m) \mid \mathsf{L}_\infty\ rexpr_P\ \mathsf{R}_0 \mid \mathsf{L}_\infty\ (\mathsf{SX}_n)^*\ \mathsf{RX}_0$$

Lemma 4.2.5: *$rexpr_P$ is closed under β-reductions.*

Proof: we will prove the following stronger result:

Let $\xi \in rexpr_P$ then $\beta\langle\xi\rangle \in rexpr_P$ and $\beta^*\langle\xi\rangle = \mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$ with $n_1 < \cdots < n_{k+1}$.

(1) For $\xi = \mathsf{X}_n$ this is trivial.

(2) For $\xi = \mathsf{SX}_n\xi'$, if SX_n can reduce with the first token of ξ', so $\xi' = \mathsf{X}_m$ or $\xi' = \mathsf{SX}_m\xi''$, then we reduce and continue by induction. Otherwise $\beta\langle\xi\rangle = \mathsf{SX}_n\beta\langle\xi'\rangle$ which is inductively in $rexpr_P$. And if $\beta\langle\xi'\rangle$'s length is decreased, we induct. Otherwise $\beta^*\langle\xi'\rangle = \xi'$ and so $\xi' = \mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$, so we get that $\xi = \mathsf{SX}_n\mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$ which we already know from the previous section converges to the desired form.

(3) For $\xi = \mathsf{X}_n\mathsf{S}_m\xi'$, we have that $\beta\langle\xi\rangle = \mathsf{SX}_m\xi'$ and so we induct.

(4) For $\xi = \mathsf{L}_\infty\xi'\mathsf{R}_0$, since L_∞ cannot reduce with the first character of ξ' (as it is not RX), $\beta\langle\xi\rangle = \mathsf{L}_\infty\beta\langle\xi'\mathsf{R}_0\rangle$. If ξ' can be reduced, then this is just $\mathsf{L}_\infty\beta\langle\xi'\rangle\mathsf{R}_0$ and we induct. Otherwise, $\xi' = \mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$ with $n_1 < \cdots < n_{k+1}$, and so this becomes $\mathsf{L}_\infty\mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{RX}_0$, which is covered by the final rule.

(5) For $\xi = \mathsf{L}_\infty \mathsf{SX}_{n_1} \cdots \mathsf{SX}_{n_k} \mathsf{RX}_0 = \mathsf{L}_\infty \xi' \mathsf{RX}_0$, if ξ' can be reduced then we simply induct. Otherwise $n_1 < \cdots < n_k$ and so a β-reduction gives $\mathsf{L}_\infty \mathsf{SX}_{n_1} \cdots \mathsf{SX}_{n_{k-1}} \mathsf{RX}_0$, and we induct. The base is taken by $k = 0$ in which case we get $\mathsf{L}_\infty \mathsf{RX}_0 \to \mathsf{X}_\infty$. ■

From this lemma, since $\beta^*\langle\xi\rangle = \mathsf{SX}_{n_1} \cdots \mathsf{SX}_{n_k} \mathsf{X}_{n_{k+1}}$, we get that $\beta_0^*\langle\xi\rangle \in \mathsf{X}$.

Further note from (4) we can infer that $\mathsf{L}_\infty \xi \mathsf{R}_0 \to^* \mathsf{L}_\infty \beta^*\langle\xi\rangle \mathsf{R}_0$. From the proof of this lemma we have that $\beta^*\langle\xi\rangle = \mathsf{s}^1_{n_1}[\mathsf{x}^1] \cdots \mathsf{s}^k_{n_k}[\mathsf{x}^k] \mathsf{x}^{k+1}_{n_{k+1}}$, and so we can see that $\mathsf{L}_\infty \beta^*\langle\xi\rangle \mathsf{R}_0 \to^* \beta_0^*\langle\xi\rangle_\infty$. Thus we have proven the following lemma:

Lemma 4.2.6: *Let $\xi \in rexpr_P$, then $\beta^*\langle \mathsf{L}_\infty \xi \mathsf{R}_0 \rangle = \beta_0^*\langle\xi\rangle_\infty$.*

Lemma 4.2.7: *Let $\xi \in rexpr_P$ and $n \le \pi(\xi)$, then $\beta^*\langle \xi \mathsf{s}_n \rangle = s_n[\beta_0^*\langle\xi\rangle]$.*

Proof: we know that

$$\xi \mathsf{s}_n \to^* \beta^*\langle\xi\rangle \mathsf{s}_n = \mathsf{s}^1_{n_1}[\mathsf{x}^1] \cdots \mathsf{s}^k_{n_k}[\mathsf{x}^k] \mathsf{x}^{k+1}_{n_{k+1}} \mathsf{s}_n$$

and so this just reduces to the same lemma as for expressions without parentheses. ■

Lemma 4.2.8: *Let $\xi \in rexpr_P$ and $n < \pi(\xi)$, then $\beta_0^*\langle \mathsf{s}_n[\mathsf{x}]\xi \rangle = f_{\mathsf{s}}(\mathsf{x}, \beta_0^*\langle\xi\rangle)$.*

Proof: this too, reduces to the same proof as for expressions without parentheses. ■

Theorem 4.2.9: *For $\xi \in expr_P$, $\textsc{eval}(\xi) = \beta_0^*\langle\xi\rangle$.*

Proof: We induct on the length of ξ. If $\xi = \mathsf{x}$ this is trivial. If $\xi = \mathsf{L}_\infty \rho \mathsf{R}_0 \xi'$ for $\rho \in expr_P$ and so

$$\xi \to^* \beta^*\langle \mathsf{L}_\infty \rho \mathsf{R}_0 \rangle \xi' = \beta_0^*\langle\rho\rangle_\infty \xi' = \textsc{eval}(\rho)\xi'$$

where the final equality is due to induction. Then inductively, we have that $\beta_0^*\langle\xi\rangle = \beta_0^*\langle \textsc{eval}(\rho)\xi' \rangle = \textsc{eval}(\textsc{eval}(\rho)\xi')$, which is just $\textsc{eval}(\mathsf{L}\rho\mathsf{R}\xi') = \textsc{eval}(\xi)$. And finally if n is the lowest operator priority in ξ, then $\xi = \xi_1 \mathsf{s}^1_n \xi_2 \cdots \xi_k \mathsf{s}^k_n \xi_{k+1}$, this just reduces to the same proof for expressions without parentheses by the above two lemmas. ■

5 Comparing Runtimes

For this paper, we developed an interpreter using both our algorithm and the tool menhir. We then compare runtimes for various different programs.

5.1 Currying

We begin with the following program:

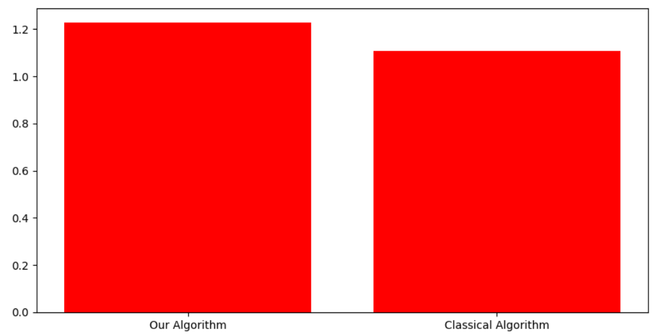
```
fun print (x) {
   _prim_print x
}

fun curry (f) {
   fun curried (x) {
      fun curriedX (y) {
         f (x,y)
      }
      curriedX
   }
   curried
}

fun plus (x,y) {
   x + y
}

print (plus (10, 20));
let curry_plus = curry plus;
print ((curry_plus 10) 20);
```

We compare the time it takes our algorithm and the classical algorithm to run this program 300 times.



It takes our algorithm 0.0040886 seconds to run this program on average, and it takes the classical algorithm 0.0036926 seconds.

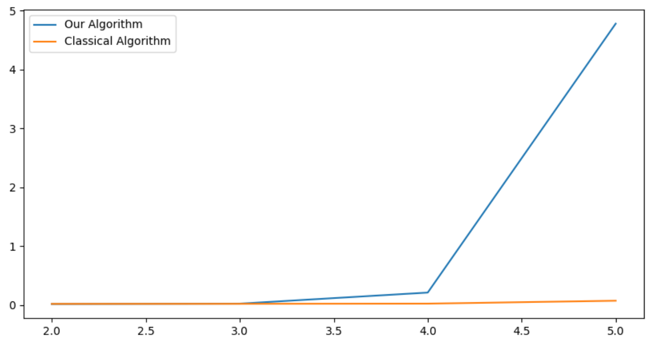
5.2 Expressions

We compare runtimes of programs of the form: `let x = <expr>;`, where `<expr>` is an arithmetical expression generated by the following algorithm:

▷ *Create an expression with ℓ subterms, with a parentheses depth of d, where numbers are chosen from* $[0, n)$.

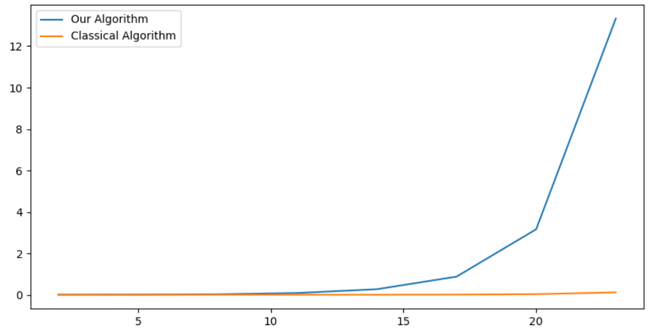
function Generate-Expression(ℓ, d, n)
  if $(d = 0)$ **return** an expression with ℓ random numbers in $[0, n)$ and $\ell - 1$ random operators in $\{+, -, \times, \div\}$.
  return $\left(\text{Generate-Expression}(\ell, d-1, n)\right) \circ_1 \cdots \circ_{\ell-1} \left(\text{Generate-Expression}(\ell, d-1, n)\right)$ where $\circ_i \in \{+, -, \times, \div\}$ are chosen randomly
end function

Running this with $n = 100$, and we parameterize $\ell = d$, we get the following:



5.3 Fibonacci

We compare runtimes of a fibonacci program, which computes the nth fibonacci number using its recursive formula, where n varies.



Using exponential regression, we estimate that our algorithm takes ab^n seconds, where $a = 0.000138585$ and $b = 1.64685$; and the classical algorithm takes ab^n seconds where $a = 0.00000362553$ and $b = 1.5809$.