Iterative Reduction: Parsing and Interpreting Code Without Trees

Ari Feiglin[1] and Yoni Zohar[1]

Bar-Ilan University, Ramat Gan, Israel

Abstract. In this paper we discuss an alternative method to parsing and interpreting human-readable code. This method utilizes an iterative approach, as opposed to a recursive tree-oriented approach found in most modern compilers and interpreters. We implement our algorithm in an interpreter and compare it with an interpreter created using more classical means.

Keywords: Parsing · Interpreting · AST · Parse Trees

1 Introduction

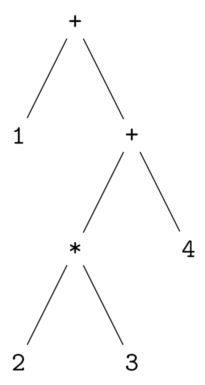
Nowadays, most programming language interpreters follow the following pipeline (see [1])

(**1**) Lexing: lexically analyze the input code, separating it into atomic *tokens*.
(**2**) Parsing: convert the parsed tokens into a *parse tree*: a tree which represents the code according to the language's grammar.
(**3**) Interpreting: recursively descend the tree and execute the code according to the instructions embedded in the tree.

For example, we can parse the following expression:

`1 + 2 * 3 + 4`

as follows



One can then descend the parse tree and compute the expression. Explicitly, if you reach a node whose operator is $\circ$, evaluate the left tree to get x, evaluate the right tree to get y, and then return $x \circ y$.

Unfortunately, constructing such a tree is not so easy. Multiple methods for doing so exist (some are listed in [1]), as well as tools for automating this process.

We propose a novel method for the process of interpretation, which doesn't use parse trees. But our algorithm is not limited to this, and can be altered to produce parse trees as well (see §6). This means that our algorithm can be utilized as part of the pipeline for compilers and traditional interpreters.

The idea of the algorithm is as follows: the input is a stream of tokens, which are indivisible characters in any language. Given a string of tokens $\boxed{\sigma^1} \cdots \boxed{\sigma^n}$ (for readability, tokens are written in boxes), we form rules for combining two tokens. For example, $\boxed{\texttt{1}}$ and $\boxed{\texttt{+}}$ can be combined together to give a token $\boxed{\texttt{1,+}}$. Then if we follow a token of the form $\boxed{\texttt{x,+}}$ with a token of the form $\boxed{\texttt{y}}$, the tokens are combined to form a token $\boxed{\texttt{x+y}}$. So for example:

$$\boxed{\texttt{1}}\,\boxed{\texttt{+}}\,\boxed{\texttt{2}}\,\boxed{\texttt{+}}\,\boxed{\texttt{3}} \to \boxed{\texttt{1,+}}\,\boxed{\texttt{2}}\,\boxed{\texttt{+}}\,\boxed{\texttt{3}} \to \boxed{\texttt{3}}\,\boxed{\texttt{+}}\,\boxed{\texttt{3}} \to \boxed{\texttt{3,+}}\,\boxed{\texttt{3}} \to \boxed{\texttt{6}}$$

But what about `1+2*3`? We'd get

$$\boxed{\texttt{1}}\,\boxed{\texttt{+}}\,\boxed{\texttt{2}}\,\boxed{\texttt{*}}\,\boxed{\texttt{3}} \to \boxed{\texttt{1,+}}\,\boxed{\texttt{2}}\,\boxed{\texttt{*}}\,\boxed{\texttt{3}} \to \boxed{\texttt{3}}\,\boxed{\texttt{*}}\,\boxed{\texttt{3}} \to \boxed{\texttt{3,*}}\,\boxed{\texttt{3}} \to \boxed{\texttt{9}}$$

So to remedy this we add the concept of *priorities*: for each token in our alphabet Σ, we assign a priority $p \in \overline{\mathbb{N}}$. This can be thought of as a *priority function* $\pi \colon \Sigma \longrightarrow \overline{\mathbb{N}}$. So if the tokens $\boxed{\sigma^1} \cdots \boxed{\sigma^n}$ have respective priorities $p_1, \ldots, p_n$ we can form the string $\boxed{\sigma^1}_{p_1} \cdots \boxed{\sigma^n}_{p_n}$ (where the subscript is part of the character). Now, we only combine two tokens if the token of the first is at least as great as the token of the second, and then the combined token gets the lower (the second) of the two priorities. So we can assign to tokens containing numbers a priority of ∞, the token `+` gets a priority of 1, and `*` gets a priority of 2. Then this becomes:

$$\boxed{\texttt{1}}_\infty\,\boxed{\texttt{+}}_1\,\boxed{\texttt{2}}_\infty\,\boxed{\texttt{*}}_2\,\boxed{\texttt{3}}_\infty \to \boxed{\texttt{1,+}}_1\,\boxed{\texttt{2}}_\infty\,\boxed{\texttt{*}}_2\,\boxed{\texttt{3}}_\infty \to \boxed{\texttt{1,+}}_1\,\boxed{\texttt{2,*}}_2\,\boxed{\texttt{3}}_\infty$$

This cannot be combined any further, so we need another rule: we give the final token a priority of 0, so this becomes:

$$\boxed{\texttt{1,+}}_1\,\boxed{\texttt{2,*}}_2\,\boxed{\texttt{3}}_0 \to \boxed{\texttt{1,+}}_1\,\boxed{\texttt{6}}_0 \to \boxed{\texttt{7}}_0$$

which is the correct result.

2 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 integer numbers.
(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 (i.e. $\operatorname{dom} f \subseteq A$).
(**6**) ε denotes the empty string.
(**7**) If $\mathcal{L}$ is a language (a set of strings), then $\mathcal{L}^{\varepsilon}$ is equal to $\mathcal{L} \cup \{\varepsilon\}$.
(**8**) A list is denoted by $(x_1, \ldots, x_n)$.
(**9**) The concatenation of two lists ℓ_1 and ℓ_2 is denoted by $\ell_1 @ \ell_2$.
(**10**) $x :: \ell$ is the list whose first element is x and tail is ℓ.

3 The Algorithm

3.1 A First Attempt – Primitive Reduction

We can formalize the algorithm given in the introduction as follows: we start with an alphabet of tokens Σ. Define $\overline{\Sigma} = \Sigma \times \overline{\mathbb{N}}$, and we assume the existence of a so-called *initial β-reducer*: $\widehat{\beta}\colon \Sigma \times \Sigma \longrightarrow \Sigma$, which is defined by the implementer of the language being interpreted, and is the function which tells us how to combine tokens together. We extend it to a *derived β-reducer* which is a function $\beta\colon \overline{\Sigma}^* \longrightarrow \overline{\Sigma}^*$ as follows.

Suppose $\xi = \sigma^1_{p_1} \cdots \sigma^n_{p_n}$, then $\beta(\xi)$ is defined recursively as follows:

(**1**) if $n = 1$, then $\beta(\xi) = \sigma^n_0$,
(**2**) if $p_1 \geq p_2$, then $\beta(\xi) = \widehat{\beta}(\sigma^1, \sigma^2)_{p_2} \sigma^3_{p_3} \cdots \sigma^n_{p_n}$,
(**3**) otherwise $\beta(\xi) = \sigma^1_{p_1}\, \beta(\sigma^2_{p_2} \cdots \sigma^n_{p_n})$.

Notice that since in our reduction we get pairs of tokens in the form $\boxed{\texttt{x,*}}$, $\boxed{\texttt{y,+}}$, we need to add extra rules to handle these cases. Specifically here we have $\widehat{\beta}(\boxed{\texttt{x,*}}, \boxed{\texttt{y,+}}) = \boxed{\texttt{x*y,+}}$.

So let us define Σ to consist of tokens of the form $\boxed{\texttt{x}}$ for numbers x, $\boxed{\circ}$ for operator symbols $\circ$ (addition, multiplication, etc.), and $\boxed{\texttt{x,}\circ}$. Then we define the initial β-reducer by

(**1**) $\widehat{\beta}(\boxed{\texttt{x}}, \boxed{\circ}) = \boxed{\texttt{x,}\circ}$ for operator symbols $\circ$,
(**2**) $\widehat{\beta}(\boxed{\texttt{x,}\circ}, \boxed{\texttt{y}}) = \boxed{\texttt{x}\circ\texttt{y}}$,
(**3**) $\widehat{\beta}(\boxed{\texttt{x,}\circ}, \boxed{\texttt{y,}\bullet}) = \boxed{\texttt{x}\circ\texttt{y,}\bullet}$,
(**4**) and any other inputs can be given an arbitrary output.

So for example we can now handle expressions of the form

$$\boxed{\texttt{1}}_\infty \boxed{\texttt{*}}_2 \boxed{\texttt{2}}_\infty \boxed{\texttt{+}}_1 \boxed{\texttt{3}}_\infty \to \boxed{\texttt{1,*}}_2 \boxed{\texttt{2}}_\infty \boxed{\texttt{+}}_1 \boxed{\texttt{3}}_\infty \to \boxed{\texttt{1,*}}_2 \boxed{\texttt{2,+}}_1 \boxed{\texttt{3}}_\infty \to \boxed{\texttt{2,+}}_1 \boxed{\texttt{3}}_\infty \\ \to \boxed{\texttt{2,+}}_1 \boxed{\texttt{3}}_0 \to \boxed{\texttt{5}}_0$$

But this has a major issue: it can't handle things like parentheses. We could of course handle parentheses by changing how we assign priorities: every time we see an open parentheses we could increase the priority of each token and then beta-reduce. But this complicates how we allocate priorities, and does not generalize well. Furthermore the notion introduced in the next section, partial reduction, generalizes well and is used as a basepoint in the remainder of this paper.

3.2 A Second Attempt – Partial Reduction

In order to handle parentheses, all we need to do is to make a simple change to our algorithm. Instead of a full function which also tells us how to compute the new priority, the initial β-reducer will be a partial function:

$$\widehat{\beta}\colon \Sigma \times \Sigma \rightharpoonup \Sigma \times (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}} \to \overline{\mathbb{Z}})$$

We add a function to compute the new priority of the token to the output of the initial β-reducer instead of taking the priorities as inputs. This is because the new token computed should not be dependent on the priorities of the input tokens, as this just adds extra unnecessary complexity.

Now we define the derived β-reducer as follows; it is a function $\beta\colon \overline{\Sigma}^* \rightharpoonup \overline{\Sigma}^*$ where for $\xi = \sigma^1_{p_1} \cdots \sigma^n_{p_n}$:

(1) if $n = 1$ then $\beta(\xi) = \sigma^n_0$,
(2) if $p_1 \geq p_2$ and $\widehat{\beta}(\sigma^1, \sigma^2) = (\tau, \rho)$ is defined, then $\beta(\xi) = \tau_{\rho(p_1,p_2)}\sigma^3_{p_3} \cdots \sigma^n_{p_n}$,
(3) otherwise $\beta(\xi) = \sigma^1_{p_1}\, \beta(\sigma^2_{p_2} \cdots \sigma^n_{p_n})$.

We define our alphabet Σ to be the same as in the previous section.

Now we define our initial β-reducer which will support parentheses. It is an extension of our previous β-reducer:

$$\widehat{\beta}(\boxed{\mathtt{x}}, \boxed{\circ}) = (\boxed{\mathtt{x},\circ}, \mathsf{snd}), \quad \widehat{\beta}(\boxed{\mathtt{x},\circ}, \boxed{\mathtt{y}}) = (\boxed{\mathtt{x} \circ \mathtt{y}}, \mathsf{snd}) \qquad \text{for } \circ \in \{\mathtt{+},\mathtt{*}\}$$
$$\widehat{\beta}(\boxed{\mathtt{x,*}}, \boxed{\mathtt{y,+}}) = (\boxed{\mathtt{x*y,+}}, \mathsf{snd})$$

where $\mathsf{fst}, \mathsf{snd}$ are the first and second projections, respectively (i.e. $(x, y) \mapsto x$ and $(x, y) \mapsto y$). We then add the rules for parentheses:

$$\widehat{\beta}(\boxed{\mathtt{x}}, \boxed{\mathtt{)}}) = (\boxed{\mathtt{x,)}}, \mathsf{snd}), \quad \widehat{\beta}\boxed{\mathtt{x},\circ}, \boxed{\mathtt{y,)}} = (\boxed{\mathtt{x} \circ \mathtt{y,)}}, \mathsf{snd}) \qquad \text{for } \circ \in \{\mathtt{+},\mathtt{*}\}$$
$$\widehat{\beta}(\boxed{\mathtt{(}}, \boxed{\mathtt{x,)}}) = (\boxed{\mathtt{x}}, \mathsf{fst})$$

And we define our priority function as follows:

$$\pi(\boxed{\mathtt{x}}) = \infty \text{ for numbers } \mathtt{x}, \quad \pi(\boxed{\mathtt{+}}) = 1, \quad \pi(\boxed{\mathtt{*}}) = 2, \quad \pi(\boxed{\mathtt{(}}) = \infty, \quad \pi(\boxed{\mathtt{)}}) = 0$$

Now take for example `2*(1+3)+4`:

$$\begin{aligned}
\boxed{\mathtt{2}}_\infty \boxed{\mathtt{*}}_2 \boxed{\mathtt{(}}_\infty \boxed{\mathtt{1}}_\infty \boxed{\mathtt{+}}_1 \boxed{\mathtt{3}}_\infty \boxed{\mathtt{)}}_0 \boxed{\mathtt{+}}_1 \boxed{\mathtt{4}}_\infty &\to \boxed{\mathtt{2,*}}_2 \boxed{\mathtt{(}}_\infty \boxed{\mathtt{1}}_\infty \boxed{\mathtt{+}}_1 \boxed{\mathtt{3}}_\infty \boxed{\mathtt{)}}_0 \boxed{\mathtt{+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{2,*}}_2 \boxed{\mathtt{(}}_\infty \boxed{\mathtt{1,+}}_1 \boxed{\mathtt{3}}_\infty \boxed{\mathtt{)}}_0 \boxed{\mathtt{+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{2,*}}_2 \boxed{\mathtt{(}}_\infty \boxed{\mathtt{1,+}}_1 \boxed{\mathtt{3,)}}_0 \boxed{\mathtt{+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{2,*}}_2 \boxed{\mathtt{(}}_\infty \boxed{\mathtt{4,)}}_0 \boxed{\mathtt{+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{2,*}}_2 \boxed{\mathtt{4}}_\infty \boxed{\mathtt{+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{2,*}}_2 \boxed{\mathtt{4,+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{8,+}}_1 \boxed{\mathtt{4}}_\infty \\
&\to \boxed{\mathtt{8,+}}_1 \boxed{\mathtt{4}}_0 \\
&\to \boxed{\mathtt{12}}_0
\end{aligned}$$

Notice that our β-reducers don't care about the exact value of the number and operation: both $\boxed{1}\ \boxed{+}$ and $\boxed{2}\ \boxed{*}$ are mapped to something of the form $\boxed{\mathtt{x,\circ}}$. We can thus abstract away the exact value of tokens and have our initial β-reducer deal only with the form of the token. We do such in our third attempt.

3.3 A Third Attempt – Valued Reduction

Here we abstract away the value of each term; the value of the new term given by the initial β-reducer will be determined by an output function. Our alphabet of types Σ, instead of having a token for every number, will just have tokens for types (number, operator, etc.). We also have another set $\mathcal{U}$, the *universe of values.*

The initial β-reducer is a function

$$\widehat{\beta}\colon \Sigma \times \Sigma \rightharpoonup \Sigma \times (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}} \to \overline{\mathbb{Z}}) \times (\mathcal{U} \times \mathcal{U} \rightharpoonup \mathcal{U})$$

Our alphabet of types is $\Sigma = \{\mathsf{Num}, \mathsf{Op}, \mathsf{Lparen}, \mathsf{Rparen}, \mathsf{OpNum}, \mathsf{RparenNum}\}$, define $\overline{\Pi} = \overline{\Sigma} \times \mathcal{U}$ to be the set of all possible terms (a term consists of a type, a priority, and a value). Since terms now consist of three parts, we write them as $\sigma_p(v)$, where σ is the type; p is the priority; and v the value.

Our rules for the derived β-reducer are similar to before: it is a function $\beta\colon \overline{\Pi}^* \rightharpoonup \overline{\Pi}^*$ where for $\xi = \sigma^1_{p_1}(v_1)\cdots\sigma^n_{p_n}(v_n)$

(1) if $n = 1$ then $\beta(\xi) = \sigma^n_0(v_n)$,
(2) if $p_1 \geq p_2$ and $\widehat{\beta}(\sigma^1, \sigma^2) = (\tau, \rho, f)$ is defined, then

$$\beta(\xi) = \tau_{\rho(p_1,p_2)}(f(v_1, v_2))\sigma^3_{p_3}(v_3)\cdots\sigma^n_{p_n}(v_n)$$

(3) otherwise $\beta(\xi) = \sigma^1_{p_1}(v_1)\beta(\sigma^2_{p_2}(v_2)\cdots\sigma^n_{p_n}(v_n))$.

Our initial β-reducer will is defined as follows:

- $\widehat{\beta}(\mathsf{Num}, \mathsf{Op}) = (\mathsf{OpNum}, \mathsf{snd}, (v, f \mapsto (v, f))$
- $\widehat{\beta}(\mathsf{OpNum}, \mathsf{Num}) = (\mathsf{Num}, \mathsf{snd}, ((v, f), u \mapsto f(v, u)))$
- $\widehat{\beta}(\mathsf{OpNum}, \mathsf{OpNum}) = (\mathsf{OpNum}, \mathsf{snd}, ((v, f), (u, g) \mapsto (f(v, u), g))$
- $\widehat{\beta}(\mathsf{Num}, \mathsf{Rparen}) = (\mathsf{RparenNum}, \mathsf{snd}, (v, \_ \mapsto v))$
- $\widehat{\beta}(\mathsf{OpNum}, \mathsf{RparenNum}) = (\mathsf{RparenNum}, \mathsf{snd}, ((v, f), u \mapsto f(v, u)))$
- $\widehat{\beta}(\mathsf{Lparen}, \mathsf{RparenNum}) = (\mathsf{Num}, \mathsf{fst}, (\_, v \mapsto v))$

So taking our previous example of `2*(1+3)+4`: we first must convert it into tokens acceptable for β-reduction (via lexing). So it is converted to

$\mathsf{Num}_\infty(2)\ \mathsf{Op}_2(*)\ \mathsf{Lparen}_\infty\ \mathsf{Num}_\infty(1)\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(3)\ \mathsf{Rparen}_0\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(4)$

The derivation is then

$$\begin{gathered}
\mathsf{OpNum}_2(2,*)\ \mathsf{Lparen}_\infty\ \mathsf{Num}_\infty(1)\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(3)\ \mathsf{Rparen}_0\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(2,*)\ \mathsf{Lparen}_\infty\ \mathsf{OpNum}_1(1,+)\ \mathsf{Num}_\infty(3)\ \mathsf{Rparen}_0\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(2,*)\ \mathsf{Lparen}_\infty\ \mathsf{OpNum}_1(1,+)\ \mathsf{RparenNum}_0(3)\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(2,*)\ \mathsf{Lparen}_\infty\ \mathsf{RparenNum}_0(4)\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(2,*)\ \mathsf{Num}_\infty(4)\ \mathsf{Op}_1(+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(2,*)\ \mathsf{OpNum}_1(4,+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(8,+)\ \mathsf{Num}_\infty(4)\\
\mathsf{OpNum}_2(8,+)\ \mathsf{Num}_0(4)\\
\mathsf{Num}_0(12)
\end{gathered}$$

But this of course is not capable of handling the semantics necessary for programming languages. We need a notion of a state, and a somewhat more refined algorithm.

3.4 A Fourth Attempt – 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*; this is the set of all printable tokens a programmer may type.
(3) $\mathcal{T}_\Sigma$ the set of *type terms*; these are terms which correspond to types in the classical sense.
(4) $\mathcal{T}_\mathcal{A}$ the set of *abstract terms*; these are intermediate terms (like OpNum from the previous section).

The sets $\mathcal{T}_\mathcal{P}, \mathcal{T}_\Sigma, \mathcal{T}_\mathcal{A}$ are disjoint, and we place no such restriction on $\mathcal{U}$.

While we don't need to place any further restrictions on $\mathcal{T}_\mathcal{P}, \mathcal{T}_\Sigma, \mathcal{T}_\mathcal{A}$, we found it useful to explicitly construct them. To construct $\mathcal{T}_\Sigma$, we begin with a set of *base types* Σ. Each base type has an associated arity in $\overline{\mathbb{N}}$; let Σ^n denote the set of all base types of arity n. Then we define $\mathcal{T}_\Sigma$ as follows:

$$\mathcal{T}_\Sigma ::= \Sigma^0 \mid \Sigma^1\mathcal{T}_\Sigma \mid \Sigma^2\mathcal{T}_\Sigma\mathcal{T}_\Sigma \mid \Sigma^3\mathcal{T}_\Sigma\mathcal{T}_\Sigma\mathcal{T}_\Sigma \mid \cdots \mid \Sigma^\infty(\mathcal{T}_\Sigma\cdots\mathcal{T}_\Sigma)$$

To construct $\mathcal{T}_\mathcal{A}$ we start with $\mathcal{A}$, a set of *atomic abstract terms*, again each with an arity in $\overline{\mathbb{N}}$. $\mathcal{A}^n$ is the set of atomic abstract terms of a set arity n. Then $\mathcal{T}_\mathcal{A}$ is defined by

$$\mathcal{T}_\mathcal{A} ::= \mathcal{A}^0 \mid \mathcal{A}^1\mathcal{T}_\Sigma^\varepsilon \mid \mathcal{A}^2\mathcal{T}_\Sigma^\varepsilon\mathcal{T}_\Sigma^\varepsilon \mid \mathcal{A}^3\mathcal{T}_\Sigma^\varepsilon\mathcal{T}_\Sigma^\varepsilon\mathcal{T}_\Sigma^\varepsilon \mid \cdots \mid \mathcal{A}^\infty\mathcal{T}_\Sigma^\varepsilon\cdots\mathcal{T}_\Sigma^\varepsilon$$

Here we allow inputs to atomic abstract terms to be empty (i.e. $\mathcal{T}_\Sigma^\varepsilon$ instead of just $\mathcal{T}_\Sigma$).

So if $\Sigma = \{\mathsf{Num}_0, \mathsf{List}_1, \mathsf{Product}_\infty\}$ (the subscripts are the arities of each base type) and $\mathcal{A} = \{\mathsf{Op}\}$, then

$$\mathsf{Num}, \mathsf{ListNum}, \mathsf{Product}(\mathsf{ListNumNum}) \in \mathcal{T}_\Sigma, \qquad \mathsf{OpListNum} \in \mathcal{T}_\mathcal{A}$$

For readability we can put parentheses and commas where necessary, but importantly these sets have the *unique reconstruction property*: it is possible to compute their construction from the string itself (i.e. a string cannot be constructed from two different patterns).

Now we define the following sets:

(1) $\mathcal{T} = \mathcal{T}_\mathcal{P} \cup \mathcal{T}_\Sigma \cup \mathcal{T}_\Sigma$, the set of *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 *internal expressions*,
(4) $\Pi = \mathcal{T}_\mathcal{I} \cup \mathcal{T}_\mathcal{P}$, the set of *expressions*.

Finally we define $\overline{\Pi} = \Pi \times \overline{\mathbb{Z}}$. Elements of $\overline{\Pi}$ will be written like $\sigma_n(v)$ as before; for printable terms we act as if the value is empty.

The *initial priority function* is a function $\pi\colon \mathcal{T}_\mathcal{P} \longrightarrow \overline{\mathbb{Z}}$. This can be naturally extended to $\pi\colon \mathcal{T}_\mathcal{P}^* \longrightarrow \overline{\Pi}^*$.

Finally, our *initial* β*-reducer* is a function (State is a set of functions $\mathcal{T}_\mathcal{P} \rightharpoonup \Pi_\mathcal{I}$ which we define later)

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

and we extend this to a *derived* β*-reducer*, a function

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

inputs in $\overline{\Pi}^* \times \text{State}$ will be written as $\langle\, \xi \mid \bar{s} \,\rangle$ for $\xi \in \overline{\Pi}^*$ and $\bar{s} \in \text{State}$. We define $\beta^* \langle\, \xi \mid \bar{s} \,\rangle$ to be the value obtained by composing β with itself of $\langle\, \xi \mid \bar{s} \,\rangle$ until the convergence of ξ. Meaning

$$\begin{aligned} &\beta^* = \beta^n \langle\, \xi \mid \bar{s} \,\rangle \\ &\qquad \text{where } n = \min\{\, n \geq 1 \mid \beta^n \langle\, \xi \mid \bar{s} \,\rangle = \langle\, \xi' \mid \bar{s}' \,\rangle, \beta \langle\, \xi' \mid \bar{s}' \,\rangle = \langle\, \xi' \mid \bar{s}'' \,\rangle \,\} \end{aligned}$$

We define $\beta \langle\, \xi \mid \bar{s} \,\rangle$ as follows, where $\xi = \sigma^1_{p_1}(v_1) \cdots \sigma^n_{p_n}(v_n)$.

(1) if $\sigma^1 \in \mathcal{T}_\mathcal{P}$ then

$$\beta \langle\, \xi \mid \bar{s} \,\rangle = \left\langle\, s(\sigma^1)_{p_1} \sigma^2_{p_2}(v_2) \cdots \sigma^n_{p_n}(v_n) \;\middle|\; \bar{s} \,\right\rangle$$

(2) if $\widehat{\beta}(\sigma^1, \varepsilon) = (\tau, \rho, f)$ is defined, then compute $f(v_1, \_, s) = (u, \xi', s')$ and $\beta^* \langle\, \pi\xi' \mid \bar{s}' \,\rangle = \langle\, \xi'' \mid \bar{s}'' \,\rangle$, and

$$\beta \langle\, \xi \mid \bar{s} \,\rangle = \left\langle\, \tau_{\rho(p_1, \_)} \xi'' \sigma^2_{p_2}(v_2) \cdots \sigma^n_{p_n}(v_n) \;\middle|\; \bar{s}'' \,\right\rangle$$

(3) if $p_1 \geq p_2$ and $\widehat{\beta}(\sigma^1, \sigma^2) = (\tau, \rho, f)$ is defined, then compute $f(v_1, v_2, s) = (u, \xi', s')$ and $\beta^* \langle\, \pi\xi' \mid \bar{s}' \,\rangle = \langle\, \xi'' \mid \bar{s}'' \,\rangle$, and

$$\beta \langle\, \xi \mid \bar{s} \,\rangle = \left\langle\, \tau_{\rho(p_1, p_2)} \xi'' \sigma^3_{p_3}(v_2) \cdots \sigma^n_{p_n}(v_n) \;\middle|\; \bar{s}'' \,\right\rangle$$

(4) otherwise compute $\beta \left\langle\, \sigma^2_{p_2}(v_2) \cdots \sigma^n_{p_n}(v_n) \;\middle|\; \bar{s} \,\right\rangle = \langle\, \xi' \mid \bar{s}' \,\rangle$ and

$$\beta \langle\, \xi \mid \bar{s} \,\rangle = \left\langle\, \sigma^1_{p_1}(v_1) \xi' \;\middle|\; \bar{s}' \,\right\rangle$$

States We begin by defining *point-states* to be partial maps $\mathcal{T}_{\mathcal{P}} \rightharpoonup \Pi_{\mathcal{I}}$. If s_1, s_2 are two point-states then we define their composition to be

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

Point-states with finite domain will be denoted by $[\sigma_i \mapsto \varkappa_i]_{i=1}^n$ where $\sigma_1, \ldots, \sigma_n$ is the domain and $\varkappa_1, \ldots, \varkappa_n$ are their respective images.

A *state* will be a pair of a sequence of point-states and a set of indices. This will be denoted by $\bar{s} = [(s_1, \ldots, s_n),\ I = (i_1, \ldots, i_k)]$ (we require that $k \leq n$). We define

(1) $\bar{s} + s = \big[(s_1, \ldots, s_n, s),\ I\big]$, for s a point-state,
(2) $\bar{s} +_c s = \big[(s_1, \ldots, s_n, s),\ (i_1, \ldots, i_k, n+1)\big]$, for s a point-state,
(3) $pop\bar{s} = \big[(s_1, \ldots, s_{n-1}),\ I\big]$ if $i_k < n$, otherwise it is defined to be equal to $\big[(s_1, \ldots, s_{n-1}),\ (i_1, \ldots, i_{k-1})\big]$,
(4) $\bar{s}s = \big[(s_1, \ldots, s_{n-1}, s_n s),\ I\big]$ for s a point-state,
(5) $\bar{s}(\sigma) = s_1 \cdots s_n(\sigma)$ for $\sigma \in \mathcal{T}_{\mathcal{P}}$,
(6) $\bar{s}_c = s_{i_k} \cdots s_n$,
(7) if $\sigma \in \mathcal{T}_{\mathcal{P}}$ and $\varkappa \in \Pi_{\mathcal{I}}$ then $\bar{s}\{\sigma \mapsto \varkappa\} = \big[(s_1, \ldots, s_k[\sigma \mapsto \varkappa], \ldots, s_n),\ I\big]$ where k is the maximum index such that $\sigma \in \operatorname{dom} s_k$.

4 Implementing the Algorithm

We implement our algorithm in an interpreter for a toy language we dub "ILang". We also create an interpreter for ILang using menhir[2]. In this section we detail the implementation of our algorithm and compare and contrast the two implementations.

4.1 The Grammar

The grammar for our toy language is pretty straight-forward:

Identifiers

$$\begin{aligned} \mathit{letter} &::= (a \ldots z \mid A \ldots Z \mid \_) \\ \mathit{digit} &::= (0 \ldots 9) \\ \mathit{non\_digit} &::= (\_ \mid \mathit{letter}) \\ \mathit{ident} &::= \mathit{letter}\ (\mathit{digit} \mid \mathit{non\_digit})^* \\ \mathit{pident} &::= \_\mathbf{prim}\_(\mathit{digit} \mid \mathit{non\_digit})^* \end{aligned}$$

Constant Expressions

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

Operators

$$\mathit{op} ::= (@ \mid + \mid - \mid * \mid / \mid <= \mid < \mid >= \mid > \mid == \mid !=)$$

Expressions

$$\mathit{expr} ::= \varepsilon$$

$$
\begin{aligned}
&\mid \textbf{let}\ \mathit{ident}\ =\ \mathit{value}\ ;\ \mathit{expr}\\
&\mid \textbf{fun}\ \mathit{ident}\ (\ \mathit{plist}\ )\ \{\ \mathit{expr}\ \}\ \mathit{expr}\\
&\mid \textbf{if}\ (\ \mathit{value}\ )\ \{\ \mathit{expr}\ \}\ \{\ \mathit{expr}\ \}\ \mathit{expr}\\
&\mid \mathit{funcall}\ ;\ \mathit{expr}\\
&\mid \mathit{pident}\ \mathit{value}\ \mathit{expr}\\
&\mid \mathit{value}
\end{aligned}
$$

Values

$$
\begin{aligned}
\mathit{value} &::= \mathit{const}\\
&\mid \mathit{ident}\\
&\mid \mathit{value}\ \mathit{op}\ \mathit{value}\\
&\mid \mathit{funcall}\\
&\mid (\ \mathit{product}\ )\\
\mathit{product} &::= \mathit{value}\\
&\mid \mathit{value}\ ,\ \mathit{product}
\end{aligned}
$$

Function Calls

$$
\begin{aligned}
\mathit{plist} &::= \mathit{ident}\\
&\mid \mathit{ident}\ ,\ \mathit{plist}\\
\mathit{funcall} &::= \mathit{value}\ \mathit{value}
\end{aligned}
$$

This basic grammar will suffice for our classical implementation. For our algorithm, we will add more features (such as lists and nested products).

4.2 The Initial β-Reducer

Our algorithm is based on an initial β-reducer instead of a grammar. Thus in order to implement our algorithm we must first form an initial β-reducer. So we must translate our previous section to an initial β-reducer.

Instead of explicitly writing the sets $\mathcal{A}$ and Σ, base types will be underlined, atomic abstract terms will be written normally, and internal terms will be boxed. General terms will be denoted by σ, τ, and lists of terms will be denoted by Ω.
End:

- $\boxed{\sigma}$ end $\longrightarrow$ $\boxed{\sigma}$ minfty $(u, \_, s \to u, \varepsilon, s)$

Arithmetic:

- $\underline{\sigma}$ op $\longrightarrow$ op$\underline{\sigma}$ snd $(u, f, s \to (u, f), \varepsilon, s)$
- op$\underline{\sigma}$ op$\underline{\sigma}$ $\longrightarrow$ op$\underline{\sigma}$ snd $\big((u, f), (v, g), s \to (f(u, v), g), \varepsilon, s\big)$
- op$\underline{\sigma}$ $\underline{\sigma}$ $\longrightarrow$ $\underline{\sigma}$ snd $\big((u, f), v, s \to f(u, v), \varepsilon, s\big)$
- pop $\underline{\sigma}$ $\longrightarrow$ $\underline{\sigma}$ snd $\big((f, g), u, s \to f(u), \varepsilon, s\big)$
- $\underline{\sigma}$ pop $\longrightarrow$ op$\underline{\sigma}$ one $\big(u, (f, g), s \to (u, g), \varepsilon, s\big)$
- $\underline{\sigma}$ rparen $\longrightarrow$ rparen$\underline{\sigma}$ snd $(u, \_, s \to u, \varepsilon, s)$
- op$\underline{\sigma}$ rparen$\underline{\sigma}$ $\longrightarrow$ rparen$\underline{\sigma}$ snd $\big((f, u), v, s \to f(u, v), \varepsilon, s\big)$
- pop rparen$\underline{\sigma}$ $\longrightarrow$ rparen$\underline{\sigma}$ snd $\big((f, g), u, s \to f(u), \varepsilon, s\big)$

- $\mathsf{lparen\ rparen}\underline{\sigma} \longrightarrow \underline{\sigma}\ \mathsf{fst}\ (\_, u, s \to u, \varepsilon, s)$

Lists:

- $\mathsf{lbrack}\ \underline{\sigma} \longrightarrow \mathsf{lbrack}\underline{\sigma}\ \mathsf{fst}\ (\_, u, s \to (u), \varepsilon, s)$
- $\mathsf{lbrack}\underline{\sigma}\ \underline{\sigma} \longrightarrow \mathsf{lbrack}\underline{\sigma}\ \mathsf{fst}\ (\ell, u, s \to (\ell, u), \varepsilon, s)$
- $\mathsf{lbrack}\underline{\sigma}\ \mathsf{rbrack} \longrightarrow \underline{\mathsf{list}\sigma}\ \mathsf{infty}\ (\ell, \_, s \to \ell, \varepsilon, s)$
- $\mathsf{period}\ \underline{\mathsf{num}} \longrightarrow \mathsf{index\ zero}\ (\_, n, s \to n, \varepsilon, s)$
- $\underline{\mathsf{list}\sigma}\ \mathsf{index} \longrightarrow \underline{\sigma}\ \mathsf{fst}\ (\ell, i, s \to \ell_i, \varepsilon, s)$

Variables:

- $\mathsf{let}\ x \longrightarrow \mathsf{letvar\ snd}\ (\_, \_, s \to (x, \varnothing), \varepsilon, s)$
- $\mathsf{letvar\ index} \longrightarrow \mathsf{letvar\ fst}\ \big((x, \ell), n, s \to (x, (\ell, n)), \varepsilon, s\big)$
- $\mathsf{letvar\ equal} \longrightarrow \mathsf{leteq\ minfty}\ \big((x, \ell), \_, s \to (x, \ell), \varepsilon, s\big)$
- $\mathsf{leteq}\ \boxed{\sigma} \longrightarrow \varepsilon\ \varnothing\ \big((x, \ell), v, s \to \varepsilon, \varepsilon, s'\big)$ 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:

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

Products:

- $\underline{\sigma}\ \mathsf{comma} \longrightarrow \mathsf{comma}(\underline{\sigma})\ \mathsf{snd}\ (u, \_, s \to (u), \varepsilon, s)$
- $\mathsf{op}\underline{\sigma}\ \mathsf{comma}(\underline{\sigma}) \longrightarrow \mathsf{comma}(\underline{\sigma})\ \mathsf{snd}\ ((f, u), (v) \to (f(u, v)), \varepsilon, s)$
- $\mathsf{pop\ comma}(\underline{\sigma}) \longrightarrow \mathsf{comma}(\underline{\sigma})\ \mathsf{snd}\ ((f, g), (u) \to (f(u), \varepsilon, s))$
- $\mathsf{comma}\underline{\Omega}\ \mathsf{comma}(\underline{\sigma}) \longrightarrow \mathsf{comma}(\underline{\Omega}, \underline{\sigma})\ \mathsf{snd}\ (\ell, \ell', s \to (\ell, \ell'), \varepsilon, s)$
- $\mathsf{comma}\underline{\Omega}\ \mathsf{rparen}\underline{\sigma} \longrightarrow \mathsf{listrparen}(\underline{\Omega}, \underline{\sigma})\ \mathsf{snd}\ (\ell, v \to (\ell, v), \varepsilon, s)$
- $\mathsf{lparen\ listrparen}\underline{\Omega} \longrightarrow \underline{\mathsf{product}\Omega}\ \mathsf{infty}\ (\_, \ell, s \to \ell, \varepsilon, s)$

Primitives:

- $\underline{\mathsf{primitive}}\ \boxed{\sigma} \longrightarrow \varepsilon\ \varnothing\ (f, v, s \to \varepsilon, w, s)$ where $f(\boxed{\sigma}, v) = (w, s')$ (the purpose is for f to have a side effect)

Code Capture

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

Parameter Capture

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

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

Function Definitions

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

Function Calls

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

If Statements

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

Types

- $\underline{\mathsf{type}\sigma}\ \underline{\mathsf{type}\tau} \longrightarrow \underline{\mathsf{type}\sigma}(\underline{\tau})\ \mathsf{snd}\ (\_, \_, s \to \underline{\sigma}(\underline{\tau}), \varepsilon, s)$
- $\underline{\mathsf{type}\sigma}\ \underline{\mathsf{product}}(\underline{\mathsf{type}\tau_1}, \ldots, \underline{\mathsf{type}\tau_n}) \longrightarrow$
  $\underline{\mathsf{type}\sigma}(\underline{\tau}_1, \ldots, \underline{\tau}_n)\ \mathsf{snd}\ (\_, \_, s \to \underline{\sigma}(\underline{\tau}_1, \ldots, \underline{\tau}_n), \varepsilon, s)$
- $\mathsf{colon}\ \underline{\mathsf{type}\sigma} \longrightarrow \mathsf{typer}\underline{\sigma}\ \mathsf{snd}\ (\_, u, s \to u, \varepsilon, s)$
- $\underline{\sigma}\ \mathsf{typer}\underline{\tau} \longrightarrow \underline{\tau}\ \mathsf{snd}\ (u, \_, s \to u, \varepsilon, s)$

4.3 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

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

Keywords

- $\text{fun} \mapsto (\mathsf{fun}, \varnothing)$
- $\text{if} \mapsto (\mathsf{if}, \varnothing)$

Types

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

5 Proving Correctness

In this section we show that valued reduction (§3.3) computes arithmetical expressions correctly. To do so, we show that valued reduction computes expressions and gives a result equal to a naïve algorithm.

But first, we must define the problem. We are given the following:

(1) a set X which is our *universe* (e.g. numbers),
(2) a set S of *operator symbols*,
(3) for every $\mathsf{s} \in \mathsf{S}$ a *priority* $\pi(\mathsf{s}) \in \overline{\mathbb{N}}$,
(4) for every $\mathsf{s} \in \mathsf{S}$ a function $f_{\mathsf{s}} \colon \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{X}$.

We also take two symbols L and R for the left and right parentheses. $\mathsf{X}, \mathsf{S}, \mathsf{L}, \mathsf{R}$ will also be our symbols for the β-reduction.

Expressions are defined by the following grammar:

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

Operators are assumed to be left-associative.

Let us define UNPAREN(ξ, i) to mean that the ith character in ξ is not within parentheses (this can be implemented easily). We define the EVAL algorithm (see algorithm 1) to evaluate strings in *expr*.

Algorithm 1 EVALuating arithmetical expressions

```
1.  function EVAL($\xi$)
2.      if ($\xi = \mathsf{x} \in \mathsf{X}$)
3.          return $\mathsf{x}$
4.      else if ($\xi[0] = \mathsf{L}$)
5.          $\xi$ is of the form $\mathsf{L}\xi'\mathsf{R}\zeta$
6.          return EVAL(EVAL($\xi'$)$\zeta$)
7.      else
            ▷ Find the greatest index with the smallest operator.
              Note that $\max_i \operatorname{argmin}_i f(i)$ is the maximal $i$ which minimizes $f$.
8.          $i := \max_i \operatorname{argmin}_i \{ \pi(\xi[i]) \mid \xi[i] \in \mathsf{S}, \text{UNPAREN}(\xi, i) \}$
9.          $\mathsf{s} := \xi[i]$
            ▷ $\xi' = \xi[: i-1]$, $\xi'' = \xi[i+1 :]$
10.         $\xi$ is of the form $\xi'\mathsf{s}\xi''$
11.         return $f_{\mathsf{s}}$(EVAL($\xi'$), EVAL($\xi''$))
12.     end if
13. end function
```

As opposed to pure valued (stateless) reduction, we adopt some conventions from stateful reduction. Since our string is a string in $(\{\mathsf{L}, \mathsf{R}\} \cup \mathsf{X} \cup \mathsf{S})^*$ and not $\{\mathsf{L}, \mathsf{R}, \mathsf{X}, \mathsf{S}\}^*$ (note the difference!), we must first convert the string into one admissable for β-reduction. To do so, we use stateful reduction and use the initial state and priority map which maps elements $\mathsf{x} \in \mathsf{X}$ to $\mathsf{X}_\infty(\mathsf{x})$, $\mathsf{S} \in \mathsf{S}$ to $\mathsf{S}_{\pi(\mathsf{s})}(f_\mathsf{s})$, L to L_∞, and R to R_0. Since our state doesn't change (our initial β-reducer will not affect the state) we can assume that our string has been mapped to its associated $\overline{\Pi}$-string from the start. That is, instead of strings like `1+2`, we can assume that our input strings are of the form $\mathsf{X}_\infty(1)\ \mathsf{S}_1(+)\ \mathsf{X}_\infty(2)$.

Secondly, we adopt the convention of having type terms, abstract terms, etc. Our base type will be $\Sigma = \{\mathsf{X}\}$, and our atomic abstract terms $\mathcal{A} = \{\mathsf{L}_0, \mathsf{R}_1, \mathsf{S}_1\}$ (printable terms are $\{\mathsf{L}, \mathsf{R}\} \cup \mathsf{X} \cup \mathsf{S}$, but as said before we assumed these are already altered by the state).

Our initial β-reducer will be

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

5.1 The Proof

We can rewrite the grammar *expr* with priorities as follows:

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

when considering an input for EVAL, we will of course neglect the priorities in the string. And when considering an input for β-reduction, we act as if each character is of the term indicated with the value in the string (i.e. $1_\infty +_1 2_\infty \in \mathit{expr}$ is $\mathsf{X}_\infty(1)\mathsf{S}_1(+)\mathsf{X}_\infty(2)$).

Let us define the set of *intermediate expressions* as

$$\begin{aligned}\mathit{intexpr} ::= \mathsf{X}_n \mid \mathsf{SX}_n\ \mathit{intexpr} \mid \mathsf{X}_n\ \mathsf{S}_m\ \mathit{intexpr} \mid \mathsf{L}_\infty\ \mathit{intexpr}\ \mathsf{R}_0\ \mathit{opintexpr}\\ \mid \mathsf{L}_\infty\ (\mathsf{SX}_n)^*\ \mathsf{RX}_0\ \mathit{opintexpr}\end{aligned}$$

$$\mathit{opintexpr} ::= \varepsilon \mid \mathsf{S}_n\ \mathit{intexpr}$$

where $m \le n \in \overline{\mathbb{N}}$.

Let β_0 be the normal derived β-reducer. And let β be the derived β-reducer without the rule that a single character's priority is changed to zero (instead it just keeps the character's priority instead). We define $\beta^*(\xi)$ to be the value of the full reduction of ξ under β (i.e. the string obtained by repeated applications of β until convergence). And $\beta_0^*(\xi)$ is defined similarly.

Definition 1. *A set of strings S is* closed under β-reductions *if when $\xi \in S$, then $\beta(\xi) \in S$ And S is* closed under β_0-reductions *if when $\xi \in S$, then $\beta_0(\xi) \in S$.*

Obviously if S is closed under β_0-reductions, it is closed under β-reductions.

Lemma 1. *intexpr is closed under β_0-reductions, and for $\xi \in$ intexpr, $\beta^*(\xi) = \mathsf{SX}_{n_1} \cdots \mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$ where $k \ge 0$ and $n_1 < \cdots < n_{k+1}$.*

Proof. We prove this by induction on the length and structure of ξ.

(**1**) For $\xi = \mathsf{X}_n$, this is trivial as ξ is a fixed-point of β.
(**2**) For $\xi = \mathsf{SX}_n\xi'$, if SX_n can be reduced with the first token in ξ' then we have the following cases:
  (a) $\xi' = \mathsf{X}_m$, then $\beta(\xi) = \mathsf{X}_m$ which is an intermediate expression and of the desired form.
  (b) $\xi' = \mathsf{SX}_m\xi''$, then $\beta(\xi) = \mathsf{SX}_m\xi''$ which is an intermediate expression, and since the length of this string is less than ξ, we can prove the second part of the lemma by induction.
  (c) $\xi' = \mathsf{X}_m\ \mathsf{S}_k\xi''$ with $\xi'' \in$ *intexpr*, in which case $\beta(\xi) = \mathsf{X}_m\mathsf{S}_k\xi''$, which is in *intexpr* (since $m \ge k$). And since the length of this string is one less than ξ, we prove the second part of the lemma by induction.

Otherwise $\beta(\xi) = \mathsf{SX}_n\beta(\xi')$, and since by induction $\beta(\xi') \in$ *intexpr* we have the first part of the lemma. For the second part, if ξ' is not a fixed point of β, $\beta(\xi')$'s length is less than ξ''s, and thus since $\beta^*(\xi) = \beta^*(\mathsf{SX}_n\beta(\xi'))$ we can proceed by induction. If ξ' is a fixed point of β, by our inductive assumption (the second part of the lemma), $\xi' = \mathsf{SX}_{n_1} \cdots \mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$. And so ξ is also a fixed point of β and has the desired form (since $n < n_1$ as otherwise we could reduce ξ).

(**3**) For $\xi = \mathsf{X}_n\mathsf{S}_m\xi'$, this can be reduced to $\beta(\xi) = \mathsf{SX}_m\xi'$ and we simply induct.
(**4**) For $\xi = \mathsf{L}_\infty\xi'\mathsf{R}_0\zeta$, note that since ξ' doesn't start with RX, L cannot be reduced with ξ'. Thus $\beta(\xi) = \mathsf{L}_\infty\beta(\xi'\mathsf{R}_0\zeta)$. If ξ' can be reduced, this is just $\mathsf{L}_\infty\beta(\xi')\mathsf{R}_0\zeta$ and so we simply induct.
Otherwise it is a fixed point and so $\xi' = \mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$ with $n_1 < \cdots < n_{k+1}$. Then $\beta(\xi) = \mathsf{L}_\infty\mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{RX}_0\zeta$ which is an intermediate expression whose length is shorter than ξ so we induct.
(**5**) For $\xi = \mathsf{L}_\infty\mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{RX}_0\zeta = \mathsf{L}_\infty\xi'\mathsf{RX}_0\zeta$, if ξ' can be reduced or is empty (because $\mathsf{L}_\infty\mathsf{RX}_0\zeta \to \mathsf{X}_\infty\zeta$), we simply induct. Otherwise $n_1 < \cdots < n_k$ and so $\beta(\xi) = \mathsf{L}_\infty\mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_{k-1}}\mathsf{RX}_0\zeta$, which is an intermediate expression whose length is shorter. ■

Notice that $\beta_0^*(\mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}) \in \mathsf{X}$ (this can be proved by simple induction), and so by the above lemma we get that for any intermediate expression ξ, $\beta_0^*(\xi) \in \mathsf{X}$ (since $\beta_0^*(\xi) = \beta_0^*(\beta^*(\xi))$).

Furthermore, from (**4**) we see that $\beta^*(\mathsf{L}_\infty\xi\mathsf{R}_0) = \mathsf{L}_\infty\beta^*(\xi)\mathsf{R}_0$. Since $\beta^*(\xi) = \mathsf{SX}_{n_1}\cdots\mathsf{SX}_{n_k}\mathsf{X}_{n_{k+1}}$, it is trivial to then see that $\beta^*(\mathsf{L}_\infty\xi\mathsf{R}_0) = \beta_0^*(\xi)_\infty$ (we can assign $\beta_0^*(\xi)$ a priority since as we said, it is in X). So we have proven:

Lemma 2. *Let $\xi \in$ intexpr, then $\beta^*(\mathsf{L}_\infty\xi\mathsf{R}_0) = \beta_0^*(\xi)_\infty$.*

We introduce some more notation. Let us write $\xi \to^* \xi'$ to mean that $\beta^n(\xi) = \xi'$ for some n and $\xi \to_0^* \xi'$ to mean that $\beta_0^n(\xi) = \xi'$ for some n. We write s_n for $\mathsf{S}_n(f_\mathsf{s})$, $\mathsf{s}_n[\mathsf{x}]$ for $\mathsf{SX}_n(\mathsf{x}, f_\mathsf{s})$, and x_n for $\mathsf{X}_n(\mathsf{x})$. And if $\xi \in$ *intexpr* we write $\pi(\xi)$ for the set of priorities in ξ, and $\pi_u(\xi)$ for the set of priorities in ξ of operators which are unparenthesized.

Lemma 3. *If $N \leq \pi_u(\xi)$ then $N \leq \pi_u(\beta(\xi))$ for $\xi \in$ intexpr. If the first inequality is strict, so is the second.*

Proof. We prove this by induction on the length of ξ.

(**1**) For $\xi = \mathsf{X}_n$ this is trivial.
(**2**) For $\xi = \mathsf{SX}_n\xi'$, if SX_n can be reduced with the first token of ξ', we simply note that it takes the second token's priority, which N is already less than. Otherwise $\beta(\xi) = \mathsf{SX}_n\beta(\xi')$ and so $\pi_u(\beta(\xi)) = \{n\} \cup \pi_u(\beta(\xi')) \geq N$ as required.
(**3**) For $\xi = \mathsf{X}_n\mathsf{S}_m\xi'$, $\beta(\xi) = \mathsf{SX}_m\xi'$ and this is trivial.
(**4**) For $\xi = \mathsf{L}_\infty\xi'\mathsf{R}_0\zeta$, a β-reduction gives a string of the form $\mathsf{L}_\infty\xi''\mathsf{RX}_0\zeta$ and so $\pi_u(\beta(\xi)) = \pi_u(\zeta) \geq N$.
(**5**) For $\xi = \mathsf{L}_\infty\xi'\mathsf{RX}_0\zeta$, a β-reduction either gives a string of the form $\mathsf{L}_\infty\xi''\mathsf{RX}_0\zeta$ as before, or $\mathsf{X}_\infty\zeta$. The first case was covered in the last point, the second case is simple: $N \leq \pi_u(\zeta)$ and certainly $N \leq \infty$.

Note that if we have a strict inequality, all of our arguments still hold. ■

From this and the previous lemma we can infer that if $n \leq \pi_u(\xi)$ then $n \leq \pi_u(\beta^*(\xi)) = \pi(\beta^*(\xi))$ (since $\beta^*(\xi)$ cannot have parentheses). So $\beta^*(\xi)$ is of the form

$$\mathsf{SX}_{n_1} \cdots \mathsf{SX}_{n_k} \mathsf{X}_{n_{k+1}}, \qquad n \leq n_1 < \cdots < n_{k+1}$$

Lemma 4. *Let ξ be an intermediate expression, and $n \leq \pi_u(\xi)$. Then $\beta^*(\xi \mathsf{s}_n) = \mathsf{s}_n[\beta_0^*(\xi)]$.*

Proof. As explained above,

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

for $n \leq n_1 < \cdots < n_k < n_{k+1}$. A β-reduction gives

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

further β-reductions give (instead of f_{s_i} we write f_i)

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

And it isn't hard to show that the β_0^*-reduction of $\mathsf{s}^1_{n_1}[\mathsf{x}^1] \cdots \mathsf{s}^k_{n_k}[\mathsf{x}^k]\mathsf{x}^{k+1}_{n_{k+1}}$ is precisely $f_1(\mathsf{x}^1, f_2(\mathsf{x}^2, \dots))$ as required. ■

Lemma 5. *If $\xi \in$ intexpr, $n < \pi_u(\xi)$. Then $\beta_0^*(\mathsf{s}_n[\mathsf{x}]\xi) = f_\mathsf{s}(\mathsf{x}, \beta_0^*(\xi))$.*

Proof. Since $n < \pi(\xi)$, $\mathsf{s}_n[\mathsf{x}]$ cannot be reduced with ξ, and since $n < \pi_u(\beta^k(\xi))$ for all k. Since $\mathsf{s}_n[\mathsf{x}]$ can only reduce with unparenthesized expressions (that is, there is no rule reducing SX and L), this means that $\mathsf{s}_n[\mathsf{x}]$ cannot be reduced until ξ is reduced fully. So

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

where $n < n_1 < \cdots < n_{k+1}$. A β_0-reduction just changes the priority of x^{k+1} to 0. Then β_0^*-reducing gives as before

$$f_\mathsf{s}(\mathsf{x}, f_{\mathsf{s}_1}(\mathsf{x}^1, f_{\mathsf{s}_2}(\mathsf{x}^2, \dots))) = f_\mathsf{s}(\mathsf{x}, \beta_0^*(\xi)) \qquad ■$$

Theorem 1. *For $\xi \in$ expr, $\textsc{eval}(\xi) = \beta_0^*(\xi)$.*

Proof. We induct on the length of ξ.

(**1**) If $\xi = \mathsf{X}_m$ this is trivial.
(**2**) If $\xi = \mathsf{L}_\infty \xi' \mathsf{R}_0 \zeta$ then $\xi \to^*= \beta_0^*(\xi')\zeta$ which inductively is equal to $\textsc{eval}(\xi')\zeta$. And inductively $\beta_0^*(\textsc{eval}(\xi')\zeta) = \textsc{eval}(\textsc{eval}(\xi')\zeta)$ which is equal to $\textsc{eval}(\xi)$ by definition.
(**3**) Otherwise let n be the lowest priority of operators in ξ which are unparenthesized, i.e. $n = \min \pi_u(\xi)$. Then $\xi = \xi' \mathsf{s}_n \xi''$ where s is the last unparenthesized operator with a priority of n, so $\xi', \xi'' \in$ *intexpr* and $n \leq \pi_u(\xi')$ and $n < \pi_u(\xi'')$. This means that

$$\xi \to^* \mathsf{s}_n[\beta_0^*(\xi')]\xi'' \to_0^* f_\mathsf{s}(\beta_0^*(\xi'), \beta_0^*(\xi''))$$

Inductively $\beta_0^*(\xi') = \textsc{eval}(\xi')$ and $\beta_0^*(\xi'') = \textsc{eval}(\xi'')$, meaning this is equal to $f_\mathsf{s}(\textsc{eval}(\xi'), \textsc{eval}(\xi''))$. By definition this is $\textsc{eval}(\xi)$ as required. ■

6 Producing Parse Trees

Iterative reduction can be used to just parse, instead of parsing and interpreting simultaneously. This has its benefits, as separating parsing and interpretation allows for the inspection of code before running it, which can help alert the programmer to syntax errors. Furthermore, this means iterative reduction can be used in the classical interpreter and compiler pipeline, as it can be used as a parser.

To do so, we define $\mathcal{T}_{ree}$, the set of trees, as follows:

$$\mathcal{T}_{ree} ::= \Sigma \mid \Sigma(\mathcal{T}_{ree}) \mid \Sigma(\mathcal{T}_{ree})(\mathcal{T}_{ree}) \mid \Sigma(\mathcal{T}_{ree})(\mathcal{T}_{ree})(\mathcal{T}_{ree}) \mid \cdots$$

So a tree is simply a multi-branched tree whose nodes are elements of our alphabet Σ. Then we define an *initial β-reducer* as

$$\widehat{\beta}\colon \Sigma \times \Sigma \rightharpoonup \Sigma \times (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}} \to \overline{\mathbb{Z}}) \times (\mathcal{T}_{ree} \times \mathcal{T}_{ree} \to \mathcal{T}_{ree})$$

This extends to a *derived β-reducer*

$$\beta\colon (\overline{\Sigma} \times \mathcal{T}_{ree})^* \longrightarrow (\overline{\Sigma} \times \mathcal{T}_{ree})^*$$

where $\beta(\xi)$ is defined as

(1) If $\xi = \sigma^1_n(t)\sigma^2_m(s)\xi'$ where $n \geq m$ and $\widehat{\beta}(\sigma^1, \sigma^2) = (\tau, \rho, f)$ is defined then $\beta(\xi) = \tau_{\rho(n,m)}(f(t,s))\xi'$.
(2) Otherwise $\beta(\xi) = \sigma^1_n(t)\beta(\sigma^2_m(s)\xi')$.

We can also add the case that if ξ is a single token then $\beta(\xi) = \xi_0$.

For example, for arithmetic we can define

- $\mathsf{Num}\ \mathsf{Op} \to \mathsf{OpNum}, \mathsf{snd}, (t, f \mapsto f(t))$
- $\mathsf{OpNum}\ \mathsf{Num} \to \mathsf{Num}, \mathsf{snd}, (f(t), s \mapsto f(t)(s))$
- $\mathsf{OpNum}\ \mathsf{OpNum} \to \mathsf{OpNum}, \mathsf{snd}, \big(f(t), g(s) \mapsto g(f(t)(s))\big)$
- $\mathsf{Num}\ \mathsf{Rparen} \to \mathsf{RparenNum}, \mathsf{snd}, (t, \_ \mapsto t)$
- $\mathsf{OpNum}\ \mathsf{RparenNum} \to \mathsf{RparenNum}, \mathsf{snd}, (f(t), s \mapsto f(t)(s))$
- $\mathsf{Lparen}\ \mathsf{RparenNum} \to \mathsf{Num}, \mathsf{fst}, (\_, t \mapsto t)$

So for example, `1 + 2 * (3 + 4)` is then parsed as

$$\begin{aligned}
\boxed{1}_\infty \boxed{+}_1 \boxed{2}_\infty \boxed{*}_2 \boxed{(}_\infty \boxed{3}_\infty \boxed{+}_1 \boxed{4}_\infty \boxed{)}_0 &\to^* \boxed{+(1)}_1 \boxed{*(2)}_2 \boxed{(}_\infty \boxed{+(3)}_1 \boxed{4}_0 \\
&\to \boxed{+(1)}_1 \boxed{*(2)}_2 \boxed{(}_\infty \boxed{+(3)(4)}_0 \\
&\to \boxed{+(1)}_1 \boxed{*(2)}_2 \boxed{+(3)(4)}_\infty \\
&\to^* \boxed{+(1)}_1 \boxed{*(2)(+(3)(4))}_0 \\
&\to \boxed{+(1)(*(2)(+(3)(4)))}_0
\end{aligned}$$

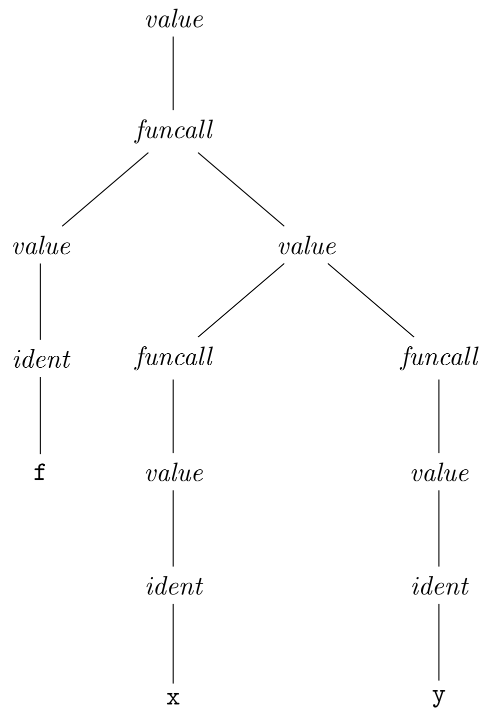
7 Comparing Implementations

In this section we compare the runtimes of both implementations (one using iterative reduction and one using menhir[2]). A few notes beforehand:

- The iterative reduction implementation has more features than the menhir implementation, this should not severely impact the comparisons and we will of course not be comparing the implementations on code which the menhir implementation does not support.
- ILang supports left composition. Meaning code of the form `f x y` is read as `(f x) y`. We found no way of supporting this with menhir. This is because our grammar for the menhir implementation contains the following fragment:

  $$\textit{value} ::= \textit{const} \mid \textit{ident} \mid \textit{funcall}, \qquad \textit{funcall} ::= \textit{value}\ \textit{value}$$

  So `f x y` can be parsed as `f (x y)` instead:



7.1 Currying

We begin with a simple program to demonstrate currying. Currying is the action of taking a function $f\colon X \times Y \longrightarrow Z$ and converting it to a function $c(f)\colon X \longrightarrow (Y \longrightarrow Z)$ where $c(f)(x)\colon y \mapsto f(x,y)$. Our ILang code is as follows:

```
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);
```

Comparing the time it takes both implementations to run this 300 times, we see that it takes iterative reduction 0.0040886 seconds, opposed to the classical implementation which takes 0.0036926 seconds.

7.2 Arithmetical Expressions

We can generate random arithmetical expressions using the following algorithm:

```
   ▷ Create an expression with $\ell$ subterms, with a parentheses depth of $d$,
     where numbers are chosen from $[0, n)$.
1. function GENERATE-EXPRESSION($\ell, d, n$)
2.     if ($d = 0$) return an expression with $\ell$ randomly chosen numbers in
                   $[0, n)$ and $\ell - 1$ random operators in $\{+, -, \times, \div\}$.
3.     return $\big($GENERATE-EXPRESSION$(\ell, d-1, n)\big) \circ_1 \cdots \circ_{\ell-1}$
                                  $\big($GENERATE-EXPRESSION$(\ell, d-1, n)\big)$
       where $\circ_i \in \{+, -, \times, \div\}$ are chosen randomly
4. end function
```

We then run ILang code of the form `let x = <expr>;` where `<expr>` is generated by the above algorithm. We measure how long it takes to run this code five times with expressions generated by GENERATE-EXPRESSION$(n, n, 100)$ as n varies (figure 1) and expressions generated by GENERATE-EXPRESSION$(n, 1, 100)$ as n varies (figure 2).

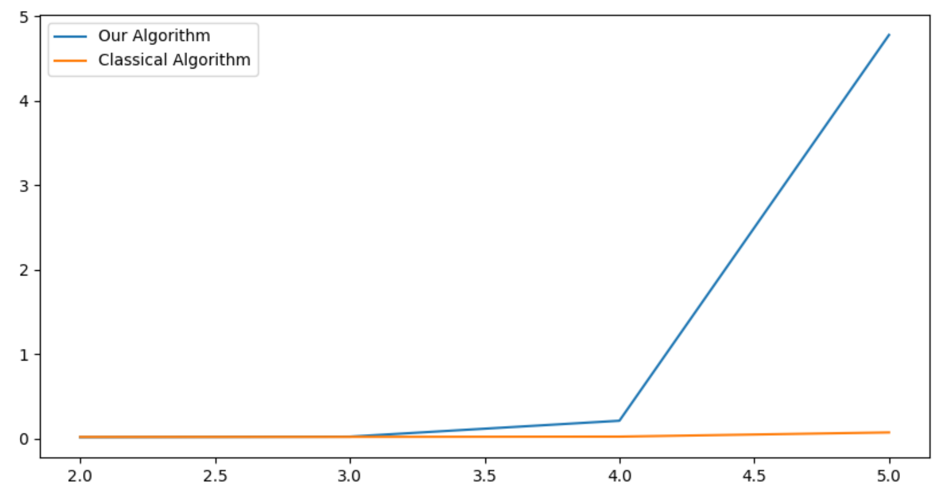


Fig. 1. Time to compute $\textsc{Generate-Expression}(n, n)$

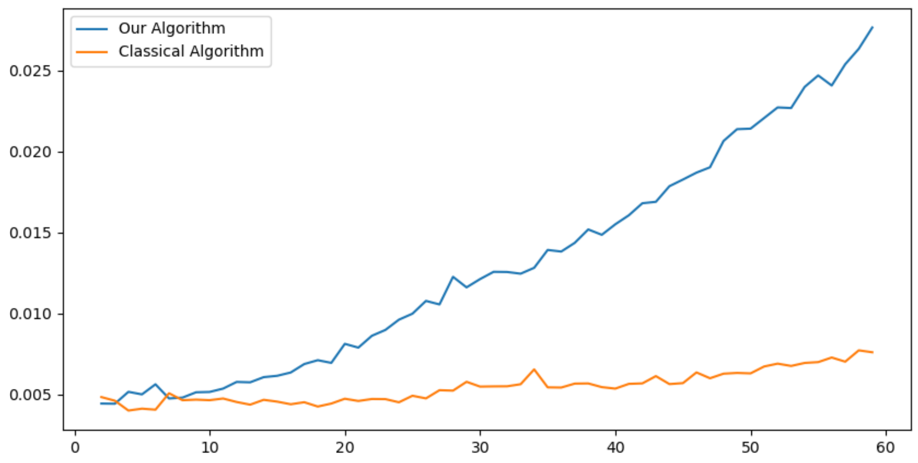


Fig. 2. Time to compute $\textsc{Generate-Expression}(n, 1)$

7.3 Fibonacci

The fibonacci sequence is a linear recurrence defined by $F_n = F_{n-1} + F_{n-2}$ for $n > 1$ and $F_1 = F_0 = 1$. We wrote a script to compute the nth fibonacci number recursively in ILang:

```
fun fib (x) {
    if (x < 2) {
        1
    }{
        (fib (x-1)) + (fib (x-2))
    }
}
let x = (fib n);
_prim_print x
```

Running each implementation 5 times for each x, we get the following graph in figure 3 (as x varies):

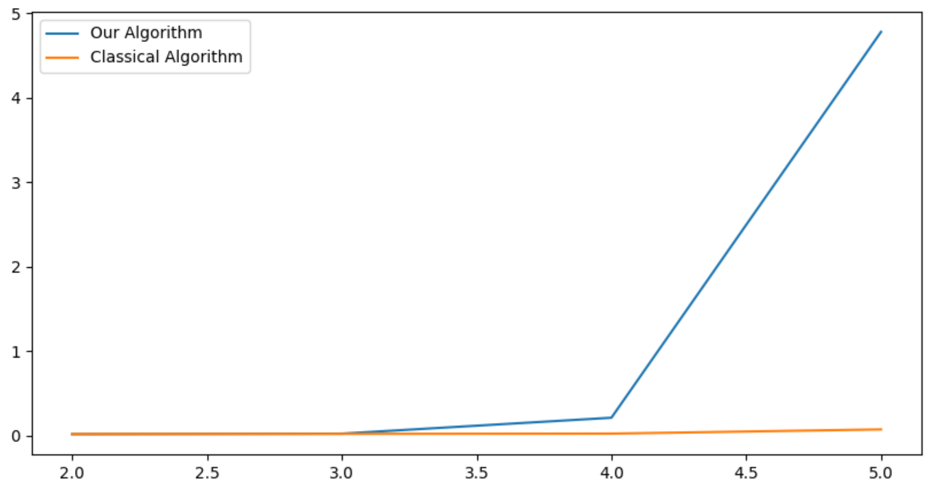


Fig. 3. Time to compute `fib`(x)

Our implementation takes approximately $\sim ab^x$ seconds to run where $a = 0.000138585, b = 1.64685$, and the menhir-based interpreter takes $\sim ab^x$ seconds where $a = 0.00000362553, b = 1.5809$.

8 Conclusion

We can see that iterative reduction is significantly slower than the menhir-based interpreter. This can be for various reasons:

(**1**) Iterative reduction is an $O(n^2)$-time algorithm (when no reduction produces a string of printable tokens, and every reduction reduces two tokens to one). This is because a string of the form $\mathsf{s}^1_1\mathsf{s}^2_2\cdots\mathsf{s}^n_n$ will take n applications of the β-reducer, each application taking $n-k$ iterations (since β is recursive). This can be made more efficient: instead of starting the next β-reduction at the beginning of a string, it can be started at the token before the one just reduced. Meaning once s^{n-1} and s^n are reduced to give x^{n-1}, we can begin the next iteration of the β-reducer at $\mathsf{s}^{n-2}\mathsf{x}^{n-1}$. Unfortunately, implementing this did not lead to substantial increases in efficiency.

(**2**) When handling function calls, iterative reduction simply re-reduces the string in the definition of the function. This requires it to parse and interpret the function again. But classically, since the function definition has already been parsed and a parse tree created, further parsing of the function is not created. This means that classical methods handle function calls better than iterative reduction, but on the other hand iterative reduction can handle macro-like programming better. For example, if one were to change the state to map `(` to lbrace, the menhir-based interpreter would not be able to reparse the function to reinterpret `(`. But this naturally happens with iterative reduction, for better or worse.

(**3**) The standard methods of parsing code are well-studied and optimized. Menhir does not naively parse code, it is an optimized tool. Thus comparing a naive implementation of iterative reduction to menhir is not a fair comparison.

References

[1] A. Aho et al. *Compilers: Principles, Techniques, and Tools*. Pearson, Addison Wesley, 2006.

[2] F. Pottier and Y. Régis-Gianis. *Menhir*. 2010. URL: `https://gallium.inria.fr/~fpottier/menhir/` (visited on 12/01/2024).