SNESL formalization Level-1

Dandan Xue

October 3, 2017

Abstract

TBD

Contents

1 Introduction **2**
1.1 Nested data parallelism 2
1.2 NESL 2
1.3 Work-depth cost model 2
1.4 SNESL 2
1.4.1 Type system 2
1.4.2 Syntax 2
1.4.3 Semantics 3
1.4.4 Cost model 3
1.5 Mathematical background and notations 3

2 Implementation **4**
2.1 SVCODE Syntax 4
2.2 SVCODE semantics 5
2.3 Xducers 6
2.3.1 SVCODE Dataflow 6
2.3.2 General semantics 7
2.3.3 Block semantics 8
2.4 SVCODE determinism 10
2.5 Streaming interpreter 13
2.6 Recursion 13

1 Introduction

1.1 Nested data parallelism

1.2 NESL

1.3 Work-depth cost model

1.4 SNESL

1.4.1 Type system

$$\tau ::= \mathbf{int} \mid \{\tau_1\}$$

Type environment $\Gamma = [x_1 \mapsto \tau_1, ..., x_i \mapsto \tau_i]$.

- Expression typing rules:

  Judgment $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\frac{\phi : (\tau_1, ..., \tau_k) \to \tau}{\Gamma \vdash \phi(x_1, ..., x_k) : \tau}\,((\Gamma(x_i) = \tau_i)_{i=1}^{k})$$

$$\frac{[x \mapsto \tau_1, (x_i \mapsto \mathbf{int})_{i=1}^{j}] \vdash e : \tau}{\Gamma \vdash \{e : x\ \mathbf{in}\ y\ \mathbf{using}\ x_1, ..., x_j\} : \{\tau\}}\,(\Gamma(y) = \{\tau_1\}, (\Gamma(x_i) = \mathbf{int})_{i=1}^{j})$$

- **Judgment** $\boxed{\phi : (\tau_1, ..., \tau_k) \to \tau}$

$$\frac{}{\mathbf{const}_n : () \to \mathbf{int}} \qquad \frac{}{\mathbf{iota} : (\mathbf{int}) \to \{\mathbf{int}\}} \qquad \frac{}{\mathbf{plus} : (\mathbf{int}, \mathbf{int}) \to \mathbf{int}}$$

- Value typing rules:

  Judgment $\boxed{v : \tau}$

$$\frac{}{n : \mathbf{int}} \qquad \frac{(v_i : \tau)_{i=1}^{k}}{\{v_1, ..., v_k\} : \{\tau\}}$$

1.4.2 Syntax

SNESL Expressions:

$$e ::= x \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \phi(x_1, ..., x_k) \mid \{e : x\ \mathbf{in}\ y\ \mathbf{using}\ x_1, ..., x_j\}$$

$$\phi = \mathbf{const}_n \mid \mathbf{iota} \mid \mathbf{plus}$$

SNESL values:

$$n \in \mathbb{Z}$$

$$v ::= n \mid \{v_1, ..., v_k\}$$

1.4.3 Semantics

$\rho = [x_1 \mapsto v_1, ..., x_i \mapsto v_i]$

- **Judgment** $\boxed{\rho \vdash e \downarrow v}$

$$\frac{}{\rho \vdash x \downarrow v}\ (\rho(x) = v) \qquad \frac{\rho \vdash e_1 \downarrow v_1 \qquad \rho[x \mapsto v_1] \vdash e_2 \downarrow v}{\rho \vdash \mathbf{let}\ e_1 = x\ \mathbf{in}\ e_2 \downarrow v}$$

$$\frac{\phi(v_1, ..., v_k) \downarrow v}{\rho \vdash \phi(x_1, ..., x_k) \downarrow v}\ ((\rho(x_i) = v_i)_{i=1}^{k})$$

$$\frac{([x \mapsto v_i, (x_i \mapsto n_i)_{i=1}^{j}] \vdash e \downarrow v_i')_{i=1}^{k}}{\rho \vdash \{e : x\ \mathbf{in}\ y\ \mathbf{using}\ x_1, ..., x_j\} \downarrow \{v_1', ..., v_k'\}}\ (\rho(y) = \{v_1, ..., v_k\}, (\rho(x_i) = n_i)_{i=1}^{j})$$

- **Judgment** $\boxed{\phi(v_1, ..., v_k) \downarrow v}$

$$\frac{}{\mathbf{const}_n() \downarrow n} \qquad \frac{}{\mathbf{iota}(n) \downarrow \{0, 1, ..., n-1\}}\ (n \geq 0)$$

$$\frac{}{\mathbf{plus}(n_1, n_2) \downarrow n_3}\ (n_3 = n_1 + n_2)$$

1.4.4 Cost model

1.5 Mathematical background and notations

- Set difference:
  For two sets A and B,
  $$A \setminus B = \{s | s \in A \land s \notin B\}$$
  It is easy to prove the following properties:
  - For any three sets A, B and C:
  $$(A \setminus B) \cap C = (A \cap C) \setminus B = A \cap (C \setminus B) \tag{1.1}$$
  - For two sets A and B,
  $$A \cap B = \emptyset \Leftrightarrow A \setminus B = A \tag{1.2}$$

2 Implementation

In [Mad13] a streaming target language for a minimal SNESL was defined. With trivial changes in the instruction set, this language, named as SVCODE (streaming VCODE), has been implemented on a multicore system in [Mad16]; the various experiment results have demonstrated single-core performance similar to sequential C code for some simple text-processing tasks as well as the potential for further performance improvment by scheduling opitmization and code analysis.

In this thesis, we put emphasis on the formalization of this low-level language's semantics. Also, to support recursion in the high-level language at the same time preserving the cost, non-trivial extension of this language is needed.

2.1 SVCODE Syntax

The data or *streams* that an SVCODE program computes are basically vectors of consts. For our minimal language, a primitive stream $\vec{a}$ can be a vector of booleans, integers or units, as the following grammar shows:

$$
\begin{aligned}
b &\in \mathbb{B} = \{\mathtt{T}, \mathtt{F}\} \\
a &::= n \mid b \mid () \\
\vec{b} &= \langle b_1, ..., b_i \rangle \\
\vec{c} &= \langle (), ..., () \rangle \\
\vec{a} &= \langle a_1, ..., a_i \rangle
\end{aligned}
$$

The grammar of SVCODE is given in Figure 2.1.

$$
\begin{array}{lll}
p & ::= \epsilon & \\
 & \mid s := \psi(s_1, ..., s_k) & (s \notin \{s_1, ..., s_k\}) \\
 & \mid S_{out} := \mathtt{WithCtrl}(s, S_{in}, p_1) & (\mathtt{fv}(p_1) \subseteq S_{in}, S_{out} \subseteq \mathtt{dv}(p_1)) \\
 & \mid p_1; p_2 & (\mathtt{dv}(p_1) \cap \mathtt{dv}(p_2) = \emptyset) \\
 & & \\
s & ::= 0 \mid 1 \; ... \in \mathbf{SId} = \mathbb{N} & \text{(stream ids)} \\
 & & \\
\psi & ::= \mathtt{Const_a} \mid \mathtt{ToFlags} \mid \mathtt{Usum} \mid \mathtt{MapTwo}_{\oplus} \mid \mathtt{ScanPlus}_{n_0} \mid \mathtt{Distr} & \text{(Xducers)} \\
\oplus & ::= + \mid - \mid \times \mid \div \mid \% \mid \leq \mid ... & \text{(binary operations)} \\
 & & \\
S & ::= \{s_1, ..., s_i\} \in \mathbb{S} & \text{(a set of stream ids)}
\end{array}
$$

Figure 2.1: Grammar of SVCODE

The instructions in SVCODE that perform the computation directly on primitive streams are stream definitions in the form

$$s := \psi(s_1, ..., s_k)$$

where ψ is a primitive function, or a $Xducer$(transducer), taking stream $s_1, ..., s_k$ as parameters and returning s.

The only essential control struture in SVCODE is the instruction

$$S_{out} := \texttt{WithCtrl}(s, S_{in}, p_1)$$

which may or may not execute a piece of SVCODE program p_1, but always defines a bunch of stream variables S_{out}. Discussions about this `WithCtrl` instruction will occur again and again throughout the thesis as it plays a significant role in dealing with most of the issues we are deeply concerned, including cost model correctness and dynamic unfolding of recursive functions.

The function `dv` returns the set of defined variables of a given SVCODE program.

$$\begin{aligned}
&\texttt{dv}(\epsilon) &&= \emptyset\\
&\texttt{dv}(s := \psi(s_1, ..., s_k)) &&= \{s\}\\
&\texttt{dv}(S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)) &&= S_{out}\\
&\texttt{dv}(p_1; p_2) &&= \texttt{dv}(p_1) \cup \texttt{dv}(p_2)
\end{aligned}$$

Correspondingly, `fv` returns the free variables set.

$$\begin{aligned}
&\texttt{fv}(\epsilon) &&= \emptyset\\
&\texttt{fv}(s := \psi(s_1, ..., s_i)) &&= \{s_1, ..., s_k\}\\
&\texttt{fv}(S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)) &&= \{s_c\} \cup S_{in}\\
&\texttt{fv}(p_1; p_2) &&= \texttt{fv}(p_1) \cup (\texttt{fv}(p_2) - \texttt{dv}(p_1))
\end{aligned}$$

An immediate property of this language is that the defined variables of a well-formed SVCODE program are always $fresh$. In other words, there is no overlapping between the free variables and the newly generated ones.

Lemma 2.1. $\texttt{fv}(p) \cap \texttt{dv}(p) = \emptyset$.

The proof is straightward by induction on the syntax of p.

2.2 SVCODE semantics

Before showing the semantics, we first introduce some notations and operations about streams for convenience.

Notation 2.2. *Let* $\langle a_0 | \vec{a} \rangle$ *denote a non-empty stream* $\langle a_0, a_1, ..., a_i \rangle$ *for some* $\vec{a} = \langle a_1, ..., a_i \rangle$.

Notation 2.3 (Stream concatenation). $\langle a_1, ..., a_i \rangle {+\!\!+} \langle a'_1, ..., a'_j \rangle = \langle a_1, ..., a_i, a'_1, ..., a'_j \rangle$

The operational semantics of SVCODE is given in Figure 2.2. The runtime environment or store σ is a map from stream variables to vectors:

$$\sigma = [s_1 \mapsto \vec{a}_1, ..., s_i \mapsto \vec{a}_i]$$

The *control stream* $\vec{c}$, which is basically a vector of units representing an unary number, indicates the *parallel degree* of the computation. The role of control stream will become

Judgment $\boxed{\langle p, \sigma \rangle \Downarrow^{\vec{c}} \sigma'}$

$$\text{P-EMPTY: } \frac{}{\langle \epsilon, \sigma \rangle \Downarrow^{\vec{c}} \sigma}$$

$$\text{P-XDUCER : } \frac{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}}{\langle s := \psi(s_1, ..., s_k), \sigma \rangle \Downarrow^{\vec{c}} \sigma[s \mapsto \vec{a}]} \; ((\sigma(s_i) = \vec{a}_i)_{i=1}^{k})$$

$$\text{P-WC-EMP: } \frac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma \rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{l}]} \begin{pmatrix} \forall s \in \{s_c\} \cup S_{in}.\sigma(s) = \langle\rangle \\ S_{out} = \{s_1, ..., s_l\} \end{pmatrix}$$

$$\text{P-WC-NONEMP : } \frac{\langle p_1, \sigma \rangle \Downarrow^{\vec{c}_1} \sigma''}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma \rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma''(s_i))_{i=1}^{l}]} \begin{pmatrix} \sigma(s_c) = \vec{c}_1 \neq \langle\rangle \\ S_{out} = \{s_1, ..., s_l\} \end{pmatrix}$$

$$\text{P-SEQ : } \frac{\langle p_1, \sigma \rangle \Downarrow^{\vec{c}} \sigma'' \qquad \langle p_2, \sigma'' \rangle \Downarrow^{\vec{c}} \sigma'}{\langle p_1; p_2, \sigma \rangle \Downarrow^{\vec{c}} \sigma'}$$

Figure 2.2: SVCODE semantics

much clearer when we come to the semantics of Xducers. It is worth noting that only in the rule P-WC-NONEMP the control stream has a chance to get changed.

The rule P-EMPTY is trivial, empty program doing nothing on the store.

The rule P-XDUCER adds the store a new stream binding where the bound vector is generated by a specific Xducer with input streams. The detailed semantics of Xducers are defined in the next subsection.

The rules P-WC-EMP and P-WC-NONEMP together show two possibilities for interpreting a `WithCtrl` instruction:

- if the new control stream s_c as well as the streams in S_{in}, which includes the free variables of p_1, are all empty, then just bind empty vectors to the streams in S_{out}, which are part of the defined streams of p_1.
- otherwise execute the code of p_1 as usual under the new control stream, ending in the store σ''; then copy the bindings of S_{out} from σ'' to the initial store.

The new control stream is crucial here, because it decides whether or not to execute p_1, which is the key to avoiding infinite unfolding of recursive funtions. For an eager interpreter of SVCODE, if we count one stream definition as one step, then this skip guarantees the low-level step cost agrees on the high-level one. Also, skipping a certain piece of code should help improve the efficiency of execution.

2.3 Xducers

2.3.1 SVCODE Dataflow

Transducers or *Xducers* are the primitive computing functions on streams in SVCODE. Each Xducer consumes a number of streams and transforms them into another.

As we can see from the side condition of the rule P-XDUCER, a stream can be consumed only after it has been produced. Thus, the dataflow among an SVCODE program constructs a DAG (directed acyclic graph), where each Xducer performs one node. Figure 2.3 shows an example program with its DAG in Figure 2.4.

```
        S1 := Const_3();
        S2 := ToFlags(S1);
        S3 := Usum(S2);
        [S4] := WithCtrl(S3,[], S4 := Const_1();)
        S5 := ScanPlus(S2, S4);
```

Figure 2.3: A small SVCODE program

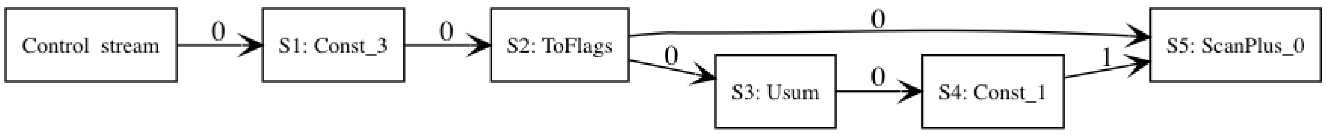


Figure 2.4: Dataflow DAG for the code in Figure 2.3. Note that, for simplicity, the control stream is added as an explicit supplier only to Xducer $\texttt{Const}_\texttt{a}$.

When we talk about two Xducers A and B connected by an arrow from A to B in the DAG, we call A a *producer* or a *supplier* to B, and B a *consumer* or a *client* of A. As an Xducer can have multiple suppliers, we distinguish these suppliers by giving each of them an index, called a *channel number*. In Figure 2.4, the channel number is labeled above each edge. For example, the Xducer S2 has two clients, S3 and S5, for both of whom it is the No.0 channel; Xducer S5 has two suppliers: S2 the No.0 channel and S3 the No.1.

2.3.2 General semantics

The semantics of Xducers are abstracted into two levels: the *general* level and the *block* level. The general level summarizes the comman property that all Xducers share, and the block level describes the specific behavior of each Xducer.

Figure 2.5 shows the semantics at the general level.

Judgment $\boxed{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}}$

$$\text{P-X-LOOP}: \frac{\psi(\vec{a}_{11}, ..., \vec{a}_{k1}) \downarrow \vec{a}_{01} \qquad \psi(\vec{a}_{12}, ..., \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle\rangle|\vec{c}_0\rangle} \vec{a}_0} \; ((\vec{a}_{i1} +\!\!+ \vec{a}_{i2} = \vec{a}_i)_{i=0}^{k})$$

$$\text{P-X-TERMI}: \frac{}{\psi(\langle\rangle_1, ..., \langle\rangle_k) \Downarrow^{\langle\rangle} \langle\rangle} \; 1$$

Figure 2.5: Semantics of SVCODE transducers

There are only two rules for the general semantics. They together say that the output stream is computed in a "loop" fashion, where the iteration uses specific block semantics

[1]For notational convenience, in this thesis we add subscripts to a sequence of constants, such as $\langle\rangle, \texttt{F}, 1$, to denote the total number of these constants.

of the Xducer and the number of iteration is the unary number that the control stream represents, i.e., the length of the control stream. In the parallel setting, we prefer to call this iteration a *block*. Recall the control stream is a representation of the parallel degree of the computation, then a block consumes exact one degree. We note that all these blocks are data-independent, which means they can be performed in parallel. Now it is clear that the control stream indeed carrys the theoretical maximum number of processors we need to execute the computation most efficiently (if the computation within the block can not be parallelized further)(???).

2.3.3 Block semantics

After abstracting the general semantics, the remaining work of formalizing the specific semantics of Xducers within a block becomes relatively clear and easy. The block semantics are defined in Figure 2.6.

Judgment $\boxed{\psi(\vec{a}_1, ..., \vec{a}_k) \downarrow \vec{a}}$

P-CONST: $\dfrac{}{\texttt{Const}_{\texttt{a}}() \downarrow \langle a \rangle}$ P-TOFLAGS: $\dfrac{}{\texttt{ToFlags}(\langle n \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle}$

P-MAPTWO : $\dfrac{}{\texttt{MapTwo}_{\oplus}(\langle n_1 \rangle, \langle n_2 \rangle) \downarrow \langle n_3 \rangle} \; (n_3 = n_1 \oplus n_2)$

P-USUMF : $\dfrac{\texttt{Usum}(\vec{b}) \downarrow \vec{a}}{\texttt{Usum}(\langle \texttt{F}|\vec{b} \rangle) \downarrow \langle () | \vec{a} \rangle}$ P-USUMT: $\dfrac{}{\texttt{Usum}(\langle \texttt{T} \rangle) \downarrow \langle \rangle}$

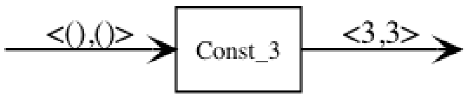
P-SCANF : $\dfrac{\texttt{ScanPlus}_{n_0+n}(\vec{b}, \vec{a}) \downarrow \vec{a}'}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F}|\vec{b} \rangle, \langle n|\vec{a} \rangle) \downarrow \langle n_0 | \vec{a}' \rangle}$ P-SCANT: $\dfrac{}{\texttt{ScanPlus}_{n_0}(\langle \texttt{T} \rangle, \langle \rangle) \downarrow \langle \rangle}$

P-DISTRF : $\dfrac{\texttt{Distr}(\vec{b}, \langle n \rangle) \downarrow \vec{a}}{\texttt{Distr}(\langle \texttt{F}|\vec{b} \rangle, \langle n \rangle) \downarrow \langle n | \vec{a} \rangle}$ P-DISTRT: $\dfrac{}{\texttt{Distr}(\langle \texttt{T} \rangle, \langle n \rangle) \downarrow \langle \rangle}$

Figure 2.6: Semantics of transducer blocks

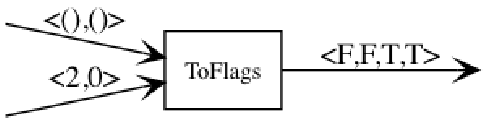
– $\texttt{Const}_{\texttt{a}}()$ outputs the const a until the control stream reaches EOS.

Example 2.1. $\texttt{Const}_3()$ with control stream $\vec{c} = \langle (), () \rangle$:



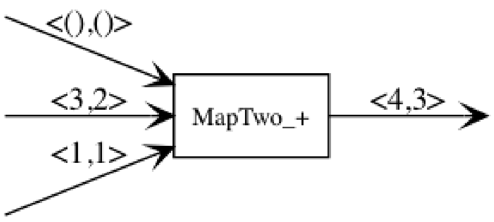
– $\texttt{ToFlags}(\langle n \rangle)$ first outputs n Fs, then one T.

Example 2.2. $\texttt{ToFlags}(\langle 2, 0 \rangle)$:



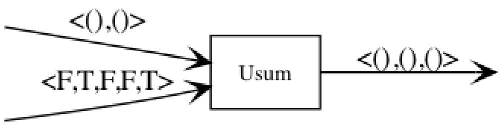
– `MapTwo`$_\oplus(\langle n_1\rangle, \langle n_2\rangle)$ outputs the binary operating result of $\oplus$ on n_1 with n_2.

Example 2.3. `MapTwo`$_+(\langle 3, 2\rangle, \langle 1, 1\rangle)$:



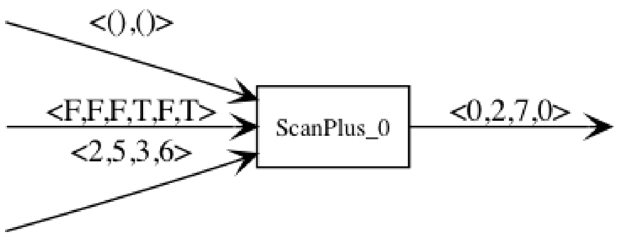
– `Usum`$(\vec{b})$ transforms an `F` to a unit, or a `T` to nothing. It is the only Xducer that can generate a unit vector, so it is mainly used when we need to replace the control stream.

Example 2.4. `Usum`$(\langle$`F, T, F, F, T`$\rangle)$:



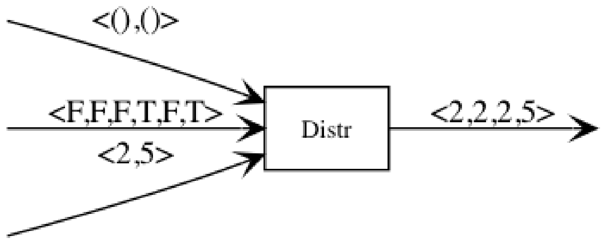
– `ScanPlus`$_{n_0}(\vec{b}, \vec{a})$ performs an exclusive scan of the binary operation plus on $\vec{a}$, segmented by $\vec{b}$, with a starting element n_0.

Example 2.5. `ScanPlus`$_0(\langle$`F, F, F, T, F, T`$\rangle, \langle 2, 5, 3, 6\rangle)$



– `Distr`$(\vec{b}, \langle n\rangle)$ replictes the const n u times where u is the unary number segmented by b.

Example 2.6. `Distr`$(\langle$`F, F, F, T, F, T`$\rangle, \langle$`2, 5`$\rangle)$



As we have discussed before, we consider a block as the minimum computing unit assigned to a single processor. This is reasonable for Xducers such as $\texttt{Const}_\texttt{a}$ and $\texttt{MapTwo}_\oplus$, because they are already sequential at the block level.

However, some other Xducers, such as `Usum`, can be parallelized further inside a block. As we extend the language with more Xducers, we could find that computations on unary numbers within blocks are common, which is mainly due to the value represenation strategy we use, but also more difficult to be regularized. For the scope of this thesis, the block semantics we have shown are already relatively clear and simple enough to reason about, and the unary level parallelism can be investegated in future work.

2.4 SVCODE determinism

As we have given formal semantics to the language, we now argue that a well-formed SVCODE program is deterministic.

Definition 2.4 (Stream prefix). *$\vec{a}$ is a prefix of $\vec{a}'$, written $\vec{a} \sqsubseteq \vec{a}'$, if one the following rules applys:*

Judgment $\boxed{\vec{a} \sqsubseteq \vec{a}'}$

Pre-Emp: $\dfrac{}{\langle\rangle \sqsubseteq \vec{a}'}$ Pre-Nonemp: $\dfrac{\vec{a} \sqsubseteq \vec{a}'}{\langle a_0|\vec{a}\rangle \sqsubseteq \langle a_0|\vec{a}'\rangle}$

Lemma 2.5. *If $\vec{a}_1{+\!\!+}\vec{a}_2 = \vec{a}$, then $\vec{a}_1 \sqsubseteq \vec{a}$.*

Proof. The proof is straightforward by induction on $\vec{a}_1$: case $\vec{a}_1 = \langle\rangle$ and case $\vec{a}_1 = \langle a_0, \vec{a}_1'\rangle$ for some $\vec{a}_1'$. ■

One may notice that in the rule P-X-Termi both the control stream and the parameter stream(s) must be all empty, and no rules apply to the other cases where one of them is empty while the other is not. The following lemma explains why the other cases can never happen: there is only one way to cut down a prefix of each input stream for a specific Xducer to be consumed in a block.

Lemma 2.6. *If*

(i) $(\vec{a}_i' \sqsubseteq \vec{a}_i$ by some derivation $\mathcal{P}_{ri})_{i=1}^k$ and $\psi(\vec{a}_1', ..., \vec{a}_k') \downarrow \vec{a}'$ by some $\mathcal{P}$,

(ii) $(\vec{a}_i'' \sqsubseteq \vec{a}_i$ by some derivation $\mathcal{P}_{ri}')_{i=1}^k$ and $\psi(\vec{a}_1'', ..., \vec{a}_k'') \downarrow \vec{a}''$ by some $\mathcal{P}'$.

then

(i) $(\vec{a}_i' = \vec{a}_i'')_{i=1}^k$

(ii) $\vec{a}' = \vec{a}''$.

Proof. The proof is by induction on the syntax of ψ. We show two cases `ToFlags` and $\texttt{ScanPlus}_{n_0}$ here; the others are analogous.

- Case $\psi = \texttt{ToFlags}$.

  Since there is only one rule for `ToFlags`, we must have
  $$\mathcal{P} = \frac{}{\texttt{ToFlags}(\langle n_1 \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_{n_1}, \texttt{T} \rangle}$$
  and
  $$\mathcal{P}' = \frac{}{\texttt{ToFlags}(\langle n_2 \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_{n_2}, \texttt{T} \rangle}$$
  so $k = 1, \vec{a}'_1 = \langle n_1 \rangle$, $\vec{a}' = \langle \texttt{F}_1, ..., \texttt{F}_{n_1}, \texttt{T} \rangle$, and $\vec{a}''_1 = \langle n_2 \rangle$, $\vec{a}'' = \langle \texttt{F}_1, ..., \texttt{F}_{n_2}, \texttt{T} \rangle$.
  Since both $\vec{a}'_1$ and $\vec{a}''_1$ are nonempty, $\mathcal{P}_{r1}$ and $\mathcal{P}'_{r1}$ must all use the rule PRE-NONEMP, which implies $n_1 = n_2$. Then it is clear that $\vec{a}'_1 = \vec{a}''_1$ and $\vec{a}' = \vec{a}''$ as required.

- Case $\psi = \texttt{ScanPlus}_{n_0}$.
  From the two rules P-SCANT and P-SCANF, it is clear that k=2, and both $\vec{a}'_1$ and $\vec{a}''_1$ must be nonempty, which means $\mathcal{P}_{r1}$ and $\mathcal{P}'_{r1}$ must all use PRE-NONEMP.
  By induction on $\vec{a}_1$, there are two subcases:

  – Subcase $\vec{a}_1 = \langle \texttt{T} | \vec{a}_{10} \rangle$.
  By PRE-NONEMP we know $\vec{a}'_1$ and $\vec{a}''_1$ must start with a `T`, thus both $\mathcal{P}$ and $\mathcal{P}'$ must use P-SCANT, and they must be identical:
  $$\mathcal{P} = \mathcal{P}' = \frac{}{\texttt{ScanPlus}_{n_0}(\langle \texttt{T} \rangle, \langle \rangle) \downarrow \langle \rangle}$$
  So immediately we have $\vec{a}'_1 = \vec{a}''_1 = \langle \texttt{T} \rangle$, $\vec{a}'_2 = \vec{a}''_2 = \langle \rangle$, and $\vec{a}' = \vec{a}'' = \langle \rangle$ as required.

  – Subcase $\vec{a}_1 = \langle \texttt{F} | \vec{a}_{10} \rangle$.
  By PRE-NONEMP we know $\vec{a}'_1$ and $\vec{a}''_1$ must start with an `F`, therefore both $\mathcal{P}$ and $\mathcal{P}'$ must use P-SCANF. Assume $\vec{a}_2 = \langle n | \vec{a}_{20} \rangle$, then we must have
  $$\mathcal{P} = \frac{\begin{array}{c}\mathcal{P}_0 \\ \texttt{ScanPlus}_{n_0+n}(\vec{a}'_{10}, \vec{a}'_{20}) \downarrow \vec{a}'_0\end{array}}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F} | \vec{a}'_{10} \rangle, \langle n | \vec{a}'_{20} \rangle) \downarrow \langle n_0 | \vec{a}'_0 \rangle}$$
  where
  $$(\vec{a}'_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^{2} \tag{2.1}$$
  So $\vec{a}'_1 = \langle \texttt{F} | \vec{a}'_{10} \rangle, \vec{a}'_2 = \langle n | \vec{a}'_{20} \rangle$, and $\vec{a}' = \langle n_0 | \vec{a}'_0 \rangle$.
  Similarly,
  $$\mathcal{P}' = \frac{\begin{array}{c}\mathcal{P}'_0 \\ \texttt{ScanPlus}_{n_0+n}(\vec{a}''_{10}, \vec{a}''_{20}) \downarrow \vec{a}''_0\end{array}}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F} | \vec{a}''_{10} \rangle, \langle n | \vec{a}''_{20} \rangle) \downarrow \langle n_0 | \vec{a}''_0 \rangle}$$
  where
  $$(\vec{a}''_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^{2} \tag{2.2}$$
  and $\vec{a}''_1 = \langle \texttt{F} | \vec{a}''_{10} \rangle, \vec{a}''_2 = \langle n | \vec{a}''_{20} \rangle$, $\vec{a}'' = \langle n_0 | \vec{a}''_0 \rangle$.
  By IH on (2.1) with $\mathcal{P}_0$, (2.2), $\mathcal{P}'_0$, we get $(\vec{a}'_{i0} = \vec{a}''_{i0})_{i=1}^{2}$, and $\vec{a}'_0 = \vec{a}''_0$.
  Thus $\langle \texttt{F} | \vec{a}'_{10} \rangle = \langle \texttt{F} | \vec{a}''_{10} \rangle$, i.e., $\vec{a}'_1 = \vec{a}''_1$. Likewise, $\vec{a}'_2 = \langle n | \vec{a}'_{20} \rangle = \langle n | \vec{a}''_{20} \rangle = \vec{a}''_2$, and $\vec{a}' = \langle n_0 | \vec{a}'_0 \rangle = \langle n_0 | \vec{a}''_0 \rangle = \vec{a}''$ as required.

■

Lemma 2.7 (Xducer determinism). *If $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}_0$ by some derivation $\mathcal{P}$, and $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}'_0$ by some derivation $\mathcal{P}'$, then $\vec{a}_0 = \vec{a}'_0$.*

Proof. The proof is by induction on the structure of $\vec{c}$. There are two cases: $\vec{c} = \langle\rangle$ and $\vec{c} = \langle()|\vec{c}_0\rangle$ for some $\vec{c}_0$. The first case is trivial, so we just show the second here.

- Case $\vec{c} = \langle()|\vec{c}_0\rangle$.
  $\mathcal{P}$ must use P-X-Loop:

$$\mathcal{P} = \dfrac{\overset{\mathcal{P}_1}{\psi(\vec{a}_{11}, ..., \vec{a}_{k1}) \downarrow \vec{a}_{01}} \qquad \overset{\mathcal{P}_2}{\psi(\vec{a}_{12}, ..., \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle()|\vec{c}_0\rangle} \vec{a}_0}$$

  where

$$(\vec{a}_{i1} +\!\!+ \vec{a}_{i2} = \vec{a}_i)_{i=1}^k \tag{2.3}$$

$$\vec{a}_{01} +\!\!+ \vec{a}_{02} = \vec{a}_0 \tag{2.4}$$

  Similarly,

$$\mathcal{P}' = \dfrac{\overset{\mathcal{P}'_1}{\psi(\vec{a}'_{11}, ..., \vec{a}'_{k1}) \downarrow \vec{a}'_{01}} \qquad \overset{\mathcal{P}'_2}{\psi(\vec{a}'_{12}, ..., \vec{a}'_{k2}) \Downarrow^{\vec{c}_0} \vec{a}'_{02}}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle()|\vec{c}_0\rangle} \vec{a}'_0}$$

  where

$$(\vec{a}'_{i1} +\!\!+ \vec{a}'_{i2} = \vec{a}_i)_{i=1}^k \tag{2.5}$$

$$\vec{a}'_{01} +\!\!+ \vec{a}'_{02} = \vec{a}'_0 \tag{2.6}$$

  Using Lemma 2.5 k times on (2.3), we have

$$(\vec{a}_{i1} \sqsubseteq \vec{a}_i)_{i=1}^k \tag{2.7}$$

  Analogously, from (2.5),

$$(\vec{a}'_{i1} \sqsubseteq \vec{a}_i)_{i=1}^k \tag{2.8}$$

  By Lemma 2.6 on (2.7) with $\mathcal{P}_1$, (2.8), $\mathcal{P}'_1$, we get

$$(\vec{a}_{i1} = \vec{a}'_{i1})_{i=1}^k \tag{2.9}$$

$$\vec{a}_{01} = \vec{a}'_{01} \tag{2.10}$$

  It is easy to show that from (2.3), (2.5) and (2.9) we can get

$$(\vec{a}_{i2} = \vec{a}'_{i2})_{i=1}^k \tag{2.11}$$

  Then by IH on $\mathcal{P}_2$ with $\mathcal{P}'_2$, we obtain $\vec{a}_{02} = \vec{a}'_{02}$.

  Therefore, with (2.4), (2.6), (2.10), we obtain $\vec{a}_0 = \vec{a}_{01} +\!\!+ \vec{a}_{02} = \vec{a}'_{01} +\!\!+ \vec{a}'_{02} = \vec{a}'_0$, as required.

■

Theorem 2.8 (SVCODE determinism). *If $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma'$ (by some derivation $\mathcal{P}$) and $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma''$ (by some derivation $\mathcal{P}'$), then $\sigma' = \sigma''$.*

Proof. The proof is by induction on the syntax of p. There are four cases: the case for $p = \epsilon$ is trivial; with the help of Lemma 2.7, the case for $p = s := \psi(s_1, ..., s_k)$ is also trivial; proof of $p = p_1; p_2$ can be done by IH; the only interesting case is $p = S_{out} :=$ `WithCtrl`(s_c, S_{in}, p_1).

- Case $p = S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)$.
  Assume $S_{out} = \{s_1, ..., s_l\}$. There are two subcases by induction on $\sigma(s_c)$:
  - Subcase $\sigma(s_c) = \langle\rangle$.
    Then $\mathcal{P}$ and $\mathcal{P}'$ must all use P-Wc-Emp, and they must be identical:
    $$\mathcal{P} = \mathcal{P}' = \frac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{l}]}$$
    with $\forall s \in S_{in}.\sigma(s) = \langle\rangle$. So $\sigma' = \sigma'' = \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{l}]$, as required.
  - Subcase $\sigma(s_c) \neq \langle\rangle$.
    Then we must have
    $$\mathcal{P} = \frac{\begin{array}{c}\mathcal{P}_1 \\ \langle p_1, \sigma\rangle \Downarrow^{\vec{c}_1} \sigma_1\end{array}}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma_1(s_i))_{i=1}^{l}]}$$
    Also, we have
    $$\mathcal{P}' = \frac{\begin{array}{c}\mathcal{P}_1' \\ \langle p_1, \sigma\rangle \Downarrow^{\vec{c}_1} \sigma_1'\end{array}}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma_1'(s_i))_{i=1}^{l}]}$$
    So $\sigma' = \sigma[(s_i \mapsto \sigma_1(s_i))_{i=1}^{l}]$, and $\sigma'' = \sigma[(s_i \mapsto \sigma_1'(s_i))_{i=1}^{l}]$.
    By IH on $\mathcal{P}_1$ and $\mathcal{P}_1'$, we obtain
    $$\sigma_1 = \sigma_1'$$
    Then it is clear that $\sigma' = \sigma''$, as required.

■

2.5 Streaming interpreter

2.6 Recursion

References

[Mad13] Frederik M. Madsen. A streaming model for nested data parallelism. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, March 2013.

[Mad16] Frederik M. Madsen. *Streaming for Functional Data-Parallel Languages*. PhD thesis, Department of Computer Science (DIKU), University of Copenhagen, September 2016.