Formalizing the implementation of Streaming NESL

Dandan Xue

Novermber 8, 2017

Department of Computer Science (DIKU)
University of Copenhagen

Contents

1. Introduction
    - NESL
    - Streaming NESL (SNESL)
2. Implementation
    - Extended target language (supporting recursion)
    - Translation
    - Streaming SVCODE interpreter
3. Formalization
    - Source and target language semantics
    - Target language well-formedness, determinism
    - Translation correctness (including work preservation)
4. Conclusion

Introduction

NESL

- A functional nested data-parallel language
- Developed by Guy E. Blelloch in 1990s at CMU
- Highlights:
  - Highly expressive for parallel algorithms.
    Data-parallel construct: *apply-to-each*

    $$\{e_1(x) : x \textbf{ in } e_0\}$$

    Example: compute $\sum_{i=0}^{k-1}$ for $k \in [2, 3, 4]$ (result: [1,3,6]):

    $$\{\textbf{sum}(\&x) : x \textbf{ in } [2, 3, 4]\}$$

  - An intuitive cost model for time complexity: work-step model
    - work cost t_1: total number of operations executed
    - step cost t_∞: the longest chain of sequential dependency

Streaming NESL (SNESL)

- Experimental refinement of NESL
- Aiming at improving space-usage efficiency
- Work from Frederik M. Madsen and Andrzej Filinski in 2010s at DIKU
- Highlights:
  - Streaming semantics

$$\pi ::= \textbf{bool} \mid \textbf{int} \mid \textbf{char} \mid \textbf{real} \mid \cdots \qquad \text{(scalar types)}$$

$$\tau ::= \pi \mid (\tau_1, ..., \tau_k) \mid [\tau] \qquad \text{(concrete types)}$$

$$\sigma ::= \tau \mid (\sigma_1, ..., \sigma_k) \mid \{\sigma\} \qquad \text{(streamable types)}$$

  - A space cost model
    - sequential space s_1: the minimal space to perform the computation
    - parallel space s_∞: space needed to achieve the maximal parallel degree (NESL's case)

SNESL syntax

- Expressions

$$
\begin{aligned}
e ::= {} & a \mid x \mid (e_1, ..., e_k) \mid \textbf{let } x = e_1 \textbf{ in } e_2 \mid \phi(e_1, ..., e_k) \\
& \mid \{e_1 : \ x \textbf{ in } e_0\} && \text{(general comprehension)} \\
& \mid \{e_1 \mid e_0\} && \text{(restricted comprehension)}
\end{aligned}
$$

- Primitive functions

$$
\begin{aligned}
\phi \ ::= {} & \oplus \mid \ \textbf{append} \mid \textbf{concat} \mid \textbf{zip} \mid \textbf{iota} \mid \textbf{part} \mid \textbf{scan}_{\otimes} \mid \textbf{reduce}_{\otimes} \\
& \mid \textbf{mkseq} \mid \textbf{the} \mid \textbf{empty} && \text{(sequence operations)} \\
& \mid \textbf{length} \mid \textbf{elt} && \text{(vector operations)} \\
& \mid \textbf{seq} \mid \textbf{tab} && \text{(convertion between vector and sequence)} \\
\oplus \ ::= {} & + \mid \ \times \ \mid / \mid == \ \mid \textbf{not} \mid \ \cdots && \text{(scalar operations)} \\
\otimes \ ::= {} & + \mid \ \times \ \mid \textbf{max} \mid ... && \text{(associative binary operations)}
\end{aligned}
$$

SNESL primitive functions

append : $(\{\sigma\},\{\sigma\}) \to \{\sigma\}$	append two sequences; syntactic sugar: $++$
concat : $\{\{\sigma\}\} \to \{\sigma\}$	flatten a sequence of sequences
zip: $(\{\sigma_1\},...,\{\sigma_k\}) \to \{(\sigma_1,...,\sigma_k)\}$	**zip**$(\{1,2\},\{\mathtt{F},\mathtt{T}\}) = \{(1,\mathtt{F}),(2,\mathtt{T})\}$
iota(&) : **int** $\to$ $\{$**int**$\}$	&5 = $\{$0,1,2,3,4$\}$
part : $(\{\sigma\},\{$**bool**$\}) \to \{\{\sigma\}\}$	**part**$(\{3,1,4\},\{\mathtt{F},\mathtt{F},\mathtt{T},\mathtt{F},\mathtt{T},\mathtt{T}\}) = \{\{3,1\},\{4\},\{\}\}$
scan$_\otimes$: $\{$**int**$\} \to \{$**int**$\}$	**scan**$_+$(&5) $= \{0,0,1,3,6\}$
reduce$_\otimes$: $\{$**int**$\} \to$ **int**	**reduce**$_+$(&5) $= 10$
mkseq : $(\overbrace{\sigma,...,\sigma}^{k}) \to \{\sigma\}$	**mkseq**$(1,2,3) = \{1,2,3\}$
length(#): $[\tau] \to$ **int**	length of a vector
elt(!): $([\tau],$ **int**$) \to \tau$	element indexing, [3,8,2] ! 1 = 8
the : $\{\sigma\} \to \sigma$	return the element of a singleton, **the**$(\{10\}) = 10$
empty : $\{\sigma\} \to$ **bool**	test a sequence empty or not
seq : $[\tau] \to \{\tau\}$	**seq**$([1,2]) = \{1,2\}$
tab : $\{\tau\} \to [\tau]$	**tab**$(\{1,2\}) = [1,2]$

??[optional] Example program: Splitting a string into words

```
-- NESL version
function str2wds(str) =
 let strl = #str;
     spc_is =  { i : c in str, i in &strl | c == ' '};
     word_ls = { id2-id1-1: id1 in [-1]++spc_is; id2 in
         spc_is++[strl]};
     valid_ls = {l : l in word_ls | l > 0};
     chars = {c : c in str | c != ' ' }  -- non-space chars
  in partition(chars, valid_ls);
```

```
-- SNESL version
function str2wds_snesl(str) =
 let flags = { x == ' ' : x in str};
     nonsps =  concat({{x | x != ' '} : x in v})
  in concat({{x|not(empty(x))}: x in part(nonsps,flags ++ {T})})
```

```
  $> str2wds("A   NESL program . ")
  [['A'], ['N', 'E', 'S', 'L'], ['p', 'r', 'o', 'g', 'r', 'a',
      'm'], ['.']] :: [[char]]
```

Implementation

Source language

- Simplified SNESL types

$$\pi ::= \textbf{bool} \mid \textbf{int} \qquad \text{(only two scalar types)}$$

$$\tau ::= \pi \mid (\tau_1, \tau_2) \mid \{\tau\} \qquad \text{(no vectors, change tuples to pairs)}$$

$$\varphi ::= (\tau_1, ..., \tau_k) \to \tau \qquad \text{(support recursion)}$$

- Syntax

$$t ::= \textbf{eval}\ e \mid d\ t \qquad \text{(top-level term)}$$

$$e ::= a \mid x \mid (e_1, e_2) \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \phi(e_1, ..., e_k)$$

$$\mid \{\}\tau \mid \{e_1, ..., e_k\}_{(k \geq 1)}$$

$$\mid \{e_1 : x\ \textbf{in}\ e_0\ \textbf{using}\ x_1, ..., x_k\} \mid \{e_1 \mid e_0\ \textbf{using}\ x_1, ..., x_k\}$$

$$\mid f(e_1, ..., e_k) \qquad \text{(user-defined function call)}$$

$$d ::= \textbf{function}\ f(x_1 : \tau_1, ..., x_k : \tau_k) : \tau = e$$

Source language

- Key typing rules:

$$\frac{\Gamma \vdash_\Sigma \; e_0 : \{\tau_0\} \qquad [x \mapsto \tau_0, (x_i \mapsto \tau_i)_{i=1}^k] \vdash_\Sigma \; e_1 : \tau}{\Gamma \vdash_\Sigma \; \{e_1 : x \textbf{ in } e_0 \textbf{ using } x_1, ..., x_k\} : \{\tau\}} \begin{pmatrix} (\Gamma(x_i) = \tau_i \\ \tau_i \textbf{ concrete})_{i=1}^k \end{pmatrix}$$

$$\frac{\Gamma \vdash_\Sigma \; e_0 : \textbf{bool} \qquad [(x_i \mapsto \tau_i)_{i=1}^k] \vdash_\Sigma \; e_1 : \tau}{\Gamma \vdash_\Sigma \; \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_k\} : \{\tau\}} \; ((\Gamma(x_i) = \tau_i)_{i=1}^k)$$

- Key evaluation rules:

$$\frac{\rho \vdash_\Phi e_0 \downarrow \{v_1, ..., v_l\} \qquad ([x \mapsto v_i, (x_j \mapsto \rho(x_j))_{j=1}^k] \vdash_\Phi e_1 \downarrow v_i')_{i=1}^l}{\rho \vdash_\Phi \{e_1 : x \textbf{ in } e_0 \textbf{ using } x_1, ..., x_k\} \downarrow \{v_1', ..., v_l'\}}$$

$$\frac{(\rho \vdash_\Phi e_i \downarrow v_i)_{i=1}^k \qquad [(x_i \mapsto v_i)_{i=1}^k] \vdash_\Phi e_0 \downarrow v}{\rho \vdash_\Phi f(e_1, ..., e_k) \downarrow v}$$

where $\Phi(f) = f(x_1 \colon \tau_1, ..., x_k \colon \tau_k) \colon \tau = e_0$

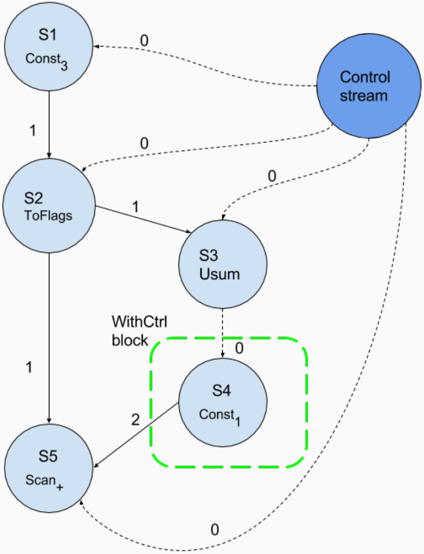
Target language: SVCODE

- SVCODE values:
  - primitive stream: $\vec{a} ::= \langle a_1, ..., a_l \rangle$
    e.g., $\vec{a}_1 = \langle 1, 2 \rangle, \langle 0 | \vec{a}_1 \rangle = \langle 0, 1, 2 \rangle$, $\vec{b} = \langle \mathtt{F}, \mathtt{T}, \mathtt{F} \rangle$
  - stream tree: $w ::= \vec{a} \mid (w_1, w_2)$
- SVCODE syntax

$$
\begin{array}{lll}
p & ::= \epsilon \mid p_1; p_2 & \\
 & \quad \mid s := \psi(s_1, ..., s_k) & \text{(single stream definition)} \\
 & \quad \mid S_{out} := \mathtt{WithCtrl}(s, S_{in}, p_1) & (\mathtt{WithCtrl}\text{ block}) \\
 & \quad \mid (s'_1, ..., s'_{k'}) := \mathtt{SCall}\ f(s_1, ..., s_k) & \text{(function call)} \\
s & ::= 0 \mid 1 \mid \cdots \in \mathbf{SId} = \mathbb{N} & \text{(stream ids)} \\
S & ::= \{s_1, ..., s_k\} \in \mathbb{S} & \text{(set of stream ids)} \\
\psi & ::= \mathtt{Const_a} \mid \mathtt{ToFlags} \mid \mathtt{Usum} \mid \mathtt{Map}_{\oplus} \mid \mathtt{Scan}_{+} \mid \mathtt{Reduce}_{+} \mid \mathtt{Distr} & \\
 & \quad \mid \mathtt{Pack} \mid \mathtt{UPack} \mid \mathtt{B2u} \mid \mathtt{SegConcat} \mid \mathtt{InterMerge} \mid \cdots & \\
 & & \text{(Xducers)}
\end{array}
$$

SVCODE dataflow

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



Value representation

- Scalars are represented as singleton primitive streams: e.g., $3 \triangleright_{\textbf{int}} \langle 3 \rangle, \mathtt{T} \triangleright_{\textbf{bool}} \langle \mathtt{T} \rangle$
- A nested sequence with a nesting depth *d* is represented as a flattening data stream and *d* descriptor streams.

$$\{\{3,1\},\{4\}\} \triangleright_{\{\{\textbf{int}\}\}} ((\langle 3,1,4 \rangle, \langle \mathtt{F},\mathtt{F},\mathtt{T},\mathtt{F},\mathtt{T} \rangle), \langle \mathtt{F},\mathtt{F},\mathtt{T} \rangle)$$

$$\{\mathtt{T},\mathtt{F}\} \triangleright_{\{\textbf{bool}\}} (\langle \mathtt{T},\mathtt{F} \rangle, \langle \mathtt{F},\mathtt{F},\mathtt{T} \rangle)$$

- A sequence of pairs is represented as a pair of sequences sharing one descriptor:

$$\{(1,\mathtt{T}),(2,\mathtt{F}),(3,\mathtt{F})\} \triangleright_{\{(\textbf{int},\textbf{bool})\}} ((\langle 1,2,3 \rangle, \langle \mathtt{T},\mathtt{F},\mathtt{F} \rangle), \langle \mathtt{F},\mathtt{F},\mathtt{F},\mathtt{T} \rangle)$$

Translation

- **STree** $\ni st ::= s \mid (st_1, st_2)$
- Translation symbol table $\delta ::= [x_1 \mapsto st_1, ..., x_k \mapsto st_k]$
- General comprehension translation:
  $\{i + x : i$ **in** $\&3$ **using** $x\} \Rightarrow$

```
  ...
  S4 := ...   -- <1 >      x
  S5 := ...   -- <F,F,F,T> descriptor of &3
  S6 := ...   -- <0,1,2>   i
  S7 := Usum S5;  -- 1. generate new control: <() () ()>
  S8 := Distr S4 S5; -- 2. replicate x 3 times: <1 1 1 >
  [S9] := WithCtrl S7 [S6,S8]: -- 3. translate  (i+x)
            S9 := Map_+ S6 S8  -- <1,2,3>
```

- Restricted comprehension translation: `Pack` free variables instead of `Distr`

Translation continue

- Built-in function translation:
  - **scan**, **reduce**, **concat**, **part**, **empty**: translated to a single stream definition, e.g., $\textbf{scan}_{+}((s_d, s_b)) \Rightarrow \texttt{Scan}_{+}(s_b, s_d)$
  - **the**, **iota** translated to a few lines of code, e.g.,

$$\textbf{iota}(s) \Rightarrow \begin{array}{l} s_0 := \texttt{ToFlags}(s); \\ s_1 := \texttt{Usum}(s_0); \\ \{s_2\} := \texttt{WithCtrl}(s_1, \{\}, s_2 := \texttt{Const}_1()); \\ s_3 := \texttt{Scan}_{+}(s_0, s_2) \end{array}$$

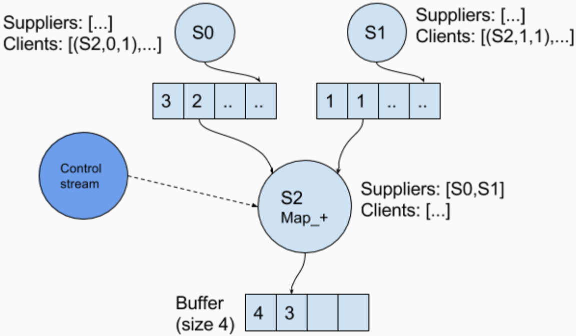
  - $++_{\tau}$: translated recursively, depending on τ
- User-defined functions: translated to SVCODE functions (i.e., SVCODE program with arguments), unfolded at runtime when interpreting a `SCall`

SVCODE interpreters

- Eager interpreter (NESL-like)
  - sufficient memory for allocating all streams at once
  - execute each instruction sequentially
  - an extreme/simplest case of the streaming one with the largest buffer size, used to compare results and analyze time complexity
- Streaming interpreter
  - limited buffer size, space-usage efficient
  - result is collected from each scheduling round
  - need effective scheduling strategy to avoid deadlock and guarantee cost preservation

SVCODE streaming interpreter

- Dataflow graph is similar to a Kahn process network
  - Graph node (a process): $\textbf{Proc} = (\textbf{BufState}, S, \textbf{Clis}, \textbf{Xducer})$
  - Buffer state maintained by process:
    $\textbf{BufState} ::= \texttt{Filling}\ \vec{a} \mid \texttt{Draining}\ \vec{a}'\ b$
  - A process example:



Recursion example

A function to compute factorial:

```
> function fact(x:int):int = if x <= 1 then 1 else x*fact(x-1)
> let x = {3,7,0,4} in {fact(y): y in x }
```

1st unfolding (will unfold 7 times in total):

```
--  Parameters: [S1]                   -- <3 7 0  4>
... -- compare parameters with 1, get S5 = <T T FT T>
S6 := Usum S5;  -- for elements <=1    -- <    ()  >
[S7] := WithCtrl S6 []: S7 := Const_1  -- <     1  >
...
S13 := Usum S11; -- for elementes >1   -- <()()  ()>
[S17] := WithCtrl S13 [S12]:
           S14 := Const_1              -- <1  1     1>
           S15 := MapTwo Minus S12 S14 -- <2  6     3>
           [S16] := SCall fact [S15]   -- <2 720    6>
               recursive call
           S17 := MapTwo Times S12 S16 -- <6 5040   24>
...  -- merge results
S19 := PriSegInterS [(S7,S5),(S17,S11)];-- <6 5040 1 24>
```

Formalization

Source language: SNESL$_0$

- Types:

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

- Expressions:

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

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

- Key evaluation rules with work cost W:
  - General comprehension:

$$\frac{([x \mapsto v_i, x_1 \mapsto n_1, ..., x_k \mapsto n_k] \ \vdash \ e \downarrow v_i' \ \$ \ W_i)_{i=1}^{l}}{\rho \ \vdash \ \{e : x\ \mathbf{in}\ y\ \mathbf{using}\ x_1, ..., x_k\} \downarrow \{v_1', ..., v_l'\} \ \$ \ W}$$

    where $\rho(y) = \{v_1, ..., v_l\}$,$(\rho(x_i) = n_i)_{i=1}^{k}$, and
    $W = (k+1) \cdot (l+1) + \sum_{i=1}^{l} W_i$

  - Built-in function:

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

Target language: SVCODE$_0$

- syntax

$$p ::= \ \epsilon \mid s := \psi(s_1, ..., s_k) \mid S_{out} := \texttt{WithCtrl}(s, S_{in}, p_1) \mid p_1; p_2$$

- key semantics with work cost
  - Empty new control stream ($\sigma(s_c) = \langle\rangle$):

$$\frac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{k}] \ \$\ 1}$$

    where $\forall s \in \{s_c\} \cup S_{in}.\sigma(s) = \langle\rangle$, $S_{out} = \{s_1, ..., s_k\}$
  - Nonempty new control stream ($\sigma(s_c) = \vec{c}_1 \neq \langle\rangle$):

$$\frac{\langle p_1, \sigma\rangle \Downarrow^{\vec{c}_1} \sigma'' \ \$\ W_1}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma''(s_i))_{i=1}^{k}] \ \$\ W_1 + 1}$$

  - Xducers, $((\sigma(s_i) = \vec{a}_i)_{i=1}^{k})$

$$\frac{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}}{\langle s := \psi(s_1, ..., s_k), \sigma\rangle \Downarrow^{\vec{c}} \sigma[s \mapsto \vec{a}] \ \$\ (\sum_{i=1}^{k} |\vec{a}_i|) + |\vec{a}|}$$

Xducer semantics

- General semantics: **Judgment** $\boxed{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}}$
  - $$\frac{\psi(\vec{a}_{11}, ..., \vec{a}_{k1}) \downarrow \vec{a}_{01} \qquad \psi(\vec{a}_{12}, ..., \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle () | \vec{c}_0 \rangle} \vec{a}_0} \; ((\vec{a}_{i1} +\!\!+ \vec{a}_{i2} = \vec{a}_i)_{i=0}^{k})$$
  - $$\frac{}{\psi(\langle\rangle_1, ..., \langle\rangle_k) \Downarrow^{\langle\rangle} \langle\rangle}$$
- Specific semantics (part): **Judgment** $\boxed{\psi(\vec{a}_1, ..., \vec{a}_k) \downarrow \vec{a}}$

$$\frac{}{\mathtt{Const}_a() \downarrow \langle a \rangle} \qquad \frac{}{\mathtt{ToFlags}(\langle n \rangle) \downarrow \langle \mathtt{F}_1, ..., \mathtt{F}_n, \mathtt{T} \rangle} \; (n \geq 0)$$

$$\frac{}{\mathtt{MapTwo}_{+}(\langle n_1 \rangle, \langle n_2 \rangle) \downarrow \langle n_3 \rangle} \; (n_3 = n_1 + n_2)$$

$$\frac{\mathtt{Usum}(\vec{b}) \downarrow \vec{a}}{\mathtt{Usum}(\langle \mathtt{F} | \vec{b} \rangle) \downarrow \langle () | \vec{a} \rangle} \qquad \frac{}{\mathtt{Usum}(\langle \mathtt{T} \rangle) \downarrow \langle\rangle}$$

SVCODE$_0$ determinism

Definition (Stream prefix)

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

$$\frac{}{\langle\rangle \sqsubseteq \vec{a}'} \qquad \frac{\vec{a} \sqsubseteq \vec{a}'}{\langle a_0|\vec{a}\rangle \sqsubseteq \langle a_0|\vec{a}'\rangle}$$

Lemma (Blocks are self-delimiting)

If (i) $(\vec{a}'_i \sqsubseteq \vec{a}_i)_{i=1}^k$ *and* $\psi(\vec{a}'_1, ..., \vec{a}'_k) \downarrow \vec{a}'$,
(ii) $(\vec{a}''_i \sqsubseteq \vec{a}_i)_{i=1}^k$ *and* $\psi(\vec{a}''_1, ..., \vec{a}''_k) \downarrow \vec{a}''$,
then $(\vec{a}'_i = \vec{a}''_i)_{i=1}^k$, *and* $\vec{a}' = \vec{a}''$.

Lemma (Xducer determinism)

If $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}_0$, *and* $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}'_0$, *then* $\vec{a}_0 = \vec{a}'_0$.

Theorem (SVCODE$_0$ determinism)

If $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma' \ \$ \ W_1$ *and* $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma'' \ \$ \ W_2$, *then* $\sigma' = \sigma''$ *and* $W_1 = W_2$.

Translation formalization

- General comprehension translation:

$$\frac{[x \mapsto st_1, (x_i \mapsto s'_i)_{i=1}^k] \ \vdash \ e \Rightarrow_{s''_1}^{s'_k+1} (p_1, st_2)}{\delta \ \vdash \ \{e : x \textbf{ in } y \textbf{ using } x_1, ..., x_k\} \Rightarrow_{s''_1}^{s'_0} (p, (st_2, s_b))} \left(\begin{array}{l} \delta(y) = (st_1, s_b), (\delta(x_i) = s_i)_{i=1}^k \\ p = \ (\ s'_0 := \texttt{Usum}(s_b); \\ \qquad (s'_i := \texttt{Distr}(s_b, s_i);)_{i=1}^k \\ \qquad S_{out} := \texttt{WithCtrl}(s'_0, S_{in}, p_1)) \\ S_{in} = \overline{\overline{st_1}} \cup \{s'_1, ..., s'_k\} \\ S_{out} = \{s \mid s \in \overline{\overline{st_2}}, s \geq s'_k + 1\} \\ s'_{i+1} = \ s'_i + 1, \forall i \in \{0, ..., k-1\} \end{array} \right)$$

- Well-formed program $\boxed{S \Vdash p : S'}$

$$\frac{}{S \Vdash s := \psi(s_1, ..., s_k) : \{s\}} \ (\{s_1, ..., s_k\} \subseteq S, s \notin S)$$

$$\frac{S_{in} \Vdash p_1 : S'}{S \Vdash S_{out} := \texttt{WithCtrl}(s, S_{in}, p_1) : S_{out}} \ ((S_{in} \cup \{s\}) \subseteq S, S_{out} \subseteq S', S \cap S' = \emptyset)$$

- **Theorem**

If $\delta \ \vdash \ e \Rightarrow_{s_1}^{s_0} (p, st)$, $\forall x \in dom(\delta).\overline{\overline{\delta(x)}} \subseteq S$, *and* $S \lessdot s_0$
then, *for some* S', $S \Vdash p : S'$, $S' \subseteq \{s_0, s_0+1, ..., s_1-1\}$, *and* $\overline{\overline{st}} \subseteq (S \cup S')$

Value representation formalization

- Value representation: **Judgment** $\boxed{v \rhd_\tau w}$

$$\frac{}{n \rhd_{\mathbf{int}} \langle n \rangle} \qquad \frac{(v_i \rhd_\tau w_i)_{i=1}^{l}}{\{v_1, \dots, v_l\} \rhd_{\{\tau\}} (w, \langle \mathtt{F}_1, \dots, \mathtt{F}_l, \mathtt{T} \rangle)} \; (w = (+\!\!+_\tau w_i)_{i=1}^{l})$$

- Value recovery: **Judgment** $\boxed{w \lhd_\tau v, w'}$

$$\frac{}{\langle n_0 | \vec{a} \rangle \lhd_{\mathbf{int}} n_0, \vec{a}} \qquad \frac{w \lhd_\tau v_1, w_1 \qquad w_1 \lhd_\tau v_2, w_2 \qquad \cdots \qquad w_{l-1} \lhd_\tau v_l, w_l}{(w, \langle \mathtt{F}_1, \dots, \mathtt{F}_l, \mathtt{T} | \vec{b} \rangle) \lhd_{\{\tau\}} \{v_1, \dots, v_l\}, (w_l, \vec{b})}$$

Lemma (Recovery correctness)

If $v \rhd_\tau w$, then $\forall w'.(w +\!\!+_\tau w') \lhd_\tau v, w'$.

Lemma (Recovery determinism)

If $w \lhd_\tau v, w'$, and $w \lhd_\tau v', w''$, then $v = v'$, and $w' = w''$.

Corollary

If $v \rhd_\tau w$, $v' \rhd_\tau w$, then $v = v'$.

Parallelism fusion lemma

Definition (Store similarity)

$\sigma_1 \overset{S}{\sim} \sigma_2$ *iff* $dom(\sigma_1) = dom(\sigma_2)$, *and* $\forall s \in S . \sigma_1(s) = \sigma_2(s)$

Definition (Store fusion)

For $\sigma_1 \overset{S}{\sim} \sigma_2$, $\sigma_1 \overset{S}{\bowtie} \sigma_2 = \sigma$ *where* $\sigma(s) = \begin{cases} \sigma_1(s)\ (= \sigma_2(s)), & s \in S \\ \sigma_1(s) +\!\!+ \sigma_2(s), & s \notin S \end{cases}$

Lemma (Xducer fusion)

If $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}$, *and* $\psi(\vec{a}'_1, ..., \vec{a}'_k) \Downarrow^{\vec{c}'} \vec{a}'$,
then $\psi(\vec{a}_1 +\!\!+ \vec{a}'_1, ..., \vec{a}_k +\!\!+ \vec{a}'_k) \Downarrow^{\vec{c} +\!\!+ \vec{c}'} \vec{a} +\!\!+ \vec{a}'$.

Lemma (Parallelism fusion)

If (i) $S_1 \Vdash p : S_2$, *(ii)* $\sigma_1 \overset{S}{\sim} \sigma_2$, *(iii)* $\langle p, \sigma_1 \rangle \Downarrow^{\vec{c}_1} \sigma'_1 \ \$\ W_1$, *(iv)* $\langle p, \sigma_2 \rangle \Downarrow^{\vec{c}_2} \sigma'_2 \ \$\ W_2$, *and (v)* $(S_1 \cup S_2) \cap S = \emptyset$,
then $\sigma'_1 \overset{S}{\sim} \sigma'_2$, $\left\langle p, \sigma_1 \overset{S}{\bowtie} \sigma_2 \right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma'_1 \overset{S}{\bowtie} \sigma'_2 \ \$\ W$, *and* $W \leq W_1 + W_2$

Correctness of translation and cost preservation

Theorem (Correctness for expressions)

For some constant C, ***if***

(i) $\Gamma \ \vdash \ e : \tau$

(ii) $\rho \ \vdash \ e \downarrow v \ \$ \ W^H$

(iii) $\delta \ \vdash \ e \Rightarrow^{s_0}_{s_1} (p, st)$

(iv) $\forall x \in dom(\Gamma). \ \vdash \ \rho(x) : \Gamma(x)$

(v) $\forall x \in dom(\Gamma).\overline{\overline{\delta(x)}} \lessdot s_0$

(vi) $\forall x \in dom(\Gamma).\rho(x) \triangleright_{\Gamma(x)} \sigma^*(\delta(x))$

then*, for some* σ' *and* W^L*,*

(vii) $\langle p, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma' \ \$ \ W^L$

(viii) $v \triangleright_\tau \sigma'^*(st)$

(ix) $W^L \leq C \cdot W^H$

(In our implementation, we have proven that C can be any number ≥ 7.)

Scaling up

- More scalar types and built-in operations: should be trivial
- Step/space cost: similar to work cost
- Pairs/tuples: require more value representation rules
- Restricted comprehension: similar to the general one, but need to take care of packing general types
- Recursion: consider termination preservation (from high-level to low-level) and reflection (from low-level to high-level)
- Error preservation: possible to support
- Streaming semantics: challenging, open problem

Conclusion

Conclusion

Main contributions:

- Extension of streaming dataflow model to account for recursion
- A formalization of the source and target language, and the correctness proof of the translation including work cost preservation

Future work:

- Formalization of the streaming semantics of the target language
- More investigation to schedulability, deadlock, etc.