

Formalizing the implementation of Streaming NESL

Dandan Xue

November 8, 2017

Department of Computer Science (DIKU)
University of Copenhagen

1. Introduction

- NESL
- Streaming NESL (SNESL)

2. Implementation

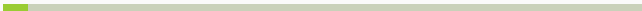
- Extended target language (supporting recursion)
- Translation
- Streaming SVCODE interpreter

3. Formalization

- Source and target language semantics
- Target language well-formedness, determinism
- Translation correctness (including work preservation)

4. Conclusion

Introduction



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

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

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

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

- 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

$\pi ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{char} \mid \mathbf{real} \mid \dots$ (scalar types)

$\tau ::= \pi \mid (\tau_1, \dots, \tau_k) \mid [\tau]$ (concrete types)

$\sigma ::= \tau \mid (\sigma_1, \dots, \sigma_k) \mid \{\sigma\}$ (general 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)

- Expressions

$e ::= a \mid x \mid (e_1, \dots, e_k) \mid \text{let } x = e_1 \text{ in } e_2 \mid \phi(e_1, \dots, e_k)$
| $\{e_1 : x \text{ in } e_0\}$ (general comprehension)
| $\{e_1 \mid e_0\}$ (restricted comprehension)

- Primitive functions

$\phi ::= \oplus \mid \text{append} \mid \text{concat} \mid \text{zip} \mid \text{iota} \mid \text{part} \mid \text{scan}_{\otimes} \mid \text{reduce}_{\otimes}$
| $\text{mkseq} \mid \text{the} \mid \text{empty}$ (sequence operations)
| $\text{length} \mid \text{elt}$ (vector operations)
| $\text{seq} \mid \text{tab}$ (conversion between vector and sequence)
 $\oplus ::= + \mid \times \mid / \mid == \mid \text{not} \mid \dots$ (scalar operations)
 $\otimes ::= + \mid \times \mid \text{max} \mid \dots$ (associative binary operations)

SNESL primitive functions

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

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

```
1  -- NESL version
2  function str2wds(str) =
3    let str1 = #str;
4      spc_is = { i : c in str, i in &str1 | c == ' ' };
5      word_ls = { id2-id1-1: id1 in [-1]++spc_is; id2 in
6                  spc_is++[str1]};
7      valid_ls = {l : l in word_ls | l > 0};
8      chars = {c : c in str | c != ' ' } -- non-space chars
9    in partition(chars, valid_ls);
```

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

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


Implementation



Source language

- Simplified SNESL types

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

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

$\varphi ::= (\tau_1, \dots, \tau_k) \rightarrow \tau$ (support recursion)

- Syntax

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

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

$\mid \{\tau\} \mid \{e_1, \dots, e_k\} \quad (k \geq 1)$

$\mid \{e_1 : x \ \mathbf{in} \ e_0 \ \mathbf{using} \ x_1, \dots, x_k\} \mid \{e_1 \mid e_0 \ \mathbf{using} \ x_1, \dots, x_k\}$

$\mid f(e_1, \dots, e_k)$ (user-defined function call)

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

$\phi ::= \oplus \mid ++_{\tau} \mid \mathbf{concat}_{\tau} \mid \&(\mathbf{iota}) \mid \mathbf{part}_{\tau} \mid \mathbf{scan}_{+} \mid \mathbf{reduce}_{+} \mid \dots$

Source language

- Key typing rules:

$$\frac{\Gamma \vdash_{\Sigma} e_0 : \{\tau_0\} \quad [x \mapsto \tau_0, (x_i \mapsto \tau_i)_{i=1}^k] \vdash_{\Sigma} e_1 : \tau}{\Gamma \vdash_{\Sigma} \{e_1 : x \text{ in } e_0 \text{ using } x_1, \dots, x_k\} : \{\tau\}} \left(\begin{array}{l} (\Gamma(x_i) = \tau_i \\ \tau_i \text{ concrete})_{i=1}^k \end{array} \right)$$

$$\frac{\Gamma \vdash_{\Sigma} e_0 : \mathbf{bool} \quad [(x_i \mapsto \tau_i)_{i=1}^k] \vdash_{\Sigma} e_1 : \tau}{\Gamma \vdash_{\Sigma} \{e_1 \mid e_0 \text{ using } x_1, \dots, x_k\} : \{\tau\}} ((\Gamma(x_i) = \tau_i)_{i=1}^k)$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \tau_1 \quad \dots \quad \Gamma \vdash_{\Sigma} e_k : \tau_k}{\Gamma \vdash_{\Sigma} f(e_1, \dots, e_k) : \tau} (\Sigma(f) = (\tau_1, \dots, \tau_k) \rightarrow \tau)$$

- Key evaluation rules:

$$\frac{\rho \vdash_{\Phi} e_0 \downarrow \{v_1, \dots, v_l\} \quad ([x \mapsto v_i, x_j \mapsto \rho(x_j)]_{j=1}^k) \vdash_{\Phi} e_1 \downarrow v'_i!_{i=1}}{\rho \vdash_{\Phi} \{e_1 : x \text{ in } e_0 \text{ using } x_1, \dots, x_k\} \downarrow \{v'_1, \dots, v'_l\}}$$

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

where $\Phi(f) = f(x_1 : \tau_1, \dots, x_k : \tau_k) : \tau = e_0$

Target language: **SVCODE**

- SVCODE values:
 - primitive stream: $\vec{a} ::= \langle a_1, \dots, a_l \rangle$, e.g., $\langle 1, 2, 3 \rangle$, $\langle F, T, F, F, T \rangle$
 - stream tree: $w ::= \vec{a} \mid (w_1, w_2)$

- SVCODE syntax

$p ::= \epsilon \mid p_1; p_2$

$\mid s := \psi(s_1, \dots, s_k)$ (single stream definition)

$\mid S_{out} := \text{WithCtrl}(s, S_{in}, p_1)$ (WithCtrl block)

$\mid (s'_1, \dots, s'_{k'}) := \text{SCall } f(s_1, \dots, s_k)$ (function call)

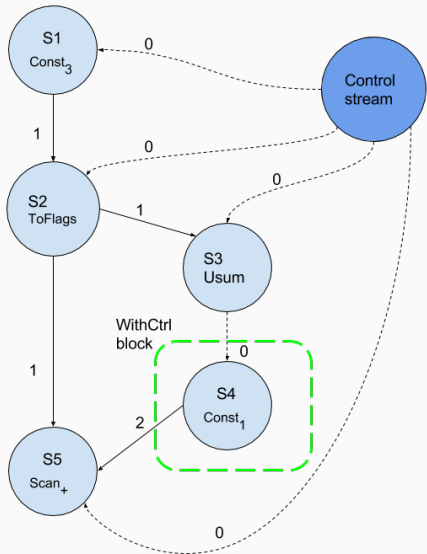
$s ::= 0 \mid 1 \mid \dots \in \mathbf{SId} = \mathbb{N}$ (stream ids)

$S ::= \{s_1, \dots, s_k\} \in \mathbb{S}$ (set of stream ids)

$\psi ::= \text{Const}_a \mid \text{ToFlags} \mid \text{Usum} \mid \text{Map}_{\oplus} \mid \text{Scan}_{+} \mid \text{Reduce}_{+} \mid \text{Distr}$
 $\mid \text{Pack} \mid \text{UPack} \mid \text{B2u} \mid \text{SegConcat} \mid \text{InterMerge} \mid \dots$
(Xducers)

SVCODE dataflow

```
1 S1 := Const_3
2 S2 := ToFlags S1
3 S3 := Usum S2
4 [S4] := WithCtrl S3 []:
5     S4 := Const_1()
6 S5 := ScanPlus S2 S4
```



Value representation

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

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

- A pair of SNESL values is represented as a pair of stream trees.

$$(\{T, F\}, 2) \triangleright_{(\{\text{bool}\}, \text{int})} ((\langle T, F \rangle, \langle F, F, T \rangle), \langle 2 \rangle)$$

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

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

Translation

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

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

- Restricted comprehension translation: Pack free variables instead of Distr

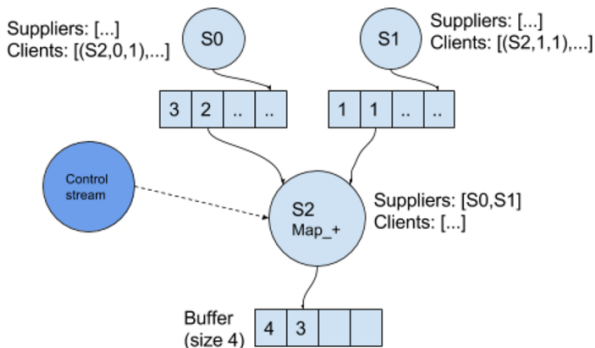
Translation continue

- Built-in function translation:
 - **scan, reduce, concat, part, empty**: translated to a single stream definition, e.g., $\mathbf{scan}_+((s_d, s_b)) \Rightarrow \mathbf{Scan}_+(s_b, s_d)$
 - **the, iota** translated to a few lines of code, e.g.,
$$\begin{aligned} s_0 &:= \mathbf{ToFlags}(s); \\ \mathbf{iota}(s) &\Rightarrow \begin{aligned} s_1 &:= \mathbf{Usum}(s_0); \\ \{s_2\} &:= \mathbf{WithCtrl}(s_1, \{\}, s_2 := \mathbf{Const}_1()); \\ s_3 &:= \mathbf{Scan}_+(s_0, s_2) \end{aligned} \end{aligned}$$
 - $++_\tau$: translated recursively, depending on τ
- User-defined functions: translated to SVCODE functions, unfolded at runtime when interpreting a SCall

- 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): **Proc** = (**BufState**, **S**, **Clis**, **Xducer**)
 - Buffer state maintained by process:
BufState ::= Filling \vec{a} | Draining \vec{a}' b
 - A process example:



Recursion example

A function to compute factorial:

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

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

```
1 -- Parameters: [S1] -- <3 7 0 4>
2 ... -- compare parameters with 1, get S5 = <T T FT T>
3 S6 := Usum S5; -- for elements <=1 -- < () >
4 [S7] := WithCtrl S6 []: S7 := Const_1 -- < 1 >
5 ...
6 S13 := Usum S11; -- for elementes >1 -- <()() ()>
7 [S17] := WithCtrl S13 [S12]:
8     S14 := Const_1 -- <1 1 1>
9     S15 := MapTwo Minus S12 S14 -- <2 6 3>
10    [S16] := SCall fact [S15] -- <2 720 6>
11        recursive call
12    S17 := MapTwo Times S12 S16 -- <6 5040 24>
13 ... -- merge results
14 S19 := PriSegInterS [(S7,S5),(S17,S11)]; -- <6 5040 1 24>
```

Formalization



- Types:

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

- Expressions:

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

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

- Key evaluation rules with work cost W :

- General comprehension:

$$\frac{([x \mapsto v_i, x_1 \mapsto n_1, \dots, x_k \mapsto n_k] \vdash e \downarrow v'_i \ \$ \ W_i)_{i=1}^l}{\rho \vdash \{e : x \ \mathbf{in} \ y \ \mathbf{using} \ x_1, \dots, x_k\} \downarrow \{v'_1, \dots, v'_l\} \ \$ \ W}$$

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

- Built-in function:

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

Target language: **SVCODE₀**

- syntax

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

- key semantics with work cost

- Empty new control stream ($\sigma(s_c) = \langle \rangle$):

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

where $\forall s \in \{s_c\} \cup S_{in}. \sigma(s) = \langle \rangle$, $S_{out} = \{s_1, \dots, s_k\}$

- Nonempty new control stream ($\sigma(s_c) = \vec{c}_1 \neq \langle \rangle$):

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

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

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

Xducer semantics

- General semantics: **Judgment**

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

-

$$\frac{\psi(\vec{a}_{11}, \dots, \vec{a}_{k1}) \Downarrow \vec{a}_{01} \quad \psi(\vec{a}_{12}, \dots, \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}}{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow^{(\langle \rangle | \vec{c}_0)} \vec{a}_0} ((\vec{a}_{i1} ++ \vec{a}_{i2} = \vec{a}_i)_{i=0}^k)$$

-

$$\overline{\psi(\langle \rangle_1, \dots, \langle \rangle_k) \Downarrow^{\langle \rangle} \langle \rangle}$$

- Specific semantics (part): **Judgment**

$$\boxed{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow \vec{a}}$$

$$\overline{\text{Const}_a() \Downarrow \langle a \rangle}$$

$$\overline{\text{ToFlags}(\langle n \rangle) \Downarrow \langle F_1, \dots, F_n, T \rangle} (n \geq 0)$$

$$\overline{\text{MapTwo}_+(\langle n_1 \rangle, \langle n_2 \rangle) \Downarrow \langle n_3 \rangle} (n_3 = n_1 + n_2)$$

$$\frac{\text{Usum}(\vec{b}) \Downarrow \vec{a}}{\text{Usum}(\langle F | \vec{b} \rangle) \Downarrow \langle () | \vec{a} \rangle}$$

$$\overline{\text{Usum}(\langle T \rangle) \Downarrow \langle \rangle}$$

SVCODE₀ determinism

Definition (Stream prefix)

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

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

Lemma (Blocks are self-delimiting)

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

Lemma (Xducer determinism)

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

Theorem (SVCODE₀determinism)

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

Translation formalization

- General comprehension translation:

$$\frac{[x \mapsto st_1, (x_i \mapsto s'_i)_{i=1}^k] \vdash e \Rightarrow_{s'_1}^{s'_k+1} (p_1, st_2)}{\delta \vdash \{e : x \text{ in } y \text{ using } x_1, \dots, x_k\} \Rightarrow_{s'_1}^{s'_0} (p, (st_2, s_b))}$$

$$\left(\begin{array}{l} \delta(y) = (st_1, s_b), (\delta(x_i) = s_i)_{i=1}^k \\ p = (s'_0 := \text{Usum}(s_b); \\ \quad (s'_k := \text{Distr}(s_b, s_k);)_{i=1}^k \\ \quad S_{out} := \text{WithCtrl}(s'_0, S_{in}, p_1)) \\ S_{in} = \overline{\overline{st_1}} \cup \{s'_1, \dots, s'_k\} \\ S_{out} = \{s \mid s \in \overline{\overline{st_2}}, s \geq s'_k + 1\} \\ s'_{i+1} = s'_i + 1, \forall i \in \{0, \dots, k-1\} \end{array} \right)$$

- translation well-formedness

Lemma

*If $\phi(st_1, \dots, st_k) \Rightarrow_{s_1}^{s_0} (p, st)$, $\bigcup_{i=1}^k \overline{\overline{st_i}} \subseteq S$, and $S \triangleleft s_0$,
then, for some S' , $S \Vdash p : S'$, $S' \subseteq \{s_0, s_0+1, \dots, s_1-1\}$, and $\overline{\overline{st}} \subseteq (S \cup S')$*

Theorem

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

Value representation formalization

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

$$\frac{}{n \triangleright_{\text{int}} \langle n \rangle} \quad \frac{(v_i \triangleright_{\tau} w_i)_{i=1}^l}{\{v_1, \dots, v_l\} \triangleright_{\{\tau\}} (w, \langle F_1, \dots, F_l, T \rangle)} (w = (++)_{\tau} w_i)_{i=1}^l$$

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

$$\frac{\langle n_0 | \vec{a} \rangle \triangleleft_{\text{int}} n_0, \vec{a}}{(w, \langle F_1, \dots, F_l, T | \vec{b} \rangle) \triangleleft_{\{\tau\}} \{v_1, \dots, v_l\}, (w_l, \vec{b})} \quad \frac{w \triangleleft_{\tau} v_1, w_1 \quad w_1 \triangleleft_{\tau} v_2, w_2 \quad \dots \quad w_{l-1} \triangleleft_{\tau} v_l, w_l}{} \quad \dots$$

Lemma (Recovery correctness)

If $v \triangleright_{\tau} w$, then $\forall w'. (w ++_{\tau} w') \triangleleft_{\tau} v, w'$.

Lemma (Recovery determinism)

If $w \triangleleft_{\tau} v, w'$, and $w \triangleleft_{\tau} v', w''$, then $v = v'$, and $w' = w''$.

Corollary

If $v \triangleright_{\tau} w$, $v' \triangleright_{\tau} w$, then $v = v'$.

Parallelism fusion lemma

Definition (Store similarity)

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

Definition (Store Concatenation)

For $\sigma_1 \stackrel{S}{\sim} \sigma_2$, $\sigma_1 \boxtimes \sigma_2 = \sigma$ 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 concatenation)

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

Lemma (Parallelism fusion)

If (i) $S_1 \Vdash p : S_2$, (ii) $\sigma_1 \stackrel{S}{\sim} \sigma_2$, (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 $\sigma'_1 \stackrel{S}{\sim} \sigma'_2$, $\left\langle p, \sigma_1 \boxtimes \sigma_2 \right\rangle \Downarrow^{\vec{c}_1 ++ \vec{c}_2} \sigma'_1 \boxtimes \sigma'_2 \$ W$, 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) $\delta \vdash e \Rightarrow_{s_1}^{s_0} (p, st)$
- (iv) $\forall x \in \text{dom}(\Gamma). \vdash \rho(x) : \Gamma(x)$
- (v) $\forall x \in \text{dom}(\Gamma). \overline{\delta(x)} \leq s_0$
- (vi) $\forall x \in \text{dom}(\Gamma). \rho(x) \triangleright_{\Gamma(x)} \sigma^*(\delta(x))$

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

- 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 some thinking about packing general types
- Error preservation: possible to support
- Recursion: consider termination preservation (from high-level to low-level) and reflection (from low-level to high-level)
- 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 working cost preservation

Future work:

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