

Formalizing the implementation of Streaming NESL

Master's Thesis

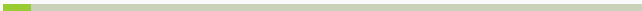
Dandan Xue

November 8, 2017

Department of Computer Science (DIKU)
University of Copenhagen

1. Introduction
2. Implementation
3. Formalization
4. Conclusion

Introduction



- A functional nested data-parallel language
- Developed by Guy Blelloch in 1990s at CMU
- Highlights:

- Highly expressive for parallel algorithms.
 - Data-parallel construct: *apply-to-each*

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

- Ex: compute $\sum_{i=1}^{10000} i^2$:

$$\text{sum}(\{i \times i : i \text{ in } [1, \dots, 10000]\})$$

! Allocate 10000 size of space for intermediate data

- 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 by Frederik 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)

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

- Expressions

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

- Expressions

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

- Primitive functions

$$\begin{aligned} \phi ::= & \oplus \mid \text{append} \mid \text{concat} \mid \text{zip} \mid \text{iota} \mid \text{part} \mid \text{scan}_{\otimes} \mid \text{reduce}_{\otimes} \\ & \mid \text{mkseq} \mid \text{the} \mid \text{empty} \mid \text{length} \mid \text{elt} \mid \text{seq} \mid \text{tab} \mid \dots \\ \oplus ::= & + \mid \times \mid / \mid == \mid \text{not} \mid \dots \\ \otimes ::= & + \mid \times \mid \text{max} \mid \dots \end{aligned}$$

- Expressions

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

- Primitive functions

$$\phi ::= \oplus \mid \text{append} \mid \text{concat} \mid \text{zip} \mid \text{iota} \mid \text{part} \mid \text{scan}_{\otimes} \mid \text{reduce}_{\otimes} \\ \mid \text{mkseq} \mid \text{the} \mid \text{empty} \mid \text{length} \mid \text{elt} \mid \text{seq} \mid \text{tab} \mid \dots$$
$$\oplus ::= + \mid \times \mid / \mid == \mid \text{not} \mid \dots$$
$$\otimes ::= + \mid \times \mid \text{max} \mid \dots$$

- $\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \equiv \text{let } b = e_0 \text{ in the}(\{e_1 \mid b\} ++ \{e_2 \mid \text{not}(b)\})$

SNESL primitive functions

append : $(\{\sigma\}, \{\sigma\}) \rightarrow \{\sigma\}$	syntactic sugar ++; $\{3, 1\}++\{4\} = \{3, 1, 4\}$
concat : $\{\{\sigma\}\} \rightarrow \{\sigma\}$	concat ($\{\{3, 1\}, \{4\}\}$) = $\{3, 1, 4\}$
zip : $(\{\sigma_1\}, \dots, \{\sigma_k\}) \rightarrow \{(\sigma_1, \dots, \sigma_k)\}$	zip ($\{1, 2\}, \{F, T\}$) = $\{(1, F), (2, T)\}$
iota ($\&$) : $\text{int} \rightarrow \{\text{int}\}$	$\&5 = \{0, 1, 2, 3, 4\}$
part : $(\{\sigma\}, \{\text{bool}\}) \rightarrow \{\{\sigma\}\}$	part ($\{3, 1, 4\}, \{F, F, T, F, T, T\}$) = $\{\{3, 1\}, \{4\}, \{\}\}$
scan $_{\otimes}$: $\{\text{int}\} \rightarrow \{\text{int}\}$	scan $_+$ ($\&5$) = $\{0, 0, 1, 3, 6\}$
reduce $_{\otimes}$: $\{\text{int}\} \rightarrow \text{int}$	reduce $_+$ ($\&5$) = 10
mkseq : $(\overbrace{(\sigma, \dots, \sigma)}^k) \rightarrow \{\sigma\}$	mkseq (1, 2, 3) = $\{1, 2, 3\}$
length (#): $[\tau] \rightarrow \text{int}$	$\#[10, 20] = 2$
elt (!): $([\tau], \text{int}) \rightarrow \tau$	$[3, 8, 2] ! 1 = 8$
the : $\{\sigma\} \rightarrow \sigma$	return the element of a singleton, the ($\{10\}$) = 10
empty : $\{\sigma\} \rightarrow \text{bool}$	empty ($\{1, 2\}$) = F, empty ($\&0$) = T
seq : $[\tau] \rightarrow \{\tau\}$	seq ($[1, 2]$) = $\{1, 2\}$
tab : $\{\tau\} \rightarrow [\tau]$	tab ($\{1, 2\}$) = $[1, 2]$

SNESL example: word count

```
-- split a string into words (delimited by spaces)
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})})
-- count the length of a stream
function slength(s) = reduce({1 : _ in s})
```

```
$> slength(str2wds_snesl(read_file(filename)))
```

SNESL example: word count

```
-- split a string into words (delimited by spaces)
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})})
-- count the length of a stream
function slength(s) = reduce({1 : _ in s})
```

```
$> slength(str2wds_snesl(read_file(filename)))
```

- Takes constant space
- Speedups of a similar word count program from [MF16]:

Benchmark	Speedup	Chunk size	Milliseconds
wc -w	0.65	N/A	19760
snesl-1	1.00	163840	12770
snesl-2	2.06	163840	6214
snesl-4	3.54	327680	3607
snesl-6	4.73	655360	2700
snesl-8	5.84	655360	2188
snesl-10	6.74	1310720	1894

Implementation



- Simplified SNESL types

$$\pi ::= \mathbf{bool} \mid \mathbf{int}$$
$$\tau ::= \pi \mid (\tau_1, \tau_2) \mid \{\tau\} \quad \text{(no vectors)}$$
$$\varphi ::= (\tau_1, \dots, \tau_k) \rightarrow \tau \quad \text{(support recursion)}$$

- Simplified SNESL types

$$\pi ::= \mathbf{bool} \mid \mathbf{int}$$
$$\tau ::= \pi \mid (\tau_1, \tau_2) \mid \{\tau\} \quad \text{(no vectors)}$$
$$\varphi ::= (\tau_1, \dots, \tau_k) \rightarrow \tau \quad \text{(support recursion)}$$

- Syntax

$$e ::= \dots \mid f(e_1, \dots, e_k) \quad \text{(user-defined function call)}$$
$$d ::= \mathbf{function} \ f(x_1: \tau_1, \dots, x_k: \tau_k): \tau = e$$

- Key typing rules, $\boxed{\Gamma \vdash_{\Sigma} e : \tau}$:

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

- Key evaluation rules, $\boxed{\rho \vdash_{\Phi} e \downarrow v}$:

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

Target language: **SVCODE**

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

Target language: SVCODE

- SVCODE values:

- primitive stream: $\vec{a} ::= \langle a_1, \dots, a_l \rangle$

- e.g., $\vec{a}_1 = \langle 1, 2 \rangle$, $\langle 0 | \vec{a}_1 \rangle = \langle 0, 1, 2 \rangle$, $\vec{b} = \langle F, T, F \rangle$

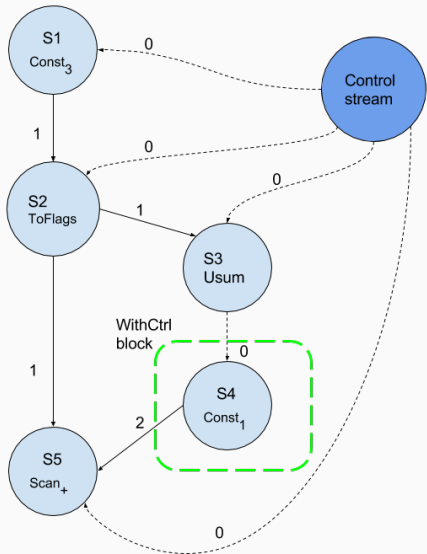
- stream tree: $w ::= \vec{a} \mid (w_1, w_2)$

- SVCODE syntax:

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

SVCODE dataflow DAG

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



Value representation

- A nested sequence with a nesting depth d is represented as a flattened data stream and d segment descriptor streams.

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

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

Translation

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

```
S4 := ...    -- <10 >      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: <10 10 10>
[S9] := WithCtrl S7 [S6,S8]: -- 3. translate (i+x)
      S9 := Map_+ S6 S8 -- <10,11,12>
```

Translation continued

- 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.,
$$\mathbf{iota}(s) \Rightarrow \left(\begin{array}{l} s_0 := \mathbf{ToFlags}(s); \\ s_1 := \mathbf{Usum}(s_0); \\ \{s_2\} := \mathbf{WithCtrl}(s_1, \emptyset, s_2 := \mathbf{Const}_1()); \\ s_3 := \mathbf{Scan}_+(s_0, s_2) \end{array} \right)$$
 - $++_\tau$: translated recursively, depending on τ

Translation continued

- 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.,
$$\mathbf{iota}(s) \Rightarrow \left(\begin{array}{l} s_0 := \mathbf{ToFlags}(s); \\ s_1 := \mathbf{Usum}(s_0); \\ \{s_2\} := \mathbf{WithCtrl}(s_1, \emptyset, s_2 := \mathbf{Const}_1()); \\ s_3 := \mathbf{Scan}_+(s_0, s_2) \end{array} \right)$$
 - $++_\tau$: translated recursively, depending on τ
- User-defined functions: translated to SVCODE functions, unfolded at runtime when interpreting a SCall

- Eager interpreter (NESL-like)
 - assumes sufficient memory for allocating all streams at once
 - executes each instruction sequentially
- Streaming interpreter
 - limited buffer size, space-usage efficient
 - result is collected from each scheduling round
 - DAG dynamically completed
 - need effective scheduling strategy to avoid deadlock and guarantee cost preservation
- Both instrumented with cost metrics (work, step), compared with the high-level one

Recursion example

A function to compute factorial:

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


Recursion example

A function to compute factorial:

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

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

```
-- Parameters: [S1]                                -- <3  7  0  4>
...-- compare parameters with 1,B2u:  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 >
      S17 := MapTwo Times S12 S16         -- <6 5040    24>
... -- merge results
S19 := PriSegInterS [(S7,S5),(S17,S11)];-- <6 5040 1  24>
```

Formalization



- Types:

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

- Key evaluation rules with work cost, $\boxed{\rho \vdash e \downarrow v \$ W}$:

- General comprehension:

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

where $\rho(y) = \{v_1, \dots, v_l\}$, 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**

- key semantics with work cost, $\boxed{\langle p, \sigma \rangle \Downarrow^{\vec{c}} \sigma' \$ W}$:
 - 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 $\forall s \in \{s_c\} \cup S_{in}. \sigma(s) = \langle \rangle$, $S_{out} = \{s_1, \dots, s_k\}$

Target language: **SVCODE₀**

- key semantics with work cost, $\boxed{\langle p, \sigma \rangle \Downarrow^{\vec{c}} \sigma' \$ W}$:

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

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

Target language: **SVCODE₀**

- key semantics with work cost, $\boxed{\langle p, \sigma \rangle \Downarrow^{\vec{c}} \sigma' \$ W}$:

- 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, $\boxed{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}}$
 - $$\frac{\psi(\vec{a}_{11}, \dots, \vec{a}_{k1}) \Downarrow \vec{a}_{01} \quad \psi(\vec{a}_{12}, \dots, \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}}{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow^{\langle () | \vec{c}_0 \rangle} \vec{a}_0} ((\vec{a}_{i1} ++ \vec{a}_{i2} = \vec{a}_i)_{i=0}^k)$$
 - $$\frac{}{\psi(\langle \rangle_1, \dots, \langle \rangle_k) \Downarrow^{\langle \rangle} \langle \rangle}$$

Xducer semantics

- General semantics, $\boxed{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}}$
 - $$\frac{\psi(\vec{a}_{11}, \dots, \vec{a}_{k1}) \Downarrow \vec{a}_{01} \quad \psi(\vec{a}_{12}, \dots, \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}}{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow^{\langle () | \vec{c}_0 \rangle} \vec{a}_0} ((\vec{a}_{i1} ++ \vec{a}_{i2} = \vec{a}_i)_{i=0}^k)$$
 - $$\overline{\psi(\langle \rangle_1, \dots, \langle \rangle_k) \Downarrow^{\langle \rangle} \langle \rangle}$$
- Block semantics (part), $\boxed{\psi(\vec{a}_1, \dots, \vec{a}_k) \Downarrow \vec{a}} :$
 - $$\overline{\text{Const}_a() \Downarrow \langle a \rangle} \quad \overline{\text{ToFlags}(\langle n \rangle) \Downarrow \langle F_1, \dots, F_n, T \rangle} \quad (n \geq 0)$$
 - $$\overline{\text{MapTwo}_+(\langle n_1 \rangle, \langle n_2 \rangle) \Downarrow \langle n_3 \rangle} \quad (n_3 = n_1 + n_2)$$
 - $$\overline{\text{Usum}(\vec{b}) \Downarrow \vec{a}} \quad \overline{\text{Usum}(\langle T \rangle) \Downarrow \langle \rangle}$$
 - $$\overline{\text{Usum}(\langle F | \vec{b} \rangle) \Downarrow \langle () | \vec{a} \rangle}$$

Translation formalization

- General comprehension, $\boxed{\delta \vdash e \Rightarrow_{s_1}^{s_0} (p, st)}$:

$$\frac{[x \mapsto st_1, (x_i \mapsto s'_i)_{i=1}^k] \vdash e_1 \Rightarrow_{s_1'}^{s_k'+1} (p_1, st_2)}{\delta \vdash \{e_1 : x \text{ in } y \text{ using } x_1, \dots, x_k\} \Rightarrow_{s_1'}^{s_0'} (p, (st_2, s_b))}$$

with

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

Value representation formalization

- Value representation, $\boxed{v \triangleright_{\tau} w}$:

$$\frac{\overline{n \triangleright_{\text{int}} \langle n \rangle}}{\frac{v_1 \triangleright_{\tau} w_1 \quad \cdots \quad v_l \triangleright_{\tau} w_l}{\{v_1, \dots, v_l\} \triangleright_{\{\tau\}} (w, \langle F_1, \dots, F_l, T \rangle)}} (w = w_1 ++_{\tau} \cdots ++_{\tau} w_l)$$

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

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

- Both representation and recovery are deterministic; high-level values and low-level ones are 1-1 corresponding.

Parallelism fusion lemma

Ex. $\text{let } x = 10 \text{ in } \{i + x : i \text{ in } \&3 \text{ using } x\} \Rightarrow:$

```
S6 := ...                -- <0, 1, 2>  -- data stream of &3
S7 := Usum S5;           -- <(),(),()>
S8 := Distr S4 S5;       -- <10,10,10>
[S9] := WithCtrl S7 [S6,S8]:
      S9 := Map_+ S6 S8   -- <10,11,12>  -- p
```

Parallelism fusion lemma

Ex. `let x = 10 in {i + x : i in &3 using x} ⇒:`

```
S6 := ...                -- <0, 1, 2 >  -- data stream of &3
S7 := Usum S5;           -- <(),(),()>
S8 := Distr S4 S5;       -- <10,10,10>
[S9] := WithCtrl S7 [S6,S8]:
      S9 := Map_+ S6 S8   -- <10,11,12>  -- p
```

σ_1

```
S6 := <0>
S7 := <()>
S8 := <10>
```

$\langle p, \sigma_1 \rangle \Downarrow^{S_7} \sigma'_1 \$ W_1$

```
...
S9 := <10>
```

Parallelism fusion lemma

Ex. `let x = 10 in {i + x : i in &3 using x} ⇒:`

```
S6 := ... -- <0, 1, 2> -- data stream of &3
S7 := Usum S5; -- <(),(),()>
S8 := Distr S4 S5; -- <10,10,10>
[S9] := WithCtrl S7 [S6,S8]:
      S9 := Map_+ S6 S8 -- <10,11,12> -- p
```

σ_1

```
S6 := <0>
S7 := <()>
S8 := <10>
```

$\langle p, \sigma_1 \rangle \Downarrow^{s_7} \sigma'_1 \$ W_1$

```
...
S9 := <10>
```

σ_2

```
S6 := <1,2>
S7 := <(),()>
S8 := <10,10>
```

$\langle p, \sigma_2 \rangle \Downarrow^{s_7} \sigma'_2 \$ W_2$

```
...
S9 := <11,12>
```

Parallelism fusion lemma

Ex. `let x = 10 in {i + x : i in &3 using x} ⇒:`

```
S6 := ... -- <0, 1, 2> -- data stream of &3
S7 := Usum S5; -- <(),(),()>
S8 := Distr S4 S5; -- <10,10,10>
[S9] := WithCtrl S7 [S6,S8]:
      S9 := Map_+ S6 S8 -- <10,11,12> -- p
```

σ_1

```
S6 := <0>
S7 := <()>
S8 := <10>
```

$\langle p, \sigma_1 \rangle \Downarrow^{S_7} \sigma'_1 \$ W_1$

```
...
S9 := <10>
```

σ_2

```
S6 := <1,2>
S7 := <(),()>
S8 := <10,10>
```

$\langle p, \sigma_2 \rangle \Downarrow^{S_7} \sigma'_2 \$ W_2$

```
...
S9 := <11,12>
```

$\sigma_1 \bowtie \sigma_2$

```
S6 := <0,1,2>
S7 := <(),(),()>
S8 := <10,10,10>
```

$\langle p, \sigma_1 \bowtie \sigma_2 \rangle \Downarrow^{S_7} \sigma'_1 \bowtie \sigma'_2 \$ W$

```
...
S9 := <10,11,12>
```

Parallelism fusion lemma

Ex. let $x = 10$ in $\{i + x : i \text{ in } \&3 \text{ using } x\} \Rightarrow$:

```
S6 := ... -- <0, 1, 2> -- data stream of &3
S7 := Usum S5; -- <(),(),()>
S8 := Distr S4 S5; -- <10,10,10>
[S9] := WithCtrl S7 [S6,S8]:
      S9 := Map_+ S6 S8 -- <10,11,12> -- p
```

σ_1

```
S6 := <0>
S7 := <()>
S8 := <10>
```

$\langle p, \sigma_1 \rangle \Downarrow^{S_7} \sigma'_1 \$ W_1$

```
...
S9 := <10>
```

σ_2

```
S6 := <1,2>
S7 := <(),()>
S8 := <10,10>
```

$\langle p, \sigma_2 \rangle \Downarrow^{S_7} \sigma'_2 \$ W_2$

```
...
S9 := <11,12>
```

$\sigma_1 \bowtie \sigma_2$

```
S6 := <0,1,2>
S7 := <(),(),()>
S8 := <10,10,10>
```

$\langle p, \sigma_1 \bowtie \sigma_2 \rangle \Downarrow^{S_7} \sigma'_1 \bowtie \sigma'_2 \$ W$

```
...
S9 := <10,11,12>
```

Lemma (Parallelism fusion, simplified version)

If $\langle p, \sigma_1 \rangle \Downarrow^{\vec{c}_1} \sigma'_1 \$ W_1$, and $\langle p, \sigma_2 \rangle \Downarrow^{\vec{c}_2} \sigma'_2 \$ W_2$,
then $\langle p, \sigma_1 \bowtie \sigma_2 \rangle \Downarrow^{\vec{c}_1 + \vec{c}_2} \sigma'_1 \bowtie \sigma'_2 \$ W$, and $W \leq W_1 + W_2$

Correctness theorem

- (i) If e (as well as its free variables) is well-typed, evaluated to v with cost W^H , translated to p
- (ii) then executing p will generate streams that can represent v , cost bounded by $C \cdot W^H$:

Theorem (Translation correctness, simplified version)

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) $\forall x \in \text{dom}(\Gamma). \rho(x) \triangleright_{\Gamma(x)} \sigma^*(\delta(x))$
then, (vii) $\langle p, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma' \$ W^L$ (viii) $v \triangleright_{\tau} \sigma'^*(st)$ (ix) $W^L \leq C \cdot W^H$

- Note: $C \geq 7$ (only for **iota**; other cases $C \sim 2$)

Correctness theorem

- (i) If e (as well as its free variables) is well-typed, evaluated to v with cost W^H , translated to p
- (ii) then executing p will generate streams that can represent v , cost bounded by $C \cdot W^H$:

Theorem (Translation correctness, simplified version)

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) $\forall x \in \text{dom}(\Gamma). \rho(x) \triangleright_{\Gamma(x)} \sigma^*(\delta(x))$
then, (vii) $\langle p, \sigma \rangle \Downarrow^{(l)} \sigma' \$ W^L$ (viii) $v \triangleright_{\tau} \sigma'^*(st)$ (ix) $W^L \leq C \cdot W^H$

- Note: $C \geq 7$ (only for **iota**; other cases $C \sim 2$)
- Corollary: Implementation correctness
 - (i) if e is well-typed, closed, evaluated to v with cost W^H
 - (ii) then e can be translated to p ; executing p generates streams that can recover v , cost bounded by $C \cdot W^H$
 - (iii) translation, execution of p , recovery are all deterministic

Conclusion

Conclusion

Main contributions:

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

Conclusion

Main contributions:

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

Future work:

- Extending the proof system to support more types, primitive functions, recursion, step & space preservation, etc.
- Formalization of the streaming semantics of the target language
- Formalization of parallel Xducers
- Investigation of schedulability, deadlock, etc.
 - a characterization of streamability
 - streamable programs do not deadlock