Formalizing the implementation of Streaming NESL

Dandan Xue

October 15, 2017

Abstract

Streaming NESL (SNESL) is a first-order functional, nested data-parallel language, employing a streaming execution model and integrating with a cost model that can predict both time and space complexity. The experimentation has demonstrated good performance of SNESL's implementation and positive empirical evidence of the validity of the code model. In this thesis, we first present non-trivial extension to SNESL's target language, SVCODE, which enables SNESL to support recursion at the same time preserve the cost. Then the formalization of the semantics of this low-level streaming language is given. Finally, we present the proof of the correctness of SNESL's implementation model.

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Nested data parallelism

1.2 NESL

NESL [Ble95] is a first-order functional nested data-parallel language. The main construct to express data-parallelism in NESL is called *apply-to-each*, whose form is similar to the list-comprehension in Haskell. As an example, adding 1 to each element of a sequence $[1, 2, 3]$ can be written as the following apply-to-each expression:

$$\{x + 1 : x \textbf{ in } [1, 2, 3]\}$$

which does the same computation as the Haskell expression

```
map (\x -> x + 1) [1,2,3]
```

but the low-level implementation is executed in parallel rather than sequentially.

The first highlight of NESL is that the design of this language makes it easy to write readable parallel algorithms. The apply-to-each construct is more expressive in its general form:

$$\{e_1 : x_1 \textbf{ in } seq_1 \ ; \ ... \ ; \ x_i \textbf{ in } seq_i \ \mid e_2\}$$

where the variables $x_1, ..., x_i$ possibly occurring in e_1 and e_2 are corresponding elements of $seq_1, ..., seq_i$ respectively; e_2, called a *sieve*, performs as a condition to filter out some elements. Also, NESL's built-in primitive functions, such as scan [Ble89], are powerful for manipulating sequences. An example program of NESL for splitting a string into words is shown in Figure 1.1.

The low-level language of NESL's implementation is VCODE. (some more about vcode)

1.2.1 Work-depth cost model

Another important idea of NESL is its language-based cost model [Ble96]. [PP93]. (some more)

```
-- split a string into words (delimited by spaces)
function str2wds(str) =
  let strl = #str;  -- string length
      spc_is =  { i : c in str, i in &strl | ord(c) == 32}; --
         space indices
      word_ls = { id2 - id1 -1 : id1 in [-1] ++ spc_is, id2 in
         spc_is++[strl]}; -- length of each word
      valid_ls = {l : l in word_ls | l > 0}; -- filter multiple
         spaces
      chars = {c : c in str | not(ord(c) == 32)}  -- non-space
         chars
   in partition(chars, valid_ls);   -- split strings into words
```

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

Figure 1.1: A NESL program for splitting a string into words

1.3 SNESL

Streaming NESL (SNESL) [Mad16] is a refinement of NESL that attempts to improve the efficiency of space usage. It extends NESL with two features: streaming semantics and a cost model for space usage. The basic idea behind the streaming semantics may be described as: data-parallelism can be realized not only in terms of space, as NESL has demonstrated, but also, for some restricted cases, in terms of time. When there is no enough space to store all the data at the same time, computing them chunk by chunk may be a way out. This idea is similar to the concept *piecewise execution* in [PPCF95], but SNESL makes the chunking exposed at the source level in the type system and the cost model instead of a low-level execution optimization.

1.3.1 Types

The types of a minimalistic version of SNESL defined in [Mad16] are:

$$\pi ::= \textbf{bool} \mid \textbf{int} \mid ...$$
$$\tau ::= \pi \mid (\tau_1, ..., \tau_k) \mid [\tau]$$
$$\sigma ::= \tau \mid (\sigma_1, ..., \sigma_k) \mid \{\sigma\}$$

Here π stands for the primitive types and τ the concrete types, both originally supported in NESL. The type $[\tau]$, which is called *sequences* in NESL and *vectors* in SNESL, represents spatial collections of homogeneous data, and must be fully allocated or *materialized* in memory at once for random access. $(\tau_1, ..., \tau_k)$ are tuples with k components that may be of different types.

The novel extension is the *streamable* types σ, which generalizes the types of data that are not necessarily entirely materialized at once, but rather in a streaming fashion. In particular, the type $\{\sigma\}$, called *sequences* in SNESL, represents collections of data computed in terms of time. So, even with a small size of memory,

SNESL could execute programs which is impossible in NESL due to space limitation or more space efficiently than in NESL.

For clarity, from now on, we will use the terms consistent with SNESL.

1.3.2 Values and expressions

The values of SNESL are as follows:

$$a ::= \mathtt{T} \mid \mathtt{F} \mid n\ (n \in \mathbb{Z}) \mid ...$$
$$v ::= a \mid (v_1, ..., v_k) \mid [v_1, ..., v_k] \mid \{v_1, ..., v_k\}$$

where a is the atomic values or constants of types π, and v are general values which can be a constant, a tuple, a vector or a sequence with k elements.

The expressions of SNESL are shown in Figure 1.2.

$e ::= a$	(constant)
$\mid x$	(variable)
$\mid (e_1, ..., e_k)$	(tuple)
$\mid$ **let** $x = e_1$ **in** e_2	(let-binding)
$\mid \phi(e_1, ..., e_k)$	(built-in function call)
$\mid \{e_1 :\ x$ **in** $e_0\}$	(general comprehension)
$\mid \{e_1 \mid e_0\}$	(restricted comprehension)

Figure 1.2: Syntax of SNESL expressions

As an extension of NESL, SNESL keeps a similar programming style of NESL. Basic expressions, such as the first five in Figure 1.2, are the same as in NESL. The apply-to-each construct in its general form splits into the general and the restricted comprehensions: the general one now is only responsible for performing parallel computation, and the restricted one can decide if a computation is necessary to do, working as the only conditional in SNESL. Also, these comprehensions extend the semantics of the apply-to-each from evaluating to vectors (i.e., type $[\tau]$) to evaluating to sequences (i.e., type $\{\sigma\}$). A notable difference between them is that the free variables of e_1 in the general comprehension can only be of concrete types, while they can be of any types in the restricted one.

1.3.3 Primitive functions

SNESL also extends the primitive functions of NESL non-trivially. The primitive functions of SNESL is shown in Figure 1.3.

The scalar functions of $\oplus$ and $\otimes$ should be self-explanatory from their conventional symbols. The types of the other functions and their brief description are given in Table 1.1.

The functions listed in (1.1) and (1.2) of Figure 1.3 are original supported in NESL, doing transformations on scalars and vectors. In SNESL, list (1.1) are adapted to streaming versions with slight changes of parameter types where necessary. By streaming version we mean that these functions in SNESL take sequences

$$\phi ::= \oplus \mid \textbf{append} \mid \textbf{concat} \mid \textbf{zip} \mid \textbf{iota} \mid \textbf{part} \mid \textbf{scan}_{\otimes} \mid \textbf{reduce}_{\otimes} \mid \textbf{mkseq} \tag{1.1}$$

$$\mid \textbf{length} \mid \textbf{elt} \tag{1.2}$$

$$\mid \textbf{the} \mid \textbf{empty} \tag{1.3}$$

$$\mid \textbf{seq} \mid \textbf{tab} \tag{1.4}$$

$$\oplus ::= + \mid - \mid \times \mid / \mid \% \mid == \mid <= \mid \textbf{not} \mid \ldots \quad \text{(scalar operations)}$$

$$\otimes ::= + \mid \times \mid \ldots \quad \text{(associative binary operations)}$$

Figure 1.3: SNESL primitive functions

as parameters instead of vectors as they do in NESL, as we can see from Table 1.1, thus most of these functions can execute in a more space-efficient way.

Functions in (1.2), i.e., **length** and **elt**, are kept their vector versions in SNESL. For **length**, this is because it is impossible to know the length of a sequence in advance before the sequence reaches its EOS (end of stream). Thus it is more reasonable to obtain the length by materializing all the elements at once. A similar reason applies to the element-indexing function **elt**.

List (1.3) are new primitives in SNESL. The function **the**, returning the sole element of a singleton sequence, can be used to simulate an if-then-else expression together with restricted comprehensions:

$$\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \equiv \textbf{the}(\{e_1 \mid e_0 \textbf{ using } \ldots\} +\!\!+ \{e_2 \mid \textbf{not}(e_0) \textbf{ using } \ldots\})$$

The function **empty**, which tests whether a sequence is empty or not, only needs to check at most one element of the sequence instead of materializing all the element. Therefore, it works in a fairly efficient way with a constant complexity both in time and space.

Finally, functions in list (1.4) connects the concrete types and streams, allowing SNESL to be flexible enough to not only write programs as NESL can do but also provide a streaming execution possibility.

The SNESL program for string splitting is shown in Figure 1.4. Compared with the NESL counterpart in Figure 1.1, the code of SNESL version is simpler, because SNESL's primitives make it good at streaming text processing. In particular, this SNESL version can be executed with even one element space.

```
1 -- partition a string to words (delimited by spaces)
2 -- SNESL version
3 function str2wds_snesl(str) =
4    let flags = {ord(x) == 32 : x in str};
5        nonsps =  concat({{x | ord(x) != 32} : x in v})
6     in concat({{x | not(empty(x))}: x in part(nonsps, flags ++
       {T})})
```

Figure 1.4: A SNESL program for splitting a string into words

Function type	Brief description
append : $\{\sigma\} \times \{\sigma\} \rightarrow \{\sigma\}$	append two sequences; syntactic sugar: infix symbol "++"
concat : $\{\{\sigma\}\} \rightarrow \{\sigma\}$	concatenates a sequence of sequences into one
zip : $\{\sigma_1\} \times ... \times \{\sigma_k\} \rightarrow \{(\sigma_1, ..., \sigma_k)\}$	convert k sequences into a sequence of k-component tuples
iota : **int** $\rightarrow$ {**int**}	generate an integer sequence starting from 0 to the given argument minus one; syntactic sugar: symbol "&"
part : $\{\sigma\} \times$ {**bool**} $\rightarrow \{\{\sigma\}\}$	partitions a sequence into subsequences segmented by Ts in the second argument; e.g., **part**$(\{3,1,4\}, \{\mathtt{F},\mathtt{F},\mathtt{T},\mathtt{F},\mathtt{T},\mathtt{T}\}) = \{\{3,1\},\{4\},\{\}\}$
scan$_\otimes$: {**int**} $\rightarrow$ {**int**}	performs an exclusive scan of $\otimes$ operation on the given sequence.
reduce$_\otimes$: {**int**} $\rightarrow$ **int**	performs a reduction of $\otimes$ operation on the given sequence
mkseq : $(\overbrace{\sigma, ..., \sigma}^{k}) \rightarrow \{\sigma\}$	make a k-component tuple to a sequence of length k
length: $[\tau] \rightarrow$ **int**	return the length of a vector; syntactic sugar: symbol "#"
elt: $[\tau] \times$ **int** $\rightarrow \tau$	return the element of a vector with the given index;syntactic sugar: infix symbol "!"
the : $\{\sigma\} \rightarrow \sigma$	return the element of a singleton sequence
empty : $\{\sigma\} \rightarrow$ **bool**	test if the given sequence is empty.
seq : $[\tau] \rightarrow \{\tau\}$	convert a vector into a sequence
tab : $\{\tau\} \rightarrow [\tau]$	tabulate a sequence into a vector

Table 1.1: SNESL primitive functions with types

1.3.4 Cost model

Based on the work-depth model, SNESL develops a third component of complexity measurement with regards to space. (more)

1.4 Mathematical background and notations

Chapter 2

Implementation

In this chapter, we will first talk about the high-level interpreter of a minimal SNESL language but with extension of user-defined functions to give the reader a more concrete feeling about SNESL. Then we introduce the streaming low-level language, SVCODE, with respect to its grammar, semantics and primitive operations. Translation from the high-level language to the low-level one will be explained to show their connections. Finally, two interpreters of SVCODE will be described and compared with emphasis on the latter one to demonstrate the streaming mechanism.

2.1 High-level interpreter

In this thesis, the high-level language we have experimented with is a subset of SNESL introduced in the last chapter but without vectors. We will call this language SNESL_1 . As our first goal is to extend SNESL with user-defined (recursive) functions, it is safe to do experiments only with SNESL_1 because removing vectors from SNESL should not affect the complexity of the problem too much; we believe that if the solution works with streams, the general type in SNESL, it should be trivial to extend it to support vectors.

Besides, only two primitive types of SNESL, **int** and **bool**, are retained in SNESL_1 . Tuples are also simplified to pairs. Thus the type structure for SNESL_1 is as follows.

$$\pi ::= \textbf{bool} \mid \textbf{int}$$
$$\tau ::= \pi \mid (\tau_1, \tau_2) \mid \{\tau\}$$

And the values in SNESL_1 are:

$$a ::= \texttt{T} \mid \texttt{F} \mid n$$
$$v ::= a \mid (v_1, v_2) \mid \{v_1, ..., v_k\}$$

The abstract syntax of SNESL_1 is given in Figure 2.1.

$$
\begin{array}{lr}
t ::= e \mid d & \text{(top-level statement)} \\
\\
e ::= a & \text{(constant)} \\
\quad \mid x & \text{(variable)} \\
\quad \mid (e_1, e_2) & \text{(pair)} \\
\quad \mid \{e_1, ..., e_k\} \qquad k \geq 1 & \text{(primitive sequence)} \\
\quad \mid \{\}\tau & \text{(empty sequence of type } \tau\text{)} \\
\quad \mid \textbf{let } x = e_1 \textbf{ in } e_2 & \text{(let-binding)} \\
\quad \mid \phi(e_1, ..., e_k) & \text{(built-in function call)} \\
\quad \mid \{e_1 : x \textbf{ in } e_0 \textbf{ using } x_1, ..., x_j\} & \text{(general comprehension)} \\
\quad \mid \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_j\} & \text{(restricted comprehension)} \\
\quad \mid f(e_1, ..., e_k) & \text{(user-defined function call)} \\
\\
d ::= \textbf{function } f(x_1 : \tau_1, ..., x_k : \tau_k) : \tau = e & \text{(user-defined function)}
\end{array}
$$

Figure 2.1: Abstract syntax of SNESL_1

In addition to the simplifications mentioned before, we also made the following changes or extensions on expressions:

- For comprehension expressions, the free variables of the comprehension body e_1 are collected as a list added after the keyword **using**. This task can simply be done by the compiler; we do this for presenting the details of implementation more conveniently.
- Added primitive sequence expression (including the empty one with its type explicitly given). They work as a replacement of the function **mkseq** or **seq** which we didn't use in SNESL_1 .
- Added user-defined functions which allow recursions. Since type inference is not incorporated in the interpreter, the types of parameters and return values need to be provided when the user defines a function.

The typing rules of SNESL_1 are given in Figure 2.2 and Figure 2.3. The type environment Γ is a mapping from variables to types:

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

and Σ from the function identities to their defined types:

$$\Sigma = [f_1 \mapsto \tau_1 \times ... \times \tau_k \to \tau, ..., f_i \mapsto \tau'_1 \times ... \times \tau'_m \to \tau']$$

Judgment $\boxed{\Gamma \vdash_\Sigma \ e : \tau}$

$$\frac{}{\Gamma \vdash_\Sigma \ a : \pi}\,(a : \pi) \qquad \frac{}{\Gamma \vdash_\Sigma \ x : \tau}\,(\Gamma(x) = \tau)$$

$$\frac{\Gamma \vdash_\Sigma \ e_1 : \tau_1 \qquad \Gamma \vdash_\Sigma \ e_2 : \tau_2}{\Gamma \vdash_\Sigma \ (e_1, e_2) : (\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash_\Sigma \ e_1 : \tau \qquad ... \qquad \Gamma \vdash_\Sigma \ e_k : \tau}{\Gamma \vdash_\Sigma \ \{e_1, ..., e_k\} : \{\tau\}} \qquad \frac{}{\Gamma \vdash_\Sigma \ \{\}\tau : \{\tau\}}$$

$$\frac{\Gamma \vdash_\Sigma \ e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash_\Sigma \ e_2 : \tau}{\Gamma \vdash_\Sigma \ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash_\Sigma \ e_1 : \tau_1 \qquad ... \qquad \Gamma \vdash_\Sigma \ e_k : \tau_k}{\Gamma \vdash_\Sigma \ \phi(e_1, ..., e_k) : \tau}\,(\Sigma(\phi) = \tau_1 \times ... \times \tau_k \to \tau)$$

$$\frac{\Gamma \vdash_\Sigma \ e_0 : \{\tau_0\} \qquad [x \mapsto \tau_0, (x_i \mapsto \tau_i)_{i=1}^j] \vdash_\Sigma \ e_1 : \tau}{\Gamma \vdash_\Sigma \ \{e_1 : x\ \textbf{in}\ e_0\ \textbf{using}\ x_1, ..., x_j\} : \{\tau\}}\,((\Gamma(x_i) = \tau_i \in \{\pi, (\pi, \pi)\})_{i=1}^j)$$

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

$$\frac{\Gamma \vdash_\Sigma \ e_1 : \tau_1 \qquad ... \qquad \Gamma \vdash_\Sigma \ e_k : \tau_k}{\Gamma \vdash_\Sigma \ f(e_1, ..., e_k) : \tau}\,(\Sigma(f) = \tau_1 \times ... \times \tau_k \to \tau)$$

Figure 2.2: Typing rules of SNESL_1 expressions

Judgment $\boxed{v : \tau}$

$$\frac{}{n : \textbf{int}} \qquad \frac{}{\texttt{T} : \textbf{bool}} \qquad \frac{}{\texttt{F} : \textbf{bool}}$$

$$\frac{v_1 : \tau_1 \qquad v_2 : \tau_2}{(v_1, v_2) : (\tau_1, \tau_2)} \qquad \frac{v_1 : \tau \qquad ... \qquad v_k : \tau}{\{v_1, ..., v_k\} : \{\tau\}}$$

Figure 2.3: Typing rules of SNESL_1 values

The type rules of SNESL_1 should be straightforward except those for comprehensions. For the general comprehension, the types of free variables of the comprehension body e_1 in the rules, i.e., $x_1, ..., x_j$, are all required to be non-sequence, i.e., only primitive types or pair of primitive types. This is because the values of these variables will be repeatedly used k times ($k \geq 0$) to evaluate the k elements of the comprehension, so they are not allowed to be streams as streams can only be traversed once.

The semantics of SNESL_1 is given in Figure 2.4 with the evaluation environment in the form $\rho = [x_1 \mapsto v_1, ..., x_i \mapsto v_i]$
(??TODO: add cost)

Judgment $\boxed{\rho \vdash_\Phi e \downarrow v}$

$$\frac{}{\rho \vdash_\Phi a \downarrow a} \qquad \frac{}{\rho \vdash_\Phi x \downarrow v}\,(\rho(x) = v) \qquad \frac{\rho \vdash_\Phi e_1 \downarrow v_1 \qquad \rho \vdash_\Phi e_2 \downarrow v_2}{\rho \vdash_\Phi (e_1, e_2) \downarrow (v_1, v_2)}$$

$$\frac{\rho \vdash_\Phi e_1 \downarrow v_1 \qquad ... \qquad \rho \vdash_\Phi e_k \downarrow v_k}{\rho \vdash_\Phi \{e_1, ..., e_k\} \downarrow \{v_1, ..., v_k\}} \qquad \frac{}{\rho \vdash_\Phi \{\}\tau \downarrow \{\}}$$

$$\frac{\rho \vdash_\Phi e_1 \downarrow v_1 \qquad \rho[x \mapsto v_1] \vdash_\Phi e_2 \downarrow v}{\rho \vdash_\Phi \textbf{let } e_1 = x \textbf{ in } e_2 \downarrow v}$$

$$\frac{\rho \vdash_\Phi e_1 \downarrow v_1 \qquad ... \qquad \rho \vdash_\Phi e_k \downarrow v_k}{\rho \vdash_\Phi \phi(e_1, ..., e_k) \downarrow v}\,((\Phi(\phi) = (v_1, ..., v_k) \downarrow v)$$

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

$$\frac{\rho \vdash_\Phi e_0 \downarrow \texttt{F}}{\rho \vdash_\Phi \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_j\} \downarrow \{\}} \qquad \frac{\rho \vdash_\Phi e_0 \downarrow \texttt{T} \qquad \rho \vdash_\Phi e_1 \downarrow v_1}{\rho \vdash_\Phi \{e_1 \mid x \textbf{ using } x_1, ..., x_j\} \downarrow \{v_1\}}$$

$$\frac{\rho \vdash_\Phi e_1 \downarrow v_1 \qquad ... \qquad \rho \vdash_\Phi e_k \downarrow v_k}{\rho \vdash_\Phi f(e_1, ..., e_k) \downarrow v}\,((\Phi(f) = (v_1, ..., v_k) \downarrow v)$$

Figure 2.4: Semantics of SNESL_1

Again, the evaluation rules are also straightforward except for the comprehensions. In the general comprehension, the bound sequence e_0 gets evaluated first, then its k elements are used to substitute the bound variable x in the comprehension body e_1; so e_1 will be evaluated k times with different substitutions of x.

For the restricted comprehension, the guard expression e_0 first gets evaluated: if it is a `T`, then e_1 will be evaluated, generating a singleton sequence; otherwise, e_1 will not be evaluated, and the comprehension only returns an empty sequence.

The built-in functions of SNESL_1 correspond to a subset of SNESL's shown in Figure 1.3 but without the vector-related ones and **mqseq**. In addition, the function **scan** and **reduce** are also taken a specific verison: picked + from $\otimes$, for simplicity, as shown in Figure 2.5.

$$\phi \ ::= \oplus \mid \textbf{append} \mid \textbf{concat} \mid \textbf{iota} \mid \textbf{part} \mid \textbf{scan}_{+} \mid \textbf{reduce}_{+} \mid \textbf{the} \mid \textbf{empty}$$
$$\oplus \ ::= + \mid - \mid \times \mid / \mid \% \mid <= \mid == \mid \textbf{not} \mid \ldots \qquad \text{(scalar operations)}$$

Figure 2.5: Primitive functions in SNESL_1

The types and semantics of these built-in functions remain the same as they are described in Table 1.1; we complement that brief version with more details where necessary and examples here.

- **append** : $\{\tau\} \times \{\tau\} \to \{\tau\}$, appends one sequence to the end of another; syntactic-sugared as the infix symbol ++.

Example 2.1.

```
> {3,1} ++ {4}
{3,1,4} :: {int}

> {{3,1},{4}} ++ {{}int} ++ {{1,5}}
{{3,1},{4},{},{1,5}} :: {{int}}
```

- **concat** : $\{\{\tau\}\} \to \{\tau\}$, concatenates a sequence of sequences into one, that is, decreases the nesting-depth by one.

Example 2.2.

```
> concat({{3,1},{4}})
{3,1,4} :: {int}

> concat({{{3,1}, {4}}, {{1}}})
{{3,1},{4},{1}} :: {{int}}
```

- **iota** : **int** $\to$ {**int**}, generates a sequence of integers starting from 0 to the given argument minus 1; syntactic-sugared as the symbol &; if the argument is negative, then reports a runtime error.

Example 2.3.

```
> &10
{0,1,2,3,4,5,6,7,8,9} :: {int}

> &0
{} :: {int}
```

- **part** : $\{\tau\} \times \{\textbf{bool}\} \to \{\{\tau\}\}$, partitions a sequence into subsequences according to the second boolean sequence, where a `F` corresponds to one element of

the first argument and a `T` indicates a segment separation; so the number of `F`s is equal to length of the first argument, and the number of `T`s is equal to the length of the returned value, and it must end with a `T`. If the second argument does not satisfy the requirement, a runtime error will be reported.

Example 2.4.

```
> part({3,1,4,1,5,9}, {F,F,T,F,T,T,F,F,F,T})
{{3,1},{4},{},{1,5,9}} :: {{int}}

> part({{F,T},{T},{}bool, {F,F}}, {F,F,T,F,F,T})
{{{F,T},{T}},{{},{F,F}}} :: {{{bool}}}
```

- **scan**$_+$: $\{\textbf{int}\} \to \{\textbf{int}\}$, performs an exclusive scan of addition operation on the given sequence, that is, assuming the argument is $\{n_1, n_2, .., n_{k-1}, n_k\}$, to compute $\{0, n_1, n_1 + n_2, ..., n_1 + ... + n_{k-1}\}$. This scan operation has its general form in full SNESL, where it supports more associative binary operations with a specified identity element e_0 such that $e_0 \otimes e_0 = e_0$.

Example 2.5.

```
> scanPlus({3,1,4,1})
{0,3,4,8} :: {int}

> scanPlus({}int)
{} :: {int}
```

- **reduce**$_+$: $\{\textbf{int}\} \to \textbf{int}$, performs a reduction of addition operation on the given sequence, i.e., computes its sum. Again, this function also has its general form in full SNESL, where it computes $a_1 \otimes a_2 \otimes ... \otimes a_k$ for an argument $\{a_1, a_2, ..., a_k\}$.

Example 2.6.

```
> reducePlus({3,1,4,1})
9 :: int

> reducePlus({}int)
0 :: int
```

- **the** : $\{\tau\} \to \tau$, returns the element of a singleton sequence; if the length of the argument is not exact one, reports a runtime error.

Example 2.7.

```
> the({3})
3 :: int

> the({(3,1)})
(3,1) :: (int,int)
```

- **empty** : $\{\tau\} \to$ **bool**, tests if the given sequence is empty; if it is empty, returns a `T`, otherwise returns a `F`.

Example 2.8.

```
> empty({3,1,4,1})
F :: bool

> empty({}int)
T :: bool
```

2.2 Value representation

At the low level, a value of SNESL$_1$ is represented as either a primitive *stream* $\vec{a}$, a collection of primitive values, or a binary tree structure w with stream leaves. We use $\langle a_1, ..., a_k \rangle$ to denote a primitive stream which consists of k elements $a_1, ..., a_k$. Thus,

$$\vec{a} ::= \langle a_1, ..., a_k \rangle$$
$$w ::= \vec{a} \mid (w_1, w_2)$$

The representation also relies on the type of the value. We use the infix symbol "$\triangleright$" subscripted by a type τ to denote a type-depended representation relation.

Some simple but representative cases are given below. More formal and general rules will be presented in the next chapter.

- A primitive value is represented as a singleton primitive stream:

  Example 2.9.

  $$3 \triangleright_{\mathbf{int}} \langle 3 \rangle$$
  $$\mathtt{T} \triangleright_{\mathbf{bool}} \langle \mathtt{T} \rangle$$

- A non-nested/flat sequence of length n is represented as a primitive *data stream* with an auxiliary boolean stream called a *descriptor*, which consists of n number of `F`s followed by one `T` denoting the end of a segment.

  Example 2.10.

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

- For a nested sequence with a nesting depth d (or a d-dimensional sequence), all the data are flattened to a data stream, but d descriptors are used to maintain the segment information at each depth. (Thus a non-nested sequence is just a special case of $d = 1$).

  Example 2.11.

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

- A pair of high-level values is a pair of streams (or stream trees) representing the two high-level components respectively.

  Example 2.12.

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

- A sequence of pair can be regarded as a pair of sequences sharing a descriptor at the low level:

  Example 2.13.

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

2.3 SVCODE

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 and near-linear scales to multicore.

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.

2.3.1 SVCODE Syntax

The abstract syntax of SVCODE is given in Figure 2.6. An SVCODE program p is basically a list of commands or instructions each of which defines one or more streams. We use s as a stream variable, called a stream *id*, and S a bunch of stream ids. As a general rule of reading an SVCODE instruction, the stream ids on the left-hand side of a symbol ":=" are the defined streams, and the right-hand side ones are possibly used to generate those new ones.

The instructions in SVCODE that define only one stream are in the form

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

where ψ is a primitive function, called a *Xducer*(transducer), taking the stream $s_1, ..., s_k$ as parameters and returning s. More detailed descriptions for specific Xducers are given in the next subsection.

The only essential control struture in SVCODE is the `WithCtrl` instruction

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

which may or may not execute a piece of SVCODE program p_1, but always defines a bunch of stream ids S_{out}. The definition instructions for all the stream ids in S_{out} are included in the code block p_1. Whether to execute p_1 or not depends on the value of the stream s, the *new control stream*, at runtime:

- If s is an empty stream, then execute p_1 and compute S_{out} as usual
- Otherwise, skip p_1 and assign S_{out} all empty streams

$$
\begin{array}{lll}
p & ::= \ \epsilon & \text{(empty program)} \\
& \mid s := \psi(s_1, ..., s_k) & \text{(single stream definition)} \\
& \mid S_{out} := \texttt{WithCtrl}(s, S_{in}, p_1) & (\texttt{WithCtrl} \text{ block}) \\
& \mid (s'_1, ..., s'_n) := \texttt{SCall}\ f(s_1, ..., s_m) & \text{(SVCODE function call)} \\
& \mid p_1; p_2 & \\
\\
s & ::= 0 \mid 1 \ ... \in \mathbf{SId} = \mathbb{N} & \text{(stream ids)} \\
S & ::= \{s_1, ..., s_i\} \in \mathbb{S} & \text{(set of stream ids)} \\
\\
\psi & ::= \texttt{Const}_{\texttt{a}} \mid \texttt{ToFlags} \mid \texttt{Usum} \mid \texttt{Map}_{\oplus} \mid \texttt{Scan}_{\otimes} \mid \texttt{Reduce}_{\otimes} \mid \texttt{Distr} & \text{(Xducers)} \\
& \quad \mid \texttt{Pack} \mid \texttt{UPack} \mid \texttt{B2u} \mid \texttt{SegConcat} \mid \texttt{USegCount} \mid \texttt{InterMerge} \mid ... & \\
\oplus & ::= + \mid \ - \ \mid \ \times \ \mid / \mid \% \mid <= \ \mid \ == \ \mid \mathbf{not} \mid ... & \text{(scalar operations)} \\
\otimes & ::= + \mid \ \times \ \mid ... & \text{(associative binary operations)}
\end{array}
$$

Figure 2.6: Abstract syntax of SVCODE

Thus the new control stream is the most important role here, because it decides whether or not to execute p_1, which is the key to avoiding infinite unfolding of recursive functions. S_{in} is the variable set including all the streams that are referred to by p_1. It will only affect the streaming execution model of SVCODE; in the eager model, it can be totally ignored.

The instruction $(s'_1, ..., s'_n) := \texttt{SCall}\ f(s_1, ..., s_m)$ can be read as: "calling function f with arguments $s_1, ..., s_m$ returns $s'_1, ..., s'_n$". The function body of f is merely another piece of SVCODE program, but without the definition instructions of its argument streams.

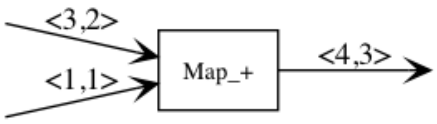
It is worth noting that a well-formed SVCODE instruction should always assign $fresh$ (never used) stream ids to the defined streams, in which way the dataflow of an SVCODE program can construct a DAG (directed acyclic graph). We will give more formal definitions of this language in the next chapter to demonstrate how the freshness property is guaranteed. In the practical implementation, we simply identify each stream with a natural number, a smaller one always defined earlier than a greater one.

2.3.2 Xducers and control stream

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

For example, the Xducer $\texttt{Map}_{+}(\langle 3, 2\rangle, \langle 1, 1\rangle)$ consumes the stream $\langle 3, 2\rangle$ and $\langle 1, 1\rangle$, then outputs the element-wise addition result $\langle 4, 3\rangle$.

Example 2.14. $\texttt{Map}_{+}(\langle 3, 2\rangle, \langle 1, 1\rangle)$:



As we have mentioned before, the dataflow of an SVCODE program is basically a DAG, where each Xducer stands for one node. The `WithCtrl` block is only a subgraph that may be added to the DAG at runtime, and `SCall` another that will be unfolded dynamically.

Figure 2.7 shows an example program, with its DAG in Figure 2.8.

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

Figure 2.7: A small SVCODE program

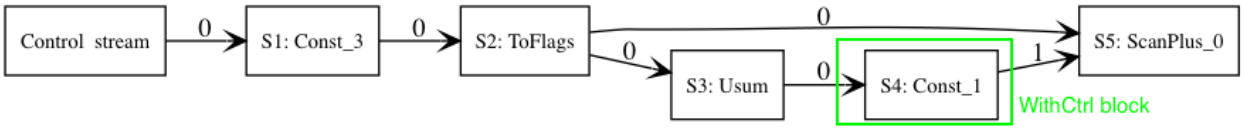


Figure 2.8: Dataflow DAG for the code in Figure 2.7 (assuming S3 is nonempty). Note that, for simplicity, the control stream is added as an explicit supplier only to Xducer $\texttt{Const}_a$.

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 2.8, 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.

An issue here is dealing with runtime errors of Xducers. For instance, the $\texttt{Map}_+$ Xducer can consume a) the stream $\langle 3, 2\rangle$ with $\langle 1, 1, 5\rangle$, or b) the stream $\langle 3, 2, 5\rangle$ with $\langle 1, 1\rangle$, and outputs the same result $\langle 4, 3\rangle$. But apparently there is a runtime error, since the two input streams are expected to be the same length, and we can not decide whether the correct length should be two or three, or maybe none of them. Another special case is the Xducer $\texttt{Const}_a()$, which outputs some number of constants a, even takes no argument stream.

In the implementation of [Mad16], a special stream of unit type, called the *control stream*, is used at the compiling time basically for replicating constants, and it has nothing to do with the runtime.

In our implementation, we move it to the runtime, so that it will not only control constants replication, but more importantly, dominate all the Xducers' behavior. Our observation is that the parallel degree throughout the whole computation is expressed by this control stream, and all the Xducers should behave correspondingly to this parallel degree, or in other words, the parallel degree propagates through the dataflow DAG.

In practice, a Xducer can first consume the control stream to "read" the parallel degree it should compute, then it consumes its normal input streams and outputs the expected number of elements. There can be three benefits by doing so.

First, Xducers now can easily check a runtime error. For example, when the parallel degree is two, i.e., the control stream is $\langle((),())\rangle$, the Xducer $\texttt{Map}_+$ will know that it only needs to consume two elements from each input stream, and output two as well; all the other cases will be reported as runtime errors. And the Xducer $\texttt{Const}_\texttt{a}$ just outputs the equal number of elements to the length of the control stream.

Another benefit is that all the Xducers will behave in a more uniform and regular way, which is easier for reason about and formalization, as we will show in the next chapter.

Finally, the functionality of Xducers can be completely independent or separated from the scheduling in the streaming execution model, which makes the Xducer easier to be extended or changed, and the implementation model more flexible and easier to debug.

2.4 Translating SNESL_1 to SVCODE

In Chapter 2.2, we have seen the idea of how a high-level value of SNESL_1 can be represented as a binary tree of low-level stream values. At the compiling time, we use a structure **STree** (stream id tree) to generalize a binary tree of stream ids, so that the high-level variables and the low-level ones can be connected:

$$\textbf{STree} \ni st ::= s \mid (st_1, st_2)$$

The translation environment δ is a mapping from high-level variables to stream trees:

$$\delta = [x_1 \mapsto st_1, ..., x_i \mapsto st_i]$$

Another important component maintained at the compiling time is a fresh stream id, which has not been used yet. It will be assigned to the defined stream(s) of the newly generated instruction.

We will use the symbol "$\Rightarrow$" superscripted with the fresh id to denote the translation relation. Also, a subscript may be used to denote the new fresh id after a certain piece of code has been generated.

2.4.1 Expression translation

An SNESL_1 expression will be translated to a pair of an SVCODE program p and a stream tree st whose stream values represents the high-level evaluation result:

$$\delta \ \vdash \ e \Rightarrow_{s_1}^{s_0} (p, st)$$

The translation for constants, variables and pairs are straightforward. For example, a pair $(x, 4)$ will be translated to a program of only one instruction $s_0 := \texttt{Const}_4()$ and a stream tree (s, s_0), assuming in the context x is bound to s and s_0 is a fresh id before the translation :

$$[x \mapsto s] \ \vdash \ (x, 4) \Rightarrow^{s_0} (s_0 := \texttt{Const}_4(), (s, s_0))$$

For a let-binding expression **let** $x = e_1$ **in** e_2, first e_1 gets translated to some code p_1 and a stream tree st_1 as usual; then the binding $[x \mapsto st_1]$ is added to δ, in which the body e_2 gets translated to p_2 with st_2; the translation of the entire expression will be the concatenation of p_1 and p_2, i.e., $p_1; p_2$, and the stream tree is only st_2.

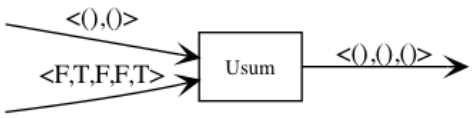
Translations for specific built-in functions and user-defined functions will be given later.

Non-empty primitive sequence looks a little bit tricky to translate as it can have arbitrary number (≥ 1) of elements, but it is basically a general version of the function **append**. For the empty sequence $\{\}\tau$, the low-level streams are all empty streams except for the outermost descriptor $\langle \texttt{T} \rangle$, and the number of those empty streams depends on the type τ.

The most interesting case may be the comprehensions. For the general one, $\{e_1 : x \ \textbf{in} \ e_0 \ \textbf{using} \ x_1, ..., x_j\}$, first we need to translate e_0, whose type must be a sequence, and we obtain a program p_0 and a stream tree that must look like (st_0, s_b) where s_b is the outermost descriptor. Recall that the number of Fs of a descriptor is equal to the length of the inner data stream, assuming it is k, and the comprehension body e_1 will be evaluated k times with x being bound to different elements of the data stream st_0, thus the parallel degree of computing e_1 can be represented by employing a Usum Xducer on s_b.

The Xducer $\texttt{Usum}(\vec{b})$ takes one boonlean stream as argument, and transforms an F to a unit, or a T to nothing. It is the only Xducer that can generate a unit vector in our instruction set, so it is mainly used when we need to replace the control stream.

Example 2.15. $\texttt{Usum}(\langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T} \rangle)$ with the control stream $= \langle (), () \rangle$:



If there is no free variables in e_1, then the next step will be translating the comprehension body e_1 in the new environment $\delta[x \mapsto st_1]$, and we are done.

For the case where e_1 uses some free variables $x_1, ..., x_j$ defined earlier than this comprehension translation, we need to generate new streams, each of which is a k-replicate or a distribution of the stream of $x_1, ..., x_j$ respectively.

The translation will look like:

$$\frac{\delta \ \vdash \ e_1 \Rightarrow (p_1, (st_1, s_b)) \qquad [x \mapsto st_1, (x_i \mapsto st'_i)_{i=1}^{j}] \ \vdash \ e \Rightarrow^{s_1} (p_2, st)}{\delta \ \vdash \ \{e_1 : x \ \textbf{in} \ e_0 \ \textbf{using} \ x_1, ..., x_j\} \Rightarrow (p, (st, s_b))} \ ((\delta(x_i) = \ st_i)_{i=1}^{j})$$

in which

$$
\begin{aligned}
p = p_1;&\\
& s_1 := \texttt{Usum}(s_b);\\
& st'_1 := \texttt{distr}(s_b, st_1);\\
& \qquad\vdots\\
& st'_j := \texttt{distr}(s_b, st_j);\\
& S_{out} := \texttt{WithCtrl}(s_1, S_{in}, p_2)
\end{aligned}
$$

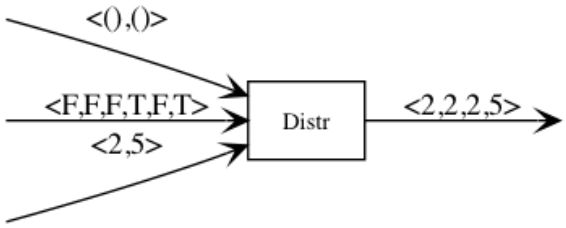
Note that we put p_2 into a `WithCtrl` block so that p_2 can be skipped when s_1 is tested to be an empty stream at runtime. S_{in} and S_{out} are some analysis results about the free stream variables and the defined ones of p_2, which can be easily obtained by traversing p_2. We will give more details about them in the next chapter.

The function `distr` is responsible for replicating streams by employing the Xducer `Distr`. We give its definition in a form close to the SVCODE style for more readability:

$$
\begin{aligned}
s' := \texttt{distr}(s_b, s); \quad &\equiv \quad s' := \texttt{Distr}(s_b, s);\\
(st'_1, st'_2) := \texttt{distr}(s_b, (st_1, st_2)); \quad &\equiv \quad st'_1 := \texttt{distr}(s_b, st_1);\\
& \qquad\quad st'_2 := \texttt{distr}(s_b, st_2);
\end{aligned}
$$

The Xducer `Distr` consumes a boolean stream as a segment descriptor of the data stream, and replicates the constants of the data stream corresponding times to their segment lengths.

Example 2.16. $\texttt{Distr}(\langle \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{T}\rangle, \langle 2, 5\rangle)$ with control stream $\langle (), ()\rangle$



The restricted comprehension is a little bit simpler compared to the general one, since there is no variable-bindings in e_1 and the parallel degree of the computation of e_1 is either one or zero, thus the free variables $x_1, ..., x_j$ does not need to be distributed, but rather, *pack*ed. Its translation will look like:

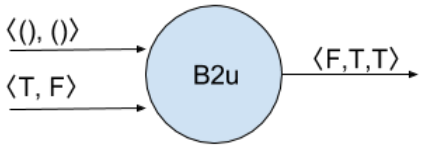
$$
\frac{\delta \;\vdash\; e_1 \Rightarrow (p_1, s_b) \qquad [(x_i \mapsto st'_i)_{i=1}^{j}] \;\vdash\; e \Rightarrow^{s_1} (p_2, st)}{\delta \;\vdash\; \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_j\} \Rightarrow (p, (st, s_1))} ((\delta(x_i) = \; st_i)_{i=1}^{j})
$$

in which

$$
\begin{aligned}
p = p_1;&\\
& s_1 := \texttt{B2u}(s_b);\\
& s_2 := \texttt{Usum}(s_1);\\
& st'_1 := \texttt{pack}_{\tau_1}(s_b, st_1);\\
& \qquad\vdots\\
& st'_j := \texttt{pack}_{\tau_j}(s_b, st_j);\\
& S_{out} := \texttt{WithCtrl}(s_2, S_{in}, p_2)
\end{aligned}
$$

The Xducer `B2u` simply transforms a boolean to a unary number, i.e., transforms $\langle \texttt{F} \rangle$ to $\langle \texttt{T} \rangle$, and $\langle \texttt{T} \rangle$ to $\langle \texttt{F}, \texttt{T} \rangle$,.

Example 2.17. $\texttt{B2u}(\langle \texttt{T}, \texttt{F} \rangle)$ with control stream $\langle (), () \rangle$

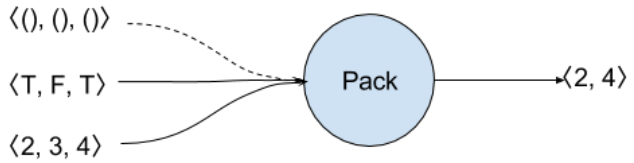


The function $\texttt{pack}_\tau$ annotated with the type of the packed variable will generate instructions using `Pack` and possibly `UPack` and `Distr`:

$$
\begin{aligned}
s' := \texttt{pack}_{\pi}(s_b, s); \quad &\equiv \quad s' := \texttt{Pack}(s_b, s);\\
(st'_1, st'_2) := \texttt{pack}_{(\tau_1,\tau_2)}(s_b, (st_1, st_2)); \quad &\equiv \quad st'_1 := \texttt{pack}_{\tau_1}(s_b, st_1);\\
& \qquad\quad st'_2 := \texttt{pack}_{\tau_2}(s_b, st_2);\\
(st'_1, s'_1) := \texttt{pack}_{\{\tau\}}(s_b, (st_1, s_1)); \quad &\equiv \quad s'_1 := \texttt{UPack}(s_b, s_1);\\
& \qquad\quad s'_2 := \texttt{Distr}(s_1, s_b);\\
& \qquad\quad st'_1 := \texttt{pack}_{\tau}(s'_2, st_1);
\end{aligned}
$$

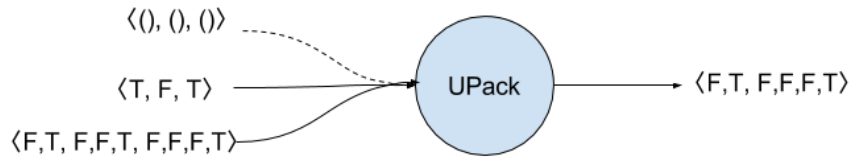
`Pack` throws the element of the second stream if the boolean of the corresponding position in the first stream is a `F`.

Example 2.18. $\texttt{Pack}(\langle \texttt{T}, \texttt{F}, \texttt{T} \rangle, \langle 2, 3, 4 \rangle)$ with control stream $\langle (), (), () \rangle$



`UPack` works similarly to `Pack`, but on data of boolean segments rather then primitive values.

Example 2.19. $\texttt{UPack}(\langle \texttt{T}, \texttt{F}, \texttt{T} \rangle, \langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T} \rangle)$ with control stream $\langle (), (), () \rangle$



2.4.2 Built-in function translation

A high-level built-in function call will be translated to a few lines of SVCODE instuctions. For example,

- Scalar operations, for instance $x_1 \oplus x_2$, will be translated to a single instruction $\texttt{Map}_{\oplus}(s_1, s_2)$, assuming $\delta(x_1) = s_1, \delta(x_2) = s_2$.
- The function **iota**$(()n)$ generates an integer sequence starting from 0 with the length of n. The translation will first use the instruction `ToFlags` to generate the descriptor of the return value, and then perform a scan operation on a stream of n 1s to generate the data stream. Its translation will look like:

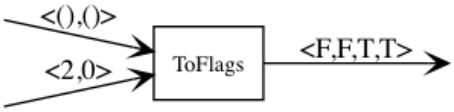
$$\frac{}{\textbf{iota}(s) \Rightarrow^{s_0} (p, (s_3, s_0))}$$

where

$$\begin{aligned} p = & s_0 := \texttt{ToFlags}(s); \\ & s_1 := \texttt{Usum}(s_0); \\ & [s_2] := \texttt{WithCtrl}(s_1, [], s_2 := \texttt{Const}_1()); \\ & s_3 := \texttt{ScanPlus}_0(s_0, s_2) \end{aligned}$$

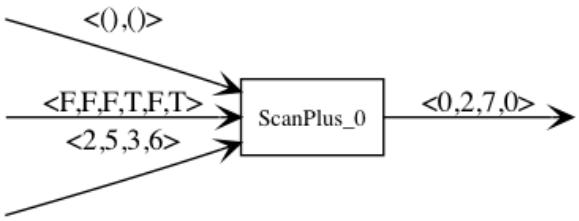
Given a stream $\langle n \rangle$, the Xducer `ToFlags` first outputs n Fs, then one T.

Example 2.20. $\texttt{ToFlags}(\langle 2, 0 \rangle)$:



$\texttt{ScanPlus}(s_b, s_d)$ performs an exclusive scan of on the data stream s_d with the descriptor s_b.

Example 2.21. $\texttt{ScanPlus}(\langle \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{T} \rangle, \langle 2, 5, 3, 6 \rangle)$



- The high-level function $\mathbf{scan}_+$ has a corresponding low-level Xducer, `ScanPlus`, thus its translation is straightforward.

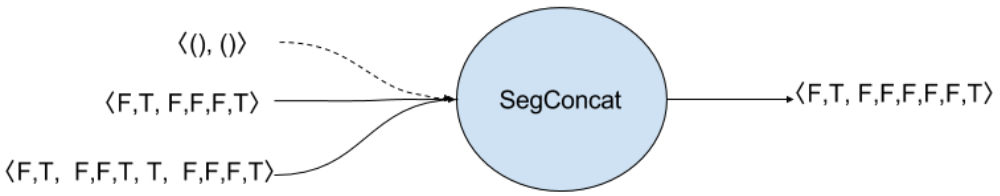
$$\overline{\mathbf{scan}_+((s_d, s_b)) \Rightarrow^{s_0} (s_0 := \texttt{ScanPlus}(s_b, s_d), (s_0, s_b))}$$

- Translating $\mathbf{reduce}_+$ is analogous to that of $\mathbf{scan}_+$ by using the Xducer `ReducePlus`.
- The translation of **concat** is also one instruction using the Xducer `SegConcat`:

$$\overline{\mathbf{concat}(((st, s_1), s_2)) \Rightarrow^{s_0} (s_0 := \texttt{SegConcat}(s_2, s_1), (st, s_0))}$$

The Xducer `SegConcat` merges the second outermost descriptors, i.e., s_1, into a new one s_0 by removing unnecessary segment boundary `T`s; the old outermost descriptor s_2 helps maintain the segmenting information.

Example 2.22. $\texttt{SegConcat}(\langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T} \rangle, \langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T} \rangle)$ with control stream $\langle (), () \rangle$. The second argument has 4 segments, and the first argument says that the first one will be merged as one segment, and the other three together as another.



- **part** can be implemented straightforward by the Xducer `USegCount`.
- Implementing the function **append** at the low level may be the most tricky one, since it needs to recursively append subsequences at each depth of the argument sequences:

$$\overline{\mathbf{append}((st_1, s_1), (st_2, s_2))) \Rightarrow^{s_0} (p, (st, s_0))}$$

where

$$\begin{aligned} p = s_0 &:= \texttt{InterMerge}([s_1, s_2]); \\ st &:= \texttt{mergeRecur}_{\{\tau\}}([(st_1, s_1), (st_2, s_2)]); \end{aligned}$$

The Xducer `InterMerge` merges two descriptors by interleaving their segments.

The function `mergeRecur` annotated by the type of the argument merges the inner segments recursively. Its definition is given below. The Xducer `PrimSegInter` merges the given data streams according to their descriptors, similar to `InterMerge` but working on primitive data instead of boolean segments.

`SegInter` merger the segments of a descriptor to segmented by the second argument.

$$s := \texttt{mergeRecur}_{\{\pi\}}([(s'_1, s_1), (s'_2, s_2)]); \quad \equiv \quad s := \texttt{PrimSegInter}([(s'_1, s_1), (s'_2, s_2)]);$$

$$\begin{aligned}(st, st') := \texttt{mergeRecur}_{\{(\tau_1,\tau_2)\}}([((st_1, st'_1), s_1), ((st_2, st'_2), s_2)]); \quad \equiv \\ st := \texttt{mergeRecur}_{\{\tau_1\}}([(st_1, s_1), (st_2, s_2)]); \\ st' := \texttt{mergeRecur}_{\{\tau_2\}}([(st'_1, s_1), (st'_2, s_2)]);\end{aligned}$$

$$\begin{aligned}(st, s_3) := \texttt{mergeRecur}_{\{\{\tau\}\}}([((st_1, s_1), s'_1), ((st_2, s_2), s'_2)]); \quad \equiv \\ s_3 := \texttt{SegInter}([(s_1, s'_1), (s_2, s'_2)]); \\ s_4 := \texttt{SegConcat}(s_1, s'_1); \\ s_5 := \texttt{SegConcat}(s_2, s'_2); \\ st := \texttt{mergeRecur}_{\{\tau\}}([(st_1, s_4), (st_2, s_5)]);\end{aligned}$$

Note that we make the argument of `InterMerge`, `mergeRecur`, `SegInter` and `PrimSegInter` all a list of stream trees instead of exact two, thus they can be used to append ≥ 1 number of sequences.

- Implementing the function **the** needs runtime check on the length of the sequence, which is done by the Xducer `Check`; if the length satisfies the requirement, then the data stream can be returned.
- **empty** also has a corresponding low-level Xducer `IsEmpty`.

2.4.3 User-defined function translation

We first define the type of SVCODE functions **SFun**: a triple in the form $([s_1, ..., s_m], p, [s'_1, ..., s'_n])$, where $s_1, ..., s_m$ are the argument stream ids, p the function body, and $s'_1, ..., s'_n$ the returne values, that is,

$$sf ::= ([s_1, ..., s_m], p, [s'_1, ..., s'_n]) \in \textbf{SFun}$$

The function $\bar{}$ returns a flat list of all the stream ids of the given stream tree:

$$\begin{aligned}&\bar{}: \textbf{STree} \to [\textbf{SId}] \\ &\overline{s} = [s] \\ &\overline{(st_1, st_2)} = \overline{st_1} +\!\!+ \overline{st_2}\end{aligned}$$

Then a user-defined function **function** $f(x_1 : \tau_1, ..., x_k : \tau_k) : \tau = e$ will be translated to an SVCODE function $([s_1, ..., s_m], p, \overline{st})$, such that

$$[x_1 \mapsto st_1, ..., x_k \mapsto st_k] \ \vdash \ e \Rightarrow (p, st)$$

where argument trees $st_1, ..., st_k$ are generated from their corresponding types $\tau_1, ..., \tau_k$ (with all fresh ids), and

$$[s_1, ..., s_m] = \overline{st_1} \mathbin{+\!\!+} ... \mathbin{+\!\!+} \overline{st_k}$$

The generated SVCODE function will be added to a user-defined function mapping Ψ from function identifiers to **SFun**:

$$\Psi = [f_1 \mapsto sf_1, ..., f_n \mapsto sf_n]$$

And Ψ will be used as a component of the runtime environment, so when we interpret the instruction $[s'_1, ..., s'_n] :=$ `SCall` $f([s_1, ..., s_m])$, the function body can be unfolded by looking up f in Ψ and then passing the arguments.

2.5 Eager interpreter

Recall that an SVCODE program is a list of instructions each of which defines one or more streams. The eager interpreter executes the instructions sequentially, assuming the available memory is infinitely large, which is the critical difference between the execution models of the eager and streaming interpreters.

For an eager interpreter, since there is always enough space, a new defined stream can be entirely allocated in memory immediately after its definition instruction is executed. In this way, traversing the whole program only once will generate the final result, even for recursions. The streaming model of SVCODE does not show any of its strengths here; the interpreter will perform just like a NESL's low-level interpreter.

As we will add a limitation to the memory size in the streaming model, it is reasonable to consider the eager version as an extreme case with the largest buffer size of the streaming one. In this case, much work can be simplified or even removed, such as the scheduling since there is only one sequential execution round. Thus the correctness, as well as the time complexity, is the easiest to analyze, which can be used as a baseline to compare with the streaming version with different buffer sizes.

2.5.1 Dataflow

In the eage model, a Xducer consumes the entire input streams at once and output the entire stream immediately. The dataflow DAG is established gradually as Xducers are activated one by one.

2.5.2 Cost model

The low-level work cost in the eager model is the total number of consumed and produced elements of all Xducers, and the step is merely the number of activated Xducers. By activated we mean the executed Xducer definitions, because the stream definitions inside a `WithCtrl` block may be skipped, in which case we will not account for the steps for those definitions.

2.6 Streaming interpreter

As we have mentioned before, the execution model of streaming interpreter does not assume an infinite memory; instead, it only uses a limited size of memory as a buffer. If the buffer size is relatively small, then most of the streams cannot be materialized entirely at once. As a result, the SVCODE program will be traversed multiple times, or there will be more scheduling rounds. The dataflow of the streaming execution model is still a DAG, but the difference from the eager one is that each Xducer maintains a small buffer, whose data is updated each round. The final result will be collected from all these scheduling rounds.

Since in most cases we will have to execute more rounds, some extra setting-up and overhead seem to be inevitable. On the other hand, exploiting only a limited buffer increases the efficiency of space usage. In particular, for some streamable cases, such as an exclusive scan, the buffer size can be as small as one (and by one we do not mean one bit or byte of physical memory, but rather a conceptual, minimal size).

2.6.1 Processes

In the streaming execution model, the buffer of a Xducer can be written only by the Xducer itself, but can be read by many other Xducers. We define two states for a buffer: And it has two states:

- `Filling` state: the buffer is not full, and the Xducer is producing or writing data to it; any other trying to read it has to wait, or more precisely, enters a read-block state.
- `Draining` state: the buffer must be full; the readers, including the read-blocked ones, can read it only in this state; if the Xducer itself tries to write the buffer, then it enters a write-block state.

The condition of switching from `Filling` to `Draining` is simple: when the buffer is fully filled. But the other switching direction takes a bit more work to detect: all the readers have read all the data in the buffer. We will come to this later.

A notable special case is when the Xducer produces its last chunk, whose size may be less than the buffer size thus can never turn the buffer to a draining mode. To deal with this case, we add a flag to the draining state to indicate if it is the last chunk of the stream. Thus, the definition of a buffer state is as follows:

$$\textbf{BufState} = \{\texttt{Filling}\ \ \vec{a},\ \texttt{Draining}\ \ \vec{a}'\ b\}$$

In addition to maintaining the buffer state, a Xducer also has to remember its suppliers so that it is not necessary to specify the suppliers repeatedly each round. Actually, once a dataflow DAG is established, it is only possible to add more subgraphs to it due to an unfolding of a `WithCtrl` block or a `SCall` instruction; the other parts uninvolved will be unchanged until the execution is done.

Since Xducers have different data rates (the size of consumed/produced data at each round), it is also important to keep track of the position of the data that has been read, which can be represented by an integer. Also, it is possible that a Xducer reads from the same supplier multiple times but with different data rates, in which case only a pair of stream id and an integer is not enough to distinguish all the

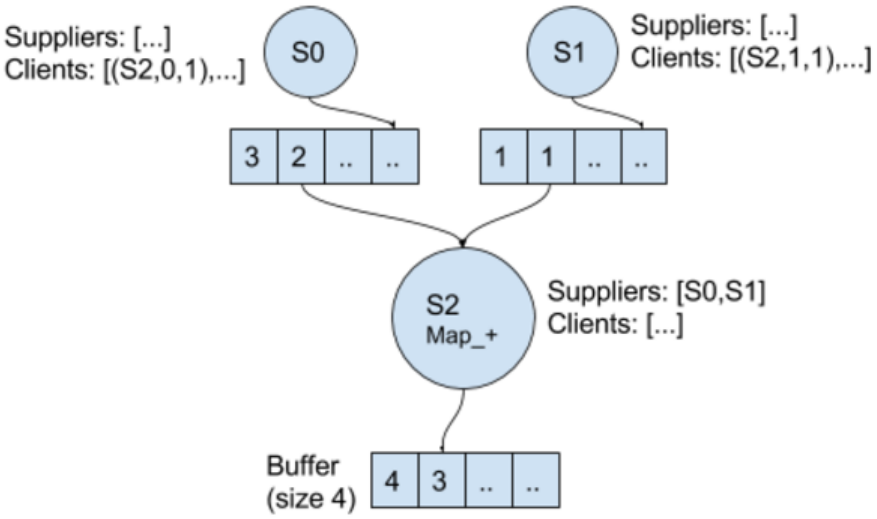


Figure 2.9: A process S2 of Xducer $\texttt{Map}_+$. It reads a 2 from S0's buffer and a 1 from S1's buffers, then writes a 3 to its own buffer.

different read cursors. Thus we need a third component, the channel number, to record the state of each reader, as we have shown in Figure 2.8. As a result, we must have a client list **Clis** of elements of type $(\mathbf{SId}, \mathbf{int}, \mathbf{int})$

Now we define a structure *process*, a tuple of four components including a Xducer, as the node on the streaming DAG:

$$\mathbf{Proc} = (\mathbf{BufState}, S, \mathbf{Clis}, \mathbf{Xducer})$$

where S is the stream ids of the suppliers. An example process of Xducer $\texttt{Map}_+$ can be found in Figure 2.9.

The Xducer inside a process is the action-performing unit. We classify the atomic actions of a Xducer into three:

- `Pin` : read one element from one supplier's buffer.
- `Pout`: write one element to its own buffer.
- `Done`: shutdown itself, no read or write any more.

In this way, a Xducer's actions can be considered as a sequential list of these three atomics. For example, the $\texttt{Map}_+$ Xducer's action will be repetitions of two `Pin` s (reads from two suppilers respectively) followed by one `Pout`, and a `Done` action can be added where we want to shut down the Xducer:

$$[\texttt{Pin}_{\ 0}, \texttt{Pin}_{\ 1}, \texttt{Pout}, \texttt{Pin}_{\ 0}, \texttt{Pin}_{\ 1}, \texttt{Pout}, ..., \texttt{Done}]$$

where the subscripts of `Pin` indicates reading different suppliers.

In our practical implementation, we use a strategy to make the Xducer self-shutdown: we add an extra input stream, the control stream, to each Xducer, and the Xducer will shut itself down when its read-cursor on the control stream reaches the end, i.e., it reads an EOS (end of stream) from the control stream.

A process is responsible for managing its buffer and Xducer. The activity of a process can be described as follows:

Buffer state / Xducer action	`Filling`	`Draining F`	`Draining T`
`Pin`	Process read	Process read	impossible
`Pout`	write one element to buffer; if buffer is full, switch to `Draining F`	enter write-block	impossible
`Done`	switch to `Draining T`	switch to `Draining T`	`skip`

Table 2.1: Process actions.

Process read:

- if the supplier's buffer state is `Draining` , and the read-cursor shows the process has not yet read all the data, then the process reads one element successfully
- if the supplier's buffer state is `Draining` , but the read-cursor shows the process has read all the data, or the supplier's buffer state is `Filling` , then the process enters a read-block state

full-check: if the buffer is full, switch it to `Draining F` state.

allread-check: if all the clients have read all the data of the buffer, switch it to `Filling` state.

2.6.2 Scheduling

The streaming execution model consists of two phases:

(1) Initialization
In this phase, the interpreter establishes the initial DAG by traversing the SV-CODE program. The cases are:

- initialize a sole stream definition $s := \psi(s_1, ..., s_k)$:
  This is to set up one process s:
  - set its suppilers $S = [s_1, ..., s_k]$
  - add itself to its suppliers' **Clis** with the corresponding channel number $\in \{1, ..., k\}$ and a read-cursor number 0
  - empty buffer of state `Filling`
  - set up the specific Xducer ψ
- initialize a function call $[s'_1, ..., s'_m] :=$ `SCall` $f([s_1, ..., s_n])$:
  A user-defined function at runtime can be considered as another DAG, whose nodes(processes) of the formal arguments are missing. So the interpreter just adds the function's DAG to the main program's DAG and replaces the function's formal arguments with the actual parameters $[s_1, ..., s_n]$, and the formal return ones with the actual ones $[s'_1, ..., s'_m]$.

- initialize a `WithCtrl` block $S_{out} :=$ `WithCtrl`(s_c, S_{in}, p)
  At the initialization phase, the interpreter does not unfold p; instead, it mainly does the following two tasks:
  - prevents all the import streams of S_{in} from producing one more chunk (a full buffer) of data before the interpreter knows whether s_c is an empty stream or not.
  - initializes all the import streams of S_{out} as dummy processes that do not produce any data

  Note that the practical approach for achieving these two goals can be various.

(2) Loop scheduling.
This phase is a looping procedure. The condition of its end is that all the Xducers have shutdown, and all the buffers are in `Draining T` state.

In a single scheduling round, the processes on the DAG are activated one by one from small to large. The active process acts as Table 2.9 shows, until it enters a read-block or write-block state, or it is skipped. The results collected from each round consist entire streams as the final result.

As long as a real (not dummy) process has been set up, it will keep working until its Xducer shutdown. The only crucial task in each round is to judge whether to unfold a `WithCtrl` block or not. The judgment depends on the buffer state of the new control stream. Note that the new control stream must be some real process defined earlier than the code inside a `WithCtrl` block.

- `Filling` $\langle\rangle$: the new control process has not produce any data yet, so the judgment cannot be made this round, thus delayed to the next round
- `Draining` $\langle\rangle$ `T` : the new control stream is empty, thus no need to unfold the code, just sets the export list streams also empty, and performs some other necessary clean-up job
- other cases: the new control stream must be nonempty, thus the interpreter can unfold the code now

2.6.3 Cost model

Since we have defined the atomic actions of Xducers, it is now easy to define the low-level cost:

Work = the total number of `Pin` and `Pout` of all processes

Step = the total number of switches from `Filling` to `Draining` of all processes

2.6.4 Recursion

In SVOCDE, a recursive function call happens when the function body of f from the instruction $[s'_1, ..., s'_n] :=$ `SCall` $f([s_1, ..., s_m])$ includes another `SCall` calling f as well.

As we have shown, for a non-recursive `SCall` , the effect of interpreting this instruction is almost transparent. For a recursive one, there is not much difference except one crucial point: the recursive `SCall` must be wrapped by a `WithCtrl` block,

otherwise it can never terminate; at each time of interpreting an inline `SCall` , the function body is unfolded, but the `WithCtrl` instruction inside it will stop its further unfolding, that is, the stack-frame number only grows by one.

At the high level, a well-defined SNESL program should use some conditional to decide when to terminate the recursion. As the only conditional of SNESL is the restricted comprehension, which is always translated to a `WithCtrl` block wrapping the expression body. Thus we can guarantee that a recursion that can terminate at the high level will also terminate at the low level.

Example 2.23.

```
-- buffer size 1

-- define a function to compute factorial
> function fact(x:int):int = if x <= 1 then 1 else x*fact(x-1)

-- running example
> {{fact(y): y in &x} : x in {5,10}}
{{1,1,2,6,24},{1,1,2,6,24,120,720,5040,40320,362880}} :: {{int}}
```

[?? Optional] The function body of `fact`:
The translated SVCODE of the expression:

```
> :c {{fact(y): y in &x} : x in {9,10}}

Parameters: []
S0 := Ctrl;
S1 := Const 9;
S2 := Const 10;
S3 := Const 1;
S4 := ToFlags 3;
S5 := ToFlags 3;
S6 := InterMergeS [4,5];
S7 := PriSegInterS [(1,4),(2,5)];
S8 := Usum 6;
WithCtrl S8 (import [7]):
        S9 := ToFlags 7
        S10 := Usum 9
        WithCtrl S10 (import []):
        S11 := Const 1
        Return: (IStr 11)
        S12 := SegscanPlus 11 9
        S13 := Usum 9
        WithCtrl S13 (import [12]):
        SCall fact [12] [14]
        Return: (IStr 14)
        Return: (SStr (IStr 14), 9)
Return: (SStr (SStr (IStr 14), 9), 6)
```

2.6.5 Deadlock

An inherent tough issue of the streaming execution model is the risk of deadlock, which is mainly due to the limitation of available memory and the irreversibility (maybe ?) of time. In general, we classify deadlock situations into two types: soft

deadlock, which can be detected and broken relatively easily but not necessarily by enlarging the buffer size, and hard deadlock, which can only be solved by enlarging the buffer size.

- Soft deadlock:

  One case of soft deadlock can be caused by trying to traverse the same sequence multiple times. A simpler example:

  Example 2.24. A soft deadlock caused by travesing the sequence x two times.

```
> let x = {1} in x ++ x
Deadlock!
```

  There are at least two feasible solutions to this case. One is manually rewriting the code to define new variables for the same sequence, as the following code shows :

```
> let x = {1}; y = {1} in x ++ y
{1,1} :: {int}
```

  The other can be done by optimizing the compiler to support multi-traversing check and automatic redefinition of retraversed sequences. As the code grows more complicated, locating the problem can become much harder, thus a smarter compiler is definitely necessary, which is worth some future investigation.

  Another case of soft deadlock can be due to the different data rates of processes, which leads to a situation where some buffer(s) of `Filling` state can never turn to `Draining` . For example, the following expression tries to negate the elements that can be divided by 5 exactly of a sequence.

  Example 2.25. A soft deadlock that can be broken by stealing

```
  -- buffer size 4
> concat({{-x | x % 5 == 0} ++ {x | x %5 != 0} :  x in &10})
{0,1,2,3,4,-5,6,7,8,9} :: {int}
```

  In this example, the sequence contains elements from 0 to 9; the subsequence of the negated numbers, which only contains 0 and -5, are concatenated with the one of the other eight numbers. Since these two subsequences are generated at different rates, If we minimize the buffer size to 1, then the deadlock can be broken since buffer of size one can always turn to `Draining` mode as long as there is one element generated. In our implementation, we use an automatic solution for this case, called *stealing*. The idea is that when a deadlock is detected, we will first switch the smallest process with a `Filling` buffer, into `Draining` mode, to see if the deadlock can be broken; if not, we repeat this switch until the deadlock is broken; otherwise, it may be a hard deadlock.

  Since the stealing strategy is basically a premature switch from `Filling` to `Draining` , the low-level step cost is possible to be affected. More precisely,

it can be increased by a certain amount, which depends on the concrete program and the buffer size. The effect of stealing on the cost model can also be investigated as future work.

- Hard deadlock:

  This type of deadlock is mainly because of insufficient space.

  Example 2.26. The following SNESL function `oeadd` risks a hard deadlock. Given an integer sequence with an equal number of odd and even numbers, this function will try to perform addition on a pair of an odd and an even number with the same index from their respective subsequences.

```
1  -- v must have equal number of odd and even numbers
2  function oeadd(v:{int}):{int} =
3    let odds = concat({{x | x %2 !=0}: x in v});
4        evens = concat({{x | x %2 == 0} : x in v})
5     in {o+e : o in odds, e in evens}
```

  If we give a proper argument, for example, an sequence of odds and evens interleaving with each other(??? not clear), it may never deadlock, even with a buffer of size one. But if there is a relatively large distance between any odd-even pair, the code will deadlock.

```
1  -- buffer size 1
2
3  > oeadd(&30)
4  {1,5,9,13,17,21,25,29,33,37,41,45,49,53,57} :: {int}
5
6  > oeadd({1,3,5,7,0,2,4,8})
7  Deadlock!
```

  This type of deadlock can only be broken by enlarging the buffer size.

2.6.6 [optional] Evaluation

Some possibilies for improving the scheduling:

- The processes in a block state will be activated in the next scheduling round, but it may still block itself immediately since the blocking condition still holds. So a better strategy can be
- The processes are activated only one by one, even though some of them can be data-independent, this simulates a SIMD machine execution. But it should be able to be optmized to support MIMD machine. Evaluation of some high-level expressions, such as the evaluation of the components of a tuple or a function call , can be executed in parallel as well.

2.6.7 [optional] Examples

```
-- united-and-conquer scan and reduce (only for n = power of 2)
function scanred(v:{int}, n:int) : ({int},int) =
    if n==1 then ({0}, the(v))
    else
      let is = scanExPlus({1 : x in v});
          odds = {x: i in is, x in v | i%2 !=0};
          evens ={x: i in is, x in v | i%2 ==0};
          ps = {x+y : x in evens, y in odds};
          (ss,r) = scanred(ps,n/2)
      in (concat({{s,s+x} : s in ss, x in evens}), r)
```

Bibliography

[Ble89] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[Ble95] Guy E Blelloch. Nesl: A nested data-parallel language.(version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, 1995.

[Ble96] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[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.

[PP93] Jan F Prins and Daniel W Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'93, pages 119–128. ACM, 1993.

[PPCF95] Daniel W Palmer, Jan F Prins, Siddhartha Chatterjee, and Rickard E Faith. Piecewise execution of nested data-parallel programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 346–361. Springer, 1995.