UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis

Dandan Xue

Formalizing the implementation of Streaming NESL

Supervisor: Andrzej Filinski

Submitted on: 24 October 2017

Revised version: 8 November 2017

Abstract

Streaming NESL (SNESL) is an experimental, 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. Practical experiments have demonstrated good performance of SNESL's implementation and provided empirical evidence of the validity of the cost model.

In this thesis, we first extend SNESL to support recursive functions. This requires non-trivial extensions to SNESL's target language, SVCODE, and the flow graph of its streaming dataflow model to be dynamically completed at runtime. Two execution models of the extended SVCODE, the NESL-like eager one, and the streaming one, are given.

Then we show the formalization of the semantics of a subset of the source and target languages, followed by a proof of the translation correctness and work cost preservation.

Contents

Preface

TBD

Chapter 1

Introduction

1.1 Background

Parallel computing has drawn increasing attention in the field of high-performance computing. Today, it is widely accepted that Moore's law will break down due to physical constraints, and future performance increase must heavily rely on increasing the number of cores, rather than making a single core run faster.

Typically, a parallel computation can be completed by splitting it into multiple subcomputations executing independently. The theoretical maximum number of these subcomputations is called the parallel degree. In practice, it is common that this theoretical parallel degree cannot be fully exploited because of the limitation of physical resources.

Parallelism can be expressed or employed in different aspects, such as algorithms, programming languages or hardware, which all have made tremendous progress in recent decades.

1.2 Nested data parallelism

Data parallelism deals with parallelism typically by applying parallel operations on data collections. The observation is that the loop structure occurs quite frequently in algorithms and usually accounts for a large proportion of the running time, which has a high potential for each iteration being executed in parallel.

Nested data parallelism allows data-parallelism to be nested. In nested data-parallel languages, function calls can be applied to a set of data that can be not only flat or one-dimension arrays, but also multi-dimension, irregular or recursive structures. So these languages can implement parallel algorithms, including nested loops and recursive ones, more expressively and closer to the programmer's intuition.

On the other hand, at such a high level, compilation becomes more complicated to achieve a performance close to the code written directly in low-level ones. Also, predicting the performance are more difficult because the details of the concrete parallel execution are usually implicit or hidden to the programmer.

Some languages, such as NESL [Ble95, BHC+93, BG96] Proteus [PP93, PPW95] and Data Parallel Haskell [CLPJ+07, PJ08, LCK+12], have pioneered NDP significantly and demonstrated its advantages and importance.

1.3 NESL

NESL 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, shown below, looks like a mathematical set comprehension (or a list comprehension in Haskell):

$$\{f(x) : x \textbf{ in } e\}$$

As its name implies, it applies the function f to each element of e, and the computation will be executed in parallel. 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 returns [2,3,4].

The strength of this construct is that it supports nested function application on irregular data sets. Continuing with the example above, the adding-1 operation can also be performed on each subsequence of a nested sequence [[1,2,3],[4],[5,6]], written as:

$$\{\{y + 1 : y \textbf{ in } x\} : x \textbf{ in } [[1, 2, 3], [4], [5, 6]]\}$$

giving [[2,3,4],[5],[6,7]].

The low-level language of NESL's implementation model is VCODE [BHC+93], which uses vectors (i.e., flat arrays of atomic values such as integers or booleans) as the primitive type, and its instruction set performs operations on vectors as a whole, such as summing. The technique *flattening nested parallelism* [BS90] used in the implementation model translates nested function calls in NESL to instructions on vectors in VCODE.

From the user's perspective, 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.

```
-- 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 | c == ' '}; -- 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 | c != ' ' }  -- non-space chars
   in partition(chars, valid_ls);   -- split strings into words
```

```
-- a running 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

Another important idea of NESL is its intuitive, language-based, high-level cost model [Ble96]. This cost model can predict the performance of a NESL program from two measures: *work*, the total number of operations executed in this program, and *depth*, or *step*, the longest chain of sequential dependency in this program. From another point of view, the work cost can be viewed as a measurement of the time complexity of the computation executed on a machine with only one processor, and the step cost corresponds to the time complexity on an ideal machine with an unlimited number of processors.

With this work-depth model, as demonstrated in [Ble90], the user can derive the asymptotic time complexity T of a NESL program executed on a machine with P processors as

$$T = O(work/P + depth)$$

However, a problem NESL suffers from is its inefficient space-usage. This is mainly due to the eager semantics that NESL uses for supporting random access in parallel execution. That is, during the execution of a NESL program, sufficient space must be provided for storing the entire evaluation values including the intermediate ones. For example, computing the multiplication of two matrices of size n-by-n will need at least n^3 space to store the intermediate element-wise multiplication result, which is a huge waste of space when n is large.

1.4 SNESL

Streaming NESL (SNESL) [MF13] is a refinement of NESL that attempts to improve the efficiency of space usage. It extends NESL with two features: a streaming semantics and a corresponding 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 of *piecewise execution* in [PPCF96], but SNESL makes the chunkability exposed at the source level in the type system and the cost model instead of a low-level execution optimization, and the chunk size should be proportional to the number of processors ($10 \sim 100$ typically) and fit in cache.

1.4.1 Types

The types of a minimalistic version of SNESL defined in [MF13] are as follows (using Haskell-style notation):

$$\begin{aligned}\pi &::= \textbf{bool} \mid \textbf{int} \mid \textbf{char} \mid \textbf{real} \mid \cdots \\ \tau &::= \pi \mid (\tau_1, ..., \tau_k) \mid [\tau] \\ \sigma &::= \tau \mid (\sigma_1, ..., \sigma_k) \mid \{\sigma\}\end{aligned}$$

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 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, which means the elements of a sequence are produced and consumed over time. So, even with a small size of memory, SNESL could execute programs which are impossible in NESL due to space limitation or more space efficiently than in NESL.

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

1.4.2 Values and expressions

The values of SNESL are as follows:

$$\begin{aligned}a &::= \texttt{T} \mid \texttt{F} \mid n \ (n \in \mathbb{Z}) \mid \cdots \\ v &::= a \mid (v_1, ..., v_k) \mid [v_1, ..., v_l] \mid \{v_1, ..., v_l\}\end{aligned}$$

where a is the atomic values or constants of types π, and v are general values which can be a constant, a tuple of k components, a vector or a sequence of l elements. Here, as part of our notation, we use k to range over "small" natural numbers (related to program size), while l are potentially "large" ones (related to data size).

The expressions of SNESL are shown in the following figure.

$$
\begin{array}{lr}
e ::= a & \text{(constant)} \\
\quad \mid x & \text{(variable)} \\
\quad \mid (e_1, ..., e_k) & \text{(tuple)} \\
\quad \mid \textbf{let } p = e_1 \textbf{ in } e_2 & \text{(let-binding)} \\
\quad \mid \phi(e_1, ..., e_k) & \text{(built-in function call)} \\
\quad \mid \{e_1 : \ p \textbf{ in } e_0\} & \text{(general comprehension)} \\
\quad \mid \{e_1 \mid e_0\} & \text{(restricted comprehension)} \\
 & \\
p ::= x \mid (p_1, ..., p_k) & \text{(pattern matching)}
\end{array}
$$

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 they are 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 expressing parallel computation, and the restricted one can decide if a computation is necessary or not, 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 (except for those bound by p) in the general comprehension can only be of concrete types, while they can be of any types in the restricted one.

Note that SNESL, as described in [Mad16], does not include programmer-defined functions. In the implementation, functions can be defined, but effectively treated as macros during compilation. In particular, they cannot be recursive.

1.4.3 Primitive functions

SNESL also refines the primitive functions of NESL to separate sequences and vectors. The primitive functions of SNESL are shown in Figure 1.3.

$$
\begin{array}{lll}
\phi & ::= \oplus \mid \textbf{append} \mid \textbf{concat} \mid \textbf{zip} \mid \textbf{iota} \mid \textbf{part} \mid \textbf{scan}_{\otimes} \mid \textbf{reduce}_{\otimes} \mid \textbf{mkseq} & (1.1) \\
 & \mid \textbf{length} \mid \textbf{elt} & (1.2) \\
 & \mid \textbf{the} \mid \textbf{empty} & (1.3) \\
 & \mid \textbf{seq} \mid \textbf{tab} & (1.4) \\
\oplus & ::= + \mid \ - \ \mid \ \times \ \mid / \mid \% \mid == \ \mid \ <= \ \mid \textbf{not} \mid \cdots & \text{(scalar operations)} \\
\otimes & ::= + \mid \ \times \ \mid \textbf{max} \mid \cdots & \text{(associative binary operations)}
\end{array}
$$

Figure 1.3: SNESL primitive functions

The scalar functions of $\oplus$ and $\otimes$ should be self-explanatory from their conventional symbols or names. The types of the other functions and their brief descriptions

are given in Table 1.1.

The functions listed in (1.1) and (1.2) of Figure 1.3 are originally 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 as parameters instead of vectors as they do in NESL, as we can see from Table 1.1, thus these functions can execute in a more space-efficient way.

Functions in (1.2), i.e., **length** and **elt**, are kept as their vector versions in SNESL. These two exploit vectors that are fully materialized, thus have constant time cost. On the other hand, while analogous functions can be defined for sequences (using other primitives), they have cost proportional to the length of the sequence.

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

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

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 elements. Therefore, it works in a fairly efficient way with a constant complexity both in time and space.

Finally, functions listed in (1.4) connects the concrete types and streams, making it possible to turn every NESL program into a SNESL one by adding suitable **seq**/**tab** calls.

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 even with a chunk size of one element.

```
1 -- partition a string to words (delimited by spaces)
2 -- SNESL version
3 function str2wds_snesl(str) =
4    let flags = { x == ' ' : x in str};
5        nonsps =  concat({{x | x != ' '} : 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

1.4.4 Cost model

Based on the work-depth model, SNESL develops another two components for estimating the space complexity [MF13]. The first one is the sequential space S_1, that is, the minimal space to perform the computation, corresponding to run the program with a buffer of size one. The other is the parallel space S_∞, the space that needed to achieve the maximal parallel degree, and it corresponds to assuming the program executes with an unlimited memory as NESL does. In [Mad16], the first component is refined further to allow vectors to be shared across parallel computations.

With this extended cost model, the user can now estimate the time complexity of a SNESL program using the same concepts as for NESL, and the space complexity

Function type	Brief description
append : $(\{\sigma\}, \{\sigma\}) \rightarrow \{\sigma\}$	append two sequences; syntactic sugar: infix symbol "++"
concat : $\{\{\sigma\}\} \rightarrow \{\sigma\}$	flatten a sequence of sequences
zip: $(\{\sigma_1\}, ..., \{\sigma_k\}) \rightarrow \{(\sigma_1, ..., \sigma_k)\}$	convert k sequences into a sequence of k-component tuple
iota : **int** $\rightarrow$ {**int**}	generate an integer sequence starting from 0 to the given argument minus one; syntactic sugar: prefix symbol "&"
part : $(\{\sigma\}, \{\textbf{bool}\}) \rightarrow \{\{\sigma\}\}$	partition a sequence into subsequences segmented by Ts in the second argument; e.g., **part**$(\{3,1,4\}, \{\texttt{F},\texttt{F},\texttt{T},\texttt{F},\texttt{T},\texttt{T}\}) = \{\{3,1\},\{4\},\{\}\}$
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: prefix symbol "#"
elt: $([\tau], \textbf{int}) \rightarrow \tau$	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 : $[\tau] \rightarrow \{\tau\}$	stream a vector as a sequence
tab : $\{\tau\} \rightarrow [\tau]$	tabulate a sequence into a vector

Table 1.1: SNESL primitive functions

with the following formula

$$S = O(min(P \cdot S_1, S_\infty))$$

where P is the number of processors.

Chapter 2

Implementation

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

2.1 High-level interpreter

In this thesis, the high-level language we have experimented with is close to the SNESL introduced in the last chapter but without vectors, and 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 so because removing vectors 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 work with vectors as well.

Besides, only two primitive types of SNESL, **int** and **bool**, are retained in SNESL_1. Tuples are also simplified to pairs.

The types of SNESL_1 are as follows.

$$\begin{aligned}\pi &::= \textbf{bool} \mid \textbf{int} \\ \tau &::= \pi \mid (\tau_1, \tau_2) \mid \{\tau\} \\ \varphi &::= (\tau_1, ..., \tau_k) \to \tau\end{aligned}$$

In SNESL_1, concrete types are either primitive types, or binary trees (i.e., nested pairs) of primitive types. We give its definition as follows:

Judgment $\boxed{\tau \textbf{ concrete}}$

$$\frac{}{\pi \textbf{ concrete}} \qquad \frac{\tau_1 \textbf{ concrete} \qquad \tau_2 \textbf{ concrete}}{(\tau_1, \tau_2) \textbf{ concrete}}$$

Figure 2.1: Concrete types

The values in SNESL$_1$ are:

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

The abstract syntax of SNESL$_1$ is given in Figure 2.2.

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

$e ::= a$ (constant)
$\mid x$ (variable)
$\mid (e_1, e_2)$ (pair)
$\mid \{e_1, ..., e_k\}$ $(k \geq 1)$ (primitive sequence)
$\mid \{\}\tau$ (empty sequence of type τ)
$\mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$ (let-binding)
$\mid \phi(e_1, ..., e_k)$ (built-in function call)
$\mid \{e_1 : x\ \textbf{in}\ e_0\ \textbf{using}\ x_1, ..., x_k\}$ (general comprehension)
$\mid \{e_1 \mid e_0\ \textbf{using}\ x_1, ..., x_k\}$ (restricted comprehension)
$\mid f(e_1, ..., e_k)$ (user-defined function call)

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

Figure 2.2: Abstract syntax of SNESL$_1$

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

- For comprehension expressions, the free variables (except x) of the comprehension body e_1 are collected as a list added after the keyword **using**. This task can simply be done by the front end of 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**.
- 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 for the expressions of SNESL$_1$ are given in Figure 2.3. The type environment Γ is a mapping from variables to types:

$$\Gamma ::= [x_1 \mapsto \tau_1, ..., x_k \mapsto \tau_k]$$

and Σ from the identifiers of user-defined functions to their types:

$$\Sigma ::= [f_1 \mapsto \varphi_1, ..., f_k \mapsto \varphi_k]$$

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

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

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

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

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

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

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

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

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

Figure 2.3: Typing rules of SNESL$_1$ expressions

The typing rules for built-in operations (using the judgment $\boxed{\phi : (\tau_1, ..., \tau_k) \to \tau}$) are given in their detailed descriptions later.

The typing rules of SNESL$_1$ should be straightforward except those for comprehensions. For the general comprehension, the *input* sequence e_0, as well as the whole expression, must be a sequence. The types of the free variables of the *comprehension body* e_1, i.e., $x_1, ..., x_k$, are all required to be concrete. That is, they cannot be sequences or contain any sequence component. However, they can be of any types in restricted comprehension. We will explain this later.

Figure 2.4 shows the typing rules for SNESL$_1$ values, and Figure 2.5 gives the rules for checking whether a top-level term is well-typed.

Judgment $\boxed{v : \tau}$

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

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

Figure 2.4: Typing rules of SNESL_1 values

Judgment $\boxed{\vdash_\Sigma t : \tau}$

$$\frac{[\,] \vdash_\Sigma \; e : \tau}{\vdash_\Sigma \textbf{eval}\ e : \tau}$$

$$\frac{[x \mapsto \tau_1, ..., x_k \mapsto \tau_k] \vdash_{\Sigma[f \mapsto (\tau_1, ..., \tau_k) \to \tau_0]} \; e : \tau_0 \qquad \vdash_{\Sigma[f \mapsto (\tau_1, ..., \tau_k) \to \tau_0]} t : \tau}{\vdash_\Sigma \textbf{function}\ f(x_1 \colon \tau_1, ..., x_k \colon \tau_k) \colon \tau_0 = e\ t : \tau}$$

Figure 2.5: Well-typed top-level terms

The high-level semantics of SNESL_1 is given in Figure 2.6. The evaluation environment ρ is in the form:

$$\rho ::= [x_1 \mapsto v_1, ..., x_k \mapsto v_k]$$

and the context Φ for looking up user-defined functions:

$$\Phi ::= [f \mapsto d_1, ..., f_k \mapsto d_k]$$

The evaluation rules are also straightforward except for the comprehensions. In the general comprehension, the input sequence e_0 gets evaluated first, generating a sequence of l elements. These l elements are used to substitute the bound variable x in the comprehension body e_1; so e_1 will be evaluated l times with different substitutions of x but with the same value of its free variables $x_1, ..., x_k$. As mentioned before, these free variables $x_1, ..., x_k$ can only be of concrete types, because their values are repeatedly used l times here, which requires these values must be already materialized before the evaluation. If any of them is a stream, then the entire stream must be allocated for each of the l evaluations of e_1.

For the restricted comprehension, the *guard* expression e_0 first gets evaluated: if it is a `T`, e_1 will be evaluated, generating a singleton sequence; otherwise, e_1 will not be evaluated, and the comprehension only returns an empty sequence. Here, since e_1 will be evaluated at most once, there will be no problem if any free variable of e_1 is a stream.

The built-in functions of SNESL_1 correspond to a subset of SNESL's ones shown in Figure 1.3 but without **mkseq**, and the vector-related ones. In addition, the function **scan** and **reduce** are also taken a specific verison: restricted $\otimes$ to $+$, for simplicity, as shown in Figure 2.7.

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

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

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

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

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

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

$$\frac{\rho \vdash_\Phi e_0 \downarrow \mathtt{F}}{\rho \vdash_\Phi \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_k\} \downarrow \{\}}$$

$$\frac{\rho \vdash_\Phi e_0 \downarrow \mathtt{T} \qquad [x_1 \mapsto \rho(x_1), ..., x_k \mapsto \rho(x_k)] \vdash_\Phi e_1 \downarrow v_1}{\rho \vdash_\Phi \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_k\} \downarrow \{v_1\}}$$

$$\frac{(\rho \vdash_\Phi e_i \downarrow v_i)_{i=1}^{k} \qquad [x_1 \mapsto v_1, ..., x_k \mapsto v_k] \vdash_\Phi e_0 \downarrow v}{\rho \vdash_\Phi f(e_1, ..., e_k) \downarrow v}\,\left(\begin{array}{l}\Phi(f) = \textbf{function} \\ \qquad f(x_1 \colon \tau_1, ..., x_k \colon \tau_k) \colon \tau = e_0\end{array}\right)$$

Figure 2.6: Semantics of SNESL_1

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

Figure 2.7: 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.

- $\mathbf{append}_\tau : (\{\tau\}, \{\tau\}) \to \{\tau\}$, 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}}
```

- $\mathbf{concat}_\tau : \{\{\tau\}\} \to \{\tau\}$, 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}}
```

- $\mathbf{iota} : \mathbf{int} \to \{\mathbf{int}\}$, 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}
```

- $\mathbf{part}_\tau : (\{\tau\}, \{\mathbf{bool}\}) \to \{\{\tau\}\}$, 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 this argument must end with a `T`. If the second argument does not satisfy these requirements, 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({{3,1},{4},{}int, {1,5}}, {F,F,T,F,F,T})
{{{3,1},{4}},{{},{1,5}}} :: {{{int}}}
```

- **scan**$_+$: **{int}** $\rightarrow$ **{int}**, performs an exclusive scan of addition operation on the given sequence, that is, assuming the argument is $\{n_1, n_2, .., n_k\}$, 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 specific identity element a_0 such that $a_0 \otimes a = a \otimes a_0 = a$ for all a.

Example 2.5.

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

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

- **reduce**$_+$: **{int}** $\rightarrow$ **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**$_\tau$: $\{\tau\} \rightarrow \tau$, returns the element of a singleton sequence; if the length of the argument is not exactly one, reports a runtime error.

Example 2.7.

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

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

- **empty**$_\tau : \{\tau\} \to$ **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 (i.e., a collection of primitive values), or a binary tree structure with stream leaves. The idea of this representation comes from [Ble90]. The primitive stream $\vec{a}$ (of l elements $a_1, ..., a_l$) and the stream tree w have the following forms:

$$\vec{a} ::= \langle a_1, ..., a_l \rangle$$

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

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

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

  Example 2.9.

$$3 \triangleright_{\textbf{int}} \langle 3 \rangle$$

$$\mathtt{T} \triangleright_{\textbf{bool}} \langle \mathtt{T} \rangle$$

- 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`.

  Example 2.10.

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

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

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

- 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.

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

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

- A pair of high-level values is a pair of 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 pairs can be regarded as a pair of sequences sharing one 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 [MF16]; the various experiment results have demonstrated single-core performance similar to sequential C code for some simple text-processing tasks and near-linear scaling on moderate multicores.

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 preserve the cost, non-trivial extension of this language is needed.

2.3.1 SVCODE Syntax

The abstract syntax of SVCODE is given in Figure 2.8. An SVCODE program p is basically a list of commands or instructions each of which defines one or more streams. We use s to range over stream variables, also called stream *id*s, and S a set 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 of the instruction.

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 section.

The only essential control struture in this language 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 set 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. We will call this special stream the *new control stream*, and we will give the explanation of the concept *control stream* later, but here we only care about if s is empty:

$$
\begin{array}{lll}
p & ::= \epsilon & \text{(empty program)} \\
& \mid s := \psi(s_1, ..., s_k) & \text{(single stream definition)} \\
& \mid S_{out} := \texttt{WithCtrl}(s, S_{in}, p_1) & (\texttt{WithCtrl} \text{ block}) \\
& \mid (s'_1, ..., s'_{k'}) := \texttt{SCall } f(s_1, ..., s_k) & \text{(SVCODE function call)} \\
& \mid p_1; p_2 &
\end{array}
$$

$$
\begin{array}{lll}
s & ::= 0 \mid 1 \mid \cdots \in \mathbf{SId} = \mathbb{N} & \text{(stream ids)} \\
S & ::= \{s_1, ..., s_k\} \in \mathbb{S} & \text{(set of stream ids)}
\end{array}
$$

$$
\begin{array}{lll}
\psi & ::= \texttt{Const}_a \mid \texttt{ToFlags} \mid \texttt{Usum} \mid \texttt{Map}_{\oplus} \mid \texttt{Scan}_{+} \mid \texttt{Reduce}_{+} \mid \texttt{Distr} & \text{(Xducers)} \\
& \quad \mid \texttt{Pack} \mid \texttt{UPack} \mid \texttt{B2u} \mid \texttt{SegConcat} \mid \texttt{USegCount} \mid \texttt{InterMerge} \mid \cdots & \\
\oplus & ::= + \mid - \mid \times \mid / \mid \% \mid <= \mid == \mid \textbf{not} \mid \cdots & \text{(scalar operations)}
\end{array}
$$

Figure 2.8: Abstract syntax of SVCODE

- If s is non-empty, then execute p_1 and generate the streams of S_{out} as usual
- Otherwise, skip p_1 and assign S_{out} all empty streams

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 a variable set including all the streams that are referred to by p_1; it will only affect the streaming execution model of SVCODE, while in the eager model it can be ignored.

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

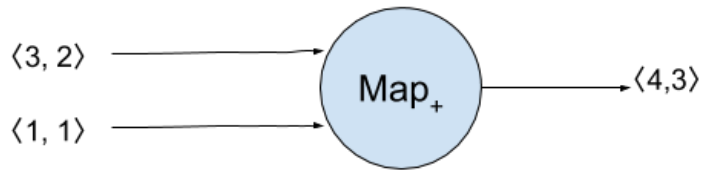
It is worth noting that a well-formed SVCODE instruction should always assign *fresh* (never previously used) stream ids to the defined streams, in which way the dataflow of an SVCODE program can construct a DAG (directed acyclic graph). We will give more formal definitions of this language in the next 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. $\texttt{Map}_{+}(\langle 3, 2\rangle, \langle 1, 1\rangle)$:



Among all the Xducers, $\texttt{Const}_a()$ is a special one, because it outputs some number of constant a but takes no argument. At the high level, it corresponds to the evaluation of constants. Some strategy must be taken here to tell this Xducer how many elements it should produce.

In the implementation of [MF16], a special stream of unit type, called *control stream*, is used at compiling time for replicating constants.

In our implementation, we move the control stream to the runtime, and let it not only control constants replication, but more importantly, dominate all the Xducers' behavior. Our observation is that the parallel degree throughout the whole computation can be expressed by this control stream, and all the Xducers can behave correspondingly to the parallel degree carried by the control stream. So we make the Xducer read the control stream firstly before consuming its normal inputs, so that it will know how many elements it should read and output.

There are 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 $\texttt{Const}_a$ 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 the Xducer 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.

As we have mentioned before, the dataflow of an SVCODE program is 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 2.9 shows an example program, with its DAG in Figure 2.10.

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

Figure 2.9: A small SVCODE program

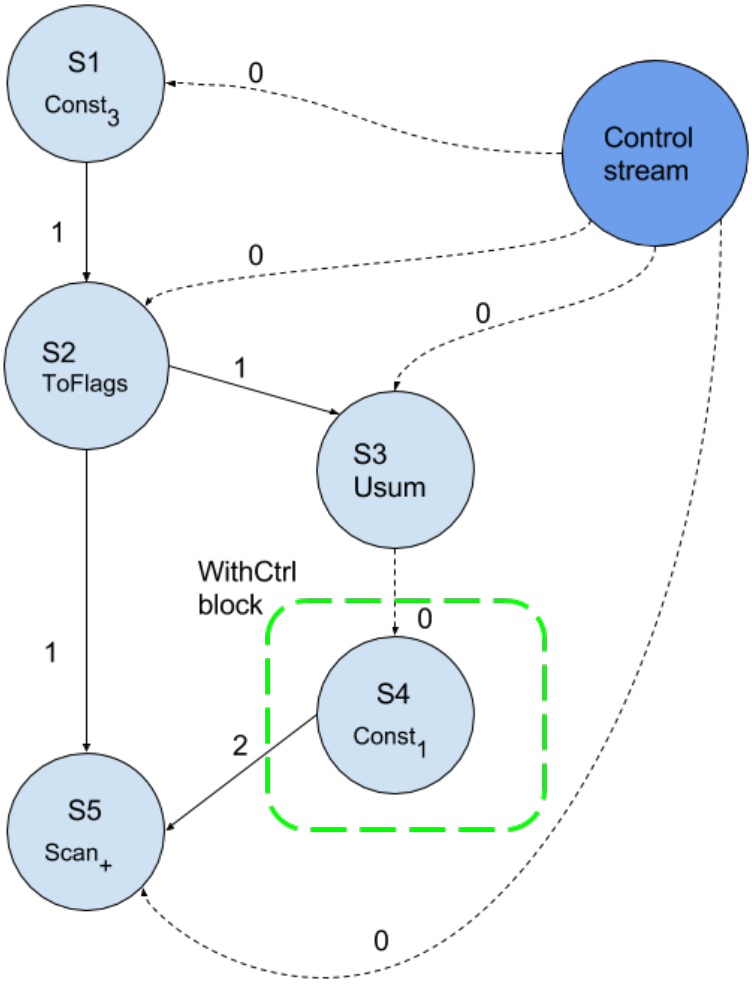


Figure 2.10: Dataflow DAG for the code in Figure 2.9 (assuming S3 is nonempty).

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 2.10, 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

(ignoring the control stream): S2 the No.1 channel and S4 the No.2.

2.4 Translating SNESL$_1$ to SVCODE

In Section 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 connect the high-level variables and the low-level ones:

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

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

$$\delta ::= [x_1 \mapsto st_1, ..., x_k \mapsto st_k]$$

Another important component maintained at the compiling time is a fresh-stream allocation counter. It will be assigned to the defined stream(s) of the generated instruction by the translation.

We will use the symbol "⇒" to denote the translation relation. To avoid clutter, in this section we assume the defined stream id(s) of the new instructions are all fresh.

2.4.1 Expression translation

A SNESL$_1$ expression will be translated to a pair of an SVCODE program p and a stream tree st whose stream values represent the high-level evaluation result:

$$\delta \ \vdash \ e \Rightarrow (p, st)$$

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 :

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

For a let-binding expression **let** $x = e_1$ **in** e_2, first e_1 gets translated to some code p_1 with 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 some 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 in the following two subsections.

Non-empty primitive sequence looks a little bit tricky to translate as it can have arbitrary number n ($n \geq 1$) of elements, but it is basically a n-argument version of the function **append**, i.e., $\{e_1, ..., e_k\}$ is compiled as **append**$(\{e_1\}, ..., \{e_k\})$. More details about translating **append** is given later. 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.

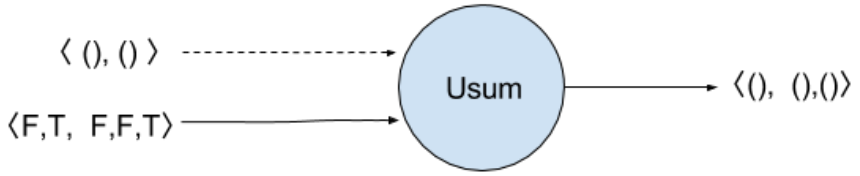
The most interesting case may be the comprehensions. For the general one, $\{e_1 : x$ **in** e_0 **using** $x_1, ..., x_k\}$, the key step is to bind the variable x to the entire

data stream of the input sequence e_0, then translate the body e_1 with this variable-binding, and put the generated code into a `WithCtrl` block. It is also important here that the parallel degrees must be changed (to the length of the input sequence). The values of the free variables $x_1, ..., x_k$ will be used many times (depending on the length of the input sequence as well), so they need to be replicated.

For example, the first step to translate the comprehension in the expression **let** $x = 1$ **in** $\{i + x : i$ **in** $\&5$ **using** $x\}$ is to translate $\&5$, which is represented as a data stream $\langle 0, 1, 2, 3, 4\rangle$ with a flag steam $\langle \mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{T}\rangle$. So i will be bound to the data stream, and the flag is used to generate the new control stream, using the Xducer `Usum`. The free variable x with value $\langle 1\rangle$ will be replicated 5 times, generating a new stream $\langle 1, 1, 1, 1, 1\rangle$, and this replication work is done by the Xducer `Distr`. Then the adding operation in the comprehension body is translated to an instruction performing vector addition on the stream bound to i and the replication stream of x, and executing the instruction will give us $\langle 1, 2, 3, 4, 5\rangle$.

The Xducer `Usum` working on a flag stream s_b is employed to generate the new control stream. It transforms an `F` to a unit, a `T` to nothing.

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



The translation for a general comprehension will look like:

$$\frac{\delta \vdash e_0 \Rightarrow (p_0, (st_0, s_b)) \qquad [x \mapsto st_0, (x_i \mapsto st'_i)_{i=1}^k] \vdash e_1 \Rightarrow (p_1, st)}{\delta \vdash \{e_1 : x \textbf{ in } e_0 \textbf{ using } x_1, ..., x_k\} \Rightarrow (p, (st, s_b))} ((\delta(x_i) = st_i)_{i=1}^k)$$

in which

$$\begin{aligned} p = (\ & p_0; \\ & s_1 := \mathtt{Usum}(s_b); \\ & st'_1 := \mathtt{distr}_{\tau_1}(s_b, st_1); \\ & \quad \vdots \\ & st'_k := \mathtt{distr}_{\tau_k}(s_b, st_k); \\ & S_{out} := \mathtt{WithCtrl}(s_1, S_{in}, p_1)) \end{aligned}$$

We put p_1 into a `WithCtrl` block so that p_1 can be skipped when s_1, the new control stream, is tested to be an empty stream. S_{in} and S_{out} are some analysis results about the free stream variables and the defined ones of p_1, which can be easily obtained by traversing p_1. We will give more details about them in the next chapter.

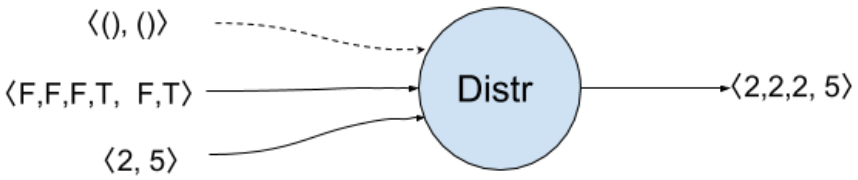
The function $\mathtt{distr}_\tau$ is responsible for replicating streams by using the Xducer `Distr`. We give its definition in a form close to the SVCODE style for more read-

ability:

$$\begin{array}{rcl} s' := \texttt{distr}_{\pi}(s_b, s); & \equiv & s' := \texttt{Distr}(s_b, s); \\ (st'_1, st'_2) := \texttt{distr}_{(\tau_1,\tau_2)}(s_b, (st_1, st_2)); & \equiv & st'_1 := \texttt{distr}_{\tau_1}(s_b, st_1); \\ & & st'_2 := \texttt{distr}_{\tau_2}(s_b, st_2); \end{array}$$

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:

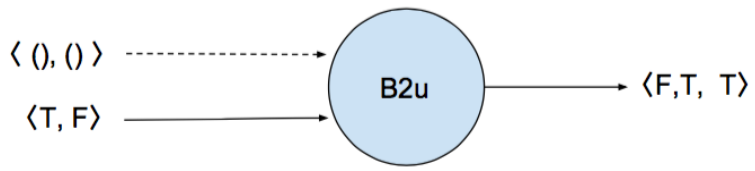
$$\frac{\delta \vdash e_0 \Rightarrow (p_1, s_b) \qquad [(x_i \mapsto st'_i)_{i=1}^{k}] \vdash e_1 \Rightarrow (p_2, st)}{\delta \vdash \{e_1 \mid e_0 \textbf{ using } x_1, ..., x_k\} \Rightarrow (p, (st, s_1))} ((\delta(x_i) = st_i)_{i=1}^{k})$$

in which

$$\begin{array}{rl} p = p_1; & \\ & s_1 := \texttt{B2u}(s_b); \\ & s_2 := \texttt{Usum}(s_1); \\ & st'_1 := \texttt{pack}_{\tau_1}(s_b, st_1); \\ & \quad \vdots \\ & st'_k := \texttt{pack}_{\tau_k}(s_b, st_k); \\ & S_{out} := \texttt{WithCtrl}(s_2, S_{in}, p_2) \end{array}$$

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 high-level type that the packed stream tree represents will generate instructions of `Pack` and possibly `UPack` and `Distr`. The function $\overline{\overline{st}}$ used in the instruction `WithCtrl` returns a flattening set of the stream ids in st; its formal definition can be found in Chapter 3.3.

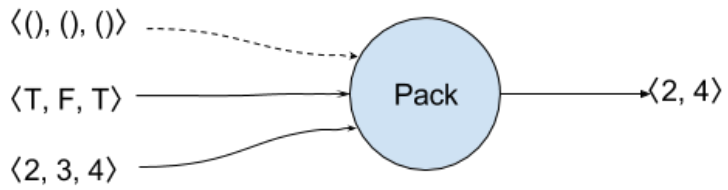
$$s' := \texttt{pack}_\pi(s_b, s); \quad \equiv \quad s' := \texttt{Pack}(s_b, s);$$

$$\begin{aligned}(st'_1, st'_2) := \texttt{pack}_{(\tau_1,\tau_2)}(s_b, (st_1, st_2)); \quad \equiv \quad & st'_1 := \texttt{pack}_{\tau_1}(s_b, st_1); \\ & st'_2 := \texttt{pack}_{\tau_2}(s_b, st_2);\end{aligned}$$

$$\begin{aligned}(st'_1, s'_1) := \texttt{pack}_{\{\tau\}}(s_b, (st_1, s_1)); \quad \equiv \quad & s'_1 := \texttt{UPack}(s_b, s_1); \\ & s'_2 := \texttt{Distr}(s_1, s_b); \\ & s_c := \texttt{Usum}(s_1); \\ & \overline{\overline{st'_1}} := \texttt{WithCtrl}(s_c, \{s'_2\} \cup \overline{\overline{st_1}}, st'_1 := \texttt{pack}_\tau(s'_2, st_1))\end{aligned}$$

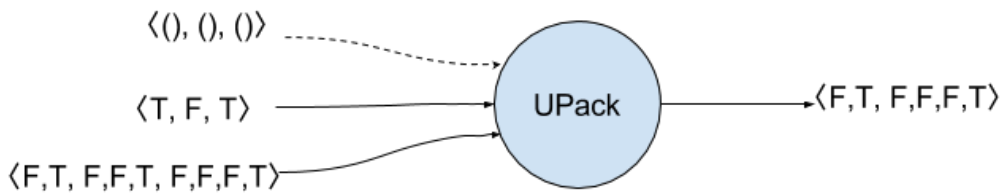
The Xducer `Pack` throws away the element of the second stream if the boolean of the corresponding position in the first stream is a `F`.

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 in a way similar to `Pack`, but on data of boolean segments rather than 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 instructions.

- 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 of length n. The translation will first use the Xducer `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:

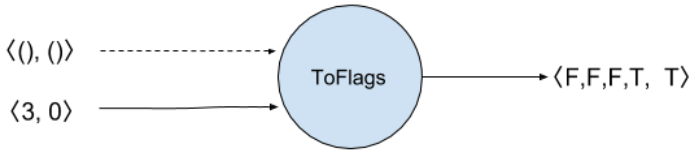
$$\frac{}{\textbf{iota}(s) \Rightarrow (p, (s_3, s_0))}$$

where

$$\begin{aligned} p = & s_0 := \texttt{ToFlags}(s); \\ & s_1 := \texttt{Usum}(s_0); \\ & \{s_2\} := \texttt{WithCtrl}(s_1, \{\}, s_2 := \texttt{Const}_1()); \\ & s_3 := \texttt{Scan}_+(s_0, s_2) \end{aligned}$$

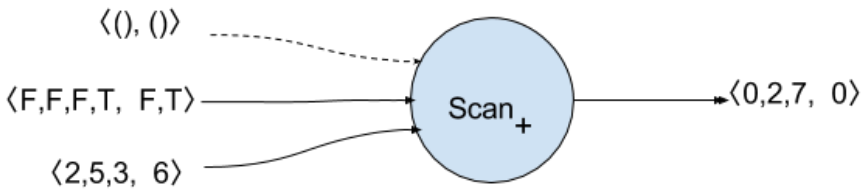
Given a stream $\langle n \rangle$, the Xducer `ToFlags` first outputs n `F`s, then one `T`.

Example 2.20. $\texttt{ToFlags}(\langle 3, 0 \rangle)$ with control stream $\langle (), () \rangle$:



$\texttt{Scan}_+(s_b, s_d)$ performs an exclusive scan of addition (with neutral element 0) on the data stream s_d segmented by s_b.

Example 2.21. $\texttt{Scan}_+(\langle \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{T} \rangle, \langle 2, 5, 3, 6 \rangle)$ with control stream $\langle (), () \rangle$:



- The high-level function **scan**$_+$ is implemented straightforwardly by using the Xducer $\texttt{Scan}_+$:

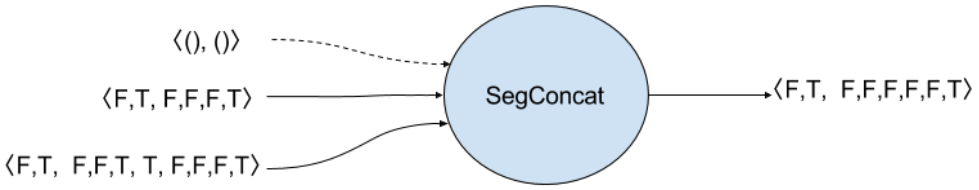
$$\frac{}{\textbf{scan}_+((s_d, s_b)) \Rightarrow (s_0 := \texttt{Scan}_+(s_b, s_d), (s_0, s_b))}$$

- Similar to **scan**$_+$, **reduce**$_+$ is translated to a low-level Xducer $\texttt{Reduce}_+$.
- The translation of **concat** is also one instruction using the Xducer `SegConcat`:

$$\frac{}{\textbf{concat}(((st, s_1), s_2)) \Rightarrow (s_0 := \texttt{SegConcat}(s_2, s_1), (st, s_0))}$$

The Xducer `SegConcat` merges the second outermost descriptors of the high-level sequence, 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.

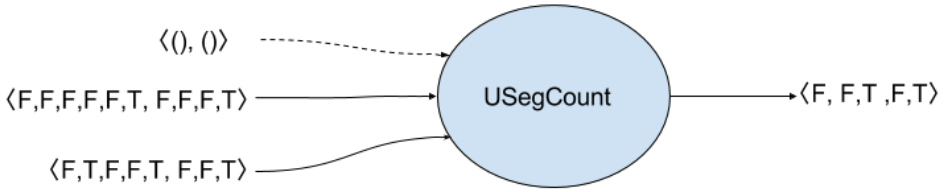


- **part** can be implemented straightforwardly by the Xducer `USegCount`.

$$\frac{}{\textbf{part}((st_1, s_1), (s_2, s_2'))) \Rightarrow (s_0 := \texttt{USegCount}(s_2, s_2'), ((st_1, s_2), s_0))}$$

`USegCount` counts the number of the segments of its second argument, segmented according to the second argument, and represents it in unary.

Example 2.23. $\texttt{USegCount}(\langle \texttt{F}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T} \rangle, \langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T} \rangle)$ with control stream $\langle (), () \rangle$. The first argument indicates that the first 5 elements of the second argument are in the same segment, which has two `T`s, and the last 3 another segment, which includes only one `T`. So the unary form of the counting result is $\langle \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{T} \rangle$



- Implementation of the function **append** may be the most tricky one, since it needs to recursively append subsequences at each depth of the argument sequences:

$$\frac{}{\textbf{append}((st_1, s_1), (st_2, s_2))) \Rightarrow (p, (st, s_0))}$$

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.

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

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

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

The Xducer `PrimSegInter` merges the given data streams according to their descriptors similarly to `InterMerge` but working on primitive data instead of boolean segments. `SegInter` merges a number of segments of a descriptor into one. 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 arbitrary number (≥ 1) 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 is one, then the data stream is returned.
- **empty** also uses a corresponding low-level Xducer `IsEmpty`.

  Example 2.24. Xducer $\texttt{IsEmpty}(\langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{T} \rangle)$ with control stream $\langle (), (), () \rangle$ outputs $\langle \texttt{F}, \texttt{F}, \texttt{T} \rangle$.

2.4.3 User-defined function translation

We first introduce 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 return values:

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

The overline function $\bar{}$ flattens a stream tree to a list of stream ids:

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

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

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

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

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

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

$$\Psi ::= [f_1 \mapsto sf_1, ..., f_i \mapsto sf_i]$$

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 will be unfolded by looking up f in Ψ and then passing the arguments.

2.5 Eager SVCODE 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 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 of the streaming one with the largest buffer size. 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. So the eager version can be used as a baseline to compare with the streaming one with different buffer sizes.

2.5.1 Dataflow

In the eager model, a Xducer consumes the entire input streams at once and outputs the whole result 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 stream definitions, because those inside a `WithCtrl` block may be skipped, thus we will not count their steps.

2.6 Streaming SVCODE interpreter

The execution model of the 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 than one 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 properly streamable SNESL programs, the buffer size can be as small as one.

2.6.1 Streamability

So far we have mentioned *streamable* for a few times, but not given a further explanation.

We consider an algorithm to be streamable if it can be executed in constant space, and more generally, in space linear in the recursion depth.

We should point out that not all algorithms are streamable. The situation where an algorithm is not streamable can be various. The first case easy to think about may be that the order of processing the data is random, not in the same direction of time, or in other words, it requires random access. For instance, most sorting algorithms, such as Mergesort, are not suitable to be streamed, since most of them involve element permutation or indexing. There are also some computations that look streamable, but can still be possible to fail, as Example 2.25 shows.

Static analysis for streamability is still an open problem [Mad16].

Example 2.25. The first expression are not properly streamable: after the stream s has been entirely consumed by `reducePlus` (i.e., $\texttt{Reduce}_+$), it is used to append to the end of the reduction result, which requires to recompute it; the second expression is properly streamable, since outputting s can be executed at the same time of computing the reduction.

```
> :bs 1    -- set buffer size = 1
>
> let s = &4 in {reducePlus(s)} ++ s    -- expression1
Deadlock!
>
> let s = &4 in s ++ {reducePlus(s)}    -- expression2
{0,1,2,3,6} :: {int}                    -- result
[W_H: 19, S_H: 5]                   -- high-level work and step cost
[W_L: 101, S_L: 37]                 -- low-level work and step cost
W_L/W_H: 5.3                            -- work bound
S_L/(W_H/bufSize + S_H): 1.5            -- step bound
>
> :bs 10    -- buffer size 10
>
> let s = &4 in {reducePlus(s)} ++ s    -- expression1
```

```
{6,0,1,2,3} :: {int}
[W_H: 19, S_H: 5]
[W_L: 101, S_L: 13]
W_L/W_H: 5.3
S_L/(W_H/bufSize + S_H): 1.9
```

2.6.2 Processes

In the streaming execution model, the output 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:

- `Filling`: the buffer is not full, and the Xducer is producing or writing data to it; any client trying to read it has to wait.
- `Draining`: the buffer must be full; the clients can read it only in this state; if the Xducer itself tries to write the buffer.

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 explain more about this later). In this way, the time of context-switching for a buffer is minimized, and the data is shared among its clients.

A notable special case is when the Xducer produces its last chunk, whose size may be less than the buffer size and 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, we have the definition of a buffer state as follows:

$$\begin{aligned}\textbf{BufState} ::= & \texttt{Filling}\ \vec{a} \\ & \mid \texttt{Draining}\ \vec{a}\ b\end{aligned}$$

where $\vec{a}$ is the data in the buffer.

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 end of the execution.

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 it has read. We will call this position the *read-cursor*. Also, it is possible that a Xducer reads from the same supplier multiple times but with different data rates, so we need to distinguish each use, rather than each client. In practice, we use the channel number, as we have shown in Figure 2.10, together with the read-cursor on the buffer to distinguish a use. As a result, we have a client list **Clis** of type $[(\textbf{SId}, \textbf{int}, \textbf{int})]$.

Now we use a structure *process*, a tuple of four components including a Xducer, to stand for one node on the streaming DAG with the type:

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

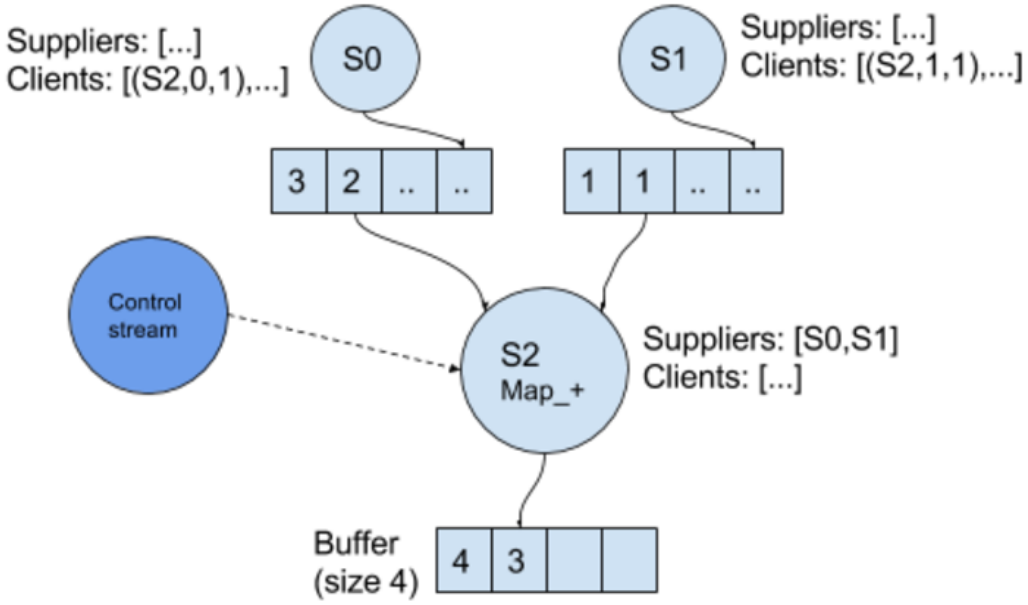


Figure 2.11: A process S2 of Xducer $\texttt{Map}_+$. It has read a 2 from S0's buffer and a 1 from S1's buffers, and it is writing a 3 to its own buffer.

where S is the stream ids of the suppliers. An example process of Xducer $\texttt{Map}_+$ can be found in Figure 2.11.

Since we do not have parallel implementation of Xducer, we can consider the Xducer inside a process as 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.

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 suppliers respectively) followed by one `Pout`, and a `Done` action can be added where the Xducer should shutdown:

$$[\texttt{Pin}_0, \texttt{Pin}_1, \texttt{Pout}, \texttt{Pin}_0, \texttt{Pin}_1, \texttt{Pout}, ..., \texttt{Done}]$$

where the subscripts of `Pin` indicate reading from different suppliers.

The process is responsible for providing its Xducer with the data and maintaining the state of the buffer. Its activities is described as the following table shows.

Xducer action \ Buffer state	`Filling`	`Draining F`	`Draining T`
`Pin`	`process-read`	`process-read`; `allread-check`	impossible
`Pout`	write one element to buffer; if buffer is full, switch to `Draining F`	wait to write, and `allread-check`	impossible
`Done`	switch to `Draining T`	switch to `Draining T`	`skip`

Table 2.1: Process activities. The description of `process-read` and `allread-check` are given later.

- `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 and increases the read-cursor by one
  - 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 waits
- `allread-check`: if all the clients have read all the data of the buffer, switch it to `Filling` state.

2.6.3 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 $(s'_1, ..., s'_{k'}) :=$ `SCall` $f(s_1, ..., s_k)$:
  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 $s_1, ..., s_k$, and the formal return ones with the actual ones $s'_1, ..., s'_{k'}$.

- initialize a `WithCtrl` block $S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p)$
  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 more than one chunk (a full buffer) of data before the interpreter knows whether s_c is an empty stream or not, i.e., add all of them a dummy client that never reads the buffer
  - initializes all the export streams of S_{out} as dummy processes that do not produce any data

(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 `DrainingT` 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 2.11 shows.

Another important 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:

- `Filling` $\langle\rangle$: the new control process has not produce any data yet, so the decision cannot be made in 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 set 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 block now, and replace the dummy clients/producers with actual clients/producers.

2.6.4 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

2.6.5 Recursion

In SVCODE, a recursive function call happens when the function body of f from the instruction $(s'_1, ..., s'_{k'}) :=$ `SCall` $f(s_1, ..., s_k)$ includes another `SCall` of f.

For a non-recursive `SCall` instruction, the effect of interpreting it is almost transparent. For a recursive one, there is not much difference except one crucial point: the recursive `SCall` can only occur within a `WithCtrl` block, otherwise it can never terminate. At each time of interpreting an inline `SCall` instruction, the function body is unfolded, but the `WithCtrl` instruction inside it will stop it from further unfolding, that is, the stack-frame number only increases by one.

At the high level, a SNESL program should use some conditional to decide when to terminate the recursion. As conditionals in SNESL are only comprehensions,

which are all translated to a `WithCtrl` block wrapping the expression body, so a recursion that can terminate at the high level will also terminate at the low level.

Example 2.26. The recursive function `fact` computes the factorial for a given number.

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

-- using eager interpreter
> let x = {3,7,0,4} in {fact(y): y in x }
{6,5040,1,24} :: {int}
[W_H: 227, S_H: 95]
[W_L: 1007, S_L: 140]
W_L/W_H: 4.4                -- work bound
S_L/S_H: 1.5                -- step bound

-- using streaming interpreter
> let x = {3,7,0,4} in {fact(y): y in x }
{6,5040,1,24} :: {int}
[W_H: 227, S_H: 95]
[W_L: 977, S_L: 128]
W_L/W_H: 4.3
S_L/(W_H/bufSize + S_H): 1.1
```

The outline of the translated SVCODE of the expression `let x = {3,7,0,4} in {fact(y): y in x}` is given in Figure 2.12. After generating the sequence `{3,7,0,4}`, the code only has two more instructions: a `Usum` to generate the new control stream, and a `WithCtrl` block only for controling the recursive function call.

```
-- initial control stream = <()>
    ...
S10 := InterMerge [S6,S7,S8,S9]; -- <F,F,F,F,T>
S11 := PriSegInter [(S1,S6),(S2,S7),(S3,S8),(S4,S9)]; -- <3,7,0,4>
S12 := Usum S10;                  -- <(),(),(),()>
S13 := WithCtrl S12 (import [S11]):
     [S13] := SCall fact [S11]    -- <6 5040 1 24> function call

Return: (S13, S10)
```

Figure 2.12: Outline SVCODE for the main function

The code of the function body is shown in Figure 2.13. The comments show the streams when the function is unfolded at its first time, i.e, the argument stream `S1 = [3,7,0,4]`. As the unfolding time increases, the graph of the dataflow grows dynamically. The total unfolding time will depend on the maximal depth of the recursive call of all elements. In our example, it is the number 7 that will unfold the function 6 times.

```

Parameters: [S1]            -- <3,7,0,4>
```

```
3
4  -- compare with 1
5  S3 := Const_1;                -- <1,1,1,1>
6  S4 := MapTwo Leq S1 S3;       -- <F F T F>
7  S5 := B2u S4;                 -- <T T FT T>
8
9  -- for elements <=1
10 S6 := Usum S5;                -- <    ()  >
11 [S7] := WithCtrl S6 []:
12           S7 := Const_1       -- <     1  >
13
14 S8 := Const_1;                -- <1>
15 S9 := MapTwo Leq S1 S8;       -- <F F T F>
16 S10 := MapOne Not S9;         -- <T T F T>
17 S11 := B2u S10;               -- <FT FT  FT>
18 S12 := Pack S1 S10;           -- <3 7   4>
19
20 -- for elementes >1
21 S13 := Usum S11;              -- <()()  ()>
22 [S17] := WithCtrl S13 [S12]:
23             S14 := Const_1   -- <1 1   1>
24             S15 := MapTwo Minus S12 S14  -- <2 6   3>
25             [S16] := SCall fact [S15]    -- <2 720   6> recursive
                  call
26             S17 := MapTwo Times S12 S16  -- <6 5040   24>
27 ...
28 S19 := PriSegInterS [(S7,S5),(S17,S11)];  -- <6 5040 1 24>
29
30 Return:  S19
```

Figure 2.13: The SVCODE of `fact` function body

2.6.6 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. In general, we classify deadlock situations into two types: soft deadlock, which can be broken not necessarily by enlarging the buffer size, and hard deadlock, which can only be solved by enlarging the buffer size or recomputation.

- Soft deadlock:

  One case of soft deadlock can be caused by the different data rates of processes that 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 2.27. A soft deadlock that can be broken by stealing

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

In this example, the input sequence contains elements from 0 to 9; the subsequence of the negated numbers, containing only 0 and -5, are concatenated with the one of the other eight numbers. Since these two subsequences are generated at different rates, the buffer for holding the shorter subsequence cannot get full when the other for the longer subsequence is already full to drain. Then the Xducer for appending their elements deadlocks. But if we minimize the buffer size to 1, then the deadlock does not happen, since buffer of size one immediately turns to `Draining` mode as soon as there is one element generated.

In our implementation, we use a *stealing* strategy to mitigate this type of deadlock. The idea is that when a deadlock is detected, we will first switch the smallest process with a `Filling` state buffer into `Draining` mode to see if the deadlock can be broken; if not, we repeat this switch until the deadlock is broken; or 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 may be affected and the effect depends on the concrete program and the buffer size. Some future work can be further investigation about the effect of this stealing strategy on the cost model.

- Hard deadlock:

  This type of deadlock is mainly caused by insufficient space or trying to traverse the same sequence multiple times.

  Example 2.28. A hard deadlock caused by traversing the sequence s two times.

```
> :bs 4
>
> let s = {1+ x : x in &5} in s ++ &2 ++ s
Deadlock!
```

  As the appending operation is sequential, which means its arguments must come one by one, the same stream s can not appear two times sequentially. One can solve the deadlock by recomputing the s stream, for example, defining another stream doing the same task so they can be used at different time, as the following code shows:

```
> let s = {1+ x : x in &5} ; s2 = {1+ x : x in &5} in s ++ &2
    ++ s2
{1,2,3,4,5,0,1,1,2,3,4,5} :: {int}
```

  Other solutions can be tabulating the sequence (into a vector), so it can be used multiple times, or enlarging the buffer size.

```
> :bs 10
>
> let s = {1+ x : x in &5} in s ++ &2 ++ s
{1,2,3,4,5,0,1,1,2,3,4,5} :: {int}
```

2.6.7 Examples

The following example program computes the united-and-conquer (exclusive) scan and reduce at the same. Due to a lack of primitive functions in the experimental language, we only implemented a version for a sequence with a length of power of 2 and require the length to be explicitly given.

For an input sequence of length n, the algorithm (for both scan and reduce) has an asymptotic cost of linear work and logarithmic step, i.e. $O(n)$ work and $O(\lg n)$ step, and its recursion depth is logarithmic as well.

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

For the following running example, the actual time of the unfolding of `scanred` is 5, i.e., $\lg 16 + 1$ (as we can see from our instrumented interpreter, but not shown here). Thus the size of the dataflow network, or the number of processes grows 4-fold after the initialization to reach its maximal size. We would expect that all streamable recursions use linear space in its recursion depth, while this requires more experiments and a formal proof.

```
1  > :bs 1
2  >
3  > scanred(&16,16)
4  ({0,0,1,3,6,10,15,21,28,36,45,55,66,78,91,105},120) :: ({int},int)
5  [W_H: 837, S_H: 188], [W_L: 4783, S_L: 1568]
6  W_L/W_H: 5.7
7  S_L/(W_H/bufSize + S_H): 1.5
8  >
9  > :bs 10
10 >
11 > scanred(&16,16)
12 ({0,0,1,3,6,10,15,21,28,36,45,55,66,78,91,105},120) :: ({int},int)
13 [W_H: 837, S_H: 188], [W_L: 4783, S_L: 351]
14 W_L/W_H: 5.7
15 S_L/(W_H/bufSize + S_H): 1.3
16 >
17 > :bs 100
18 >
19 > scanred(&16,16)
20 ({0,0,1,3,6,10,15,21,28,36,45,55,66,78,91,105},120) :: ({int},int)
21 [W_H: 837, S_H: 188], [W_L: 4783, S_L: 300]
22 W_L/W_H: 5.7
23 S_L/(W_H/bufSize + S_H): 1.5
```

Chapter 3

Formalization

In this chapter, we will present the formal proof of the correctness of the translation and the work cost preservation for the language SNESL_0, a core subset of SNESL_1. First its formal definition and semantics will be given. Then SVCODE_0, the target language of SNESL_0, is defined, and proofs of some of its properties including freshness and determinism are given. As the first step to formalize the full language, we only consider the eager semantics of this target language here. The value representation and translation from SNESL_0 to SVCODE_0 are also formalized. Finally, we put emphasis on the proof of the translation correctness theorem including work cost preservation.

3.1 SNESL_0

The language SNESL_0 we will formalize in this chapter is a subset of SNESL_1 with its core semantics. The simplifications we have made from SNESL_1 to SNESL_0 are listed below:

- only one primitive type **int**
- no pair types or zip-like comprehensions
- selected built-in functions
- no restricted comprehension
- no user-defined functions

3.1.1 Syntax

(1) The types of SNESL_0 are:

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

(2) The synatx of SNESL_0 values :

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

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

(3) The syntax of SNESL_0 expressions and the built-in functions are shown in Figure 3.1.

$$
\begin{array}{lr}
e ::= x & \text{(variable)} \\
\quad \mid \textbf{let } x = e_1 \textbf{ in } e_2 & \text{(let-binding)} \\
\quad \mid \phi(x_1, ..., x_k) & \text{(built-in function call)} \\
\quad \mid \{e : x \textbf{ in } y \textbf{ using } x_1, ..., x_k\} & \text{(general comprehension)}
\end{array}
$$

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

Figure 3.1: SNESL_0 expressions and built-in functions

Note that constants now are generated by calling the built-in function $\textbf{const}_n()$ to limit the number of expression forms. Also, the arguments of built-in functions as well as the input sequence in general comprehension are variables instead of expressions; the front end can simply convert the general forms of these expressions into the restricted forms by adding let-bindings for the variables. For example, the expression $\{x + y : x \textbf{ in } \&5 \textbf{ using } y\}$ in a general form can be turned to

$$
\begin{array}{l}
\textbf{let } t_1 = \textbf{const}_5() \textbf{ in} \\
\quad \textbf{let } t_2 = \textbf{iota}(t_1) \textbf{ in} \\
\qquad \{\textbf{plus}(x, y) : x \textbf{ in } t_2 \textbf{ using } y\}
\end{array}
$$

where t_1, t_2 are temporary variables used by the front-end compiler.

3.1.2 Typing rules

(1) Expression typing rules:

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

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

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

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

Since now there is only one concrete type in SNESL_0 (i.e., **int**), the free variables $x_1, ..., x_k$ in the rule for general comprehension must be all of type **int**.

(2) Built-in function typing rules:

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

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

(3) Value typing rules:

Judgment $\boxed{v : \tau}$

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

Notation 3.1. *Let* $|v|$ *denote the* size *of a value:*

$$|n| = 1$$
$$|\{v_1, ..., v_l\}| = 1 + \sum_{i=1}^{l} |v_i|$$

3.1.3 Semantics

(1) The evaluation rules with work cost of SNESL_0 are given below. The W in the judgment ($W \in \mathbb{N}$) stands for the work cost.

Judgment $\boxed{\rho \ \vdash \ e \downarrow v \ \$ \ W}$

$$\frac{}{\rho \ \vdash \ x \downarrow v \ \$ \ 0} \ (\rho(x) = v) \qquad \frac{\rho \ \vdash \ e_1 \downarrow v_1 \ \$ \ W_1 \qquad \rho[x \mapsto v_1] \ \vdash \ e_2 \downarrow v \ \$ \ W_2}{\rho \ \vdash \ \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \downarrow v \ \$ \ W_1 + W_2}$$

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

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

For variables, we consider its evaluation is zero-cost. Let-bindings need to evaluate both the bound expression e_1 and the body e_2, so the cost is the sum of these two evaluations. A function call simply costs the total size of its arguments and return value. The cost for evaluating a general comprehension is the sum of the evaluations for each element in the sequence, plus some extra $(k+1) \cdot (l+1)$ [1].

(2) Built-in function evaluation rules:

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

[1] In [Mad16], the extra is $k \cdot (l+1)$ (when the concrete type is restricted to only **int**), which should be a bug. In a special case where k=0 and the evaluation of the comprehension body does not cost anything, for example, $\{x : x \ \mathbf{in} \ \&5 \ \mathbf{using} \ \cdot\}$, the total cost for evaluating the comprehension will be zero as well. However, the low-level execution does pay some work cost (because of `Usum`) which is still proportional to the length of the sequence.

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

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

3.2 SVCODE$_0$

The target language of SNESL$_0$ is also a subset of SVCODE presented in the last chapter without the `SCall` instruction. We will call it SVCODE$_0$.

3.2.1 Syntax

In this minimal language, a primitive stream $\vec{a}$ can be a vector of booleans, integers or units, as the following grammar shows:

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

The syntax of SVCODE$_0$ is given in Figure 3.2.

$$\begin{aligned} p \;&::=\; \epsilon \\ &\mid s := \psi(s_1, ..., s_k) \\ &\mid S_{out} := \mathtt{WithCtrl}(s_c, S_{in}, p_1) \\ &\mid p_1; p_2 \end{aligned}$$

$$s \;::= 0 \mid 1 \mid \cdots \in \mathbf{SId} = \mathbb{N} \qquad \text{(stream ids)}$$

$$S \;::= \{s_1, ..., s_i\} \in \mathbb{S} \qquad \text{(a set of stream ids)}$$

$$\psi \;::= \mathtt{Const}_a \mid \mathtt{ToFlags} \mid \mathtt{Usum} \mid \mathtt{MapTwo}_+ \mid \mathtt{ScanPlus}_{n_0} \mid \mathtt{Distr} \qquad \text{(Xducers)}$$

Figure 3.2: Abstract syntax of SVCODE$_0$

We consider a well-formed program p must be able to be represented in a form $S \Vdash p : S'$, where S is a superset of all the free variables of p and S' a set of all the defined ones of p, and most importantly, there is no overlapping between S and S'.

Definition 3.2 (**Well-formedness**). *p is a well-formed SVCODE$_0$ program, written as $S \Vdash p : S'$ for some S and S', if it can be shown so by the following rules:*

Judgment $\boxed{S \Vdash p : S'}$

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

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

$$\frac{S \Vdash p_1 : S_1 \qquad S \cup S_1 \Vdash p_2 : S_2}{S \Vdash (p_1; p_2) : S_1 \cup S_2}$$

Lemma 3.3. *If $S \Vdash p : S'$, then $S \cap S' = \emptyset$.*

Proof. The proof is straightforward by induction on the derivation of $S \Vdash p : S'$. ■

3.2.2 Instruction semantics

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

Notation 3.4. *Let $\langle a_1, ..., a_i | \vec{a} \rangle$ denote a non-empty stream $\langle a_1, ..., a_i, a'_1, ..., a'_j \rangle$ for some $\vec{a} = \langle a'_1, ..., a'_j \rangle$;*

Notation 3.5 (Stream concatenation). $\langle a_1, ..., a_i \rangle +\!\!+ \langle a'_1, ..., a'_j \rangle = \langle a_1, ..., a_i, a'_1, ..., a'_j \rangle$

Notation 3.6 (Stream length). *For a stream $\vec{a} = \langle a_1, ..., a_l \rangle$, $|\vec{a}| = l$.*

The operational semantics of SVCODE_0 is given in Figure 3.3, and the W in the judgment stands for the work cost. The runtime environment or store σ is a mapping from stream variables to vectors:

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

The control stream $\vec{c}$, which is a vector of units, indicates the parallel degree of the computation, as we have discussed in the last chapter. It is worth noting that only in the rule P-Wc-Nonemp the control stream gets changed (from the conclusion to the premise).

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

P-Empty: $\dfrac{}{\langle \epsilon, \sigma\rangle \Downarrow^{\vec{c}} \sigma \$ 0}$

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

P-Wc-Emp :

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

P-Wc-Nonemp :

$$\frac{\langle p_1, \sigma\rangle \Downarrow^{\vec{c}_1} \sigma'' \$ W_1}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma''(s_i))_{i=1}^{k}] \$ W_1 + 1} \begin{pmatrix} \sigma(s_c) = \vec{c}_1 \neq \langle\rangle \\ S_{out} = \{s_1, ..., s_k\} \end{pmatrix}$$

P-Seq : $\dfrac{\langle p_1, \sigma\rangle \Downarrow^{\vec{c}} \sigma'' \$ W_1 \qquad \langle p_2, \sigma''\rangle \Downarrow^{\vec{c}} \sigma' \$ W_2}{\langle p_1; p_2, \sigma\rangle \Downarrow^{\vec{c}} \sigma' \$ W_1 + W_2}$

Figure 3.3: SVCODE$_0$ semantics

The rule P-Empty is simple: an empty program does nothing on the store, thus it does not cost anything.

The rule P-Xducer adds the store a new stream binding where the bound vector is generated by a specific Xducer. The detailed semantics of Xducers will be given in the next subsection. Its cost is the total size of the input and output streams.

The rules P-Wc-Emp and P-Wc-Nonemp together show two possibilities for interpreting a `WithCtrl` instruction:

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

The rule P-Seq is for executing two pieces of code sequentially, so the cost is the sum of these two.

3.2.3 Xducer semantics

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

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

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

$$\text{P-X-Termi}: \frac{}{\psi(\langle\rangle_1, ..., \langle\rangle_k) \Downarrow^{\langle\rangle} \langle\rangle}\ ^{2}$$

Figure 3.4: General semantics of SVCODE_0 Xducers

Figure 3.4 shows the semantics at the general level.

There are only two rules for the general semantics. They together say that the output stream of a Xducer is computed in a "loop" fashion, where each iteration uses specific block semantics of the Xducer and the number of iterations is the unary number that the control stream represents, i.e., the length of the control stream. In the parallel setting, we prefer to call this iteration a *block*. Recall that the control stream is a representation of the parallel degree of the computation, then a block stands for the subcomputation of exact one degree. It is worth noting that all these blocks are data-independent, which means they can be executed in parallel. So the control stream indeed carries the theoretical maximum number of processors we need to execute the computation most efficiently, if the computation within a block cannot be parallelized further.

After abstracting the general semantics, the remaining work of formalizing the specific semantics of Xducers within a block becomes relatively clear and simple. The block semantics are defined in Figure 3.5.

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

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

The formalization for these Xducers' semantics we have shown here is mainly for theoretical analysis and reasoning about the computation, but not for what the real implementation of them should be. In practice, most of the Xducers can be realized in a more efficient way to exploit parallelism by using a segmented scan operation [Ble89].

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

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

P-X-CONST: $\dfrac{}{\texttt{Const}_a() \downarrow \langle a \rangle}$

P-X-TOFLAGS: $\dfrac{}{\texttt{ToFlags}(\langle n \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle}\ (n \geq 0)$

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

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

P-X-SCANF : $\dfrac{\texttt{ScanPlus}_{n_0+n}(\vec{b}, \vec{a}) \downarrow \vec{a}'}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F} | \vec{b} \rangle, \langle n | \vec{a} \rangle) \downarrow \langle n_0 | \vec{a}' \rangle}$

P-SCANT: $\dfrac{}{\texttt{ScanPlus}_{n_0}(\langle \texttt{T} \rangle, \langle \rangle) \downarrow \langle \rangle}$

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

Figure 3.5: Block semantics of Xducers

3.2.4 SVCODE$_0$ determinism

Definition 3.7 (**Stream prefix**). *$\vec{a}$ is a* prefix *of $\vec{a}'$, written as $\vec{a} \sqsubseteq \vec{a}'$, if it can be shown so using the following rules:*

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

I-EMP: $\dfrac{}{\langle \rangle \sqsubseteq \vec{a}'}$ I-NONEMP: $\dfrac{\vec{a} \sqsubseteq \vec{a}'}{\langle a_0 | \vec{a} \rangle \sqsubseteq \langle a_0 | \vec{a}' \rangle}$

Lemma 3.8. *If $\vec{a}_1 {+\!\!+} \vec{a}_2 = \vec{a}$, then $\vec{a}_1 \sqsubseteq \vec{a}$.*

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

The following lemma says that a Xducer always knows how many elements it should consume and produce within a block.

Lemma 3.9 (**Blocks are self-delimiting and deterministic**). *If*

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

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

then

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

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

Proof. The proof is by induction on $\mathcal{P}$. We show three cases P–X-ToFlags, P–X-ScanT and P–X-ScanF here; the others are analogous.

- Case $\mathcal{P}$ uses P–X-ToFlags.
  Then
  $$\mathcal{P} = \frac{}{\texttt{ToFlags}(\langle n_1 \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_{n_1}, \texttt{T} \rangle}$$
  and
  $$\mathcal{P}' = \frac{}{\texttt{ToFlags}(\langle n_2 \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_{n_2}, \texttt{T} \rangle}$$
  so $k = 1, \vec{a}'_1 = \langle n_1 \rangle$, $\vec{a}' = \langle \texttt{F}_1, ..., \texttt{F}_{n_1}, \texttt{T} \rangle$, and $\vec{a}''_1 = \langle n_2 \rangle$, $\vec{a}'' = \langle \texttt{F}_1, ..., \texttt{F}_{n_2}, \texttt{T} \rangle$. Since both $\vec{a}'_1$ and $\vec{a}''_1$ are nonempty, $\mathcal{I}_1$ and $\mathcal{I}'_1$ must both use the rule I-Nonemp, which implies $n_1 = n_2$. Then it is clear that $\vec{a}'_1 = \vec{a}''_1$ and $\vec{a}' = \vec{a}''$, as required.

- Case $\mathcal{P}$ uses P–X-ScanT.
  Then
  $$\mathcal{P} = \frac{}{\texttt{ScanPlus}_{n_0}(\langle \texttt{T} \rangle, \langle\rangle) \downarrow \langle\rangle}$$
  so k=2, $\vec{a}'_1 = \langle \texttt{T} \rangle$. Since $\vec{a}'_1$ is nonempty, then $\mathcal{I}_1$ must use I-Nonemp, which implies the first element of $\vec{a}_1$ is T.
  There are two possibilities for $\mathcal{P}'$:

  - Subcase $\mathcal{P}'$ uses P–X-ScanF.
    This subcase is impossible, because it requires $\vec{a}_1$ to start with a F, which is contradictory to what we already know.
  - Subcase $\mathcal{P}'$ uses P–X-ScanT.
    Then
    $$\mathcal{P}' = \frac{}{\texttt{ScanPlus}_{n_0}(\langle \texttt{T} \rangle, \langle\rangle) \downarrow \langle\rangle}$$
    So $\vec{a}''_1 = \langle \texttt{T} \rangle = \vec{a}'_1$, $\vec{a}''_2 = \langle\rangle = \vec{a}'_2$, and $\vec{a}'' = \langle\rangle = \vec{a}''$, as required.

- Case $\mathcal{P}$ uses P–X-ScanF.
  Then
  $$\mathcal{P} = \frac{\mathcal{P}_0}{\texttt{ScanPlus}_{n_0+n}(\vec{a}'_{10}, \vec{a}'_{20}) \downarrow \vec{a}'_0}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F} | \vec{a}'_{10} \rangle, \langle n | \vec{a}'_{20} \rangle) \downarrow \langle n_0 | \vec{a}'_0 \rangle}$$
  So k=2, $\vec{a}'_1 = \langle \texttt{F} | \vec{a}'_{10} \rangle$, and $\vec{a}'_2 = \langle n | \vec{a}'_{20} \rangle$. $\mathcal{I}_1$ must use I-Nonemp, which implies the first element of $\vec{a}_1$ is F. So $\vec{a}_1 = \langle \texttt{F} | \vec{a}_{10} \rangle$ for some $\vec{a}_{10}$. By the rule I-Nonemp, we have
  $$\frac{\vec{a}'_{10} \sqsubseteq \vec{a}_{10}}{\langle \texttt{F} | \vec{a}'_{10} \rangle \sqsubseteq \langle \texttt{F} | \vec{a}_{10} \rangle}$$
  Similarly, we can assume $\vec{a}_2 = \langle n | \vec{a}_{20} \rangle$, and we must have $\vec{a}'_{20} \sqsubseteq \vec{a}_{20}$. Thus,
  $$(\vec{a}'_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^{2} \tag{3.1}$$
  There are two possibilities for $\mathcal{P}'$:

– Subcase $\mathcal{P}'$ uses P–X-ScanT.
This subcase is impossible, because $\vec{a}_1$ does not start with a `T`.
– Subcase $\mathcal{P}'$ uses P–X-ScanF.
Then

$$\mathcal{P}' = \frac{\begin{array}{c}\mathcal{P}'_0\\ \texttt{ScanPlus}_{n_0+n}(\vec{a}''_{10}, \vec{a}''_{20}) \downarrow \vec{a}''_0\end{array}}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F}|\vec{a}''_{10}\rangle, \langle n|\vec{a}''_{20}\rangle) \downarrow \langle n_0|\vec{a}''_0\rangle}$$

so $\vec{a}'_1 = \langle \texttt{F}|\vec{a}''_{10}\rangle$, $\vec{a}'_2 = \langle n|\vec{a}''_{20}\rangle$, $\vec{a}' = \langle n_0|\vec{a}''_0\rangle$, and it is easy to show

$$(\vec{a}''_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^2 \tag{3.2}$$

Here we first prove the following inner lemma:
if $(\vec{a}'_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^2$ and $\texttt{ScanPlus}_{n_0}(\vec{a}'_{10}, \vec{a}'_{20}) \downarrow \vec{a}'$ by some derivation $\mathcal{P}_0$, $(\vec{a}''_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^2$ and $\texttt{ScanPlus}_{n_0}(\vec{a}''_{10}, \vec{a}''_{20}) \downarrow \vec{a}''$, then $\vec{a}'_{10} = \vec{a}''_{10}, \vec{a}'_{20} = \vec{a}''_{20}$, and $\vec{a}' = \vec{a}''$.
The proof is by induction on $\mathcal{P}_0$. There are two subcases: for the case $\mathcal{P}_0$ uses P–X-ScanT, the proof is analogous to the outer proof case where $\mathcal{P}$ uses P–X-ScanT; for the case $\mathcal{P}_0$ uses P–X-ScanF, the proof can be done by the inner IH.
Then, by this inner lemma on (3.1) with $\mathcal{P}_0$, (3.2), $\mathcal{P}'_0$, we get $(\vec{a}'_{i0} = \vec{a}''_{i0})_{i=1}^2$, and $\vec{a}'_0 = \vec{a}''_0$.
Thus $\langle \texttt{F}|\vec{a}'_{10}\rangle = \langle \texttt{F}|\vec{a}''_{10}\rangle$, i.e., $\vec{a}'_1 = \vec{a}''_1$. Likewise, $\vec{a}'_2 = \langle n|\vec{a}'_{20}\rangle = \langle n|\vec{a}''_{20}\rangle = \vec{a}''_2$, and $\vec{a}' = \langle n_0|\vec{a}'_0\rangle = \langle n_0|\vec{a}''_0\rangle = \vec{a}''$, as required.

■

Lemma 3.10 (Xducer determinism). *If $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}_0$ by some derivation $\mathcal{P}$, and $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}'_0$ by some derivation $\mathcal{P}'$, then $\vec{a}_0 = \vec{a}'_0$.*

Proof. The proof is by induction on the structure of $\vec{c}$.

- Case $\vec{c} = \langle\rangle$
Then both $\mathcal{P}$ and $\mathcal{P}'$ must use P-X-Termi:

$$\mathcal{P} = \mathcal{P}' = \frac{}{\psi(\langle\rangle_1, ..., \langle\rangle_k) \Downarrow^{\langle\rangle} \langle\rangle}$$

so $\vec{a}_0 = \vec{a}'_0 = \langle\rangle$, as required.

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

$$\mathcal{P} = \frac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ \psi(\vec{a}_{11}, ..., \vec{a}_{k1}) \downarrow \vec{a}_{01} & \psi(\vec{a}_{12}, ..., \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02}\end{array}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle ()|\vec{c}_0\rangle} \vec{a}_0}$$

where

$$(\vec{a}_{i1} +\!\!+ \vec{a}_{i2} = \vec{a}_i)_{i=1}^k \tag{3.3}$$

$$\vec{a}_{01} +\!\!+ \vec{a}_{02} = \vec{a}_0 \tag{3.4}$$

Similarly,

$$\mathcal{P}' = \frac{\begin{array}{cc}\mathcal{P}'_1 & \mathcal{P}'_2\\ \psi(\vec{a}'_{11}, ..., \vec{a}'_{k1}) \downarrow \vec{a}'_{01} & \psi(\vec{a}'_{12}, ..., \vec{a}'_{k2}) \Downarrow^{\vec{c}_0} \vec{a}'_{02}\end{array}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle ()|\vec{c}_0\rangle} \vec{a}'_0}$$

where

$$(\vec{a}'_{i1} +\!\!+ \vec{a}'_{i2} = \vec{a}_i)_{i=1}^{k} \tag{3.5}$$

$$\vec{a}'_{01} +\!\!+ \vec{a}'_{02} = \vec{a}'_0 \tag{3.6}$$

Using Lemma 3.8 on each of the k equations of (3.3), we have

$$(\vec{a}_{i1} \sqsubseteq \vec{a}_i)_{i=1}^{k} \tag{3.7}$$

Analogously, from (3.5),

$$(\vec{a}'_{i1} \sqsubseteq \vec{a}_i)_{i=1}^{k} \tag{3.8}$$

By Lemma 3.9 on (3.7) with $\mathcal{P}_1$, (3.8), $\mathcal{P}'_1$, we get

$$(\vec{a}_{i1} = \vec{a}'_{i1})_{i=1}^{k} \tag{3.9}$$

$$\vec{a}_{01} = \vec{a}'_{01} \tag{3.10}$$

It is easy to show that from (3.3), (3.5) and (3.9) we can get

$$(\vec{a}_{i2} = \vec{a}'_{i2})_{i=1}^{k} \tag{3.11}$$

Then by IH on $\mathcal{P}_2$ with $\mathcal{P}'_2$, we obtain $\vec{a}_{02} = \vec{a}'_{02}$.
Therefore, with (3.4), (3.6), (3.10), we obtain $\vec{a}_0 = \vec{a}_{01} +\!\!+ \vec{a}_{02} = \vec{a}'_{01} +\!\!+ \vec{a}'_{02} = \vec{a}'_0$, as required.

■

Theorem 3.11 (SVCODE$_0$determinism). *If $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma' \$ W$ (by some derivation $\mathcal{P}$) and $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma'' \$ W'$ (by some derivation $\mathcal{P}'$), then $\sigma' = \sigma''$ and $W = W'$.*

Proof. The proof is by induction on the syntax of p. There are four cases: the case for $p = \epsilon$ is trivial; with the help of Lemma 3.10, the case for $p = s := \psi(s_1, ..., s_k)$ is immediate; proof of $p = p_1; p_2$ can be done by IH; the only interesting case is $p = S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)$.

- Case $p = S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)$.
  Assume $S_{out} = \{s_1, ..., s_k\}$. There are two subcases by induction on $\sigma(s_c)$:
  - Subcase $\sigma(s_c) = \langle\rangle$.
    Then $\mathcal{P}$ and $\mathcal{P}'$ must both use P-Wc-Emp, and they must be identical:

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

    with $\forall s \in S_{in}.\sigma(s) = \langle\rangle$. So $\sigma' = \sigma'' = \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{k}]$, and $W = W' = 1$, as required.
  - Subcase $\sigma(s_c) \neq \langle\rangle$.
    Then we must have

    $$\mathcal{P} = \frac{\begin{array}{c}\mathcal{P}_1 \\ \langle p_1, \sigma\rangle \Downarrow^{\vec{c}_1} \sigma_1 \$ W_1\end{array}}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma_1(s_i))_{i=1}^{k}] \$ W_1 + 1}$$

Also, we have

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

So $\sigma' = \sigma[(s_i \mapsto \sigma_1(s_i))_{i=1}^k]$, and $\sigma'' = \sigma[(s_i \mapsto \sigma'_1(s_i))_{i=1}^k]$.
By IH on p_1 with $\mathcal{P}_1$ and $\mathcal{P}'_1$, we obtain $\sigma_1 = \sigma'_1$ and $W_1 = W'_1$. Then it is clear that $\sigma' = \sigma''$, and $W = W_1 + 1 = W'_1 + 1 = W'$, as required.

■

3.3 Translation

(1) Since we do not have pairs, the type of stream trees here will be :

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

(2) A function for obtaining the set of stream ids in a stream tree (or flattening a stream tree to a set of stream ids):

$$\begin{aligned}\overline{\cdot} : \textbf{STree} &\to \mathbb{S} \\ \overline{s} &= \{s\} \\ \overline{(st_1, s)} &= \overline{st_1} \cup \{s\}\end{aligned}$$

The translation judgments shown below can be read as: " in the environment δ, the expression e (or the function call $\phi(st_1, ..., st_k)$) will be translated to an SVCODE$_0$ program p, and a stream tree st representing the evaluation result; the fresh stream allocation counter is increased from s_0 to s_1".

- Expression translation rules:

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

$$\frac{}{\delta \ \vdash \ x \Rightarrow_{s_0}^{s_0} (\epsilon, st)} \ (\delta(x) = st)$$

$$\frac{\delta \ \vdash \ e_1 \Rightarrow_{s'_0}^{s_0} (p_1, st_1) \qquad \delta[x \mapsto st_1] \ \vdash \ e_2 \Rightarrow_{s_1}^{s'_0} (p_2, st)}{\delta \ \vdash \ \textbf{let} \ x = e_1 \ \textbf{in} \ e_2 \Rightarrow_{s_1}^{s_0} (p_1; p_2, st)}$$

$$\frac{\phi(st_1, ..., st_k) \Rightarrow_{s_1}^{s_0} (p, st)}{\delta \ \vdash \ \phi(x_1, ..., x_k) \Rightarrow_{s_1}^{s_0} (p, st)} \ ((\delta(x_i) = st_i)_{i=1}^k)$$

$$\frac{[x \mapsto st_1, (x_i \mapsto s'_i)_{i=1}^k] \ \vdash \ e_1 \Rightarrow_{s''_1}^{s'_k+1} (p_1, st_2)}{\delta \ \vdash \ \{e_1 : x \ \textbf{in} \ y \ \textbf{using} \ x_1, ..., x_k\} \Rightarrow_{s''_1}^{s'_0} (p, (st_2, s_b))} \begin{pmatrix} \delta(y) = (st_1, s_b) \\ (\delta(x_i) = s_i)_{i=1}^k \\ p = (\ s'_0 := \texttt{Usum}(s_b); \\ \quad s'_1 := \texttt{Distr}(s_b, s_1); \\ \quad \vdots \\ \quad s'_k := \texttt{Distr}(s_b, s_k); \\ \quad S_{out} := \texttt{WithCtrl}(s'_0, S_{in}, p_1)) \\ S_{in} = \overline{\overline{st_1}} \cup \{s'_1, ..., s'_k\} \\ S_{out} = \{s \mid s \in \overline{\overline{st_2}}, s \geq s'_k + 1\} \\ s'_{i+1} = s'_i + 1, \forall i \in \{0, ..., k-1\} \end{pmatrix}$$

- Built-in function translation rules:

  Judgment $\boxed{\phi(st_1, ..., st_k) \Rightarrow_{s_1}^{s_0} (p, st)}$

  $$\frac{}{\textbf{const}_n() \Rightarrow_{s_0+1}^{s_0} (s_0 := \texttt{Const}_n(), s_0)}$$

  $$\frac{}{\textbf{plus}(s_1, s_2) \Rightarrow_{s_0+1}^{s_0} (s_0 := \texttt{MapTwo}_+(s_1, s_2), s_0)}$$

  $$\frac{}{\textbf{iota}(s) \Rightarrow_{s_4}^{s_0} (p, (s_3, s_0))} \begin{pmatrix} s_{i+1} = s_i + 1, \forall i \in \{0, ..., 3\} \\ p = s_0 := \texttt{ToFlags}(s); \\ \quad s_1 := \texttt{Usum}(s_0); \\ \quad \{s_2\} := \texttt{WithCtrl}(s_1, \emptyset, s_2 := \texttt{Const}_1()); \\ \quad s_3 := \texttt{ScanPlus}_0(s_0, s_2) \end{pmatrix}$$

Notation 3.12. *Let $S \lessdot s$ denote $\forall s' \in S.s' < s$.*

The following Lemma 3.13 and Theorem 3.14 together show that the translated SVCODE_0 program is well-formed and the defined stream ids are fresh (given the conditions are all satisfied).

Lemma 3.13. *If*

(i) $\phi(st_1, ..., st_k) \Rightarrow_{s_1}^{s_0} (p, st)$

(ii) $\bigcup_{i=1}^k \overline{\overline{st_i}} \subseteq S$

(iii) $S \lessdot s_0$

then, for some S',

(iv) $S \Vdash p : S'$

(v) $S' \subseteq \{s_0, s_0+1, ..., s_1-1\}$

(vi) $\overline{\overline{st}} \subseteq (S \cup S')$

Proof. The proof is by cases of the syntax of ϕ. ■

Theorem 3.14. ***If***

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

(ii) $\forall x \in dom(\delta).\overline{\overline{\delta(x)}} \subseteq S$

(iii) $S \lessdot s_0$

then, for some S',

(iv) $S \Vdash p : S'$

(v) $S' \subseteq \{s_0, s_0{+}1, ..., s_1{-}1\}$

(vi) $\overline{\overline{st}} \subseteq (S \cup S')$

Proof. The proof can be done by induction on the syntax of e. The proof case of $e = \phi(x_1, ..., x_k)$ uses Lemma 3.13. ■

3.4 Value representation

(1) Just like **STree**, the stream value tree also needs to be redefined as follows:

$$\mathbf{SvVal} \ni w ::= \vec{a} \mid (w, \vec{b})$$

(2) A function for SVCODE$_0$ value construction from a stream-id tree:

$$\sigma^* : \mathbf{STree} \to \mathbf{SvVal}$$
$$\sigma^*(s) = \sigma(s)$$
$$\sigma^*((st, s)) = (\sigma^*(st), \sigma(s))$$

(3) We annotate the operation $+\!+$ with the high-level type to represent the concatenation of SVCODE$_0$ values:

$$+\!+_\tau : \mathbf{SvVal} \to \mathbf{SvVal} \to \mathbf{SvVal}$$
$$\vec{a}_1 \ +\!+_{\mathbf{int}} \ \vec{a}_2 = \vec{a}_1 \ +\!+ \ \vec{a}_2$$
$$(w_1, \vec{b}_1) \ +\!+_{\{\tau\}} \ (w_2, \vec{b}_2) = (w_1 \ +\!+_\tau \ w_2, \vec{b}_1 \ +\!+ \ \vec{b}_2)$$

(4) The notation $\langle\rangle_\tau$ can represent a stream tree whose streams are all empty (i.e., an empty stream tree), or it can be viewed as that given a high-level type τ, a corresponding empty stream tree can be generated.

Notation 3.15. *Let* $\langle\rangle_\tau$ *denote an empty stream tree:*

$$\langle\rangle_{\mathbf{int}} = \langle\rangle$$
$$\langle\rangle_{\{\tau\}} = (\langle\rangle_\tau, \langle\rangle)$$

It is easy to check that, for any τ, $+\!+_\tau$ is associative with $\langle\rangle_\tau$ as neutral element.

(5) Value representation rules:

Judgment $\boxed{v \rhd_\tau w}$

$$\frac{}{n \rhd_{\mathbf{int}} \langle n \rangle} \qquad \frac{(v_i \rhd_\tau w_i)_{i=1}^{l}}{\{v_1, ..., v_l\} \rhd_{\{\tau\}} (w, \langle \mathtt{F}_1, ..., \mathtt{F}_l, \mathtt{T} \rangle)} \, (w = w_1 \ +\!+_\tau \cdots +\!+_\tau \ w_l)$$

(6) The following set of rules are used to recover/reconstruct a SNESL_0 value from a SVCODE_0 value. The judgment can be read as:" given a SVCODE_0 value w and a high-level type τ, a high-level value v can be recovered from the prefix of each stream of w, and the remaining part is w'".

Judgment $\boxed{w \lhd_\tau v, w'}$

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

The following two lemmas say that if a high-level value can be represented as a low-level one, then using the value recovery rules above, we can translate the low-level one back to the original high-level one, and the recovery is deterministic. A corollary that can be derived from these lemmas is that if two high-level values are represented as the same low-level one, then they must be identical.

Lemma 3.16 (**Recovery determinism**). *If* $w \lhd_\tau v, w'$, *and* $w \lhd_\tau v', w''$, *then* $v = v'$, *and* $w' = w''$.

Proof. The proof is by induction on the derivation of $w \lhd_\tau v, w'$. ■

Lemma 3.17 (**Recovery correctness**). *If* $v \rhd_\tau w$ *(by some derivation* $\mathcal{R}$*), then* $\forall w'.(w +\!\!+_\tau w') \lhd_\tau v, w'$.

Proof. The proof is by induction on $\mathcal{R}$. ■

Corollary 3.18. *If* $v \rhd_\tau w$, $v' \rhd_\tau w$, *then* $v = v'$.

Proof. The proof will use Lemma 3.17 (take $w' = \langle\rangle_\tau$) and Lemma 3.16. ■

3.5 Correctness

3.5.1 Definitions

We first define a binary relation $\overset{S}{\sim}$ on stores to denote that two stores are *similar*: they have identical domains, and their bound values of the stream ids in S are the same.

Definition 3.19 (**Store similarity**). $\sigma_1 \overset{S}{\sim} \sigma_2$ *iff*
(1) $dom(\sigma_1) = dom(\sigma_2)$
(2) $\forall s \in S.\sigma_1(s) = \sigma_2(s)$

According to this definition, it is only meaningful to have $S \subseteq dom(\sigma_1)$ $(= dom(\sigma_2))$. When $S = dom(\sigma_1) = dom(\sigma_2)$, σ_1 and σ_2 are identical. It is easy to show that this relation $\overset{S}{\sim}$ is symmetric and transitive.

- If $\sigma_1 \overset{S}{\sim} \sigma_2$, then $\sigma_2 \overset{S}{\sim} \sigma_1$.
- If $\sigma_1 \overset{S}{\sim} \sigma_2$ and $\sigma_2 \overset{S}{\sim} \sigma_3$, then $\sigma_1 \overset{S}{\sim} \sigma_3$.

We also define a binary operation $\overset{S}{\bowtie}$ on stores to represent a kind of fusion of two similar stores: the *fusion* of two similar stores is a new store, in which the bound values by S are from any of the parameter stores, and the others are the concatenation of the values from the two stores.

Definition 3.20 (**Store fusion**). *For* $\sigma_1 \overset{S}{\sim} \sigma_2$, $\sigma_1 \overset{S}{\bowtie} \sigma_2 = \sigma$ *where*

$$\sigma(s) = \begin{cases} \sigma_1(s)\ (= \sigma_2(s)), & s \in S \\ \sigma_1(s) {+\!+} \sigma_2(s), & s \notin S \end{cases}$$

Clearly, if $\sigma_1 \overset{S}{\bowtie} \sigma_2 = \sigma$, then $\sigma_1 \overset{S}{\sim} \sigma$ and $\sigma_2 \overset{S}{\sim} \sigma$.

Lemma 3.21 (**Xducer fusion**). *If*

(i) $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}} \vec{a}$ *by some derivation* $\mathcal{P}$

(ii) $\psi(\vec{a}'_1, ..., \vec{a}'_k) \Downarrow^{\vec{c}'} \vec{a}'$ *by some derivation* $\mathcal{P}'$,

then $\psi(\vec{a}_1 {+\!+} \vec{a}'_1, ..., \vec{a}_k {+\!+} \vec{a}'_k) \Downarrow^{\vec{c}{+\!+}\vec{c}'} \vec{a}{+\!+}\vec{a}'$ *by some* $\mathcal{P}''$.

Proof. By induction on the structure of $\vec{c}$.

- Case $\vec{c} = \langle\rangle$.
  Then $\mathcal{P}$ must use P-X-Termi:
  $$\mathcal{P} = \frac{}{\psi(\langle\rangle, ..., \langle\rangle) \Downarrow^{\langle\rangle} \langle\rangle}$$
  so $(\vec{a}_i = \langle\rangle)_{i=1}^k$ and $\vec{a} = \langle\rangle$. Then $(\vec{a}_i {+\!+} \vec{a}'_i = \vec{a}'_i)_{i=1}^k$, $\vec{c}{+\!+}\vec{c}' = \vec{c}'$ and $\vec{a}{+\!+}\vec{a}' = \vec{a}'$. Take $\mathcal{P}'' = \mathcal{P}'$ and we are done.

- Case $\vec{c} = \langle()|\vec{c}_0\rangle$ for some $\vec{c}_0$.
  Then $\mathcal{P}$ must use P-X-Loop:
  $$\mathcal{P} = \frac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}_2 \\ \psi(\vec{a}_{11}, ..., \vec{a}_{k1}) \downarrow \vec{a}_{01} & \psi(\vec{a}_{12}, ..., \vec{a}_{k2}) \Downarrow^{\vec{c}_0} \vec{a}_{02} \end{array}}{\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\langle()|\vec{c}_0\rangle} \vec{a}}$$
  with $(\vec{a}_i = \vec{a}_{i1} {+\!+} \vec{a}_{i2})_{i=1}^k$, and $\vec{a} = \vec{a}_{01} {+\!+} \vec{a}_{02}$.

  By IH on $\vec{c}_0$ with $\mathcal{P}_2$ and $\mathcal{P}'$, we get a derivation $\mathcal{P}'_2$ of
  $$\psi(\vec{a}_{12} {+\!+} \vec{a}'_1, ..., \vec{a}_{k2} {+\!+} \vec{a}'_k) \Downarrow^{\vec{c}_0{+\!+}\vec{c}'} \vec{a}_{02}{+\!+}\vec{a}'$$
  Then using the rule P-X-Loop we can build a derivation $\mathcal{P}'''$ as follows:
  $$\frac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}'_2 \\ \psi(\vec{a}_{11}, ..., \vec{a}_{k1}) \downarrow \vec{a}_{01} & \psi(\vec{a}_{12}{+\!+}\vec{a}'_1, ..., \vec{a}_{k2}{+\!+}\vec{a}'_k) \Downarrow^{\vec{c}_0{+\!+}\vec{c}'} \vec{a}_{02}{+\!+}\vec{a}' \end{array}}{\psi(\vec{a}_{11}{+\!+}(\vec{a}_{12}{+\!+}\vec{a}'_1), ..., \vec{a}_{k1}{+\!+}(\vec{a}_{k2}{+\!+}\vec{a}'_k)) \Downarrow^{\langle()|\vec{c}_0{+\!+}\vec{c}'\rangle} \vec{a}_{01}{+\!+}(\vec{a}_{02}{+\!+}\vec{a}')}$$
  Since it is clear that
  $$\forall i \in \{1, ..., k\}.\ \vec{a}_{i1}{+\!+}(\vec{a}_{i2}{+\!+}\vec{a}'_i) = (\vec{a}_{i1}{+\!+}\vec{a}_{i2}){+\!+}\vec{a}'_i = \vec{a}_i{+\!+}\vec{a}'_i$$
  $$\langle()|\vec{c}'_1{+\!+}\vec{c}_2\rangle = \langle()|\vec{c}'_1\rangle{+\!+}\vec{c}_2 = \vec{c}_1{+\!+}\vec{c}_2$$
  $$\vec{a}_{01}{+\!+}(\vec{a}_{02}{+\!+}\vec{a}') = (\vec{a}_{01}{+\!+}\vec{a}_{02}){+\!+}\vec{a}' = \vec{a}{+\!+}\vec{a}'$$
  so we take $\mathcal{P}'' = \mathcal{P}'''$, and we are done.

■

Notation 3.22. *Let* $\sigma \overset{S}{=\!=} \sigma'$ *denote:* $\forall s \in S.\sigma(s) = \sigma'(s)$.

The relation $\sigma \overset{S}{=\!=} \sigma'$ looks similar to $\sigma \overset{S}{\sim} \sigma'$, but it does not require σ and σ' have the same domain.

Lemma 3.23. *If* $S \Vdash p : S'$, $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma' \$ W$, *then* $\sigma' \overset{S}{=\!=} \sigma$.

Proof. The proof is by induction on the derivation of $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma' \$ W$. ■

Lemma 3.24. *If*

(i) $S \Vdash p : S'$

(ii) $\langle p, \sigma_1\rangle \Downarrow^{\vec{c}} \sigma_1' \$ W$

(iii) $\sigma_2 \overset{S}{=\!=} \sigma_1$

then, for some σ_2',

(iv) $\langle p, \sigma_2\rangle \Downarrow^{\vec{c}} \sigma_2' \$ W$

(v) $\sigma_2' \overset{S'}{=\!=} \sigma_1'$

Proof. The proof is by induction on the derivation of $\langle p, \sigma_1\rangle \Downarrow^{\vec{c}} \sigma_1' \$ W$. ■

The following lemma says that two separate executions of the same program p with different parallel degrees ($\vec{c}_1$ and $\vec{c}_2$) can be fused as one computation with a parallel degree of $\vec{c}_1 {+\!+} \vec{c}_2$.

Lemma 3.25 (Parallelism fusion). *If*

(i) $S_1 \Vdash p : S_2$ *(by some derivation* $\mathcal{W}$*)*

(ii) $\sigma_1 \overset{S}{\sim} \sigma_2$

(iii) $\langle p, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1' \$ W_1$ *(by some derivation* $\mathcal{P}_1$*)*

(iv) $\langle p, \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2' \$ W_2$ *(by some derivation* $\mathcal{P}_2$*)*

(v) $(S_1 \cup S_2) \cap S = \emptyset$

then, for some W,

(vi) $\sigma_1' \overset{S}{\sim} \sigma_2'$

(vii) $\left\langle p, \sigma_1 \overset{S}{\bowtie} \sigma_2 \right\rangle \Downarrow^{\vec{c}_1 {+\!+} \vec{c}_2} \sigma_1' \overset{S}{\bowtie} \sigma_2' \$ W$ *(by* $\mathcal{P}$*)*

(viii) $W \leq W_1 + W_2$

Proof. By induction on the syntax of p.

- Case $p = \epsilon$.
  $\mathcal{P}_1$ must be $\dfrac{}{\langle \epsilon, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1 \$ 0}$, and $\mathcal{P}_2$ must be $\dfrac{}{\langle \epsilon, \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2 \$ 0}$.
  So $\sigma_1' = \sigma_1$, and $\sigma_2' = \sigma_2$, thus $\sigma_1' \overset{S}{\sim} \sigma_2'$, and $\sigma_1' \overset{S}{\bowtie} \sigma_2' = \sigma_1 \overset{S}{\bowtie} \sigma_2$.

  By P-Empty, we take $\mathcal{P} = \dfrac{}{\left\langle \epsilon, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 {+\!+} \vec{c}_2} (\sigma_1 \overset{S}{\bowtie} \sigma_2) \$ 0}$ and it's clear $W = 0 \leq 0 + 0$, as required.

- Case $p = s := \psi(s_1, ..., s_k)$.
  $\mathcal{P}_1$ must look like

$$\dfrac{\begin{array}{c}\mathcal{P}_1' \\ \psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}_1} \vec{a}\end{array}}{\langle s := \psi(s_1, ..., s_k), \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1[s \mapsto \vec{a}] \ \$ \ (\sum_{i=1}^{k} |\vec{a}_i|) + |\vec{a}|}$$

  and we have

$$(\sigma_1(s_i) = \vec{a}_i)_{i=1}^{k} \tag{3.12}$$

  Similarly, $\mathcal{P}_2$ must look like

$$\dfrac{\begin{array}{c}\mathcal{P}_2' \\ \psi(\vec{a}_1', ..., \vec{a}_k') \Downarrow^{\vec{c}_2} \vec{a}'\end{array}}{\langle s := \psi(s_1, ..., s_k), \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2[s \mapsto \vec{a}'] \ \$ \ (\sum_{i=1}^{k} |\vec{a}_i'|) + |\vec{a}'|}$$

  and we have

$$(\sigma_2(s_i) = \vec{a}_i')_{i=1}^{k} \tag{3.13}$$

  So $\sigma_1' = \sigma_1[s \mapsto \vec{a}], \sigma_2' = \sigma_2[s \mapsto \vec{a}']$, and $W_1 = (\sum_{i=1}^{k} |\vec{a}_i|) + |\vec{a}|$, $W_2 = (\sum_{i=1}^{k} |\vec{a}_i'|) + |\vec{a}'|$. Clearly, $\sigma_1' \overset{S}{\sim} \sigma_2'$.

  From assumption (i) and (v) we have $\{s_1, ..., s_k\} \subseteq S_1 \cap S = \emptyset$, that is,

$$\{s_1, ..., s_k\} \cap S = \emptyset \tag{3.14}$$

  By Lemma 3.21 on $\mathcal{P}_1'$, $\mathcal{P}_2'$, we get a derivation $\mathcal{P}'$ of

$$\psi(\vec{a}_1{+\!+}\vec{a}_1', ..., \vec{a}_k{+\!+}\vec{a}_k') \Downarrow^{\vec{c}_1{+\!+}\vec{c}_2} \vec{a}{+\!+}\vec{a}'$$

  Since $\sigma_1 \overset{S}{\sim} \sigma_2$, with (3.12),(3.13) and (3.14), by Definition 3.20 we have

$$\forall i \in \{1, ..., k\}.(\sigma_1 \overset{S}{\bowtie} \sigma_2)(s_i) = \sigma_1(s_i){+\!+}\sigma_2(s_i) = \vec{a}_i{+\!+}\vec{a}_i' \tag{3.15}$$

  Also, it is easy to prove

$$\sigma_1[s \mapsto \vec{a}] \overset{S}{\bowtie} \sigma_2[s \mapsto \vec{a}'] = (\sigma_1 \overset{S}{\bowtie} \sigma_2)[s \mapsto \vec{a}{+\!+}\vec{a}'] \tag{3.16}$$

  Using the rule P-Xducer with (3.15), we can build $\mathcal{P}''$ as follows

$$\dfrac{\begin{array}{c}\mathcal{P}' \\ \psi(\vec{a}_1{+\!+}\vec{a}_1', ..., \vec{a}_k{+\!+}\vec{a}_k') \Downarrow^{\vec{c}_1{+\!+}\vec{c}_2} \vec{a}{+\!+}\vec{a}'\end{array}}{\left\langle s := \psi(s_1, ..., s_k), (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1{+\!+}\vec{c}_2} (\sigma_1 \overset{S}{\bowtie} \sigma_2)[s \mapsto \vec{a}{+\!+}\vec{a}'] \ \$ \ (\sum_{i=1}^{k} |\vec{a}_i{+\!+}\vec{a}_i'|) + |\vec{a}{+\!+}\vec{a}'|}$$

  With (3.16), we take $\mathcal{P} = \mathcal{P}''$, and it is clear that $W = (\sum_{i=1}^{k} |\vec{a}_i{+\!+}\vec{a}_i'|) + |\vec{a}{+\!+}\vec{a}'| = W_1 + W_2$ as required.

- Case $p = S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0)$.
  We must have

$$\mathcal{W} = \frac{\begin{array}{c}\mathcal{W}_0 \\ S_{in} \Vdash p_0 : S_2\end{array}}{S_1 \Vdash S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0) : S_{out}}$$

  where

$$(S_{in} \cup \{s_c\}) \subseteq S_1, \quad S_{out} \subseteq S_2 \tag{3.17}$$

  From (v) we have $(S_{in} \cup \{s_c\}) \subseteq S_1 \cap S = \emptyset$, that is,

$$\{s_c\} \cap S = \emptyset \tag{3.18}$$

$$S_{in} \cap S = \emptyset \tag{3.19}$$

  Assume $S_{out} = \{s_1, ..., s_k\}$.
  There are four possibilities:

  – Subcase both $\mathcal{P}_1$ and $\mathcal{P}_2$ use P-Wc-Emp.

    So $\mathcal{P}_1$ must look like

$$\frac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1[s_1 \mapsto \langle\rangle, ..., s_k \mapsto \langle\rangle] \ \$ \ 1}$$

    and we have

$$\forall s \in \{s_c\} \cup S_{in}.\sigma_1(s) = \langle\rangle \tag{3.20}$$

    Similarly,$\mathcal{P}_2$ must look like

$$\frac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2[s_1 \mapsto \langle\rangle, ..., s_k \mapsto \langle\rangle] \ \$ \ 1}$$

    and

$$\forall s \in \{s_c\} \cup S_{in}.\sigma_2(s) = \langle\rangle \tag{3.21}$$

    So $\sigma_1' = \sigma_1[(s_i \mapsto \langle\rangle)_{i=1}^k]$, $\sigma_2' = \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^k], W_1 = 1, W_2 = 1$.

    Since $\sigma_1 \overset{S}{\sim} \sigma_2$, by Definition 3.20 with (3.18), (3.19), and (3.20), (3.21), we have

$$\forall s \in \{s_c\} \cup S_{in}.(\sigma_1 \overset{S}{\bowtie} \sigma_2)(s) = \sigma_1(s) +\!\!+ \sigma_2(s) = \langle\rangle \tag{3.22}$$

    Also, it is easy to show that $\sigma_1' \overset{S}{\sim} \sigma_2'$ and

$$\sigma_1' \overset{S}{\bowtie} \sigma_2' = (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \langle\rangle)_{i=1}^k] \tag{3.23}$$

    Using P-Wc-Emp with (3.22), we can build a derivation $\mathcal{P}'$ as follows

$$\frac{}{\left\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \langle\rangle)_{i=1}^k] \ \$ \ 1}$$

    With (3.23), we take $\mathcal{P} = \mathcal{P}'$. And $W = 1 \leq 1 + 1 = W_1 + W_2$ as requierd.

– Subcase $\mathcal{P}_1$ uses P-Wc-Nomemp, $\mathcal{P}_2$ uses P-Wc-Emp.

$\mathcal{P}_1$ must look like

$$\frac{\begin{array}{c}\mathcal{P}_1' \\ \langle p_0, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1'' \ \$ \ W_0\end{array}}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^k] \ \$ \ W_0 + 1}$$

and we have

$$\sigma_1(s_c) = \vec{c}_1' \neq \langle\rangle \tag{3.24}$$

$\mathcal{P}_2$ must look like

$$\frac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^k] \ \$ \ 1}$$

and we have $\forall s \in \{s_c\} \cup S_{in}.\sigma_2(s) = \langle\rangle$ thus

$$\sigma_2(s_c) = \langle\rangle \tag{3.25}$$

$$\forall s \in S_{in}.\sigma_2(s) = \langle\rangle \tag{3.26}$$

So $\sigma_1' = \sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^k]$, $W_1 = W_0 + 1$, $\sigma_2' = \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^k]$ and $W_2 = 1$.
Since $\sigma_1 \overset{S}{\sim} \sigma_2$, it is easy to show that

$$\forall s \in S_{in}.(\sigma_1 \overset{S}{\bowtie} \sigma_2)(s) = \sigma_1(s) +\!\!+ \sigma_2(s) = \sigma_1(s) \tag{3.27}$$

Then by Lemma 3.24 on $\mathcal{W}_0$ with $\mathcal{P}_1'$, (3.27), we obtain a derivation $\mathcal{P}_0$ of

$$\left\langle p_0, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1} \sigma_0 \ \$ \ W_0$$

for some σ_0, and

$$\sigma_0 \overset{S_2}{=\!=} \sigma_1''$$

Then, with (3.17), we have

$$(\sigma_0(s_i) = \sigma_1''(s_i))_{i=1}^k \tag{3.28}$$

Since $\sigma_1 \overset{S}{\sim} \sigma_2$, by Definition 3.20 with (3.24), (3.25), we have

$$(\sigma_1 \overset{S}{\bowtie} \sigma_2)(s_c) = \sigma_1(s_c) +\!\!+ \sigma_2(s_c) = \vec{c}_1' \neq \langle\rangle \tag{3.29}$$

and it is also easy to show $\sigma_1' \overset{S}{\sim} \sigma_2'$ and

$$(\sigma_1' \overset{S}{\bowtie} \sigma_2') = (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \sigma_1''(s_1))_{i=1}^k] \tag{3.30}$$

With (3.28), we replace $\sigma_1''(s_i)$ with $\sigma_0(s_i)$ for $\forall i \in \{1, ..., k\}$ in (3.30), giving us

$$(\sigma_1' \overset{S}{\bowtie} \sigma_2') = (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \sigma_0(s_1))_{i=1}^k] \tag{3.31}$$

Since (3.29), we can use the rule P-Wc-Nonemp to build a derivation $\mathcal{P}'$ as follows

$$\frac{\begin{array}{c}\mathcal{P}_0\\ \left\langle p_0, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1} \sigma_0 \ \$ \ W_0\end{array}}{\left\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \sigma_0(s_i))_{i=1}^k] \ \$ \ W_0 + 1}$$

Then with (3.31), we take $\mathcal{P} = \mathcal{P}'$, and it is clear $W = W_0 + 1 = W_1 \leq W_1 + 1 = W_1 + W_2$, as required.

– Subcase $\mathcal{P}_1$ uses P-Wc-Emp and $\mathcal{P}_2$ uses P-Wc-Nonemp.
This subcase is analogous to the previous one.

– Subcase both $\mathcal{P}_1$ and $\mathcal{P}_2$ use P-Wc-Nonemp.
$\mathcal{P}_1$ must look like

$$\frac{\begin{array}{c}\mathcal{P}_1'\\ \langle p_0, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1'' \ \$ \ W_1'\end{array}}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^k] \ \$ \ W_1' + 1}$$

and

$$\sigma_1(s_c) = \vec{c}_1' \neq \langle\rangle \tag{3.32}$$

Similarly, $\mathcal{P}_2$ must look like

$$\frac{\begin{array}{c}\mathcal{P}_2'\\ \langle p_0, \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2'' \ \$ \ W_2'\end{array}}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2[(s_i \mapsto \sigma_2''(s_i))_{i=1}^k] \ \$ \ W_2' + 1}$$

and

$$\sigma_2(s_c) = \vec{c}_2' \neq \langle\rangle \tag{3.33}$$

So $\sigma_1' = \sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^k], W_1 = W_1' + 1, \sigma_2' = \sigma_2[(s_i \mapsto \sigma_2''(s_i))_{i=1}^k]$ and $W_2 = W_2' + 1$.

By IH on p_0 with $\mathcal{W}_0$, $\mathcal{P}_1'$, $\mathcal{P}_2'$, we get $\sigma_1'' \overset{S}{\sim} \sigma_2''$, a derivation $\mathcal{P}_0$ of

$$\left\langle p_0, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1' +\!\!+ \vec{c}_2'} \sigma_1'' \overset{S}{\bowtie} \sigma_2'' \ \$ \ W'$$

and $W' \leq W_1' + W_2'$.
Since $\forall i \in \{1, ..., k\}.s_i \in S_{out}$ and $S_{out} \cap S = \emptyset$, then by Definition 3.20, we know

$$(\sigma_1'' \overset{S}{\bowtie} \sigma_2'')(s_i) = \sigma_1''(s_i) +\!\!+ \sigma_2''(s_i) \tag{3.34}$$

Also, it is easy to show that $\sigma_1' \overset{S}{\sim} \sigma_2'$, and

$$(\sigma_1' \overset{S}{\bowtie} \sigma_2') = (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \sigma_1''(s_i) +\!\!+ \sigma_2''(s_i))_{i=1}^k] \tag{3.35}$$

Then, with (3.34), we replace $\sigma_1''(s_i) +\!\!+ \sigma_2''(s_i)$ with $(\sigma_1'' \overset{S}{\bowtie} \sigma_2'')(s_i)$ for $\forall i \in \{1, ..., k\}$ in (3.35), giving us

$$(\sigma_1' \overset{S}{\bowtie} \sigma_2') = (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto (\sigma_1'' \overset{S}{\bowtie} \sigma_2'')(s_i))_{i=1}^k] \tag{3.36}$$

Since (3.18) with (3.32), (3.33), we know $(\sigma_1 \overset{S}{\bowtie} \sigma_2)(s_c) = \vec{c}_1 +\!\!+ \vec{c}_2 \neq \langle\rangle$, therefore we can use the rule P-Wc-Nonemp to build a derivation $\mathcal{P}'$ as follows:

$$\frac{\mathcal{P}_0}{\left\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0), (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} (\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto (\sigma_1'' \overset{S}{\bowtie} \sigma_2'')(s_i))_{i=1}^{k}] \ \$ \ W' + 1} \quad \frac{\left\langle p_0, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma_1'' \overset{S}{\bowtie} \sigma_2'' \ \$ \ W'}{}.$$

Therefore, with (3.36), we take $\mathcal{P} = \mathcal{P}'$, and it is clear that $W = W' + 1 < W_1' + 1 + W_2' + 1 = W_1 + W_2$ as required.

- Case $p = p_1; p_2$
  We must have

$$\mathcal{W} = \frac{\mathcal{W}_1 \qquad \mathcal{W}_2}{S_0 \Vdash p_1 : S_1 \qquad S_0 \cup S_1 \Vdash p_2 : S_2}{S_0 \Vdash (p_1; p_2) : S_1 \cup S_2}$$

$$\mathcal{P}_1 = \frac{\mathcal{P}_1' \qquad \mathcal{P}_1''}{\langle p_1, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1'' \ \$ \ W_1' \qquad \langle p_2, \sigma_1''\rangle \Downarrow^{\vec{c}_1} \sigma_1' \ \$ \ W_1''}{\langle p_1; p_2, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1' \ \$ \ W_1' + W_1''}$$

  and

$$\mathcal{P}_2 = \frac{\mathcal{P}_2' \qquad \mathcal{P}_2''}{\langle p_1, \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2'' \ \$ \ W_2' \qquad \langle p_2, \sigma_2''\rangle \Downarrow^{\vec{c}_2} \sigma_2' \ \$ \ W_2''}{\langle p_1; p_2, \sigma_2\rangle \Downarrow^{\vec{c}_1} \sigma_2' \ \$ \ W_2' + W_2''}$$

  From (v) it is easy to show that

$$S_0 \cap S = \emptyset \tag{3.37}$$

$$S_1 \cap S = \emptyset \tag{3.38}$$

$$S_2 \cap S = \emptyset \tag{3.39}$$

  Then by IH on p_1 with $\mathcal{W}_1$, $\mathcal{P}_1'$, $\mathcal{P}_2'$, (3.37), we get

$$\sigma_1'' \overset{S}{\sim} \sigma_2'' \tag{3.40}$$

  and a derivation $\mathcal{P}'$ of

$$\left\langle p_1, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma_1'' \overset{S}{\bowtie} \sigma_2'' \ \$ \ W'$$

  where $W' \leq W_1' + W_2'$.

  Likewise, by IH on p_2 with $\mathcal{W}_2$, (3.40), $\mathcal{P}_1''$, $\mathcal{P}_2''$, we get $\sigma_1' \overset{S}{\sim} \sigma_2'$, and a derivation $\mathcal{P}''$ of

$$\left\langle p_2, \sigma_1'' \overset{S}{\bowtie} \sigma_2''\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} (\sigma_1' \overset{S}{\bowtie} \sigma_2') \ \$ \ W''$$

  where $W'' \leq W_1'' + W_2''$.

  Therefore, we use the rule P-Seq to build $\mathcal{P}$ as follows:

$$\frac{\mathcal{P}' \qquad \mathcal{P}''}{\left\langle p_1, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma_1'' \overset{S}{\bowtie} \sigma_2'' \ \$ \ W' \qquad \left\langle p_2, \sigma_1'' \overset{S}{\bowtie} \sigma_2''\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} (\sigma_1' \overset{S}{\bowtie} \sigma_2') \ \$ \ W''}{\left\langle p_1; p_2, (\sigma_1 \overset{S}{\bowtie} \sigma_2)\right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} (\sigma_1' \overset{S}{\bowtie} \sigma_2') \ \$ \ W}$$

  and it is clear $W = W' + W'' \leq W_1' + W_2' + W_1'' + W_2'' = W_1 + W_2$, as required.

■

3.5.2 Correctness proof

Lemma 3.26 (Correctness of built-in functions). *For some constant C, if*

(i) $\phi : (\tau_1, ..., \tau_k) \to \tau$ *(by some derivation $\mathcal{T}$)*

(ii) $\phi(v_1, ..., v_k) \downarrow v$ *(by $\mathcal{E}$)*

(iii) $\phi(st_1, ..., st_k) \Rightarrow_{s_1}^{s_0} (p, st)$ *(by $\mathcal{C}$)*

(iv) $(v_i \triangleright_{\tau_i} \sigma^*(st_i))_{i=1}^k$

then, for some σ' and W,

(v) $\langle p, \sigma\rangle \Downarrow^{\langle()\rangle} \sigma'$ \$ W *(by $\mathcal{P}$)*

(vi) $v \triangleright_\tau \sigma'^*(st)$ *(by $\mathcal{R}$)*

(vii) $W \leq C \cdot (\sum_{i=1}^k |v_i| + |v|)$

Proof. The proof is by cases on the syntax of ϕ. We will show that we can take $C = 7$ here, which will satisfy all the proof cases.

- Case $\phi = \mathbf{const}_n$
  There is only one possibility for each of $\mathcal{T}$, $\mathcal{E}$ and $\mathcal{C}$:

$$\mathcal{T} = \frac{}{\mathbf{const}_n : () \to \mathbf{int}}$$

$$\mathcal{E} = \frac{}{\mathbf{const}_n() \downarrow n}$$

$$\mathcal{C} = \frac{}{\mathbf{const}_n() \Rightarrow_{s_0+1}^{s_0} (s_0 := \mathtt{Const_n}(), s_0)}$$

  So $k = 0, \tau = \mathbf{int}, v = n, p = s_0 := \mathtt{Const_n}()$, $s_1 = s_0 + 1$, and $st = s_0$

  By P-Xducer, P-X-Loop, P-X-Termi and P-X-Const, we can construct $\mathcal{P}$ as follows:

$$\mathcal{P} = \frac{\dfrac{\dfrac{}{\mathtt{Const_n}() \downarrow \langle n\rangle} \quad \dfrac{}{\mathtt{Const_n}() \Downarrow^{\langle\rangle} \langle\rangle}}{\mathtt{Const_n}() \Downarrow^{\langle()\rangle} \langle n\rangle}}{\langle s_0 := \mathtt{Const_n}(), \sigma\rangle \Downarrow^{\langle()\rangle} \sigma[s_0 \mapsto \langle n\rangle] \text{ \$ } 1}$$

  So $\sigma' = \sigma[s_0 \mapsto \langle n\rangle]$.

  Then we take $\mathcal{R} = \dfrac{}{n \triangleright_{\mathbf{int}} \sigma'(s_0)}$

  Also clearly, $W = 1 = |v|$, so C can be any number ≥ 1.

- Case $\phi = \mathbf{plus}$
  We must have

$$\mathcal{T} = \frac{}{\mathbf{plus} : (\mathbf{int}, \mathbf{int}) \to \mathbf{int}}$$

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

  where $n_3 = n_2 + n_1$, and

$$\mathcal{C} = \frac{}{\mathbf{plus}(s_1, s_2) \Rightarrow_{s_0+1}^{s_0} (s_0 := \mathtt{MapTwo}_+(s_1, s_2), s_0)}$$

So $k = 2$ and $v_1 = n_1, v_2 = n_2, v = n_3, st = s_0$.

Assumption (iv) gives us $\overline{n_1 \triangleright_{\mathbf{int}} \sigma(s_1)}$ and $\overline{n_2 \triangleright_{\mathbf{int}} \sigma(s_2)}$, which implies $\sigma(s_1) = \langle n_1 \rangle$ and $\sigma(s_2) = \langle n_2 \rangle$ respectively.
Then using P-XDUCER with $\sigma(s_1) = \langle n_1 \rangle$ and $\sigma(s_2) = \langle n_2 \rangle$, P-X-LOOP, P-X-TERMI, and P-X-MAPTWO we can build $\mathcal{P}$ as follows:

$$\dfrac{\dfrac{\overline{\texttt{MapTwo}_+(\langle n_1 \rangle, \langle n_2 \rangle) \downarrow \langle n_3 \rangle} \quad \overline{\texttt{MapTwo}_+(\langle\rangle, \langle\rangle) \Downarrow^{\langle\rangle} \langle\rangle}}{\texttt{MapTwo}_+(\langle n_1 \rangle, \langle n_2 \rangle) \Downarrow^{\langle()\rangle} \langle n_3 \rangle}}{\left\langle s_0 := \texttt{MapTwo}_+(s_1, s_2), \sigma \right\rangle \Downarrow^{\langle()\rangle} \sigma[s_0 \mapsto \langle n_3 \rangle] \ \$ \ 3}$$

Therefore, $\sigma' = \sigma[s_0 \mapsto \langle n_3 \rangle]$.
Now we can take $\mathcal{R} = \overline{n_3 \triangleright_{\mathbf{int}} \sigma'(s_0)}$, and it is clear that $W = 3 = |v_1| + |v_2| + |v|$ and C can be any number ≥ 1.

- Case $\phi = \mathbf{iota}$.
We have

$$\mathcal{T} = \overline{\mathbf{iota} : (\mathbf{int}) \to \{\mathbf{int}\}}$$
$$\mathcal{E} = \overline{\mathbf{iota}(n) \downarrow \{0, 1, ..., n-1\}}$$

where $n \geq 0$, and

$$\mathcal{C} = \overline{\mathbf{iota}(s) \Rightarrow_{s_4}^{s_0} (p, (s_3, s_0))}$$

where

$$\begin{aligned} s_{i+1} &= s_i + 1, \forall i \in \{0, ..., 3\} \\ p = (&s_0 := \texttt{ToFlags}(s); \\ &s_1 := \texttt{Usum}(s_0); \\ &\{s_2\} := \texttt{WithCtrl}(s_1, \emptyset, s_2 := \texttt{Const}_1()); \\ &s_3 := \texttt{ScanPlus}_0(s_0, s_2)) \end{aligned}$$

So $k = 1, v_1 = n, \tau = \{\mathbf{int}\}$ and $v = \{0, 1, ..., n-1\}$.
From (iv): $\overline{n \triangleright_{\mathbf{int}} \sigma(s)}$, which implies $\sigma(s) = \langle n \rangle$.
Let $p = p_0; (p_1; (p_2; p_3))$. Then using P-SEQ 3 times, we construct $\mathcal{P}$ as follows:

$$\dfrac{\mathcal{P}_0 \quad \dfrac{\mathcal{P}_1 \quad \dfrac{\mathcal{P}_2 \quad \mathcal{P}_3}{\langle p_2; p_3, \sigma_2 \rangle \Downarrow^{\langle()\rangle} \sigma' \ \$ \ W_2 + W_3}}{\langle p_1; (p_2; p_3), \sigma_1 \rangle \Downarrow^{\langle()\rangle} \sigma' \ \$ \ W_1 + W_2 + W_3}}{\langle p_0; (p_1; (p_2; p_3)), \sigma \rangle \Downarrow^{\langle()\rangle} \sigma' \ \$ \ W_0 + W_1 + W_2 + W_3}$$

For $p_0 = s_0 := \texttt{ToFlags}(s)$, with $\sigma(s) = \langle n \rangle$, we can build $\mathcal{P}_0$ as follows:

$$\dfrac{\dfrac{\text{by P-X-TOFLAGS } \overline{\texttt{ToFlags}(\langle n \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle} \quad \text{by P-X-TERMI } \overline{\texttt{ToFlags}(\langle\rangle) \Downarrow^{\langle\rangle} \langle\rangle}}{\text{by P-X-LOOP } \texttt{ToFlags}(\langle n \rangle) \Downarrow^{\langle()\rangle} \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle}}{\text{by P-XDUCER } \langle p_0, \sigma \rangle \Downarrow^{\langle()\rangle} \sigma[s_0 \mapsto \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle] \ \$ \ 1 + n + 1}$$

So $\sigma_0 = \sigma[s_0 \mapsto \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle]$ and $W_0 = n + 2$.

Similarly, for $p_1 = s_1 := \texttt{Usum}(s_0)$, we can build $\mathcal{P}_1$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{Usum}(\langle \texttt{T} \rangle) \downarrow \langle\rangle}\ \text{by P-X-UsumT}}{\vdots}}{\texttt{Usum}(\langle \texttt{F}_2, ..., \texttt{F}_n, \texttt{T} \rangle) \downarrow \langle ()_2, ..., ()_n \rangle}\ \text{by P-X-UsumF} \quad \dfrac{}{\texttt{Usum}(\langle\rangle) \Downarrow^{\langle\rangle} \langle\rangle}\ \text{by P-X-Termi}}{\texttt{Usum}(\langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle) \downarrow \langle ()_1, ..., ()_n \rangle}\ \text{by P-X-Loop}$$

$$\dfrac{\texttt{Usum}(\langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle) \Downarrow^{\langle()\rangle} \langle ()_1, ..., ()_n \rangle}{\langle p_1, \sigma_0 \rangle \Downarrow^{\langle()\rangle} \sigma_0[s_1 \mapsto \langle ()_1, ..., ()_n \rangle]\ \$\ n + 1 + n}\ \text{by P-Xducer}$$

So $\sigma_1 = \sigma[s_0 \mapsto \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle, s_1 \mapsto \langle ()_1, ..., ()_n \rangle]$, and $W_1 = 2n + 1$.

Now we build $\mathcal{P}_2$ for $p_2 = \{s_2\} := \texttt{WithCtrl}(s_1, \emptyset, s_2 := \texttt{Const}_1())$. There are two possibilities:

- Subcase $n = 0$, then $\sigma_1(s_1) = \langle\rangle$, so we use P-Wc-Emp to build $\mathcal{P}_2$ as follows:
$$\dfrac{}{\langle p_2, \sigma_1 \rangle \Downarrow^{\langle()\rangle} \sigma_1[s_2 \mapsto \langle\rangle]\ \$\ 1}$$
thus $\sigma_2 = \sigma[s_0 \mapsto \langle \texttt{T} \rangle, s_1 \mapsto \langle\rangle, s_2 \mapsto \langle\rangle]$, and $W_2 = 1$.
- Subcase $n > 0$, then $\sigma_1(s_1) = \langle ()_1, ..., ()_n \rangle \neq \langle\rangle$. It is easy to show that we can build $\mathcal{P}_2$ ending with using the rule P-Wc-Nonemp:
$$\dfrac{\langle s_2 := \texttt{Const}_1(), \sigma_1 \rangle \Downarrow^{\langle()_1, ..., ()_n\rangle} \sigma_1[s_2 \mapsto \langle 1_1, ..., 1_n \rangle]\ \$\ n}{\langle p_2, \sigma_1 \rangle \Downarrow^{\langle()\rangle} \sigma_1[s_2 \mapsto \langle 1_1, ..., 1_n \rangle]\ \$\ n + 1}$$
So in this subcase, $\sigma_2 = \sigma[s_0 \mapsto \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle, s_1 \mapsto \langle ()_1, ..., ()_n \rangle, s_2 \mapsto \langle 1_1, ..., 1_n \rangle]$, and $W_2 = n + 1$.

For $p_3 = s_3 := \texttt{ScanPlus}_0(s_0, s_2)$, it is easy to show that

$$\langle p_3, \sigma_2 \rangle \Downarrow^{\langle()\rangle} \sigma_2[s_3 \mapsto \langle 0, ..., n - 1 \rangle]\ \$\ n + 1 + n + n$$

thus $\sigma' = \sigma[s_0 \mapsto \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle, s_1 \mapsto \langle ()_1, ..., ()_n \rangle, s_2 \mapsto \langle 1_1, ..., 1_n \rangle, s_3 \mapsto \langle 0, ..., n-1 \rangle]$ and $W_3 = 3n + 1$.

Therefore, with $\sigma'^*((s_3, s_0)) = (\langle 0, ..., n-1 \rangle, \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle)$, we can build

$$\mathcal{R} = \dfrac{\dfrac{}{0 \triangleright_{\textbf{int}} \langle 0 \rangle}\quad ...\quad \dfrac{}{n-1 \triangleright_{\textbf{int}} \langle n-1 \rangle}}{\{0, ..., n-1\} \triangleright_{\{\textbf{int}\}} (\langle 0, ..., n-1 \rangle, \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle)}$$

Since $W = W_0 + W_1 + W_2 + W_3 = 6n + 3 + W_2$, and $|v_1| + |v| = n + 1$, for $n = 0$, we have $W_2 = 1$, so $W = 4$, and C can be any number ≥ 4; for $n \neq 0$, $W_2 = n + 1$, so $W = 7n + 4$, thus $C \geq 7$.

■

Theorem 3.27 (**Correctness for expressions**). *For some constant C, if*

(i) $\Gamma \vdash e : \tau$ *(by some derivation $\mathcal{T}$)*

(ii) $\rho \vdash e \downarrow v\ \$\ W^H$ *(by some $\mathcal{E}$)*

(iii) $\delta \vdash e \Rightarrow_{s_1}^{s_0} (p, st)$ *(by some $\mathcal{C}$)*

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

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

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

then, for some σ' *and* W^L*,*

(vii) $\langle p, \sigma\rangle \Downarrow^{\langle()\rangle} \sigma'\ \$\ W^L$ *(by some derivation* $\mathcal{P}$*)*

(viii) $v \rhd_\tau \sigma'^*(st)$ *(by some* $\mathcal{R}$*)*

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

Proof. The proof is by induction on the syntax of e. We will show that we can take $C = 7$ here, which will satisfy all the proof cases.

- Case $e = x$.
  We must have
  $$\mathcal{T} = \frac{}{\Gamma\ \vdash\ x : \tau}\,(\Gamma(x) = \tau)$$
  $$\mathcal{E} = \frac{}{\rho\ \vdash\ x \downarrow v\ \$\ 0}\,(\rho(x) = v)$$
  $$\mathcal{C} = \frac{}{\delta\ \vdash\ x \Rightarrow^{s_0}_{s_0} (\epsilon, st)}\,(\delta(x) = st)$$
  So $p = \epsilon$.
  Immediately we have $\mathcal{P} = \dfrac{}{\langle \epsilon, \sigma\rangle \Downarrow^{\langle()\rangle} \sigma\ \$\ 0}$
  From the assumptions $(iv), (v)$ and (vi) we already have $v \rhd_\tau \sigma^*(st)$. Finally it's clear that $W^L = W^H = 0$, so C can be any number.

- Case $e = \textbf{let}\ x = e_1\ \textbf{in}\ e_2$.
  We must have:
  $$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma\ \vdash\ e_1 : \tau_1} \qquad \overset{\mathcal{T}_2}{\Gamma[x \mapsto \tau_1]\ \vdash\ e_2 : \tau}}{\Gamma\ \vdash\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$
  $$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\rho\ \vdash\ e_1 \downarrow v_1\ \$\ W_1^H} \qquad \overset{\mathcal{E}_2}{\rho[x \mapsto v_1]\ \vdash\ e_2 \downarrow v\ \$\ W_2^H}}{\rho\ \vdash\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \downarrow v\ \$\ W_1^H + W_2^H}$$
  $$\mathcal{C} = \frac{\overset{\mathcal{C}_1}{\delta\ \vdash\ e_1 \Rightarrow^{s_0}_{s_0'} (p_1, st_1)} \qquad \overset{\mathcal{C}_2}{\delta[x \mapsto st_1]\ \vdash\ e_2 \Rightarrow^{s_0'}_{s_1} (p_2, st)}}{\delta\ \vdash\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow^{s_0}_{s_1} (p_1; p_2, st)}$$
  So $p = p_1; p_2$.
  By IH on e_1 with $\mathcal{T}_1$, $\mathcal{E}_1, \mathcal{C}_1$, we have
  (a) $\mathcal{P}_1$ of $\langle p_1, \sigma\rangle \Downarrow^{\langle()\rangle} \sigma_1\ \$\ W_1^L$
  (b) $\mathcal{R}_1$ of $v_1 \rhd_{\tau_1} \sigma_1^*(st_1)$
  (c) $W_1^L \leq C \cdot W_1^H$

From (b), we must have $\rho[x \mapsto v_1](x) : \Gamma[x \mapsto \tau_1](x)$, and $\rho[x \mapsto v_1](x) \triangleright_{\Gamma[x \mapsto \tau_1](x)} \overline{\overline{\sigma_1^*(\delta[x \mapsto st_1](x))}}$ must hold. By Theorem 3.14 on $\mathcal{C}_1$, we get $\overline{\overline{st_1}} \lessdot s_0'$, hence $\overline{\overline{\delta[x \mapsto st_1](x)}} \lessdot s_0'$.

Then by IH on e_2 with $\mathcal{T}_2$, $\mathcal{E}_2, \mathcal{C}_2$, we get

(d) $\mathcal{P}_2$ of $\langle p_2, \sigma_1 \rangle \Downarrow^{\langle () \rangle} \sigma_2 \$ W_2^L$
(e) $\mathcal{R}_2$ of $v \triangleright_\tau \sigma_2^*(st)$
(f) $W_2^L \leq C \cdot W_2^H$

So using P-Seq we can construct:

$$\mathcal{P} = \frac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2 \\ \langle p_1, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma_1 \$ W_1^L & \langle p_2, \sigma_1 \rangle \Downarrow^{\langle () \rangle} \sigma_2 \$ W_2^L\end{array}}{\langle p_1; p_2, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma_2 \$ W_1^L + W_2^L}$$

Take $\sigma' = \sigma_2$ (thus $\mathcal{R} = \mathcal{R}_2$), and we have $W^L = W_1^L + W_2^L \leq C(W_1^H + W_2^H) = C \cdot W^H$, as required.

- Case $e = \phi(x_1, ..., x_k)$
  We must have

$$\mathcal{T} = \frac{\mathcal{T}_1}{\frac{\phi : (\tau_1, ..., \tau_k) \to \tau}{\Gamma \vdash \phi(x_1, ..., x_k) : \tau} ((\Gamma(x_i) = \tau_i)_{i=1}^k)}$$

$$\mathcal{E} = \frac{\mathcal{E}_1}{\frac{\phi(v_1, ..., v_k) \downarrow v}{\rho \vdash \phi(x_1, ..., x_k) \downarrow v \$ (\sum_{i=1}^k |v_i| + |v|)} ((\rho(x_i) = v_i)_{i=1}^k)}$$

$$\mathcal{C} = \frac{\mathcal{C}_1}{\frac{\phi(st_1, ..., st_k) \Rightarrow_{s_1}^{s_0} (p, st)}{\delta \vdash \phi(x_1, ..., x_k) \Rightarrow_{s_1}^{s_0} (p, st)} ((\delta(x_i) = st_i)_{i=1}^k)}$$

  From the assumptions (iv) and (vi), for all $i \in \{1, ..., k\}$:

  (a) $\vdash \rho(x_i) : \Gamma(x_i)$, that is, $\vdash v_i : \tau_i$
  (b) $\rho(x_i) \triangleright_{\Gamma(x_i)} \sigma^*(st_i)$, that is, $v_i \triangleright_{\tau_i} \sigma^*(st_i)$

  Since $W^H = |v| + \sum_{i=1}^k |v_i|$, using Lemma 3.26 on $\mathcal{T}_1$ with $\mathcal{E}_1, \mathcal{C}_1, (a), (b)$ will give us exactly what we shall show.

- Case $e = \{e_1 : x \textbf{ in } y \textbf{ using } x_1, ..., x_k\}$.

  From the corresponding assumptions, we have:

  (i)

$$\mathcal{T} = \frac{\mathcal{T}_1}{\frac{[x \mapsto \tau_1, x_1 \mapsto \textbf{int}, ..., x_k \mapsto \textbf{int}] \vdash e_1 : \tau_2}{\Gamma \vdash \{e_1 : x \textbf{ in } y \textbf{ using } x_1, ..., x_k\} : \{\tau_2\}}}$$

  with

$$\Gamma(y) = \{\tau_1\}$$
$$(\Gamma(x_i) = \textbf{int})_{i=1}^k$$

(ii)

$$\mathcal{E} = \frac{\left(\begin{array}{c}\mathcal{E}_i \\ [x \mapsto v_i, x_1 \mapsto n_1, ..., x_k \mapsto n_k] \ \vdash \ e_1 \downarrow v_i' \ \$ \ W_i^H\end{array}\right)_{i=1}^{l}}{\rho \ \vdash \ \{e_1 : x \ \textbf{in} \ y \ \textbf{using} \ x_1, ..., x_k\} \downarrow \{v_1', ..., v_l'\} \ \$ \ W^H}$$

with

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

(iii)

$$\mathcal{C} = \frac{\begin{array}{c}\mathcal{C}_1 \\ [x \mapsto st_1, x_1 \mapsto s_1', ..., x_k \mapsto s_k'] \ \vdash \ e_1 \Rightarrow_{s_1''}^{s_k'+1} (p_1, st_2)\end{array}}{\delta \ \vdash \ \{e_1 : x \ \textbf{in} \ y \ \textbf{using} \ x_1, ..., x_k\} \Rightarrow_{s_1''}^{s_0'} (p, (st_2, s_b))}$$

with

$$\begin{aligned}
\delta(y) &= (st_1, s_b) \\
(\delta(x_i) &= s_i)_{i=1}^{k} \\
p &= (\ s_0' := \texttt{Usum}(s_b); \\
&\quad\ s_1' := \texttt{Distr}(s_b, s_1); \\
&\quad\ \vdots \\
&\quad\ s_k' := \texttt{Distr}(s_b, s_k); \\
&\quad\ S_{out} := \texttt{WithCtrl}(s_0', S_{in}, p_1)) \\
S_{in} &= \overline{\overline{st_1}} \cup \{s_1', ..., s_k'\} \\
S_{out} &= \{s \mid s \in \overline{\overline{st_2}}, s \geq s_k' + 1\} \\
s_{i+1}' &= s_i' + 1, \forall i \in \{0, ..., k-1\}
\end{aligned}$$

So $\tau = \{\tau_2\}, v = \{v_1', ..., v_l'\}, st = (st_2, s_b)$.

(iv) $\vdash \ \rho(y) : \Gamma(y)$ gives us $\vdash \ \{v_1, ..., v_l\} : \{\tau_1\}$, which must have the derivation:

$$\frac{(\ \vdash \ v_i : \tau_1)_{i=1}^{l}}{\vdash \ \{v_1, ..., v_l\} : \{\tau_1\}} \tag{3.41}$$

and clearly for $\forall i \in \{1, ..., k\}$. $\vdash \ \rho(x_i) : \Gamma(x_i)$, that is

$$(\overline{\vdash \ n_i : \textbf{int}})_{i=1}^{k} \tag{3.42}$$

(v) $\overline{\overline{\delta(y)}} \lessdot s_0$ gives us

$$\overline{\overline{\delta(y)}} = \overline{\overline{(st_1, s_b)}} = \overline{\overline{st_1}} \cup \{s_b\} \lessdot s_0' \tag{3.43}$$

and $(\overline{\overline{\delta(x_i)}})_{i=1}^{k} \lessdot s_0'$ implies $\{s_1, ..., s_j\} \lessdot s_0'$.

(vi) Since $\rho(y) \rhd_{\Gamma(y)} \sigma^*(\delta(y)) = \{v_1, ..., v_l\} \rhd_{\{\tau_1\}} \sigma^*((st_1, s_b))$, which must have the derivation:

$$\frac{\begin{pmatrix} \mathcal{R}_i \\ v_i \rhd_{\tau_1} w_i \end{pmatrix}_{i=1}^{l}}{\{v_1, ..., v_l\} \rhd_{\{\tau_1\}} (w, \langle \mathtt{F}_1, ..., \mathtt{F}_l, \mathtt{T} \rangle)} \tag{3.44}$$

where $w = w_1 +\!\!+_{\tau_1} \cdots +\!\!+_{\tau_1} w_l$, therefore we have

$$\sigma^*(st_1) = w \tag{3.45}$$

$$\sigma(s_b) = \langle \mathtt{F}_1, ..., \mathtt{F}_l, \mathtt{T} \rangle. \tag{3.46}$$

Also, for $\forall i \in \{1, ..., k\}$. $\rho(x_i) \rhd_{\Gamma(x_i)} \sigma(\delta(x_i))$ must have the derivation $\dfrac{}{n_i \rhd_{\mathbf{int}} \sigma(s_i)}$, which implies

$$(\sigma(s_i) = \langle n_i \rangle)_{i=1}^{k} \tag{3.47}$$

First we shall show (vii): $\langle p, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma' \$ W^L$ by some $\mathcal{P}$.

Let

$$\begin{aligned} p_{\mathrm{u}} = s'_0 &:= \mathtt{Usum}(s_b) \\ p_{\mathrm{d}1} = s'_1 &:= \mathtt{Distr}(s_b, s_1) \\ &\vdots \\ p_{\mathrm{d}k} = s'_k &:= \mathtt{Distr}(s_b, s_k) \\ p_{\mathrm{w}} = S_{out} &:= \mathtt{WithCtrl}(s'_0, S_{in}, p_1) \end{aligned}$$

That is, $p = p_{\mathrm{u}};\ p_{\mathrm{d}1};\ \cdots;\ p_{\mathrm{d}k};\ p_{\mathrm{w}}$.

Using P-SEQ $(k{+}1)$ times (which generates $(k{+}1)$ intermediate stores $\sigma_0, ..., \sigma_k$), we can build $\mathcal{P}$ as follows:

$$\dfrac{\mathcal{P}_{\mathrm{u}} \quad \langle p_{\mathrm{u}}, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma_0 \$ W_{\mathrm{u}}^L \qquad \dfrac{\mathcal{P}_{\mathrm{d}1} \quad \langle p_{\mathrm{d}1}, \sigma_0 \rangle \Downarrow^{\langle () \rangle} \sigma_1 \$ W_{\mathrm{d}1}^L \qquad \dfrac{\mathcal{P}_{\mathrm{d}k} \quad \langle p_{\mathrm{d}k}, \sigma_{k-1} \rangle \Downarrow^{\langle () \rangle} \sigma_k \$ W_{\mathrm{d}k}^L \qquad \mathcal{P}_{\mathrm{w}} \quad \langle p_{\mathrm{w}}, \sigma_k \rangle \Downarrow^{\langle () \rangle} \sigma' \$ W_{\mathrm{w}}^L}{\vdots \quad \left\langle \begin{matrix} (p_{\mathrm{d}i};)_{i=2}^{k}, \\ p_{\mathrm{w}} \end{matrix}, \sigma_1 \right\rangle \Downarrow^{\langle () \rangle} \sigma' \$ W_{\mathrm{w}}^L + \sum_{i=2}^{k} W_{\mathrm{d}i}^L}}{\left\langle \begin{matrix} (p_{\mathrm{d}i};)_{i=1}^{k}, \\ p_{\mathrm{w}} \end{matrix}, \sigma_0 \right\rangle \Downarrow^{\langle () \rangle} \sigma' \$ W_{\mathrm{w}}^L + \sum_{i=1}^{k} W_{\mathrm{d}i}^L}}{\left\langle \begin{matrix} p_{\mathrm{u}}; \\ (p_{\mathrm{d}i};)_{i=1}^{k}, \\ p_{\mathrm{w}} \end{matrix}, \sigma \right\rangle \Downarrow^{\langle () \rangle} \sigma' \$ W^L}$$

thus

$$W^L = W_{\mathrm{u}}^L + (\sum_{i=1}^{k} W_{\mathrm{d}i}^L) + W_{\mathrm{w}}^L$$

It is easy to show that, (as we have presented some analogous cases in the proof of Lemma 3.26,) with $\sigma(s_b) = \langle \mathtt{F}_1, ..., \mathtt{F}_l, \mathtt{T} \rangle$, we can build $\mathcal{P}_{\mathrm{u}}$ ending in

$$\langle s'_0 := \mathtt{Usum}(s_b), \sigma \rangle \Downarrow^{\langle () \rangle} \sigma[s'_0 \mapsto \langle ()_1, ..., ()_l \rangle] \$ \ 2l + 1$$

and, with $(\sigma(s_i) = \langle n_i \rangle)_{i=1}^{k}$ from (3.47), $\mathcal{P}_{\text{di}}$ ending in

$$\langle s'_i := \texttt{Distr}(s_b, s_i), \sigma_{i-1} \rangle \Downarrow^{\langle () \rangle} \sigma_{i-1}[s_i \mapsto \langle \overbrace{n_i, ..., n_i}^{l} \rangle] \ \$ \ 2l + 2$$

Thus $\sigma_k = \sigma[s'_0 \mapsto \langle ()_1, ..., ()_l \rangle, s'_1 \mapsto \langle \overbrace{n_1, ..., n_1}^{l} \rangle, ..., s'_k \mapsto \langle \overbrace{n_k, ..., n_k}^{l} \rangle]$, and $W^L_{\text{u}} + (\sum_{i=1}^{k} W^L_{\text{di}}) = (2l + 1) + k \cdot (2l + 2) = 2(k + 1) \cdot l + 1 + 2k$.

Now it remains to build $\mathcal{P}_{\text{w}}$.
Assume $S_{out} = \{s'_{k+1}, ..., s'_{k+k'}\}$. There are two possibilities for $\sigma_k(s'_0)$, for each of which we will build a $\mathcal{P}_{\text{w}}$ and show (viii) and (ix).

– Subcase $\sigma_k(s'_0) = \langle \rangle$, i.e., $l = 0$.
Then $(\sigma_k(s'_i) = \langle \rangle)_{i=1}^{k}$. Also, with (3.44) and (3.45), we have $\sigma^*_k(st_1) = \sigma^*(st_1) = \langle \rangle_{\tau_1}$; with (3.46), $\sigma_k(s_b) = \langle \texttt{T} \rangle$. Thus

$$\forall s \in (\{s'_0\} \cup S_{in}).\sigma_k(s) = \langle \rangle$$

Then we can use the rule P-Wc-Emp to build $\mathcal{P}_{\text{w}}$ as follows:

$$\frac{}{\langle S_{out} := \texttt{WithCtrl}(s'_0, S_{in}, p_1), \sigma_k \rangle \Downarrow^{\langle () \rangle} \sigma_k[s_{k+1} \mapsto \langle \rangle, ..., s_{k+k'} \mapsto \langle \rangle] \ \$ \ 1}$$

So in this subcase, we take

$$\sigma' = \sigma[s'_0 \mapsto \langle \rangle, s'_1 \mapsto \langle \rangle, ..., s'_k \mapsto \langle \rangle, s'_{k+1} \mapsto \langle \rangle, ..., s'_{k+k'} \mapsto \langle \rangle] \tag{3.48}$$

and $W^L_{\text{w}} = 1$.

Therefore, $\sigma'^*(st) = \sigma'^*((st_2, s_b)) = (\sigma'^*(st_2), \sigma'(s_b)) = (\langle \rangle_{\tau_2}, \langle \texttt{T} \rangle)$, with which we construct

$$\mathcal{R} = \frac{}{\{\} \rhd_{\{\tau_2\}} (\langle \rangle_{\tau_2}, \langle \texttt{T} \rangle)}$$

Finally, $W^L = (2l + 1) + k \cdot (2l + 2) + W^L_{\text{w}} = 2 + 2k$, and $W^H = (l + 1) \cdot (1 + k) + \sum_{i=1}^{l} W^H_i = 1 + k$, thus C can be any number ≥ 2.

– Subcase $\sigma_k(s_0) \neq \langle \rangle$, i.e., $l > 0$.
Since we have $\mathcal{T}_1$, $(\mathcal{E}_i)_{i=1}^{l}$, and $\mathcal{C}_1$, let Γ_1, $(\rho_i)_{i=1}^{l}$, and δ_1 be the initial environment in these derivations respectively, i.e.,

$$\begin{aligned} \Gamma_1 &= [x \mapsto \tau_1, x_1 \mapsto \textbf{int}, ..., x_k \mapsto \textbf{int}] \\ \rho_i &= [x \mapsto v_i, x_1 \mapsto n_1, ..., x_k \mapsto n_k] \\ \delta_1 &= [x \mapsto st_1, x_1 \mapsto s'_1, ..., x_k \mapsto s'_k] \end{aligned}$$

From what we already shown (i.e., (3.41), (3.42)), (3.43)), it is clear that
(a) $\forall i \in \{1, ..., l\}. \forall x' \in dom(\Gamma_1). \ \vdash \ \rho_i(x') : \Gamma_1(x')$
(b) $\forall x' \in dom(\Gamma_1). \overline{\overline{\delta_1(x')}} \lessdot s'_k + 1$

Let $S = dom(\sigma_k) - S_{in}$. For $\forall i \in \{1, ..., l\}$, we take σ'_i as

$$\begin{aligned} dom(\sigma'_i) &= dom(\sigma_k) \\ \forall s \in S.\sigma'_i(s) &= \sigma_k(s) \\ \sigma'^{*}_i(st_1) &= w_i \\ \sigma'_i(s'_1) &= \langle n_1 \rangle \\ &\vdots \\ \sigma'_i(s'_k) &= \langle n_k \rangle \end{aligned}$$

Then it is eays to show that
(c) $\forall i \in \{1, ..., l\}.\forall x' \in dom(\Gamma_1).\rho_i(x') \rhd_{\Gamma_1(x')} \sigma'^{*}_i(\delta_1(x'))$
and

$$\sigma'_1 \overset{S}{\sim} \sigma'_2 \overset{S}{\sim} ... \overset{S}{\sim} \sigma'_l \overset{S}{\sim} \sigma_k \tag{3.49}$$

$$\sigma'_1 \overset{S}{\bowtie} \sigma'_2 \overset{S}{\bowtie} ... \overset{S}{\bowtie} \sigma'_l = (\overset{S}{\bowtie} \sigma'_i)^l_{i=1} = \sigma_k \tag{3.50}$$

By IH (l times) on e_1 with $\mathcal{T}_1$, $(\mathcal{E}_i)^l_{i=1}$, $\mathcal{C}_1, (a), (b), (c)$, we obtain:

$$(\langle p_1, \sigma'_i \rangle \Downarrow^{\langle () \rangle} \sigma''_i \ \$ \ W_i^{L'})^l_{i=1} \tag{3.51}$$

$$(v'_i \rhd_{\tau_2} \sigma''^{*}_i(st_2))^l_{i=1} \tag{3.52}$$

$$(W_i^{L'} \leq C \cdot W_i^H)^l_{i=1} \tag{3.53}$$

By Theorem 3.14 on $\mathcal{C}_1$, we have

$$S_{in} \Vdash p_1 : S' \tag{3.54}$$

for some $S' \subseteq \{s'_k + 1, s'_k + 2, ..., s''_1 - 1\}$.
Since we have (3.54), (3.49), (3.51) and clearly $(S_{in} \cup S') \cap S = \emptyset$, it is easy to show that using Lemma 3.25 (the parallelism fusion lemma) $(l-1)$ times we can obtain a derivation $\mathcal{P}'_{\mathrm{w}}$ of

$$\left\langle p_1, (\overset{S}{\bowtie} \sigma'_i)^l_{i=1} \right\rangle \Downarrow^{\langle ()_1, ..., ()_l \rangle} (\overset{S}{\bowtie} \sigma''_i)^l_{i=1} \ \$ \ W_{\mathrm{w}}^{L'} \tag{3.55}$$

and, together with (3.53),

$$W_{\mathrm{w}}^{L'} \leq \sum_{i=1}^{l} W_i^{L'} \leq C \cdot \sum_{i=1}^{l} W_i^H$$

Let $\sigma'' = (\overset{S}{\bowtie} \sigma''_i)^l_{i=1}$, then (with (3.50)), we have $\mathcal{P}'_{\mathrm{w}}$ as:

$$\langle p_1, \sigma_k \rangle \Downarrow^{\langle ()_1, ..., ()_l \rangle} \sigma'' \ \$ \ W_{\mathrm{w}}^{L'}$$

Now we build $\mathcal{P}_{\mathrm{w}}$ using the rule P-Wc-Nonemp as follows:

$$\frac{\begin{array}{c} \mathcal{P}'_{\mathrm{w}} \\ \langle p_1, \sigma_k \rangle \Downarrow^{\langle ()_1, ..., ()_l \rangle} \sigma'' \ \$ \ W_{\mathrm{w}}^{L'} \end{array}}{\langle p_{\mathrm{w}}, \sigma_k \rangle \Downarrow^{\langle () \rangle} \sigma_k[s'_{k+1} \mapsto \sigma''(s'_{k+1}), ..., s'_{k+k'} \mapsto \sigma''(s'_{k+k'})] \ \$ \ 1 + W_{\mathrm{w}}^{L'}}$$

So in this subcase we take

$$\sigma' = \sigma[s'_0 \mapsto \langle ()_1, ..., ()_l \rangle, s'_1 \mapsto \langle \overbrace{n_1, ..., n_1}^{l} \rangle, ..., s'_k \mapsto \langle \overbrace{n_k, ..., n_k}^{l} \rangle, \\ s'_{k+1} \mapsto \sigma''(s'_{k+1}), ..., s'_{k+k'} \mapsto \sigma''(s'_{k+k'})] \tag{3.56}$$

and $W^L_w = 1 + W^{L'}_w \leq 1 + C \cdot \sum_{i=1}^{l} W^H_i$.

Let $\sigma'^*(st_2) = w'$, and $\sigma''^*_i(st_2) = w'_i$ for $\forall i \in \{1, ..., l\}$.
Then it is easy to show:

$$w' = \sigma''^*(st_2) = w'_1 +\!\!+_{\tau_2} \cdots +\!\!+_{\tau_2} w'_l$$

So we now have $\sigma'^*(st) = \sigma'^*((st_2, s_b)) = (w', \langle \mathtt{F}_1, ..., \mathtt{F}_l, \mathtt{T} \rangle)$. With (3.52), we can construct $\mathcal{R}$ as follows:

$$\frac{(v'_i \rhd_{\tau_2} w'_i)^l_{i=1}}{\{v'_1, ..., v'_l\} \rhd_{\{\tau_2\}} (w', \langle \mathtt{F}_1, ..., \mathtt{F}_l, \mathtt{T} \rangle)}$$

as required.
Finally, in this subcase

$$\begin{aligned} W^L &= (2l+1) + k \cdot (2l+2) + W^L_{\mathrm{w}} \\ &\leq (2l+1) + k \cdot (2l+2) + 1 + C \cdot \sum_{i=1}^{l} W^H_i \\ &= 2(l+1) \cdot (1+k) + C \cdot \sum_{i=1}^{l} W^H_i \end{aligned}$$

$$W^H = (l+1) \cdot (1+k) + \sum_{i=1}^{l} W^H_i$$

therefore, C can be any number ≥ 2.

■

Corollary 3.28 (**Correctness of implementation**). *For some constant C, if*

(i) $[\,] \vdash e : \tau$

(ii) $[\,] \vdash e \downarrow v \ \$ \ W^H$

then

(iii) $\exists!\ p, st, s.\ [\,] \vdash e \Rightarrow^0_s (p, st)$

(iv) $\exists!\ \sigma, W^L.\ \langle p, [\,] \rangle \Downarrow^{\langle () \rangle} \sigma \ \$ \ W^L$

(v) $\exists!\ v'.\ \sigma^*(st) \lhd_\tau v', \langle \rangle_\tau$

(vi) $v' = v$ *and* $W^L \leq C \cdot W^H$

Proof. The proof will use Theorem 3.27, Theorem 3.11, Lemma 3.16 and Lemma 3.17.

■

3.6 Scaling up

We have presented the formal proof of the correctness of SNESL_0. It is a tiny language, compared to the full SNESL, or even SNESL_1. However, it maintains the most important properties and the core semantics of full SNESL. In particular, it includes the general comprehension, the expression for expressing nested data-parallelism in SNESL.

The proofs of the most important lemmas and theorems shown in this chapter, such as the block self-delimiting lemma, the parallelism fusion lemma, the determinism theorem and the correctness theorem of the translation, have shed light on the extension of the formal validation of full SNESL. For scaling up, we need to consider at least the following points:

- Extension to more primitive types and associative scalar operations should be trivial.
- Adding pairs should mainly increase one more rule for value representation. Since the low-level stream trees are compatible with pairs, the effect at the low-level would be almost transparent. However, the translation of the function **zip** (for generating pairs) will need to include code checking that the segment descriptors of the two sequence arguments are identical.
- Error preservation is also a desirable part that can be easily supported in principle. We would expect the runtime error can be simulated accurately from the high level to the low level.
- Adding more built-in functions are basically to add more Xducers to the target language, as most of them are implemented by only one counterpart low-level Xducer, such as `ScanPlus` for **scan** and `SegConcat` for **concat**, or a few lines. The two-level semantics of Xducers (the general level and the block level) has reduced much of the work to formalize the semantics of a new Xducer. In addition, a non-trivial built-in function **iota**, implemented with a `WithCtrl` instruction, is already given in SNESL_0 as a representative example.
- For the restricted comprehension, its implementation in SVCODE is simpler than the general one, because it does not include variable-bindings. However, the type-dependent `pack` may need some consideration to deal with.
- Extension to step and space cost should be similar to the work cost.
- With the `SCall` instruction being added to the target language, it will be simple to support recursion. However, we will also need to prove the preservation of termination to show that the translated SVCODE program from a terminating SNESL one will terminate as well. This should be provable with a generalized IH using induction on the high-level evaluation derivation.
- The real challenge may be to scale up to streaming semantics. At this point, there are still some problems about streaming left open, such as streamability and deadlock, so it is not that meaningful to give a precise prediction about how they will affect the proof system.

Chapter 4

Conclusion

Based on the practical experimentation that has demonstrated good performance and time and space efficiency from previous work of SNESL [Mad16], this thesis has moved the development of SNESL one step forward.

The main contributions of this thesis are:

- Extension of the dataflow model of streaming to account for recursion.
  The challenge of supporting recursion is that it can cause an infinite increase of the dataflow network in the execution model. Our solution is to extend the target language to make sure that the dataflow graph will be completed dynamically during execution, but not grow infinitely, if all recursions are guarded by conditionals (i.e., comprehensions in SNESL). We have developed the implementation as three instrumented interpreters, the high-level one for SNESL_1, the eager one with sufficiently large space and the streaming one with a limited buffer for SVCODE, to compare their results and costs. Although without a formal proof, we have demonstrated result and cost preservation in our experiments, and provided some representative examples. The space usage from our experiment results also shows an increase proportional to the depth of the recursion.

- A formalization of the source and target languages, and the correctness of the translation including work cost preservation.
  Formal semantics for the high-level NDP language can be found from previous related research, such as NESL and Proteus. However, none of them has given a formal semantics of the target language. This thesis has presented a formalization of the target language for a core subset of SNESL, and also given a detailed proof for the correctness of translation and work cost preservation, as well as some other important properties of the language including well-formedness and determinism. The work we have done in this thesis should lay out the possibility for a formal validation of the translation correctness and cost preservation for full SNESL.

While investigating the solution to recursion, we have also touched some crucial, open problems in this streaming language, such as streamability, scheduling, and deadlocks. We have explored a possibility of scheduling that can be as efficient and lightweight as the one used in the previous work of SNESL, but also preserves the step cost.

The future work on SNESL related to the scope of this thesis can fall into the following two points.

- Formalization of the streaming semantics of the target language. The formalization work we have done in this thesis is only for its eager semantics, while the streaming semantics is also an area that has not been covered much yet.
- Schedulability. As a streaming language aiming at both time and space efficiency, SNESL should be equipped with a type system or some static/dynamic analysis that can prevent a measure of problematic programs. The user may expect that, at least for a streamable program, deadlock will not happen. This will require a good high-level characterization of streamability, and a formal demonstration that all streamable programs (in above sense) indeed execute without deadlock.

Bibliography

[BG96] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. ACM.

[BHC+93] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993. ACM.

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

[Ble90] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[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.

[BS90] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, February 1990.

[CLPJ+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[LCK+12] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. Work efficient higher-order vectorisation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 259–270, New York, NY, USA, 2012. ACM.

[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.

[MF13] Frederik M. Madsen and Andrzej Filinski. Towards a streaming model for nested data parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 13–24, New York, NY, USA, 2013. ACM.

[MF16] Frederik M. Madsen and Andrzej Filinski. Streaming nested data parallelism on multicores. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, FHPC 2016, pages 44–51, New York, NY, USA, 2016. ACM.

[PJ08] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[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.

[PPCF96] Daniel W. Palmer, Jan Prins, Siddhartha Chatterjee, and Rickard E. Faith. Piecewise execution of nested data-parallel programs. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '95, pages 346–361, London, UK, UK, 1996. Springer-Verlag.

[PPW95] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '95, pages 186–193, Washington, DC, USA, 1995. IEEE Computer Society.