Formalizing the implementation of Streaming NESL

Dandan Xue

October 10, 2017

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Nested data parallelism

1.2 NESL

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

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

which does the same computation as the Haskell expression

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

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

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

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

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

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

1.2.1 Work-depth cost model

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

1.3 SNESL

Streaming NESL (SNESL) [Mad16] is a refinement of NESL that attempts to improve the efficiency of space usage. It extends NESL with two features: streaming semantics

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

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

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

and a cost model for space usage. The basic idea behind the streaming semantics may be described as: data-parallelism can be realized not only in terms of space, as NESL has demonstrated, but also, for some restricted cases, in terms of time. When there is no enough space to store all the data at the same time, computing them chunk by chunk may be a way out.

1.3.1 Type system

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

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

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

The novel extension is the *streamable* types σ, which generalizes the types of data that are not necessarily completely materialized at once, but rather in a streaming fashion. In particular, the type $\{\sigma\}$, called *sequences* in SNESL, represents collections of data computed in terms of time. So, even with a small size of memory, SNESL could execute programs which is impossible in NESL due to space limitation or more space efficiently than in NESL.

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

1.3.2 Values and expressions

The values of SNESL are as follows:

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

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

The expressions of SNESL are shown in Figure 1.2.

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

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 in NESL. Perhaps for easier experimental analysis, the apply-to-each construct is splitted to the general and the restricted comprehensions, both added an explicit list of the free variables (listed after the keyword **using**) occur in the body e_1, and the restricted one is the only expression that can work as a conditional itself in SNESL (???). These comprehensions also extended the semantics of the apply-to-each from evaluating to vectors (i.e., type $[\tau]$) to evaluating to sequences (i.e., type $\{\sigma\}$).

Another notable difference from NESL to SNESL occurs in the built-in functions. The primitive functions of SNESL is shown in Figure 1.3.

$$
\begin{array}{lll}
\phi \; ::= \oplus \mid \textbf{append} \mid \textbf{concat} \mid \textbf{zip} \mid \textbf{iota} \mid \textbf{part} \mid \textbf{scan}_{\otimes} \mid \textbf{reduce}_{\otimes} \mid \textbf{mkseq} & & (1.1) \\
\quad \mid \textbf{length} \mid \textbf{elt} & & (1.2) \\
\quad \mid \textbf{the} \mid \textbf{empty} & & (1.3) \\
\quad \mid \textbf{seq} \mid \textbf{tab} & & (1.4) \\
\oplus \; ::= + \mid - \mid \times \mid \div \mid \% \mid \leq \mid ... & \text{(consts operations)} & \\
\otimes \; ::= + \mid \times \mid ... & \text{(associative binary operations)} &
\end{array}
$$

Figure 1.3: SNESL primitive functions

The functions listed in (1.1) and (1.2) of Figure 1.3 are original supported in NESL, doing transformations on consts 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, thus most of these functions can be executed in a more space-efficient way. There will be more detailed descriptions of these functions in the next chapter.

Functions in (1.2) are kept their vector versions in SNESL to guarantee the efficiency, although it is possible to realize stream versions for them as well.

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

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

where ++ is a syntactic sugar of **append**. **empty**, which tests whether a sequence is empty or not, is much more efficient in a streaming setting as it takes only constant complexity both in time and space.

Finally, list (1.4) are functions performing conversions between concrete types and streams. **seq**, typed as $[\tau] \to \{\tau\}$, converts a vector to a sequence, and **tab** does the contrary work, tabulating the sequence into a vector.

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

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

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

1.3.3 Cost model

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

1.4 Mathematical background and notations

Chapter 2

Implementation

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

2.1 High-level interpreter

In this thesis, the high-level language we have experimented with is a subset of SNESL introduced in the last chapter but without vectors. We will call this language SNESL1. As our first goal is to extend SNESL with user-defined (recursive) functions, it is safe to do experiments only with SNESL1 because removing vectors from SNESL should not affect the complexity of the problem too much; we believe that if the solution works with streams, the general type in SNESL, it should be trivial to make it support vectors.

Tuples are also simplified as pairs in SNESL1. Then the type system for SNESL1 is as follows.

$$\varphi ::= \textbf{bool} \mid \textbf{int} \mid (\varphi_1, \varphi_2)$$
$$\tau ::= \varphi \mid (\tau_1, \tau_2) \mid \{\tau\}$$

The values in SNESL1:

$$a ::= \texttt{T} \mid \texttt{F} \mid n \mid (a_1, a_2)$$
$$v ::= a \mid (v_1, v_2) \mid \{v_1, ..., v_k\}$$

The grammar of the interpreter for SNESL1 is given in 2.1.

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

Figure 2.1: Grammar of high-level interpreter

The high-level interpreter we have implemented supports interpreting both SNESL1 expressions and function definitions. Since type inference is not incorporated in the interpreter, the types of parameters and return values need to be provided when the user defines a function.

The typing rules of SNES1 is given in Figure 2.2. The type environment Γ is a mapping from variables to types:

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

. (may add explaination of how to type user-defined functions)

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

$$\frac{}{\Gamma \vdash a : \varphi}\,(a : \varphi) \qquad \frac{}{\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau} \qquad \frac{\phi : (\tau_1, ..., \tau_k) \to \tau}{\Gamma \vdash \phi(x_1, ..., x_k) : \tau}\,((\Gamma(x_i) = \tau_i)_{i=1}^k)$$

$$\frac{[x \mapsto \tau_1, (x_i \mapsto \varphi_i)_{i=1}^j] \vdash e_1 : \tau}{\Gamma \vdash \{e_1 : x \textbf{ in } y \textbf{ using } x_1, ..., x_j\} : \{\tau\}}\,(\Gamma(y) = \{\tau_1\}, (\Gamma(x_i) = \varphi_i)_{i=1}^j)$$

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

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

Figure 2.2: Typing rules of SNESL1

Value typing rules:

Judgment $\boxed{a : \varphi}$

$$\frac{}{n : \textbf{int}} \qquad \frac{}{\texttt{T} : \textbf{bool}} \qquad \frac{}{\texttt{F} : \textbf{bool}} \qquad \frac{a_1 : \varphi_1 \qquad a_2 : \varphi_2}{(a_1, a_2) : (\varphi_1, \varphi_2)}$$

Judgment $\boxed{v : \tau}$

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

Semantics of SNESL1 with cost (TODO: add cost):
Evaluation environment $\rho = [x_1 \mapsto v_1, ..., x_i \mapsto v_i]$

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

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

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

$$\frac{\rho \vdash e_1 \downarrow v_1 \qquad \rho[x \mapsto v_1] \vdash e_2 \downarrow v}{\rho \vdash \textbf{let } e_1 = x \textbf{ in } e_2 \downarrow v} \qquad \frac{\phi(v_1, ..., v_k) \downarrow v}{\rho \vdash \phi(x_1, ..., x_k) \downarrow v}\ ((\rho(x_i) = v_i)_{i=1}^k)$$

$$\frac{([x \mapsto v_i, (x_i \mapsto a_i)_{i=1}^j] \vdash e_1 \downarrow v_i')_{i=1}^k}{\rho \vdash \{e_1 : x \textbf{ in } y \textbf{ using } x_1, ..., x_j\} \downarrow \{v_1', ..., v_k'\}}\ (\rho(y) = \{v_1, ..., v_k\}, (\rho(x_i) = a_i)_{i=1}^j)$$

$$\frac{}{\rho \vdash \{e_1 \mid x \textbf{ using } x_1, ..., x_j\} \downarrow \{\}}\ (\rho(x) = \texttt{F})$$

$$\frac{\rho \vdash e_1 \downarrow v_1}{\rho \vdash \{e_1 \mid x \textbf{ using } x_1, ..., x_j\} \downarrow \{v_1\}}\ (\rho(x) = \texttt{T})$$

$$\rho \vdash f(x_1, ..., x_k) \downarrow ???$$

The built-in functions are also a subset of SNESL built-in functions shown in Figure 1.3 but the vector-related ones are removed, as shown in Figure 2.3.

$$\begin{aligned}
\phi &::= \oplus \mid \textbf{append} \mid \textbf{concat} \mid \textbf{iota} \mid \textbf{part} \mid \textbf{scan}_{\otimes} \mid \textbf{reduce}_{\otimes} \mid \textbf{the} \mid \textbf{empty} \\
\oplus &::= + \mid - \mid \times \mid \div \mid \% \mid \leq \mid ... && \text{(consts operations)} \\
\otimes &::= + \mid \times \mid ... && \text{(associative binary operations)}
\end{aligned}$$

Figure 2.3: Primitive functions in SNESL1

The consts operations of $\oplus$ should be self-explained from their conventional names. The types, short descriptions and examples of the other streamable operations are given below.

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

Example 2.1.

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

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

- **concat** : $\{\{\tau\}\} \rightarrow \{\tau\}$, concatenates the elements of type sequence into one sequence.

Example 2.2.

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

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

- **iota** : **int** $\rightarrow$ {**int**}, generates a sequence of integers starting from 0 to the given argument integer minus 1; syntactic-sugared as the symbol &.

Example 2.3.

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

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

- **part** : ($\{\tau\} \times$ {**bool**}) $\rightarrow \{\{\tau\}\}$, partitions a sequence into subsequences segmented by the second descriptor argument.

Example 2.4.

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

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

- **scan**$_+$: {**int**} $\rightarrow$ {**int**}, performs an exclusive scan of plus operation on the given sequence.

Example 2.5.

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

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

- **reduce**$_+$: {**int**} $\rightarrow$ **int**, performs a reduction of plus operation on the given sequence, i.e., compute its sum.

Example 2.6.

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

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

- **the** : $\{\tau\} \rightarrow \tau$, returns the element of a singleton sequence.

Example 2.7.

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

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

- **empty** : $\{\tau\} \rightarrow$ **bool**, tests if the given sequence is empty.

Example 2.8.

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

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

Example 2.9. A user-defined function written in SNESL1 to compute the multiplication of a square matrix and its transpose.

```
-- code desplay format changed for readability
function matmul(n:int) :{{int}} =
  let matA = {&n : _ in &n};
      matB = {{x : _ in &n} : x in &n} -- transposition of matA
  in {{ reducePlus({x*y : x in a, y in b}) : a in matA} : b in
     matB}

> matmul(4)
{{0,0,0,0},{6,6,6,6},{12,12,12,12},{18,18,18,18}} :: {{int}}
```

2.2 SVCODE

In [Mad13] a streaming target language for a minimal SNESL was defined. With trivial changes in the instruction set, this language, named as SVCODE (Streaming VCODE), has been implemented on a multicore system in [Mad16]; the various experiment results have demonstrated single-core performance similar to sequential C code for some simple text-processing tasks as well as the potential for further performance improvment by scheduling opitmization and code analysis.

In this thesis, we put emphasis on the formalization of this low-level language's semantics. Also, to support recursion in the high-level language at the same time preserving the cost, non-trivial extension of this language is needed.

2.2.1 SVCODE Syntax

The primitive data structure that SVCODE manipulates, called *stream*, is similar to the one-dimensional array in C language; the main difference is that the elements of a stream can be collected through not only a piece of memory, but also a period of time.

For our minimal language, a primitive stream $\vec{a}$ can be a stream 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 grammar of SVCODE is given in Figure 2.4. An SVCODE program is basically a list of command or instructions of ecah defines one or more new streams. As a general rule of reading an SVCODE instruction, the stream ids on the left-hand side of a symbol ":=" are the defined streams, and the right-hand side ones are possibly used to generate those new ones.

$$
\begin{array}{lll}
p & ::= \ \epsilon & \text{(empty program)} \\
& \mid s := \psi(s_1, ..., s_k) & \text{(single stream definition)} \\
& \mid S_{out} := \mathtt{WithCtrl}(s, S_{in}, p_1) & (\mathtt{WithCtrl} \text{ block}) \\
& \mid S_2 := \mathtt{SCall}(f, S_1) & \text{(SVCODE function call)} \\
& \mid p_1; p_2 & \\
\\
s & ::= 0 \mid 1 \ ... \in \mathbf{SId} = \mathbb{N} & \text{(stream ids)} \\
\\
\psi & ::= \mathtt{Const_a} \mid \mathtt{ToFlags} \mid \mathtt{Usum} \mid \mathtt{Map}_{\oplus} \mid \mathtt{Scan}_{\otimes} \mid \mathtt{Reduce}_{\otimes} \mid \mathtt{Distr} & \text{(Xducers)} \\
& \mid \mathtt{Pack} \mid \mathtt{UPack} \mid \mathtt{B2u} \mid \mathtt{SegConcat} \mid \mathtt{USegCount} \mid \mathtt{InterMerge} \mid ... & \\
\oplus & ::= + \mid \ - \ \mid \ \times \ \mid \ \div \ \mid \% \mid \leq \ \mid ... & \text{(consts operations)} \\
\otimes & ::= + \mid \ \times \ \mid ... & \text{(associative binary operations)} \\
\\
S & ::= \{s_1, ..., s_i\} \in \mathbb{S} & \text{(set of stream ids)}
\end{array}
$$

Figure 2.4: Grammar of SVCODE

A well-formed SVCODE instruction is expected to always assign $fresh$ 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 we guarantee the freshness of stream ids. In the practical implementation, we simply identify each stream with a natural number, a smaller one always defined eariler than a greater one.

The primitive 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 stream $s_1, ..., s_k$ as parameters and returning s.

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

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

which may or may not execute a piece of SVCODE program p_1, but always defines a bunch of stream ids S_{out}. s is the new control stream used when the code block p_1 is executed, and S_{in} is the set of stream ids that can be used in p_1. Discussions about this instruction will occur many times throughout the thesis as it plays a significant role in dealing with most of the issues we are deeply concerned, including cost model correctness and dynamic unfolding of recursive functions.

As the high-level language supports recursive functions, the function body must be unfolded dynamically in runtime. A recursive call should be included in some condition statement to decide when the recursion can terminate, which is impossible to do during compiling time. The instruction $S_2 := \texttt{SCall}(f, S_1)$ can be read as: "calling function f with arguments S_1 returns S_2".

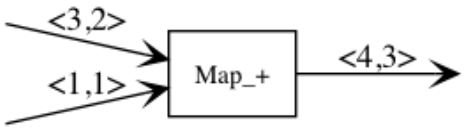
(may add more informal explanation of SVCODE semantics)

2.2.2 Xducers and dataflow

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



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

Figure 2.5 shows an example program, with its DAG in Figure 2.6.

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

Figure 2.5: A small SVCODE program

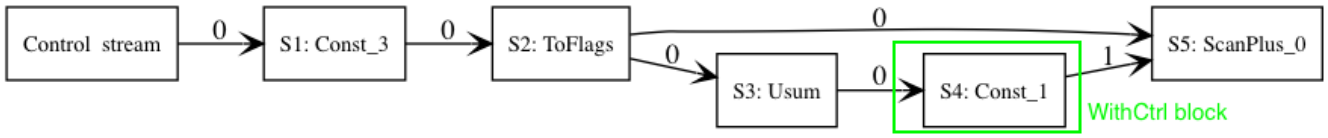


Figure 2.6: Dataflow DAG for the code in Figure 2.5 (assuming S3 is nonempty). Note that, for simplicity, the control stream is added as an explicit supplier only to Xducer $\texttt{Const}_\texttt{a}$.

When we talk about two Xducers A and B connected by an arrow from A to B in the DAG, we call A a *producer* or a *supplier* to B, and B a *consumer* or a *client* of A. As an Xducer can have multiple suppliers, we distinguish these suppliers by giving each of them an index, called a *channel number*. In Figure 2.6, the channel number is labeled above each edge. For example, the Xducer S2 has two clients, S3 and S5, for both of whom it is the No.0 channel; Xducer S5 has two suppliers: S2 the No.0 channel and S3 the No.1.

2.3 Value representation

In SVCODE, the data structures are only primitive streams and pairs of streams. To support the various high-level value types, that is, to design streamable counterparts to all the values of SNESL1, some technical transformation is necessary. The technique is similar to the idea described in [PP93], but the descriptors in SVCODE are boolean streams instread of interger ones.

- A const is represented as a singleton stream.

  Example 2.11.

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

- A non-nested primitive 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 and one `T`.

  Example 2.12.

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

- For a nested sequence with a nesting depth d, all the data are flattened to one data stream, but d descriptors are used to represent the segments at each depth. Thus a non-nested sequence is just a special case of depth $d = 1$.

  Example 2.13.

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

- A high-level pair is a pair of streams at the low level.

Example 2.14.

$$(1, 2) \triangleright (\langle 1 \rangle, \langle 2 \rangle)$$

(will add one or two examples about sequence of pairs and some explainations)

2.4 Translating SNESL1 to SVCODE

As we have seen in the last section, a high-level sequence corresponds to a number of low-level streams. We use a structure **STree** (stream tree) to generalize the relation between the high-level values and the low-level streams during compiling time:

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

Thus a stream tree is a binary tree whose leaves are stream ids.

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

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

Another essential component maintained in the translation procedure is an unused stream id called the next fresh id because it will be assigned to the defined stream(s) of the next generated instruction.

2.4.1 Expression translation

The translation rules for SNESL1 expressions is shown in Figure 2.7. The jugement can be read as: " in the environment δ, the expression e is translated to an SVCODE program p, whose stream ids starts from s_0 and ends at $s_1 - 1$ (both included), and the evaluatation result corresponds to st".

(should have more fixes and informal explanations in the rest of this section)

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

$$\frac{}{\delta \ \vdash \ a \Rightarrow^{s_0}_{s_0+1} (s_0 := \texttt{Const}_\texttt{a}(), s_0)} \qquad \frac{}{\delta \ \vdash \ x \Rightarrow^{s_0}_{s_0} (\epsilon, st)} \, (\delta(x) = st)$$

$$\frac{\delta \ \vdash \ e_1 \Rightarrow^{s_0}_{s_0'} (p_1, st_1) \qquad \delta \ \vdash \ e_2 \Rightarrow^{s_0'}_{s_1} (p_2, st_2)}{\delta \ \vdash \ (e_1, e_2) \Rightarrow^{s_0}_{s_1} (p_1; p_2, (st_1, st_2))}$$

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

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

$$\left(\begin{array}{rl} \delta(y) = & (st_1, s_b) \\ (\delta(x_i) = & st_i')^j_{i=1} \\ p = & s_0 := \texttt{Usum}(s_b); \\ & (s_i := \texttt{Distr}(s_\texttt{b}, s_\texttt{i}');)^j_{i=1} \\ & S_{out} := \texttt{WithCtrl}(s_0, S_{in} \\ S_{in} = & \texttt{fv}(p_1) \\ S_{out} = & \overline{st} \cap \texttt{dv}(p_1) \\ s_{i+1} = & s_i + 1, \forall i \in \{0, ..., j-1\} \end{array} \right.$$

???! TODO:fix $\dfrac{[x \mapsto st_1, (x_i \mapsto st_i)^j_{i=1}] \ \vdash \ e \Rightarrow^{s_0+1+j}_{s_1} (p_1, st)}{\delta \ \vdash \ \{e : x \ \textbf{in} \ y \ \textbf{using} \ x_1, ..., x_j\} \Rightarrow^{s_0}_{s_1} (p, (st, s_b))}$

Figure 2.7: Translation rules for SNESL1 expressions

2.4.2 Built-in function translation

The function call of a high-level built-in function will be translated to a few lines of SVCODE instuctions.

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

$$\frac{}{\oplus(s_1, ..., s_k) \Rightarrow^{s_0}_{s_0+1} (s_0 := \texttt{Map}_\oplus(s_1, ..., s_k), s_0)}$$

$$\frac{}{\textbf{iota}(s) \Rightarrow^{s_0}_{s_4} (p, (s_3, s_0))} \left(\begin{array}{l} s_{i+1} = s_i + 1, \forall i \in \{0, ..., 3\} \\ p = s_0 := \texttt{ToFlags}(s); \\ \quad s_1 := \texttt{Usum}(s_0); \\ \quad [s_2] := \texttt{WithCtrl}(s_1, [s_1], s_2 := \texttt{Const}_\texttt{1}()); \\ \quad s_3 := \texttt{ScanPlus}_0(s_0, s_2) \end{array} \right)$$

(will add more)

Figure 2.8: Translation of built-in functions

2.4.3 User-defined function translation

A user-defined function $function\ f(x_1 : \tau_1, ..., x_k : \tau_k) : \tau = e$ will be translated to an SVCODE function $([S_1], p, \overline{st})$, where $[x_1 \mapsto st_1, ..., x_k \mapsto st_k] \vdash e \Rightarrow^0_{s_1} (p, st)$ and ...

$$tp2tree : \tau \rightarrow \textbf{STree}$$

2.5 Eager interpreter

Recall that an SVCODE program is a list of instructions of each defines one or more streams. The eager interpreter executes the instructions sequentially, assuming the available memory is infinitely large, which is the critical difference between the execution models of the eager and streaming interpreters. For an eager interpreter, since there is always enough space, a new defined stream can be completely 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. And 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.

2.5.1 Dataflow

2.5.2 Cost model

2.6 Streaming interpreter

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

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

2.6.1 Processes

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

Filling state : the buffer is not full, and the Xducer is producing or writing data to it; any other trying to read it has to wait, or more precisely, enters a read-block state.

Draining state : the buffer must be full; the readers, including the read-blocked ones, can start reading it now; if the Xducer tries to write the buffer, then it enters a write-block state.

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

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

$$\textbf{BufState} = \{\textbf{Filling}, \textbf{Draining}\ b\}$$

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

Since Xducers have different data rates (the size of consumed/produced data at each round), it is also important to keep track of the position of the data that has been read, which can be represented by an integer. Also, it is possible that a Xducer reads from the same supplier multiple times but with different data rates, in which case only a pair of stream id and an integer is not enough to distinguish all the different read cursors. Thus we need a third component, the channel number, to record the state of each reader, as we have shown in Figure 2.6. As a result, we must have a client list **Clis** of elements of type $(\textbf{SId}, \textbf{int}, \textbf{int})$

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

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

where S is the stream ids of the suppliers.

2.6.2 Scheduling

2.6.3 Recursion

2.6.4 Deadlock

2.6.5 Examples

Chapter 3

Formalization

3.1 SNESL0

3.1.1 Type system

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

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

- Expression typing rules:

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

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

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

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

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

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

- Value typing rules:

  Judgment $\boxed{v : \tau}$

  $$\frac{}{n : \textbf{int}} \qquad \frac{(v_i : \tau)_{i=1}^k}{\{v_1, ..., v_k\} : \{\tau\}}$$

3.1.2 Syntax

SNESL Expressions:

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

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

SNESL values:

$$n \in \mathbb{Z}$$
$$v ::= n \mid \{v_1, ..., v_k\}$$

3.1.3 Semantics

$\rho = [x_1 \mapsto v_1, ..., x_i \mapsto v_i]$

- **Judgment** $\boxed{\rho \ \vdash \ e \downarrow v}$

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

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

$$\frac{([x \mapsto v_i, (x_i \mapsto n_i)_{i=1}^j] \ \vdash \ e \downarrow v_i')_{i=1}^k}{\rho \ \vdash \ \{e : x\ \mathbf{in}\ y\ \mathbf{using}\ x_1, ..., x_j\} \downarrow \{v_1', ..., v_k'\}}\,(\rho(y) = \{v_1, ..., v_k\}, (\rho(x_i) = n_i)_{i=1}^j)$$

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

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

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

3.2 SVCODE0

3.3 SVCODE Syntax

The data or *streams* that an SVCODE program computes are basically vectors of consts. For our minimal language, a primitive stream $\vec{a}$ can be a vector of booleans, integers or units, as the following grammar shows:

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

The grammar of SVCODE is given in Figure 3.1.

$$
\begin{array}{lll}
p & ::= & \epsilon \\
& | & s := \psi(s_1, ..., s_k) \qquad (s \notin \{s_1, ..., s_k\}) \\
& | & S_{out} := \texttt{WithCtrl}(s, S_{in}, p_1) \qquad (\texttt{fv}(p_1) \subseteq S_{in}, S_{out} \subseteq \texttt{dv}(p_1)) \\
& | & p_1; p_2 \qquad (\texttt{dv}(p_1) \cap \texttt{dv}(p_2) = \emptyset) \\
\\
s & ::= & 0 \mid 1 \mid ... \in \mathbf{SId} = \mathbb{N} \qquad \text{(stream ids)} \\
\\
\psi & ::= & \texttt{Const}_{\texttt{a}} \mid \texttt{ToFlags} \mid \texttt{Usum} \mid \texttt{MapTwo}_{\oplus} \mid \texttt{ScanPlus}_{n_0} \mid \texttt{Distr} \qquad \text{(Xducers)} \\
\oplus & ::= & + \mid - \mid \times \mid \div \mid \% \mid \leq \mid ... \qquad \text{(binary operations)} \\
\\
S & ::= & \{s_1, ..., s_i\} \in \mathbb{S} \qquad \text{(a set of stream ids)}
\end{array}
$$

Figure 3.1: Grammar of SVCODE

The instructions in SVCODE that perform the computation directly on primitive streams are stream definitions in the form

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

where ψ is a primitive function, or a *Xducer*(transducer), taking stream $s_1, ..., s_k$ as parameters and returning s.

The only essential control struture in SVCODE is the instruction

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

which may or may not execute a piece of SVCODE program p_1, but always defines a bunch of stream variables S_{out}. Discussions about this `WithCtrl` instruction will occur again and again throughout the thesis as it plays a significant role in dealing with most of the issues we are deeply concerned, including cost model correctness and dynamic unfolding of recursive functions.

The function `dv` returns the set of defined variables of a given SVCODE program.

$$
\begin{array}{ll}
\texttt{dv}(\epsilon) & = \emptyset \\
\texttt{dv}(s := \psi(s_1, ..., s_k)) & = \{s\} \\
\texttt{dv}(S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)) & = S_{out} \\
\texttt{dv}(p_1; p_2) & = \texttt{dv}(p_1) \cup \texttt{dv}(p_2)
\end{array}
$$

Correspondingly, `fv` returns the free variables set.

$$
\begin{aligned}
&\texttt{fv}(\epsilon) && = \emptyset \\
&\texttt{fv}(s := \psi(s_1, ..., s_i)) && = \{s_1, ..., s_k\} \\
&\texttt{fv}(S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1)) && = \{s_c\} \cup S_{in} \\
&\texttt{fv}(p_1; p_2) && = \texttt{fv}(p_1) \cup (\texttt{fv}(p_2) - \texttt{dv}(p_1))
\end{aligned}
$$

An immediate property of this language is that the defined variables of a well-formed SVCODE program are always $fresh$. In other words, there is no overlapping between the free variables and the newly generated ones.

Lemma 3.1. $\texttt{fv}(p) \cap \texttt{dv}(p) = \emptyset$.

The proof is straightward by induction on the syntax of p.

3.4 SVCODE semantics

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

Notation 3.2. *Let $\langle a_0 | \vec{a} \rangle$ denote a non-empty stream $\langle a_0, a_1, ..., a_i \rangle$ for some $\vec{a} = \langle a_1, ..., a_i \rangle$.*

Notation 3.3 (Stream concatenation). $\langle a_1, ..., a_i \rangle +\!\!+ \langle a'_1, ..., a'_j \rangle = \langle a_1, ..., a_i, a'_1, ..., a'_j \rangle$

The operational semantics of SVCODE is given in Figure 3.2. The runtime environment or store σ is a map 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 basically a vector of units representing an unary number, indicates the *parallel degree* of the computation. The role of control stream will become much clearer when we come to the semantics of Xducers. It is worth noting that only in the rule P-Wc-Nonemp the control stream has a chance to get changed.

The rule P-Empty is trivial, empty program doing nothing on the store.

The rule P-Xducer adds the store a new stream binding where the bound vector is generated by a specific Xducer with input streams. The detailed semantics of Xducers are defined in the next subsection.

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 as well as the streams in S_{in}, which includes the free variables of p_1, are all empty, then just bind empty vectors to the streams in S_{out}, which are part of the defined streams of p_1.
- 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 new control stream is crucial here, because it decides whether or not to execute p_1, which is the key to avoiding infinite unfolding of recursive funtions. For an eager interpreter of SVCODE, if we count one stream definition as one step, then this skip guarantees the low-level step cost agrees on the high-level one. Also, skipping a certain piece of code should help improve the efficiency of execution.

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

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

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

P-Wc-Emp: $\dfrac{}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{l}]} \begin{pmatrix} \forall s \in \{s_c\} \cup S_{in}.\sigma(s) = \langle\rangle \\ S_{out} = \{s_1, ..., s_l\} \end{pmatrix}$

P-Wc-Nonemp : $\dfrac{\langle p_1, \sigma\rangle \Downarrow^{\vec{c}_1} \sigma''}{\langle S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_1), \sigma\rangle \Downarrow^{\vec{c}} \sigma[(s_i \mapsto \sigma''(s_i))_{i=1}^{l}]} \begin{pmatrix} \sigma(s_c) = \vec{c}_1 \neq \langle\rangle \\ S_{out} = \{s_1, ..., s_l\} \end{pmatrix}$

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

Figure 3.2: SVCODE semantics

3.4.1 General semantics

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

Figure 3.3 shows the semantics at the general level.

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

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

P-X-Termi: $\dfrac{}{\psi(\langle\rangle_1, ..., \langle\rangle_k) \Downarrow^{\langle\rangle} \langle\rangle}$ [1]

Figure 3.3: Semantics of SVCODE transducers

There are only two rules for the general semantics. They together say that the output stream is computed in a "loop" fashion, where the iteration uses specific block semantics of the Xducer and the number of iteration 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 the control stream is a representation of the parallel degree of the computation, then a block consumes exact one degree. We note that all these blocks are data-independent, which means they can be performed in parallel. Now it is clear that the control stream indeed carrys the theoretical maximum number of processors we

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

need to execute the computation most efficiently (if the computation within the block can not be parallelized further)(???).

3.4.2 Block semantics

After abstracting the general semantics, the remaining work of formalizing the specific semantics of Xducers within a block becomes relatively clear and easy. The block semantics are defined in Figure 3.4.

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

P-Const: $\dfrac{}{\texttt{Const}_\texttt{a}() \downarrow \langle a \rangle}$ P-ToFlags: $\dfrac{}{\texttt{ToFlags}(\langle n \rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_n, \texttt{T} \rangle}$

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

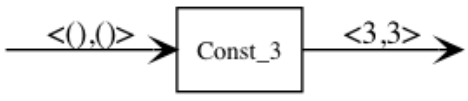
P-UsumF : $\dfrac{\texttt{Usum}(\vec{b}) \downarrow \vec{a}}{\texttt{Usum}(\langle \texttt{F} | \vec{b} \rangle) \downarrow \langle () | \vec{a} \rangle}$ P-UsumT: $\dfrac{}{\texttt{Usum}(\langle \texttt{T} \rangle) \downarrow \langle \rangle}$

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

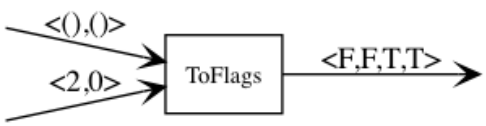
P-DistrF : $\dfrac{\texttt{Distr}(\vec{b}, \langle n \rangle) \downarrow \vec{a}}{\texttt{Distr}(\langle \texttt{F} | \vec{b} \rangle, \langle n \rangle) \downarrow \langle n | \vec{a} \rangle}$ P-DistrT: $\dfrac{}{\texttt{Distr}(\langle \texttt{T} \rangle, \langle n \rangle) \downarrow \langle \rangle}$

Figure 3.4: Semantics of transducer blocks

- $\texttt{Const}_\texttt{a}()$ outputs the const a until the control stream reaches EOS.
  Example 3.1. $\texttt{Const}_3()$ with control stream $\vec{c} = \langle (), () \rangle$:

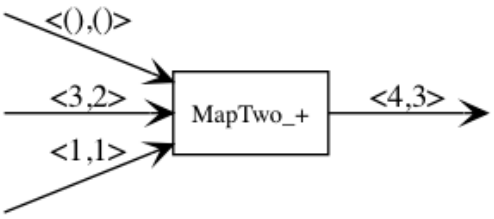


- $\texttt{ToFlags}(\langle n \rangle)$ first outputs n Fs, then one T.
  Example 3.2. $\texttt{ToFlags}(\langle 2, 0 \rangle)$:



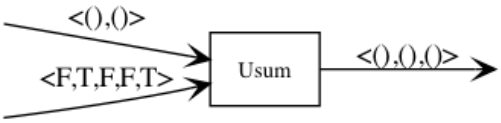
– $\texttt{MapTwo}_{\oplus}(\langle n_1 \rangle, \langle n_2 \rangle)$ outputs the binary operating result of $\oplus$ on n_1 with n_2.

Example 3.3. $\texttt{MapTwo}_{+}(\langle 3, 2 \rangle, \langle 1, 1 \rangle)$:



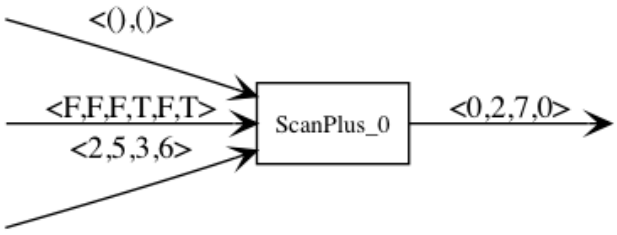
– $\texttt{Usum}(\vec{b})$ transforms an `F` to a unit, or a `T` to nothing. It is the only Xducer that can generate a unit vector, so it is mainly used when we need to replace the control stream.

Example 3.4. $\texttt{Usum}(\langle \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T} \rangle)$:



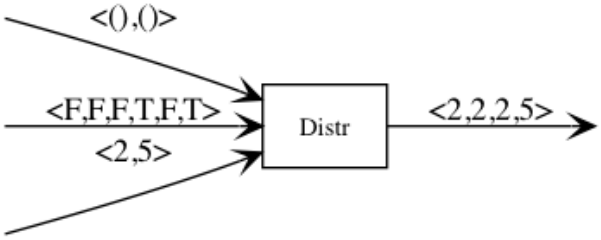
– $\texttt{ScanPlus}_{n_0}(\vec{b}, \vec{a})$ performs an exclusive scan of the binary operation plus on $\vec{a}$, segmented by $\vec{b}$, with a starting element n_0.

Example 3.5. $\texttt{ScanPlus}_0(\langle \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{T} \rangle, \langle 2, 5, 3, 6 \rangle)$



– $\texttt{Distr}(\vec{b}, \langle n \rangle)$ replictes the const n u times where u is the unary number segmented by b.

Example 3.6. $\texttt{Distr}(\langle \texttt{F}, \texttt{F}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{T} \rangle, \langle 2, 5 \rangle)$



As we have discussed before, we consider a block as the minimum computing unit assigned to a single processor. This is reasonable for Xducers such as $\texttt{Const}_{\texttt{a}}$ and $\texttt{MapTwo}_{\oplus}$, 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 extend the language with more Xducers, we could find that computations on unary numbers within blocks are common, which is mainly due to the value represenation 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 unary level parallelism can be investegated in future work.

3.5 SVCODE determinism

As we have given formal semantics to the language, we now argue that a well-formed SVCODE program is deterministic.

Definition 3.4 (**Stream prefix**). *$\vec{a}$ is a prefix of $\vec{a}'$, written $\vec{a} \sqsubseteq \vec{a}'$, if one the following rules applys:*

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

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

Lemma 3.5. *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 $\vec{a}_1 = \langle a_0, \vec{a}_1'\rangle$ for some $\vec{a}_1'$. ■

One may notice that in the rule P-X-TERMI both the control stream and the parameter stream(s) must be all empty, and no rules apply to the other cases where one of them is empty while the other is not. The following lemma explains why the other cases can never happen: there is only one way to cut down a prefix of each input stream for a specific Xducer to be consumed in a block.

Lemma 3.6. *If*

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

then

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

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

Proof. The proof is by induction on the syntax of ψ. We show two cases `ToFlags` and $\texttt{ScanPlus}_{n_0}$ here; the others are analogous.

- Case $\psi = \texttt{ToFlags}$.

  Since there is only one rule for `ToFlags`, we must have

$$\mathcal{P} = \frac{}{\texttt{ToFlags}(\langle n_1\rangle) \downarrow \langle \texttt{F}_1, ..., \texttt{F}_{n_1}, \texttt{T}\rangle}$$

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, $\mathcal{P}_{r1}$ and $\mathcal{P}'_{r1}$ must all use the rule Pre-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 $\psi = \texttt{ScanPlus}_{n_0}$.
From the two rules P-ScanT and P-ScanF, it is clear that k=2, and both $\vec{a}'_1$ and $\vec{a}''_1$ must be nonempty, which means $\mathcal{P}_{r1}$ and $\mathcal{P}'_{r1}$ must all use Pre-Nonemp.
By induction on $\vec{a}_1$, there are two subcases:
  - Subcase $\vec{a}_1 = \langle \texttt{T} | \vec{a}_{10} \rangle$.
  By Pre-Nonemp we know $\vec{a}'_1$ and $\vec{a}''_1$ must start with a T, thus both $\mathcal{P}$ and $\mathcal{P}'$ must use P-ScanT, and they must be identical:

$$\mathcal{P} = \mathcal{P}' = \frac{}{\texttt{ScanPlus}_{n_0}(\langle \texttt{T} \rangle, \langle \rangle) \downarrow \langle \rangle}$$

  So immediately we have $\vec{a}'_1 = \vec{a}''_1 = \langle \texttt{T} \rangle$, $\vec{a}'_2 = \vec{a}''_2 = \langle \rangle$, and $\vec{a}' = \vec{a}'' = \langle \rangle$ as required.
  - Subcase $\vec{a}_1 = \langle \texttt{F} | \vec{a}_{10} \rangle$.
  By Pre-Nonemp we know $\vec{a}'_1$ and $\vec{a}''_1$ must start with an F, therefore both $\mathcal{P}$ and $\mathcal{P}'$ must use P-ScanF. Assume $\vec{a}_2 = \langle n | \vec{a}_{20} \rangle$, then we must have

$$\mathcal{P} = \frac{\mathcal{P}_0}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F} | \vec{a}'_{10} \rangle, \langle n | \vec{a}'_{20} \rangle) \downarrow \langle n_0 | \vec{a}'_0 \rangle} \quad \text{where} \quad (\vec{a}'_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^2 \tag{3.1}$$

  So $\vec{a}'_1 = \langle \texttt{F} | \vec{a}'_{10} \rangle, \vec{a}'_2 = \langle n | \vec{a}'_{20} \rangle$, and $\vec{a}' = \langle n_0 | \vec{a}'_0 \rangle$.
  Similarly,

$$\mathcal{P}' = \frac{\mathcal{P}'_0}{\texttt{ScanPlus}_{n_0}(\langle \texttt{F} | \vec{a}''_{10} \rangle, \langle n | \vec{a}''_{20} \rangle) \downarrow \langle n_0 | \vec{a}''_0 \rangle} \quad \text{where} \quad (\vec{a}''_{i0} \sqsubseteq \vec{a}_{i0})_{i=1}^2 \tag{3.2}$$

  and $\vec{a}''_1 = \langle \texttt{F} | \vec{a}''_{10} \rangle, \vec{a}''_2 = \langle n | \vec{a}''_{20} \rangle$, $\vec{a}'' = \langle n_0 | \vec{a}''_0 \rangle$.
  By IH 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 3.7 (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 $\vec{c}$. There are two cases: $\vec{c} = \langle \rangle$ and $\vec{c} = \langle () | \vec{c}_0 \rangle$ for some $\vec{c}_0$. The first case is trivial, so we just show the second here.

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

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

  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,

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

  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 3.5 k times on (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 3.6 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 3.8 (**SVCODE determinism**). *If $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma'$ (by some derivation $\mathcal{P}$) and $\langle p, \sigma\rangle \Downarrow^{\vec{c}} \sigma''$ (by some derivation $\mathcal{P}'$), then $\sigma' = \sigma''$.*

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 3.7, the case for $p = s := \psi(s_1, ..., s_k)$ is also trivial; proof of $p = p_1; p_2$ can be done by IH; the only interesting case is $p = S_{out} :=$ `WithCtrl`(s_c, S_{in}, p_1).

- Case $p = S_{out} :=$ `WithCtrl`(s_c, S_{in}, p_1).
  Assume $S_{out} = \{s_1, ..., s_l\}$. There are two subcases by induction on $\sigma(s_c)$:

– Subcase $\sigma(s_c) = \langle\rangle$.
Then $\mathcal{P}$ and $\mathcal{P}'$ must all use P-Wc-Emp, and they must be identical:

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

with $\forall s \in S_{in}.\sigma(s) = \langle\rangle$. So $\sigma' = \sigma'' = \sigma[(s_i \mapsto \langle\rangle)_{i=1}^{l}]$, as required.

– Subcase $\sigma(s_c) \neq \langle\rangle$.
Then we must have

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

Also, we have

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

So $\sigma' = \sigma[(s_i \mapsto \sigma_1(s_i))_{i=1}^{l}]$, and $\sigma'' = \sigma[(s_i \mapsto \sigma'_1(s_i))_{i=1}^{l}]$.
By IH on $\mathcal{P}_1$ and $\mathcal{P}'_1$, we obtain

$$\sigma_1 = \sigma'_1$$

Then it is clear that $\sigma' = \sigma''$, as required.

■

3.6 Translation

3.6.1 Translation rules

(1) Stream tree:

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

(2) Convert a stream tree to a list of stream ids:

$$\begin{array}{l}\bar{}: \textbf{STree} \to S \\ \overline{s} = [s] \\ \overline{(st, s)} = \overline{st} +\!\!+ [s]\end{array}$$

(3) Translation environment:

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

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

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

$$\frac{\phi(st_1,...,st_k) \Rightarrow_{s_1}^{s_0} (p, st)}{\delta \vdash \phi(x_1,...,x_k) \Rightarrow_{s_1}^{s_0} (p, st)} ((\delta(x_i) = st_i)_{i=1}^k) \qquad \begin{pmatrix} \delta(y) = (st_1, s_b) \\ (\delta(x_i) = s_i')_{i=1}^j \\ p = s_0 := \texttt{Usum}(s_b); \\ \quad (s_i := \texttt{Distr}(s_\texttt{b}, s_\texttt{i}');)_{i=1}^j \\ \quad S_{out} := \texttt{WithCtrl}(s_0, S_{in}, p_1) \\ S_{in} = \texttt{fv}(p_1) \\ S_{out} = \overline{st} \cap \texttt{dv}(p_1) \\ s_{i+1} = s_i + 1, \forall i \in \{0,...,j-1\} \end{pmatrix}$$

$$\frac{[x \mapsto st_1, (x_i \mapsto s_i)_{i=1}^j] \vdash e \Rightarrow_{s_1}^{s_0+1+j} (p_1, st)}{\delta \vdash \{e : x \textbf{ in } y \textbf{ using } x_1,...,x_j\} \Rightarrow_{s_1}^{s_0} (p, (st, s_b))}$$

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

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

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

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

3.6.2 Value representation

1. SVCODE values:
$$\textbf{SvVal} \ni w ::= \vec{a} \mid (w, \vec{b})$$
2. SVCODE values concatenation:
$$\begin{aligned} &++ : \textbf{SvVal} \to \textbf{SvVal} \to \textbf{SvVal} \\ &\langle \vec{a}_1, ..., \vec{a}_i \rangle ++ \langle \vec{a}_1', ..., \vec{a}_j' \rangle = \langle \vec{a}_1, ..., \vec{a}_i, \vec{a}_1', ..., \vec{a}_j' \rangle \\ &(w_1, \vec{b}_1) ++ (w_2, \vec{b}_2) = (w_1 ++ w_2, \vec{b}_1 ++ \vec{b}_2) \end{aligned}$$
3. SVCODE value construction from a stream tree:
$$\begin{aligned} &\sigma : \textbf{STree} \to \textbf{SvVal} \\ &\sigma(s) = \vec{a} \\ &\sigma((st, s)) = (\sigma(st), \sigma(s)) \end{aligned}$$
4. Value representation rules
   - **Judgment** $\boxed{v \rhd_\tau w}$

$$\frac{}{n \rhd_{\textbf{int}} \langle n \rangle} \qquad \frac{(v_i \rhd_\tau w_i)_{i=1}^k}{\{v_1,...,v_k\} \rhd_{\{\tau\}} (w, \langle \texttt{F}_1, ..., \texttt{F}_k, \texttt{T} \rangle)} (w = w_1 ++ ... ++ w_k)$$

Lemma 3.9 (**Value translation backwards determinism**). *If* $v \triangleright_\tau w$, $v' \triangleright_\tau w$, *then* $v = v'$.

3.7 Correctness

3.7.1 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 by S are the same. We call this S an *overlap* of these two stores.

Definition 3.10 (**Stores 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 denote a kind of specical concatenation of two similar stores: the *concatenation* 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. In other words, a *concatenation* of two similar stores is only a concatenation of the bound values that *maybe* different in these stores.

Definition 3.11 (**Store Concatenation**). *For* $\sigma_1 \overset{S}{\sim} \sigma_2$, $\sigma_1 \overset{S}{\bowtie} \sigma_2 = \sigma$ *where*

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

Lemma 3.12. *If* $\sigma_1 \overset{S}{\bowtie} \sigma_2 = \sigma$, *then* $\sigma_1 \overset{S}{\sim} \sigma$ *and* $\sigma_2 \overset{S}{\sim} \sigma$.

This lemma says that the concatenation result of two similar stores is still similar to each of them.

Lemma 3.13. *If*

(i) $\psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}_1} \vec{a}_0$ *by some derivation* $\mathcal{P}_1$

(ii) $\psi(\vec{a}'_1, ..., \vec{a}'_k) \Downarrow^{\vec{c}_2} \vec{a}'_0$ *by some* $\mathcal{P}_2$,

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* $\mathcal{P}_3$.

Proof. There are two possibilities:

- Case $\mathcal{P}_1$ uses P-X-Termi.
  We must have $(\vec{a}_i = \langle\rangle)_{i=0}^k$ and $\vec{c}_1 = \langle\rangle$. Then $(\vec{a}_i +\!+ \vec{a}'_i = \vec{a}'_i)_{i=0}^k$, and $\vec{c}_1 +\!+ \vec{c}_2 = \vec{c}_2$. Take $\mathcal{P}_3 = \mathcal{P}_2$ and we are done.

- Case $\mathcal{P}_1$ uses P-X-Loop.
  Assume $\vec{c}_1 = \langle ()|\vec{c}'_1\rangle$, then we have

$$\mathcal{P}_1 = \frac{\begin{array}{cc}\mathcal{P}_{11} & \mathcal{P}_{12}\\ \psi(\vec{a}_{11},...,\vec{a}_{k1}) \downarrow \vec{a}_{01} & \psi(\vec{a}_{12},...,\vec{a}_{k2}) \Downarrow^{\vec{c}'_1} \vec{a}_{02}\end{array}}{\psi(\vec{a}_1,...,\vec{a}_k) \Downarrow^{\langle c_0|\vec{c}'_1\rangle} \vec{a}_0}$$

  with $(\vec{a}_i = \vec{a}_{i1}{+\!+}\vec{a}_{i2})_{i=0}^{k}$.

  By IH on $\mathcal{P}_{12}$ with $\mathcal{P}_2$, we get a derivation $\mathcal{P}_4$ of

$$\psi(\vec{a}_{12}{+\!+}\vec{a}'_1,...,\vec{a}_{k2}{+\!+}\vec{a}'_k) \Downarrow^{\vec{c}'_1{+\!+}\vec{c}_2} \vec{a}_{02}{+\!+}\vec{a}'_0$$

  Then using the rule P-X-Loop we can build a derivation $\mathcal{P}_5$ as follows:

$$\frac{\begin{array}{cc}\mathcal{P}_{11} & \mathcal{P}_4\\ \psi(\vec{a}_{11},...,\vec{a}_{k1}) \downarrow \vec{a}_{01} & \psi(\vec{a}_{12}{+\!+}\vec{a}'_1,...,\vec{a}_{k2}{+\!+}\vec{a}'_k) \Downarrow^{\vec{c}'_1{+\!+}\vec{c}_2} \vec{a}_{02}{+\!+}\vec{a}'_0\end{array}}{\psi(\vec{a}_{11}{+\!+}(\vec{a}_{12}{+\!+}\vec{a}'_1),...,\vec{a}_{k1}{+\!+}(\vec{a}_{k2}{+\!+}\vec{a}'_k)) \Downarrow^{\langle ()|\vec{c}'_1{+\!+}\vec{c}_2\rangle} \vec{a}_{01}{+\!+}(\vec{a}_{02}{+\!+}\vec{a}'_0)}$$

  Since it is clear that

$$\forall i \in \{0,...,k\}.\ \vec{a}_{i1}{+\!+}(\vec{a}_{i2}{+\!+}\vec{a}'_i) = (\vec{a}_{i1}{+\!+}\vec{a}_{i2}){+\!+}\vec{a}'_i = \vec{a}_i{+\!+}\vec{a}'_i$$

$$\langle ()|\vec{c}'_1{+\!+}\vec{c}_2\rangle = \langle ()|\vec{c}'_1\rangle{+\!+}\vec{c}_2 = \vec{c}_1{+\!+}\vec{c}_2$$

  so take $\mathcal{P}_3 = \mathcal{P}_5$ and we are done.

■

Lemma 3.14. *If*

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

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

(iii) $\mathtt{fv}(p) \cap S = \emptyset$

(iv) $\forall s \in \mathtt{fv}(p).\sigma_2(s) = \langle\rangle$

then

(v) $\left\langle p, \sigma_1 \overset{S}{\bowtie} \sigma_2\right\rangle \Downarrow^{\vec{c}} \sigma'$

(vi) $\forall s' \in \mathtt{dv}(p).\sigma(s') = \sigma'(s')$

Lemma 3.15. *If*

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

(ii) $\langle p, \sigma_2\rangle \Downarrow^{\vec{c}} \sigma$

(iii) $\mathtt{fv}(p) \cap S = \emptyset$

(iv) $\forall s \in \mathtt{fv}(p).\sigma_1(s) = \langle\rangle$

then

(v) $\left\langle p, \sigma_1 \overset{S}{\bowtie} \sigma_2\right\rangle \Downarrow^{\vec{c}} \sigma'$

(vi) $\forall s' \in \mathtt{dv}(p).\sigma(s') = \sigma'(s')$

Lemma 3.16 (**Stores concatenation lemma**). *If*

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

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

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

(iv) $\mathtt{fv}(p) \cap S = \emptyset$

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

We need this lemma to prove that the results of single computations inside a comprehension body (i.e. p in the lemma) can be concatenated to express a parallel computation. From the other direction, we can consider this process as distributing or splitting the compuation p on even smaller degree of parallel computations, in which all the supplier streams, i.e., $\mathtt{fv}(p)$, are splitted to feed the transducers. The splitted parallel degrees are specified by the control streams, i.e., $\vec{c}_1$ and $\vec{c}_2$ in the lemma. Other untouched **SId**s in all σs (i.e., S) have no change throughout the process.

Proof. By induction on the syntax of p.

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

  By P-Empty, we take $\mathcal{P} = \overline{\left\langle \epsilon, \sigma_1 \overset{S}{\bowtie} \sigma_2 \right\rangle \Downarrow^{\vec{c}_1 +\!+ \vec{c}_2} \sigma_1 \overset{S}{\bowtie} \sigma_2}$ and we are done.

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

$$\frac{\begin{array}{c}\mathcal{P}_1' \\ \psi(\vec{a}_1, ..., \vec{a}_k) \Downarrow^{\vec{c}_1} \vec{a}\end{array}}{\langle s_l := \psi(s_1, ..., s_k), \sigma_1 \rangle \Downarrow^{\vec{c}_1} \sigma_1[s_l \mapsto \vec{a}]}$$

  and we have

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

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

$$\frac{\begin{array}{c}\mathcal{P}_2' \\ \psi(\vec{a}_1', ..., \vec{a}_k') \Downarrow^{\vec{c}_2} \vec{a}'\end{array}}{\langle s_l := \psi(s_1, ..., s_k), \sigma_2 \rangle \Downarrow^{\vec{c}_2} \sigma_2[s_l \mapsto \vec{a}']}$$

  and we have

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

  So $\sigma_1' = \sigma_1[s_l \mapsto \vec{a}], \sigma_2' = \sigma_2[s_l \mapsto \vec{a}']$.

From assumption (iv) we have $\texttt{fv}(s_l := \psi(s_1, ..., s_k)) \cap S = \emptyset$, that is,

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

By Lemma 3.13 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 3.11 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 that $\sigma_1[s_l \mapsto \vec{a}] \overset{S}{\bowtie} \sigma_2[s_l \mapsto \vec{a}'] \overset{S}{\sim} \sigma_1 \overset{S}{\bowtie} \sigma_2[s_l \mapsto \vec{a}{+\!+}\vec{a}']$ and

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

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

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

Replacing $\sigma_1 \overset{S}{\bowtie} \sigma_2[s_l \mapsto \vec{a}{+\!+}\vec{a}']$ in $\mathcal{P}''$ with the left-hand side of (3.16) gives us $\mathcal{P}$

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

as required.

- Case $p = S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0)$ where

$$\texttt{fv}(p_0) \subseteq S_{in} \tag{3.17}$$

$$S_{out} \subseteq \texttt{dv}(p_0) \tag{3.18}$$

From the assumption (iv), we have

$$\begin{aligned} \texttt{fv}(S_{out} := \texttt{WithCtrl}(s_c, S_{in}, p_0)) \cap S &= \emptyset \\ (\{s_c\} \cup S_{in}) \cap S &= \emptyset \qquad \text{(by definition of } \texttt{fv}()) \end{aligned}$$

thus

$$\{s_c\} \cap S = \emptyset \tag{3.19}$$

$$S_{in} \cap S = \emptyset \tag{3.20}$$

Since (3.17) with (3.20), we also have

$$\texttt{fv}(p_0) \cap S = \emptyset \tag{3.21}$$

Assume $S_{out} = [s_1, ..., s_j]$.
There are four possibilities:

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

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

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

and we have

$$\forall s \in \{s_c\} \cup S_{in}.\sigma_1(s) = \langle\rangle \tag{3.22}$$

thus

$$\sigma_1(s_c) = \langle\rangle \tag{3.23}$$

$$\forall s \in \texttt{fv}(p_0).\sigma_1(s) = \langle\rangle \tag{3.24}$$

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

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

and we have

$$\forall s \in \{s_c\} \cup S_{in}.\sigma_2(s) = \langle\rangle \tag{3.25}$$

thus

$$\sigma_2(s_c) = \langle\rangle \tag{3.26}$$

$$\forall s \in \texttt{fv}(p_0).\sigma_2(s) = \langle\rangle \tag{3.27}$$

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

Since $\sigma_1 \overset{S}{\sim} \sigma_2$, by Definition 3.11 with (3.19), (3.20), and (3.22), (3.25), we have

$$\forall s \in \{s_c\} \cup S_{in}.\sigma_1 \overset{S}{\bowtie} \sigma_2(s) = \sigma_1(s) +\!\!+ \sigma_2(s) = \langle\rangle \tag{3.28}$$

Also, it is easy to show that $\sigma_1[(s_i \mapsto \langle\rangle)_{i=1}^{j}] \overset{S}{\sim} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^{j}]$ and

$$\sigma_1[(s_i \mapsto \langle\rangle)_{i=1}^{j}] \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^{j}] = \sigma_1 \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^{j}] \tag{3.29}$$

Using P-Wc-Emp with (3.28), we build $\mathcal{P}'$ as follows

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

Then replcaing $\sigma_1 \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^{j}]$ in $\mathcal{P}'$ with the left-hand side of (3.29) gives us $\mathcal{P}$ of

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

as required.

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

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

and we have

$$\sigma_1(s_c) = \vec{c}_1' = \langle ()|... \rangle \tag{3.30}$$

$\mathcal{P}_2$ must look like

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

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

$$\sigma_2(s_c) = \langle\rangle \tag{3.31}$$

$$\forall s \in \texttt{fv}(p_0).\sigma_2(s) = \langle\rangle \tag{3.32}$$

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

By Lemma 3.14 on $\sigma_1 \overset{S}{\sim} \sigma_2$ with $\mathcal{P}_1'$, (3.32), (3.21), we obtain a derivation $\mathcal{P}_0$ of

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

for some σ_0, and

$$\forall s \in dv(p_0).\sigma_0(s) = \sigma_1''(s),$$

thus, with (3.18), we have

$$(\sigma_0(s_i) = \sigma_1''(s_i))_{i=1}^{j} \tag{3.33}$$

Since $\sigma_1 \overset{S}{\sim} \sigma_2$, by Definition 3.11 with (3.30), (3.31), we have

$$\sigma_1 \overset{S}{\bowtie} \sigma_2(s_c) = \sigma_1(s_c) +\!\!+ \sigma_2(s_c) = \vec{c}_1' = \langle ()|... \rangle \tag{3.34}$$

and it is also easy to prove $\sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}] \overset{S}{\sim} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^{j}]$ and

$$\sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}] \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \langle\rangle)_{i=1}^{j}] = \sigma_1 \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \sigma_1''(s_1))_{i=1}^{j}] \tag{3.35}$$

Using the rule P-Wc-Nonemp with (3.34) we can build a derivation $\mathcal{P}'$ as follows

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

With (3.33), we replace $\sigma_0(s_i)$ with $\sigma_1''(s_i)$ for $\forall i \in \{1, ..., j\}$ in $\mathcal{P}'$, obtaining

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

Then replacing $(\sigma_1 \overset{S}{\bowtie} \sigma_2)[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}]$ in $\mathcal{P}'$ with the left-hand side of (3.35), we get $\mathcal{P}$ of

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

as required.

– Subcase $\mathcal{P}_1$ uses P-Wc-Emp and $\mathcal{P}_2$ uses P-Wc-Nonemp.
This subcase is symmetric to the second one, so the proof is analogous except that this subcase uses Lemma 3.15 rather than Lemma 3.14.

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

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

and

$$\sigma_1(s_c) = \vec{c}_1 = \langle ()|...\rangle \tag{3.36}$$

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

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

and

$$\sigma_2(s_c) = \vec{c}_2 = \langle ()|...\rangle \tag{3.37}$$

So $\sigma_1' = \sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}], \sigma_2' = \sigma_2[(s_i \mapsto \sigma_2''(s_i))_{i=1}^{j}]$.
By IH on $\mathcal{P}_1'$, $\mathcal{P}_2'$ with (3.21), we get a derivation $\mathcal{P}_0$ of

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

Since $\forall i \in \{1, ..., j\}.s_i \notin S$, then by Definition 3.11, we know

$$\sigma_1'' \overset{S}{\bowtie} \sigma_2''(s_i) = \sigma_1''(s_i) +\!\!+ \sigma_2''(s_i) \tag{3.38}$$

Also, it is easy to show that $\sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}] \overset{S}{\sim} \sigma_2[(s_i \mapsto \sigma_2''(s_i))_{i=1}^{j}]$, and

$$\begin{aligned}\sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}] \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \sigma_2''(s_i))_{i=1}^{j}] \\ = \sigma_1 \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \sigma_1''(s_i) +\!\!+ \sigma_2''(s_i))_{i=1}^{j}]\end{aligned} \tag{3.39}$$

thus, with (3.38),

$$\begin{aligned}\sigma_1[(s_i \mapsto \sigma_1''(s_i))_{i=1}^{j}] \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \sigma_2''(s_i))_{i=1}^{j}] \\ = \sigma_1 \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \sigma_1'' \overset{S}{\bowtie} \sigma_2''(s_i))_{i=1}^{j}]\end{aligned} \tag{3.40}$$

Since (3.19) with (3.36), (3.37), we know $\sigma_1 \overset{S}{\bowtie} \sigma_2(s_c) = \vec{c}_1 +\!\!+ \vec{c}_2 = \langle ()|...\rangle$, therefore we can use the rule P-Wc-Nonemp to build a derivation $\mathcal{P}'$ as follows:

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

.

Then replacing $\sigma_1 \overset{S}{\bowtie} \sigma_2[(s_i \mapsto \sigma_1'' \overset{S}{\bowtie} \sigma_2''(s_j))_{i=1}^{j}]$ with the left-hand side of (3.40), we obtain $\mathcal{P}$ of

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

as required.

- Case $p = p_1; p_2$

  We must have

$$\mathcal{P}_1 = \frac{\begin{array}{cc}\mathcal{P}_1' & \mathcal{P}_1''\end{array}}{\begin{array}{cc}\langle p_1, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1'' & \langle p_2, \sigma_1''\rangle \Downarrow^{\vec{c}_1} \sigma_1'\end{array}}{\langle p_1; p_2, \sigma_1\rangle \Downarrow^{\vec{c}_1} \sigma_1'}$$

  and

$$\mathcal{P}_2 = \frac{\begin{array}{cc}\mathcal{P}_2' & \mathcal{P}_2''\end{array}}{\begin{array}{cc}\langle p_1, \sigma_2\rangle \Downarrow^{\vec{c}_2} \sigma_2'' & \langle p_2, \sigma_2''\rangle \Downarrow^{\vec{c}_2} \sigma_2'\end{array}}{\langle p_1; p_2, \sigma_2\rangle \Downarrow^{\vec{c}_1} \sigma_2'}$$

  Since $\texttt{fv}(p_1; p_2) \cap S = \emptyset$, we have $(\texttt{fv}(p_1) \cup \texttt{fv}(p_2) - \texttt{dv}(p_1)) \cap S = \emptyset$, thus

$$\texttt{fv}(p_1) \cap S = \emptyset \tag{3.41}$$

$$\texttt{fv}(p_2) \cap S = \emptyset \tag{3.42}$$

  By IH on $\mathcal{P}_1'$, $\mathcal{P}_2'$, (3.41), we get $\mathcal{P}'$ of

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

  By Definition 3.11, we must have $\sigma_1'' \overset{S}{\sim} \sigma_2''$.
  Then by IH on $\mathcal{P}_1''$, $\mathcal{P}_2''$ with (3.42), we get $\mathcal{P}''$ of

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

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

$$\frac{\begin{array}{cc}\mathcal{P}' & \mathcal{P}''\end{array}}{\begin{array}{cc}\left\langle p_1, \sigma_1 \overset{S}{\bowtie} \sigma_2 \right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma_1'' \overset{S}{\bowtie} \sigma_2'' & \left\langle p_2, \sigma_1'' \overset{S}{\bowtie} \sigma_2'' \right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma_1' \overset{S}{\bowtie} \sigma_2'\end{array}}{\left\langle p_1; p_2, \sigma_1 \overset{S}{\bowtie} \sigma_2 \right\rangle \Downarrow^{\vec{c}_1 +\!\!+ \vec{c}_2} \sigma_1' \overset{S}{\bowtie} \sigma_2'}$$

  and we are done.

■

Let $\sigma_1 \overset{<s}{=\!=} \sigma_2$ denote $\forall s' < s. \sigma_1(s') = \sigma_2(s')$.

Lemma 3.17. *If* $\sigma_1 \overset{S_1}{\sim} \sigma_1'$, $\sigma_2 \overset{S_2}{\sim} \sigma_2'$, $\sigma_1 \overset{<s}{=\!=} \sigma_2$, *and* $\sigma_1' \overset{<s}{=\!=} \sigma_2'$ *then* $\sigma_1 \overset{S_1}{\bowtie} \sigma_1' \overset{<s}{=\!=} \sigma_2 \overset{S_2}{\bowtie} \sigma_2'$.

3.7.2 Correctness proof

Lemma 3.18. *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) $(v_i \rhd_{\tau_i} \sigma(st_i))_{i=1}^k$

(v) $\bigcup_{i=1}^k \texttt{sids}(st_i) \lessdot s_0$

then

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

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

(viii) $\sigma' \overset{< s_0}{=\!=} \sigma$

(ix) $s_0 \leq s_1$

(x) $\texttt{sids}(st) \lessdot s_1$

Proof. By inducntion on the syntax of ϕ.

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

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

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

$$\mathcal{C} = \frac{}{\mathbf{const}_n() \Rightarrow_{s_0+1}^{s_0} (s_0 := \texttt{Const}_\texttt{n}(), s_0)}$$

  So $k = 0, \tau = \mathbf{int}, v = n, p = s_0 := \texttt{Const}_\texttt{n}()$, $s_1 = s_0 + 1$, and $st = s_0$
  By P-Xducer, P-X-Loop, P-X-Termi and P-Const, we can construct $\mathcal{P}$ as follows:

$$\mathcal{P} = \frac{\dfrac{\dfrac{}{\texttt{Const}_\texttt{n}() \downarrow \langle n \rangle} \quad \dfrac{}{\texttt{Const}_\texttt{n}() \Downarrow^{\langle \rangle} \langle \rangle}}{\texttt{Const}_\texttt{n}() \Downarrow^{\langle () \rangle} \langle n \rangle}}{\langle s_0 := \texttt{Const}_\texttt{n}(), \sigma \rangle \Downarrow^{\langle () \rangle} \sigma[s_0 \mapsto \langle n \rangle]}$$

  So $\sigma' = \sigma[s_0 \mapsto \langle n \rangle]$.
  Then we take $\mathcal{R} = \dfrac{}{n \rhd_{\mathbf{int}} \sigma'(s_0)}$.
  Also clearly, $\sigma' \overset{< s_0}{=\!=} \sigma$, $s_0 \leq s_0 + 1$, $\texttt{sids}(s_0) \lessdot s_0 + 1$, and we are done.

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

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

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

  where $n_3 = n_2 + n_1$, and

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

So $k = 2, \tau_1 = \tau_2 = \tau = \textbf{int}, v_1 = n_1, v_2 = n_2, v = n_3, st_1 = s_1, st_2 = s_2, st = s_0, s_1 = s_0 + 1$ and $p = s_0 := \texttt{MapTwo}_+(s_1, s_2)$.

Assumption (iv) gives us $\overline{n_1 \rhd_{\textbf{int}} \sigma(s_1)}$ and $\overline{n_2 \rhd_{\textbf{int}} \sigma(s_2)}$, which implies $\sigma(s_1) = \langle n_1 \rangle$ and $\sigma(s_2) = \langle n_2 \rangle$ respectively.

For (v) we have $s_1 < s_0$ and $s_2 < s_0$.

Then using P-Xducer with $\sigma(s_1) = \langle n_1 \rangle$ and $\sigma(s_2) = \langle n_2 \rangle$, and using P-X-Loop and P-X-Termi, we can build $\mathcal{P}$ as follows:

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

Therefore, $\sigma' = \sigma[s_0 \mapsto \langle n_3 \rangle]$.
Now we can take $\mathcal{R} = \overline{n_3 \rhd_{\textbf{int}} \sigma'(s_0)}$, and it is clear that $\sigma' \overset{<s_0}{=\!=} \sigma$, $s_0 \leq s_0 + 1$ and $\texttt{sids}(s_0) \lessdot s_0 + 1$ as required.

- Case $\phi = \textbf{iota}$

■

Theorem 3.19. ***If***

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

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

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

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

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

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

then

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

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

(ix) $\sigma' \overset{<s_0}{=\!=} \sigma$

(x) $s_0 \leq s_1$

(xi) $\overline{st} \lessdot s_1$

Proof. By induction on the syntax of e.

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

  We must have:

(i)

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

with

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

(ii)

$$\mathcal{E} = \frac{\left(\begin{array}{c}\mathcal{E}_i \\ [x \mapsto v_i, x_1 \mapsto n_1, ..., x_j \mapsto n_j] \ \vdash \ e_1 \downarrow v_i'\end{array}\right)_{i=1}^{k}}{\rho \ \vdash \ \{e_1 : x \ \textbf{in} \ y \ \textbf{using} \ x_1, ..., x_j\} \downarrow \{v_1', ..., v_k'\}}$$

with

$$\rho(y) = \{v_1, ..., v_k\}$$
$$(\rho(x_i) = n_i)_{i=1}^{j}$$

(iii)

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

with

$$\begin{aligned}
\delta(y) &= (st_1, s_b) \\
(\delta(x_i) &= s_i')_{i=1}^{j} \\
p &= s_0 := \texttt{Usum}(s_b); \\
&\quad (s_i := \texttt{Distr}(s_\texttt{b}, s_\texttt{i}');)_{i=1}^{j} \\
&\quad S_{out} := \texttt{WithCtrl}(s_0, S_{in}, p_1) \\
S_{in} &= \texttt{fv}(p_1) \\
S_{out} &= \overline{st_2} \cap \texttt{dv}(p_1) \\
s_{i+1} &= \ s_i + 1, \forall i \in \{0, ..., j-1\}
\end{aligned} \tag{3.43}$$

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

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

$$\frac{(\vdash \ v_i : \tau_1)_{i=1}^{k}}{\vdash \ \{v_1, ..., v_k\} : \{\tau_1\}} \tag{3.44}$$

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

$$(\overline{\vdash \ n_i : \textbf{int}})_{i=1}^{j} \tag{3.45}$$

.

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

$$\overline{\delta(y)} = \overline{(st_1, s_b)} = \overline{st_1}{+\!+}[s_b] \lessdot s_0 \tag{3.46}$$

and $(\overline{\delta(x_i)})_{i=1}^{j} \lessdot s_0$ implies $[s'_1, ..., s'_j] \lessdot s_0$.

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

$$\frac{\begin{pmatrix} \mathcal{R}_i \\ v_i \triangleright_{\tau_1} w_i \end{pmatrix}_{i=1}^{k}}{\{v_1, ..., v_k\} \triangleright_{\{\tau_1\}} (w, \langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle)} \tag{3.47}$$

where $w = w_1 {+\!+} ... {+\!+} w_k$, therefore we have

$$\sigma(st_1) = w \tag{3.48}$$

$$\sigma(s_b) = \langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle. \tag{3.49}$$

Also, for $\forall i \in \{1, ..., j\}$, $\rho(x_i) \triangleright_{\Gamma(x_i)} \sigma(\delta(x_i)) = n_i \triangleright_{\mathbf{int}} \sigma(s'_i)$, which implies

$$(\sigma(s'_i) = \langle n_i \rangle)_{i=1}^{j} \tag{3.50}$$

First we shall show:

(vii) $\left\langle \begin{array}{l} s_0 := \mathtt{Usum}(s_b); \\ (s_i := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{i});)_{i=1}^{j} \\ S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1) \end{array} , \sigma \right\rangle \Downarrow^{\langle () \rangle} \sigma'$ by some $\mathcal{P}$

(viii) $\{v'_1, ..., v'_k\} \triangleright_{\{\tau_2\}} \sigma'((st_2, s_b))$ by some $\mathcal{R}$

Using P-Seq $(j+1)$ times, we can build $\mathcal{P}$ as follows:

$$\frac{\mathcal{P}_0 \quad \langle s_0 := \mathtt{Usum}(s_b), \sigma \rangle \Downarrow^{\langle () \rangle} \sigma_0 \qquad \dfrac{\mathcal{P}_1 \quad \langle s_1 := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{1}), \sigma_0 \rangle \Downarrow^{\langle () \rangle} \sigma_1 \qquad \dfrac{\dfrac{\mathcal{P}_{j+1}}{\mathcal{P}_j \quad \langle S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1), \sigma_j \rangle \Downarrow^{\langle () \rangle} \sigma'} \vdots}{\left\langle \begin{array}{l} (s_i := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{i}))_{i=2}^{j}; \\ S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1) \end{array} , \sigma_1 \right\rangle \Downarrow^{\langle () \rangle} \sigma'}}{\left\langle \begin{array}{l} (s_i := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{i});)_{i=1}^{j} \\ S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1) \end{array} , \sigma_0 \right\rangle \Downarrow^{\langle () \rangle} \sigma'}}{\left\langle \begin{array}{l} s_0 := \mathtt{Usum}(s_b); \\ (s_i := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{i});)_{i=1}^{j} \\ S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1) \end{array} , \sigma \right\rangle \Downarrow^{\langle () \rangle} \sigma'}$$

in which for $\forall i \in \{1, ..., j\}$, $\mathcal{P}_i$ is a derivation of $\langle s_i := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{i}), \sigma_{i-1} \rangle \Downarrow^{\langle () \rangle} \sigma_i$. For $\mathcal{P}_0$, with $\sigma(s_b) = \langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle$, we can build it as follows:

$$\text{by P-Xducer} \; \frac{\text{by P-X-Loop} \; \dfrac{\text{by P-UsumF} \; \dfrac{\text{by P-UsumT} \; \overline{\mathtt{Usum}(\langle \mathtt{T} \rangle) \downarrow \langle \rangle}}{\vdots \; \mathtt{Usum}(\langle \mathtt{F}_2, ..., \mathtt{F}_k, \mathtt{T} \rangle) \downarrow \langle ()_2, ..., ()_k \rangle} \quad \text{by P-X-Termi} \; \dfrac{}{\mathtt{Usum}(\langle \rangle) \Downarrow^{\langle \rangle} \langle \rangle}}{\mathtt{Usum}(\langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle) \downarrow \langle ()_1, ..., ()_k \rangle}}{\mathtt{Usum}(\langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle) \Downarrow^{\langle () \rangle} \langle ()_1, ..., ()_k \rangle}}{\langle s_0 := \mathtt{Usum}(s_b), \sigma \rangle \Downarrow^{\langle () \rangle} \sigma[s_0 \mapsto \langle ()_1, ..., ()_k \rangle]}$$

So $\sigma_0 = \sigma[s_0 \mapsto \langle ()_1, ..., ()_k \rangle]$.

Similarly, with $\sigma(s_b) = \langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle$ and $(\sigma(s'_i) = \langle n_i \rangle)_{i=1}^{j}$ from (3.50), we can build each $\mathcal{P}_i$ for $\forall i \in \{1, ..., j\}$ as follows:

$$\text{by P-DistrT } \overline{\mathtt{Distr}(\langle \mathtt{T} \rangle, \langle n_i \rangle) \downarrow \langle \rangle}$$

$$\vdots$$

$$\text{by P-DistrF } \frac{\mathtt{Distr}(\langle \mathtt{F}_2, ..., \mathtt{F}_k, \mathtt{T} \rangle, \langle n_i \rangle) \downarrow \langle \overbrace{n_i, ..., n_i}^{k-1} \rangle}{}$$

$$\text{by P-X-Loop } \frac{\mathtt{Distr}(\langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle, \langle n_i \rangle) \downarrow \langle \overbrace{n_i, ..., n_i}^{k} \rangle \qquad \text{by P-X-Termi } \dfrac{}{\mathtt{Distr}(\langle \rangle, \langle \rangle) \Downarrow^{\langle \rangle} \langle \rangle}}{}$$

$$\text{by P-Xducer } \frac{\mathtt{Distr}(\langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle, \langle n_i \rangle) \Downarrow^{\langle () \rangle} \langle \overbrace{n_i, ..., n_i}^{k} \rangle}{\langle s_i := \mathtt{Distr}(s_\mathtt{b}, s'_\mathtt{i}), \sigma_{i-1} \rangle \Downarrow^{\langle () \rangle} \sigma_{i-1}[s_i \mapsto \langle \overbrace{n_i, ..., n_i}^{k} \rangle]}$$

So $\forall i \in \{1, ..., j\}.\sigma_i = \sigma_{i-1}[s_i \mapsto \langle \overbrace{n_i, ..., n_i}^{k} \rangle]$.

Thus $\sigma_j = \sigma[s_0 \mapsto \langle ()_1, ..., ()_k \rangle, s_1 \mapsto \langle \overbrace{n_1, ..., n_1}^{k} \rangle, ..., s_j \mapsto \langle \overbrace{n_j, ..., n_j}^{k} \rangle]$.

Now it remains to build $\mathcal{P}_{j+1}$.

Since we have

$$\mathcal{T}_1 = \ [x \mapsto \tau_1, x_1 \mapsto \mathbf{int}, ..., x_j \mapsto \mathbf{int}] \ \vdash \ e_1 : \tau_2$$

$$(\mathcal{E}_i = \ [x \mapsto v_i, x_1 \mapsto n_1, ..., x_j \mapsto n_j] \ \vdash \ e_1 \downarrow v'_i \qquad)_{i=1}^{k}$$

$$\mathcal{C}_1 = \ [x \mapsto st_1, x_1 \mapsto s_1, ..., x_j \mapsto s_j] \ \vdash \ e_1 \Rightarrow_{s_1}^{s_0+1+j} (p_1, st_2)$$

Let $\Gamma_1 = [x \mapsto \tau_1, x_1 \mapsto \mathbf{int}, ..., x_j \mapsto \mathbf{int}], \rho_i = [x \mapsto v_i, x_1 \mapsto n_1, ..., x_j \mapsto n_j]$ and $\delta_1 = [x \mapsto st_1, x_1 \mapsto s_1, ..., x_j \mapsto s_j]$.
For $\forall i \in \{1, ..., k\}$, we show the following three conditions, which allows us to use IH with $\mathcal{T}_1$, $\mathcal{E}_i$, $\mathcal{C}_1$ later.

(a) $\forall x \in dom(\Gamma_1). \ \vdash \ \rho_i(x) : \Gamma_1(x)$

(b) $\forall x \in dom(\Gamma_1).\overline{\delta_1(x)} \lessdot s_0 + 1 + j$

(c) $\forall x \in dom(\Gamma_1).\rho_i(x) \rhd_{\Gamma_1(x)} \sigma_{ji}(\delta_1(x))$

TS: (a)
From (3.44) and (3.45) it is clear that

$$\forall x \in dom(\Gamma_1). \ \vdash \ \rho_i(x) : \Gamma_1(x)$$

TS: (b)
From (3.46), it is clear that $\overline{\delta_1(x)} = \overline{st_1} \lessdot s_0 + 1 + j$. From (3.43), for $\forall i \in \{1, .., j\}.\delta_1(x_i) = s_0 + i < s_0 + 1 + j$. Therefore,

$$\forall x \in dom(\Gamma_1).\overline{\delta_1(x)} \lessdot s_0 + 1 + j$$

TS: (c)
For $\forall i \in \{1,..,k\}$, we take $\sigma_{ji} \overset{S}{\sim} \sigma_j$ where $S = dom(\sigma_j) - (\overline{st_1} \cup \{s_1, ..., s_j\})$, such that

$$\begin{aligned} \sigma_{ji}(st_1) &= w_i \\ \sigma_{ji}(s_1) &= \langle n_1 \rangle \\ &\vdots \\ \sigma_{ji}(s_j) &= \langle n_j \rangle \end{aligned}$$

It is easy to show that

$$\sigma_{j1} \overset{S}{\sim} \sigma_{j2} \overset{S}{\sim} ... \overset{S}{\sim} \sigma_{jk} \overset{S}{\sim} \sigma_j \tag{3.51}$$

$$\sigma_{j1} \overset{S}{\bowtie} \sigma_{j2} \overset{S}{\bowtie} ... \overset{S}{\bowtie} \sigma_{jk} = \sigma_j \tag{3.52}$$

Also note that

$$S_{in} = \mathtt{fv}(p_1) \subseteq (\overline{st_1} \cup \{s_1, ..., s_j\}) \cap S = \emptyset \tag{3.53}$$

$$\overline{st_2} \subseteq (\overline{st_1} \cup \{s_1, ...s_j\} \cup \mathtt{dv}(p_1)) \cap S = \emptyset \tag{3.54}$$

From $\mathcal{R}_i$ in (3.47) we have $\rho_i(x) \triangleright_{\Gamma_1(x)} \sigma_{ji}(\delta_1(x))$
and it is clear that

$$\begin{aligned} \rho_i(x_1) &\triangleright_{\Gamma_1(x_1)} \sigma_{ji}(\delta_1(x_j)) \\ &\vdots \\ \rho_i(x_j) &\triangleright_{\Gamma_1(x_j)} \sigma_{ji}(\delta_1(x_j)) \end{aligned}$$

Therefore, $\forall x \in dom(\Gamma_1).\rho_i(x) \triangleright_{\Gamma_1(x)} \sigma_{ji}(\delta_1(x))$.
Then by IH (k times) on $\mathcal{T}_1$ with $\mathcal{E}_i$, $\mathcal{C}_1$ we obtain the following result:

$$(\langle p_1, \sigma_{ji} \rangle \Downarrow^{\langle () \rangle} \sigma'_{ji})_{i=1}^{k} \tag{3.55}$$

$$(v'_i \triangleright_{\tau_2} \sigma'_{ji}(st_2))_{i=1}^{k} \tag{3.56}$$

$$(\sigma'_{ji} \overset{<s_0+j+1}{=\!=\!=\!=\!=\!=\!=\!=} \sigma_{ji})_{i=1}^{k} \tag{3.57}$$

$$s_0 + 1 + j \leq s_1 \tag{3.58}$$

$$\overline{st_2} \lessdot s_1 \tag{3.59}$$

Assume $S_{out} = \{s_{j+1}, ..., s_{j+l}\}$.(Note here s_{j+i} is not necessary equal to $s_j + i$, but must be $\geq s_j$).

There are two possibilities for $\mathcal{P}_{j+1}$:

– Subcase $\sigma_j(s_0) = \langle\rangle$, i.e., $k = 0$.
Then $(\sigma_j(s_i) = \langle\rangle)_{i=1}^{j}$. Also, with (3.4) and (3.5), we have $\forall s \in \overline{st_1}.\sigma_j(s) = \langle\rangle$; with (3.6), $\sigma_j(s_b) = \langle \mathtt{T} \rangle$. Thus

$$\forall s \in (\{s_0\} \cup S_{in}).\sigma_j(s) = \langle\rangle$$

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

$$\frac{}{\langle S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1), \sigma_j \rangle \Downarrow^{\langle () \rangle} \sigma_j[(s_{j+i} \mapsto \langle\rangle)_{i=1}^{l}]}$$

So in this subcase, we take

$$\sigma' = \sigma_j[(s_{j+i} \mapsto \langle\rangle)_{i=1}^{l}] = \sigma[s_0 \mapsto \langle\rangle, s_1 \mapsto \langle\rangle, ..., s_{j+l} \mapsto \langle\rangle] \tag{3.60}$$

TS: (viii)
Since $k = 0$, then $v = \{\}$. Also, we have

$$\sigma'(s_b) = \sigma(s_b) = \langle \mathtt{T} \rangle$$
$$\forall s \in \overline{st_2}.\sigma'(s) = \langle\rangle$$

Therefore, $\sigma'((st_2, s_b)) = (\sigma'(st_2), \sigma'(s_b))$, with which we construct

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

as required.

– Subcase $\sigma_j(s_0) = \langle()|...\rangle$, i.e., $k > 0$.
Since we have (3.51), (3.55) and $\mathtt{fv}(()p_1) \cap S = \emptyset$ from (3.53), it is easy to show that using Lemma 3.16 at most (k-1) times we can obtain

$$\left\langle p_1, (\overset{S}{\bowtie} \sigma_{ji})_{i=1}^{k} \right\rangle \Downarrow^{\langle()_1,...,()_k\rangle} (\overset{S}{\bowtie} \sigma'_{ji})_{i=1}^{k} \tag{3.61}$$

Let $\sigma'' = (\overset{S}{\bowtie} \sigma'_{ji})_{i=1}^{k}$. Also with (3.52), we replace both the start and ending stores in (3.61), giving us a derivation $\mathcal{P}'_{j+1}$ of

$$\langle p_1, \sigma_j \rangle \Downarrow^{\langle()_1,...,()_k\rangle} \sigma''$$

Now we build $\mathcal{P}_{j+1}$ using the rule P-Wc-Nonemp as follows:

$$\frac{\begin{array}{c}\mathcal{P}'_{j+1} \\ \langle p_1, \sigma_j \rangle \Downarrow^{\langle()_1,...,()_k\rangle} \sigma''\end{array}}{\langle S_{out} := \mathtt{WithCtrl}(s_0, S_{in}, p_1), \sigma_j \rangle \Downarrow^{\langle()\rangle} \sigma_j[(s_{j+i} \mapsto \sigma''(s_{j+i}))_{i=1}^{l}]}$$

So in this subcase we take

$$\begin{aligned}\sigma' &= \sigma_j[(s_{j+i} \mapsto \sigma''(s_{j+i}))_{i=1}^{l}] \\ &= \sigma[s_0 \mapsto \langle()_1, ..., ()_k\rangle, s_1 \mapsto \langle\overbrace{n_1, ..., n_1}^{k}\rangle, ..., s_j \mapsto \langle\overbrace{n_j, ..., n_j}^{k}\rangle, \\ &\quad s_{j+1} \mapsto \sigma''(s_{j+1}), ..., s_{j+l} \mapsto \sigma''(s_{j+l})]\end{aligned} \tag{3.62}$$

TS : (viii)
Let $\sigma'(st_2) = w'$, and $\sigma'_{ji}(st_2) = w'_i$.
For $\forall i \in \{1, ..., k\}$, by Definition 3.11 with (3.54), we get

$$w' = \sigma''(st_2) = w'_1 ++ ... ++ w'_k$$

Also, $\sigma'(s_b) = \sigma(s_b) = \langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle$, we now have $\sigma'((st_2, s_b)) = (\sigma'(st_2), \sigma'(s_b)) = (w', \langle \mathtt{F}_1, ..., \mathtt{F}_k, \mathtt{T} \rangle)$. With (3.56), we can construct $\mathcal{R}$ as follows:

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

as required.

(ix) TS: $\sigma' \overset{<s_0}{=\!=} \sigma$

Since $\forall s \in \{s_0\} \cup \{s_1, ..., s_j\} \cup \{s_{j+1}, ..., s_{j+l}\}.s \geq s_0$, with (3.60) and (3.62), it is clear $\forall s < s_0.\sigma'(s) = \sigma(s)$, i.e., $\sigma' \overset{<s_0}{=\!=} \sigma$ as required.

(x) TS: $s_0 \leq s_1$
From (3.58) we immediately get $s_0 \leq s_1 - 1 - j < s_1$.

(xi) TS: $\overline{(st_2, s_b)} \lessdot s_1$
From (3.46) we know $s_b < s_0$, thus $s_b < s_0 \leq s_1$. And we already have (3.59). Therefore,

$$\overline{(st_2, s_b)} = \overline{st_2} +\!+ [s_b] \lessdot s_1.$$

- Case $e = x$.
We must have

$$\mathcal{T} = \frac{}{\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau)$$
$$\mathcal{E} = \frac{}{\rho \vdash x \downarrow v}\,(\rho(x) = v)$$
$$\mathcal{C} = \frac{}{\delta \vdash x \Rightarrow^{s_0}_{s_0} (\epsilon, st)}\,(\delta(x) = st)$$

So $p = \epsilon$.
Immediately we have $\mathcal{P} = \dfrac{}{\langle \epsilon, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma}$
So $\sigma' = \sigma$, which implies $\sigma' \overset{<s_0}{=\!=} \sigma$.
From the assumptions (iv),(v) and(vi) we already have $v \triangleright_\tau \sigma(st)$, and $\overline{st} \lessdot s_0$.
Finally it's clear that $s_0 \leq s_0$, and we are done.

- Case $e = \textbf{let } x = e_1 \textbf{ in } e_2$.
We must have:

$$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma \vdash e_1 : \tau_1} \qquad \overset{\mathcal{T}_2}{\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau}}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau}$$

$$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\rho \vdash e_1 \downarrow v_1} \qquad \overset{\mathcal{E}_2}{\rho[x \mapsto v_1] \vdash e_2 \downarrow v}}{\rho \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \downarrow v}$$

$$\mathcal{C} = \frac{\overset{\mathcal{C}_1}{\delta \vdash e_1 \Rightarrow^{s_0}_{s_0'} (p_1, st_1)} \qquad \overset{\mathcal{C}_2}{\delta[x \mapsto st_1] \vdash e_2 \Rightarrow^{s_0'}_{s_1} (p_2, st)}}{\delta \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Rightarrow^{s_0}_{s_1} (p_1; p_2, st)}$$

So $p = p_1; p_2$.

By IH on $\mathcal{T}_1$ with $\mathcal{E}_1, \mathcal{C}_1$, we get

(a) $\mathcal{P}_1$ of $\langle p_1, \sigma \rangle \Downarrow^{\langle () \rangle} \sigma_1$
(b) $\mathcal{R}_1$ of $v_1 \triangleright_{\tau_1} \sigma_1(st_1)$
(c) $\sigma_1 \overset{<s_0}{=\!=} \sigma$

(d) $s_0 \leq s_0'$
(e) $\overline{st_1} \lessdot s_0'$

From (b), we know $\rho[x \mapsto v_1](x) : \Gamma[x \mapsto \tau_1](x)$ and $\rho[x \mapsto v_1](x) \triangleright_{\Gamma[x \mapsto \tau_1](x)} \sigma_1(\delta[x \mapsto st_1](x))$ must hold. From (e), we have $\overline{\delta[x \mapsto st_1](x)} \lessdot s_0'$.
Then by IH on $\mathcal{T}_2$ with $\mathcal{E}_2, \mathcal{C}_2$, we get

(f) $\mathcal{P}_2$ of $\langle p_2, \sigma_1 \rangle \Downarrow^{\langle () \rangle} \sigma_2$
(g) $\mathcal{R}_2$ of $\sigma_2 \triangleright_\tau \sigma_2(st)$
(h) $\sigma_2 \overset{<s_0'}{=\!=} \sigma_1$
(i) $s_0' \leq s_1$
(j) $\overline{st} \lessdot s_1$

So we can construct:

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

From (c), (d) and (h), it is clear that $\sigma_2 \overset{<s_0}{=\!=} \sigma_1 \overset{<s_0}{=\!=} \sigma$. From (d) and (i), $s_0 \leq s_1$.
Take $\sigma' = \sigma_2$ (thus $\mathcal{R} = \mathcal{R}_2$) and we are done.

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

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

$$\mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ \vdash \ (v_1, ..., v_k)(\downarrow) \downarrow v \end{array}}{\rho \ \vdash \ \phi(x_1, ..., x_k) \downarrow v} \left((\rho(x_i) = v_i)_{i=1}^k\right)$$

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

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

(iv) $\vdash \ \rho(x_i) : \Gamma(x_i)$, that is, $\vdash \ v_i : \tau_i$
(v) $\overline{\delta(x_i)} \lessdot s_0$, that is, $\overline{st_i} \lessdot s_0$
(vi) $\rho(x_i) \triangleright_{\Gamma(x_i)} \sigma(st_i)$, that is, $v_i \triangleright_{\tau_i} \sigma(st_i)$

So using Lemma 3.18 on $\mathcal{T}_1, \mathcal{E}_1, \mathcal{C}_1, (a), (b)$ and (c) gives us exactly what we shall show.

■

3.8 Scaling up

Chapter 4

Conclusion

Bibliography

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

[Ble95] Guy E Blelloch. Nesl: A nested data-parallel language.(version 3.1). Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1995.

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

[Mad13] Frederik M. Madsen. A streaming model for nested data parallelism. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, March 2013.

[Mad16] Frederik M. Madsen. *Streaming for Functional Data-Parallel Languages*. PhD thesis, Department of Computer Science (DIKU), University of Copenhagen, September 2016.

[PP93] Jan F Prins and Daniel W Palmer. Transforming high-level data-parallel programs into vector operations. In *ACM SIGPLAN Notices*, volume 28, pages 119–128. ACM, 1993.