Rate Types for Stream Programs

Thomas W. Bartenstein and Yu David Liu

SUNY Binghamton
Binghamton, NY13902, USA
{tbarten1, davidL}@binghamton.edu

Abstract

We introduce RATE TYPES, a novel type system to reason about and optimize data-intensive programs. Built around stream languages, RATE TYPES performs static quantitative reasoning about *stream rates* — the frequency of data items in a stream being consumed, processed, and produced. Despite the fact that streams are fundamentally dynamic, we find two essential concepts of stream rate control — *throughput ratio* and *natural rate* — are intimately related to the program structure itself and can be effectively reasoned about by a type system. RATE TYPES is proven to correspond with a time-aware and parallelism-aware operational semantics. The strong correspondence result tolerates arbitrary schedules, and does not require any synchronization between stream filters. We further implement RATE TYPES, demonstrating its effectiveness in predicting stream data rates in real-world stream programs.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications — Data-flow languages; D.4.8 [*Programming Languages*]: Performance — Modeling and Prediction; F.3.3 [*Theory of Computation*]: Studies of Program Constructs — Type structure

Keywords Stream programming; data processing rates; data throughput; performance reasoning; type systems

1. Introduction

Big Data and parallelism are two dominant themes in modern computing, both of which call for language support offered naturally by the stream programming model. A *stream program* consists of data-processing units (called *filters*) connected by paths to indicate the data flow. At run time, each such path is occupied by an ordered, potentially large sequence of data items, called *data streams*. Stream programming — together with its close relatives of signal programming and (more generally) dataflow programming — are successful in scientific computing [1], graphics [2], databases [3], GUI design [4, 5], robotics [6], sensor networks [7], and network switches [8]. Its growing popularity also generates significant interest in developing theoretical foundations for stream programming [9–11].

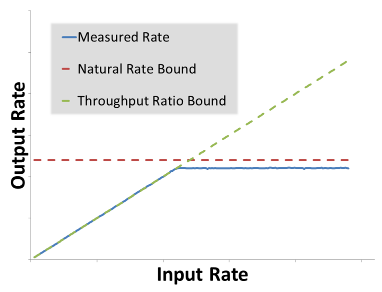


Figure 1. Throughput Ratio and Natural Rate

In this paper, we develop a novel theoretical foundation and practical implementation to reason about the *data rate* aspect of stream programming. The centerpiece of this work is RATE TYPES, a type system to reason about *stream rates*, *i.e.*, how frequently data can be consumed, processed, and produced. Despite the fundamentally dynamic nature of streams, we show that two crucial characteristics of stream applications can be reasoned about:

Throughput Ratio: the relative ratio between the output stream rate and the input stream rate of a stream program.

Natural Rate: the output stream rate of a stream program given an unlimited input stream rate.

We use Figure 1 to illustrate these concepts. Intuitively, as the the rate of data flowing into the stream program (the *input stream rate*) increases, the rate of data flowing out the same program (the *output stream rate*) should increase. The proportionality of increase is represented by the throughput ratio, with a bound illustrated by the slope of the slanted dotted (green) line in Figure 1. However, the output stream rate cannot grow infinitely even if the input stream rate

OOPSLA '14, October 20–24, 2014, Portland, Oregon, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10...$15.00.
http://dx.doi.org/10.1145/2660193.2660225

keeps increasing, because *it takes time to process data*, *i.e.*, taking data from the input stream, performing calculations on that data, and putting the results on the output stream. Eventually, the output stream will reach a "plateau" — the bound of the natural rate — as illustrated by the horizontal dotted (red) line in Figure 1. This phenomenon is supported by our experiments. The solid (blue) curve in the same Figure illustrates the experimentally measured output stream rates during a real program execution, as measured for many different input stream rates.

Our key insight is that both throughput ratio and natural rate are closely related to the *program structure*, which in turn can be effectively reasoned about by type systems. RATE TYPES models both concepts as types, and provides a unified type checking and inference framework to help answer a wide range of performance-related questions, such as whether a video "decoder" stream program can produce 3241 data items (*e.g.*, pixels) a second when being fed with 1208 raw data items per second. Make no mistake: it would be unreasonable to expect such performance-focused questions to be answered completely without any knowledge of the run-time. What is less obvious — and what RATE TYPES illuminates — is how *little* such knowledge is required to enable full-fledged reasoning, so that crucial performance questions such as data throughput can largely be answered analytically rather than experimentally. Along this line, RATE TYPES is an instance of *quantitative reasoning* about performance-related properties, an active area of research (*e.g.*, [12–15]), with a focus on *data-flow* programming models. To the best of our knowledge, RATE TYPES is the first result for stream rate reasoning, in a general context that requires neither static scheduling [16] nor inter-filter synchronization [17].

RATE TYPES promotes a type-theoretic approach to reason about data rates, bringing benefits long known to type system research to the emerging application domain of data-intensive software: (1) type systems excel at relating and propagating information characteristic of program structures, throughput ratio and natural rate in our system; (2) type systems have strong support for modularity, which in our case spearheads a flavor of *compositional* performance reasoning; (3) type systems have "standard" and provably correct ways to construct and connect a series of algorithms — such as establishing the connection between type checking and type inference, and determining principal types — which in RATE TYPES happen to unify many interesting practical algorithms in stream rate control.

To bring RATE TYPES closer to real-world computing, we implement our type inference system on top of the StreamIt language [18], predicting data throughput under different settings of data input/output stream rates. Our experimental results demonstrate close resemblance between the reasoning results and the measured results.

This paper makes the following contributions:

- It develops a type system to reason about throughput ratio and natural rate of stream programs, and formally establishes the correlation between the stream rates from the reasoning system and those manifest at run time – as a strong correspondence property between the static and the dynamic.
- It defines a type inference to infer the throughput ratio and the natural rate. The inference is sound and complete relative to type checking, and further enjoys principal typing — the existence of upper bound for throughput ratio and natural rate.
- It describes a prototyped implementation of the type system, with experimental results demonstrating the effectiveness of the formal reasoning in predicting runtime stream program behaviors.

2. Stream Programming and Reasoning

We now outline the basics of stream programming, and informally describe how RATE TYPES can help reason about stream programs. Our type system can be built around a variety of programming languages. Here we choose StreamIt [18] as the host language, and discuss other applicable languages at the end of the section as well as in later sections of the paper.

Stream Programming Figure 2 outlines an example stream program for *simulated annealing* [19], a classic optimization algorithm that probabilistically finds globally optimal solutions using a randomized locally optimal search. Given seed coordinates as the input stream, this program fragment (entry at `anneal` in Line 3) takes each input coordinate, checks its 8 neighbors in the 2D space, and picks the coordinate in the neighborhood with best benefit (`checkNeighbors` in Line 10). This coordinate is fed back for the next round of space search. The neighborhood-based strategy may trap the search to a local, but not global, optimality. Thus, the program has with a "jump" strategy to allow some coordinates to be randomly mutated (`randomJump` in Line 45).

The base processing unit of a stream program is a *filter*, like `getNeighbors` in Line 15, whose body is a function labeled with keyword `work`. A filter execution instance, informally called a *filter firing*, takes in a pre-defined finite number of data items from the input stream (through `pop`) and places a finite number of data items to the output stream (through `push`). For instance in Lines 17-21, the filter places 9 coordinates on the output stream for each coordinate it reads from the input stream. A filter can only be launched when there are enough data items on the input stream, as specified by the `pop` declaration in the filter signature (the declarations immediately after the keyword `work`).

A filter such as `getNeighbors` can execute in parallel with a different filter, such as `evalNeighbors`, as long as each filter has sufficient data items in their respective input streams, Similar to other concurrency models such as

```
1  struct xy { int x; int y; int ben; }
2
3  xy→xy feedbackloop anneal() {
4     join roundrobin(1,99);
5     body checkNeighbors;
6     loop randomJump;
7     split roundrobin(1,99);
8  }
9
10 xy→xy pipeline checkNeighbors() {
11    add getNeighbors;
12    add computeProfits;
13    add evalNeighbors;
14 }
15 xy→xy filter getNeighbors() {
16     work push 9 pop 1 {
17         xy p = pop();
18         push(p);
19         push({p.x+1, p.y, p.ben});
20         push({p.x, p.y+1, p.ben});
21         ...
22 }}
23 xy→xy splitjoin computeProfits() {
24     split roundrobin(1,1);
25     add getProfit;
26     add getProfit;
27     join roundrobin(1,1);
28 }
29 xy→xy filter getProfit() {
30     work push 1 pop 1 {
31         xy loc =pop();
32         loc.ben= ...; // find profit
33         push(xy);
34 }}
35 xy→xy filter evalNeighbors() {
36     work push 1 pop 9 {
37         xy p1 = pop(),
38            p2=pop(),
39            ...
40            p9 = pop();
41         xy best = ...; // best of 9
42         push(best);
43 }}
44
45 xy→xy filter randomJump() {
46     work push 1 pop 1 {
47         xy p = pop();
48         xy rand = ...;
49         if (...) push(p)
50         else push(rand);
51 }}
```

Figure 2. A StreamIt Program for Simulated Annealing

actors [20], a filter execution instance abides by a "one firing at a time" rule: the evaluation of each filter function body upon application must be completed before the second filter execution instance can start.

To stay neutral to the terminology of host languages, we name the 3 most commonly used filter combinators as follows:

- *Chain* (`pipeline` in StreamIt): connects the output stream of one sub-program to the input stream of another. For example, `checkNeighbors` in Line 10 "chains" the output stream of `getNeighbors` with the input stream of `computeProfits`.
- *Diamond* (`splitjoin` in StreamIt): dissembles and assembles data streams. For example, `computeProfits` in Line 23 says that the data items on the input stream will be alternatively placed to the input streams of the two `getProfit` execution instances, whose respective output streams will be alternatively selected to assemble the output stream of `computeProfits`. Declaration `roundrobin (1,1)` indicates a 1:1 alternation.
- *Circle* (`feedbackloop` in StreamIt): a combinator to support data feedback. For example, `anneal` in Line 3 says that for every 100 coordinates produced by `checkNeighbors`, 99 are fed back for `random-Jump` processing. Every time 99 coordinates are produced by `randomJump`, 1 more new coordinate (additional "seed" coordinates) will be admitted for annealing.

Stream Reasoning For stream applications such as simulated annealing, high performance is often a matter of necessity due to copious data volume. High on the wish list of a data engineer is the ability to reason about performance, with questions such as:

Q1: Is it possible for a program to sustain the production of n_2 data items per second when its input is fed with n_1 items per second?

Q2: Is it possible for a program to sustain the production of n data items per second given unlimited rates for data inputs?

Q3: If a program is targeted at producing n data items per second, what is the minimal rate of feeding data at its input?

Q4: Given the program is fed with n data items per second, what is the expected rate for its data production?

RATE TYPES addresses **Q1-2** through type checking, and **Q3-4** through type inference. It further demonstrates the relationship among these questions in a unified, provably correct framework.

For example, through answering **Q4**, RATE TYPES is capable of demonstrating that the output rate is limited both by the rate at which data items are supplied to the program, and by the inherent limits of the program sub-components. Overall, it provides a rigorous explanation to the shape of the curve we introduced in Figure 1. Consider a simple filter such as `getProfit` at line 29 in Figure 2 and assume it takes three seconds from the pop to the push. If data arrives at the input stream every five seconds, say at times 3, 8, 13, ..., then the filter will process that data and write to the output stream once every five seconds as well, for instance at times 6, 11, 16, ... Even though these times are offset from the input, the *rate* at which they appear is still directly proportional to the input rate. If, however, data arrive at `getProfit` every two seconds, say at times 5, 7, 9, ..., then the performance of the `getProfit` filter dominates the output rate, and pushes to the output stream will occur at times 8, 10, 12, ..., once every three seconds. RATE TYPES generalizes this simple concept to reason about the stream rates throughout the program, and shows that an entire program can be characterized by the proportion of the input rate to the output rate, or the *throughput ratio*, as well as an internal stream program "speed limit" - the maximum possible output rate, independent from the input stream rate, which we call the *natural rate* of the program.

The formal sections of this paper define a stream-oriented language core, and formally define

- an operational semantics (Section 3) where the notion of stream rates can be established;
- a type checking system (Section 4.1) where *throughput ratio* and *natural rate* can be reasoned about, *e.g.*, answering questions such as if one filter has throughput ratio of $\frac{1}{3}$ and the other has throughput ratio of $\frac{1}{4}$, then the program with the two filters chained together should have throughput ratio of $\frac{1}{12}$. The type checking system is important for compositional reasoning, in that throughput ratio and natural rate of a program can still be reasoned about, even though the implementation of sub-stream graphs of the program may not be known (as long as their types are known).
- a type inference algorithm (Section 4.2) where the dependencies between the rates of individual streams in a stream program runtime are abstracted as linear constraints. The type inference algorithm is important for answering questions such as what the maximum throughput ratio and natural rate are for a program.

The paper further establishes several important properties of RATE TYPES. To highlight a few:

- Static/Dynamic Correspondence (Theorem 1): we show that RATE TYPES correctly captures the dynamic behavior as defined by the operational semantics: specifically, both the throughput ratio and natural rate we reason about statically are a faithful approximation of dynamic stream rates.
- Soundness and Completeness of Inference (Theorem 3 and Theorem 4): we show that the inference algorithm correctly infers the throughput ratio and natural rate of a program, and can infer any throughput ratio and natural rate for which the type checking algorithm can establish a type derivation.
- Principal Typing of Inference (Theorem 5): the maximum throughput ratio and natural rate exist.

Assumption Every reasoning framework needs to address the *base case* of reasoning: to type an arithmetic expression, the assumption is that integers are of `int` type; to verify a program is secure, one needs to know password strings are properly associated with `high` security labels. In RATE TYPES, the base case is the filter, and the assumption we make is its execution time can be predetermined and specified.

At a first glance, this assumption may seem counter-intuitive to what is conventionally considered as "static." We however believe this is reasonable because (a) filter behaviors are much more predictable than full-fledged programs, thanks to the non-shared memory model they routinely adopt and their lack of side effects often called for in real-world stream languages [18, 21, 22]; (b) formal systems to reason about individual filter behaviors exist [10]; (c) Experimentally, filter execution time is known to be stable through profiling; Core optimizations of the StreamIt compiler [23] rely on it; (d) real-world software development is iterative. Profiling-guided typing is not new [24]. What ultimately matters is to help programmers reason about performance at some point during the software life cycle.

Applicability RATE TYPES is primarily designed for expressive and general-purpose stream languages. The framework may be applied more broadly to systems where data processing is periodic, and/or where rates matter: (a) sensor network languages (*e.g.*, [7]), where determining the lowest sensing rate possible is relevant; (b) signal languages such as FRP [6]. Even though the input signals are theoretically continuous in this context, practical implementations are still concerned with sampling rate. In addition, even if all input signals are continuous — a case analogous to **Q2** — the output signal is still discrete where rates may matter. (c) high-performance-oriented dataflow composition frameworks such as FlumeJava [25] and ParaTimer [26], where single MapReduce-like units are composed together in expressive ways. In Section 6, we will have a more in-depth discussion on how RATE TYPES can be mapped to a variety of existing frameworks.

3. Syntax and Dynamic Semantics

Abstract Syntax The abstract syntax of our language is defined as follows:

$$\begin{array}{rcll} P & ::= & F^L[n_i, n_o] \mid P \rhd^{\ell} P' \mid & \textit{program} \\ & & P \Diamond_{\delta,\alpha} P' \mid P \circlearrowleft^{\ell',\ell}_{\alpha,\delta} P' & \\ \delta & ::= & \langle n; n' \rangle & \textit{distribution factor} \\ \alpha & ::= & \langle n; n' \rangle & \textit{aggregation factor} \end{array}$$

The building block of a stream program, a filter, takes the syntactic form of $F^L[n_i, n_o]$, where F is the filter function body, and L is a unique program label called a *filter label*. Filter labels range over the set of $\mathbb{FLAB}$. Each filter is further declared with two natural numbers: n_i for the number of items to be consumed by an invocation of the filter, and n_o for the number of items to be produced by an invocation. The two numbers correspond to the `pop` and `push` declarations in Figure 2. Let $\mathbb{NAT}$ represent natural numbers starting from 1. Metavariable n ranges over $\mathbb{NAT}$. For each filter, we further require F to be an element of $\mathbb{DATA}^{n_i} \rightarrow \mathbb{DATA}^{n_o}$ where $\mathbb{DATA}$ is the set of data items.

The remaining syntactic forms of a program P correspond to the three combinators in stream programming: a chain composition, a diamond composition, and a circle composition; represented in the grammar in that order. If a program P takes the form of one of these combinators, we informally say that P_A and P_B are *sub-programs* of P. For the circle composition, $P_A \circlearrowleft^{\ell',\ell}_{\alpha,\delta} P_B$, we further informally call P_A the *forward* sub-program and P_B the *feedback* sub-program.

For both diamond and circle compositions, metavariables $\delta = \langle n; n' \rangle$ and $\alpha = \langle n; n' \rangle$ represent the *distribution factor* and the *aggregation factor* respectively. They correspond to the tuples succeeding the `split` keyword and the `join` keyword in the Figure 2 example respectively. Intuitively, the distribution factor specifies the proportion of splitting a data stream into two, and the aggregation factor specifies that of joining two data streams into one.

For the purpose of formal development, each chain expression is associated with a distinct *stream label*, ℓ, and each circle expression is associated with two stream labels. The motivation of representing them in the syntax will be made clear later. Metavariable ℓ ranges over set $\mathbb{SLAB}$. Given a program P, we use $\texttt{flabels}\,(P)$ to enumerate all filter labels in P, and $\texttt{slabels}\,(P)$ to enumerate all stream labels in P. The `flabels` definition is obvious, and `slabels` is inductively defined as follows:

$$\begin{aligned}
\texttt{slabels}\left(F^L[n_i, n_o]\right) &\triangleq \emptyset \\
\texttt{slabels}\left(P_A \rhd^{\ell} P_B\right) &\triangleq \texttt{slabels}\,(P_A) \cup \\
&\quad\ \texttt{slabels}\,(P_B) \cup \{\ell\} \\
\texttt{slabels}\,(P_A \Diamond_{\delta,\alpha} P_B) &\triangleq \texttt{slabels}\,(P_A) \cup \texttt{slabels}\,(P_B) \\
\texttt{slabels}\left(P_A \circlearrowright^{\ell',\ell}_{\alpha,\delta} P_B\right) &\triangleq \texttt{slabels}\,(P_A) \cup \\
&\quad\ \texttt{slabels}\,(P_B) \cup \{\ell, \ell'\}
\end{aligned}$$

In Appendix A, we show the simple syntax core is capable of encoding other programming patterns, such as k-way split-join and non-round-robin distribution/aggregation.

The program in Figure 2 can be represented by our syntax as $P_{\texttt{anneal}}$ where:

$$\begin{aligned}
P_{\texttt{anneal}} =& P_{\texttt{checkNeighbors}} \circlearrowright^{\ell_0,\ell_1}_{\langle 1;99\rangle,\langle 1;99\rangle} P_{\texttt{randomJump}} \\
P_{\texttt{checkNeighbors}} =& (P_{\texttt{getNeighbors}} \rhd^{\ell_2} P_{\texttt{computeProfits}}) \\
& \rhd^{\ell_3} P_{\texttt{evalNeighbors}} \\
P_{\texttt{getNeighbors}} =& F^{L_1}[1, 9] \\
P_{\texttt{computeProfits}} =& F^{L_2}[1, 1] \Diamond_{\langle 1;1\rangle;\langle 1;1\rangle} F^{L_3}[1, 1] \\
P_{\texttt{evalNeighbors}} =& F^{L_4}[9, 1] \\
P_{\texttt{randomJump}} =& F^{L_5}[1, 1]
\end{aligned}$$

In the rest of the paper, we use notation $[X_1, \ldots, X_n]$ to represent a sequence with elements X_1, ..., X_n, or simply $\overline{X}$ when sequence length does not matter. Furthermore, we define $|[X_1, \ldots, X_n]| \overset{\text{def}}{=} n$ and use comma for sequence concatenation, *i.e.* $[X_1, \ldots, X_n], [Y_1, \ldots, Y_{n'}] \overset{\text{def}}{=} [X_1, \ldots, X_n, Y_1, \ldots Y_{n'}]$. We call a sequence in the form of $[X_1 \mapsto Y_1, \ldots X_n \mapsto Y_n]$ a mapping sequence if X_1, ..., X_n are distinct. We equate two mapping sequences if one is a permutation of elements of the other. Let mapping sequence $M = [X_1 \mapsto Y_1, \ldots X_n \mapsto Y_n]$. We further define $\texttt{dom}\,(M) \overset{\text{def}}{=} \{X_1, \ldots, X_n\}$ and $\texttt{ran}\,(M) \overset{\text{def}}{=} \{Y_1, \ldots, Y_n\}$ and use notation $M(X_i)$ to refer Y_i for any $1 \leq i \leq n$. Binary operator $\uplus$ is defined as $M_1 \uplus M_2 = M_1, M_2$ iff $\texttt{dom}\,(M_1) \cap \texttt{dom}\,(M_2) = \emptyset$. The operator is otherwise undefined.

$$\frac{|S| < (n + n')}{S \curlywedge_{\langle n;n'\rangle} \emptyset; \emptyset}$$

$$\frac{\begin{array}{c} S_2 = S_{A2}, S_{B2} \\ |S_{A2}| = n \qquad |S_{B2}| = n' \\ S_1 \curlywedge_{\langle n,n'\rangle} S_{A1}, S_{B1} \end{array}}{S_1, S_2 \curlywedge_{\langle n;n'\rangle} S_{A1}, S_{A2}; S_{B1}, S_{B2}}$$

$$\frac{(|S_A| < n) \vee (|S_B| < n')}{S_A; S_B \curlyvee_{\langle n;n'\rangle} \emptyset}$$

$$\frac{\begin{array}{c} |S_{A2}| = n \qquad |S_{B2}| = n' \\ S_2 = S_{A2}, S_{B2} \\ S_{A1}, S_{B1} \curlyvee_{\langle n;n'\rangle} S_1 \end{array}}{S_{A1}, S_{A2}; S_{B1}, S_{B2} \curlyvee_{\langle n;n'\rangle} S_1, S_2}$$

Figure 3. Distribution and Aggregation

Stream Runtime The following structures are used for defining the stream runtime:

R	$::=$	$\overline{\ell \mapsto S}$	*program runtime*
S	$::=$	$\overline{d}$	*stream*
ℓ	$\in$	$\mathbb{SLAB} \cup \{\ell_{\texttt{IN}}, \ell_{\texttt{OUT}}\}$	*stream label*
t	$\in$	$\mathbb{TIME} \subseteq \mathbb{REAL}+$	*time*
d	$\in$	$\mathbb{DATA}$	*data item*
Π	$\in$	$\mathbb{FLAB} \mapsto \mathbb{TIME}$	*filter time mapping*

The runtime, R, consists of a mapping sequence from stream labels to streams. A stream is defined as a list of data items. Two built-in stream labels, $\ell_{\texttt{IN}}$ and $\ell_{\texttt{OUT}}$ are used to facilitate the semantics development. Given a program P, $\ell_{\texttt{IN}}$ and $\ell_{\texttt{OUT}}$ labels the input stream and the output stream of P. Mapping function Π maps each filter (label) to its execution time.

Ternary predicate $S \curlywedge_{\delta} S_A, S_B$ holds when S can be "split" to S_A and S_B using distribution factor δ, and $S_A, S_B \curlyvee_{\alpha} S$ holds when S_A and S_B can be "joined" together as S with aggregation factor α. Their formal definitions are in Figure 3. Some examples should demonstrate the functionality of these predicates.

$$\begin{array}{rcl}
[1, 2, 3, 4, 5, 6] & \curlywedge_{\langle 1;2\rangle} & [1, 4], [2, 3, 5, 6] \\
[1, 4], [2, 3, 5, 6] & \curlyvee_{\langle 1;2\rangle} & [1, 2, 3, 4, 5, 6] \\
[1, 2, 3, 4, 5, 6, 7] & \curlywedge_{\langle 1;2\rangle} & [1, 4], [2, 3, 5, 6] \\
[1, 4, 7], [2, 3, 5, 6] & \curlyvee_{\langle 1;2\rangle} & [1, 2, 3, 4, 5, 6]
\end{array}$$

$$\frac{}{R \xrightarrow[t]{P} R}\ \text{[D-Reflex]} \qquad \frac{R \xrightarrow[t_1]{P} R' \quad R' \xrightarrow[t_2]{P} R'' \quad t_1 + t_2 \leq t}{R \xrightarrow[t]{P} R''}\ \text{[D-Trans]}$$

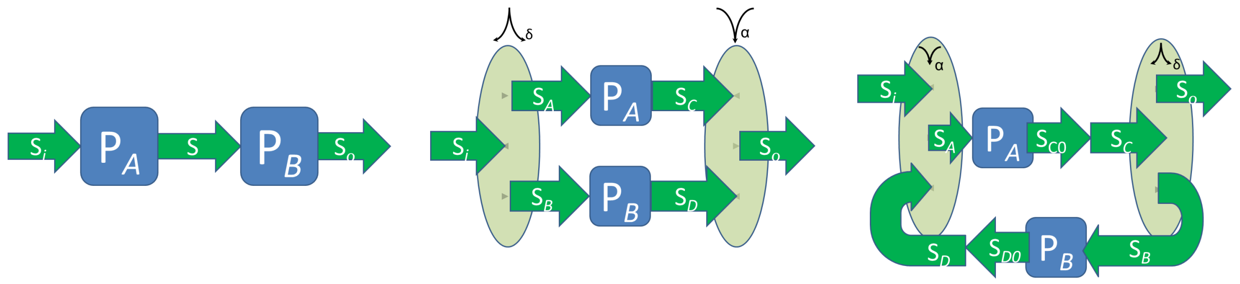
$$\frac{S_{o2} = F(S_{i1}) \quad |S_{i1}| = n_i \quad |S_{o2}| = n_o \quad \Pi(L) \leq t}{[\ell_{\mathtt{IN}} \mapsto (S_{i2}, S_{i1}), \ell_{\mathtt{OUT}} \mapsto S_{o1}] \xrightarrow[t]{F^L[n_i, n_o]} [\ell_{\mathtt{IN}} \mapsto S_{i2}, \ell_{\mathtt{OUT}} \mapsto (S_{o2}, S_{o1})]}\ \text{[D-Filter]}$$

$$\frac{\begin{array}{c} R = R_A \uplus R_B \uplus [\ell_{\mathtt{IN}} \mapsto S_i, \ell_{\mathtt{OUT}} \mapsto S_o, \ell \mapsto S] \\ R' = R'_A \uplus R'_B \uplus [\ell_{\mathtt{IN}} \mapsto S'_i, \ell_{\mathtt{OUT}} \mapsto S'_o, \ell \mapsto S'] \\ S = S_2, S_1 \qquad\qquad S' = S_3, S_2 \\ R_A \uplus [\ell_{\mathtt{IN}} \mapsto S_i, \ell_{\mathtt{OUT}} \mapsto S_2] \xrightarrow[t_A]{P_A} R'_A \uplus [\ell_{\mathtt{IN}} \mapsto S'_i, \ell_{\mathtt{OUT}} \mapsto S'] \quad t_A \leq t \\ R_B \uplus [\ell_{\mathtt{IN}} \mapsto S, \ell_{\mathtt{OUT}} \mapsto S_o] \xrightarrow[t_B]{P_B} R'_B \uplus [\ell_{\mathtt{IN}} \mapsto S_2, \ell_{\mathtt{OUT}} \mapsto S'_o] \quad t_B \leq t \end{array}}{R \xrightarrow[t]{P_A \rhd^{\ell} P_B} R'}\ \text{[D-Chain]}$$

$$\frac{\begin{array}{c} R = R_A \uplus R_B \uplus [\ell_{\mathtt{IN}} \mapsto S_i, \ell_{\mathtt{OUT}} \mapsto S_o] \\ R' = R'_A \uplus R'_B \uplus [\ell_{\mathtt{IN}} \mapsto S'_i, \ell_{\mathtt{OUT}} \mapsto S'_o] \\ S_i \curlywedge_\delta S_A, S_B \qquad S'_i \curlywedge_\delta S'_A, S'_B \qquad S_C, S_D \curlyvee_\alpha S_o \qquad S'_C, S'_D \curlyvee_\alpha S'_o \\ R_A \uplus [\ell_{\mathtt{IN}} \mapsto S_A, \ell_{\mathtt{OUT}} \mapsto S_C] \xrightarrow[t_A]{P_A} R'_A \uplus [\ell_{\mathtt{IN}} \mapsto S'_A, \ell_{\mathtt{OUT}} \mapsto S'_C] \quad t_A \leq t \\ R_B \uplus [\ell_{\mathtt{IN}} \mapsto S_B, \ell_{\mathtt{OUT}} \mapsto S_D] \xrightarrow[t_B]{P_B} R'_B \uplus [\ell_{\mathtt{IN}} \mapsto S'_B, \ell_{\mathtt{OUT}} \mapsto S'_D] \quad t_B \leq t \end{array}}{R \xrightarrow[t]{P_A \diamond_{\delta,\alpha} P_B} R'}\ \text{[D-Diamond]}$$

$$\frac{\begin{array}{c} R = R_A \uplus R_B \uplus [\ell_{\mathtt{IN}} \mapsto S_i, \ell_a \mapsto S_{C0}, \ell_b \mapsto S_{D0}, \ell_{\mathtt{OUT}} \mapsto S_o] \\ R' = R'_A \uplus R'_B \uplus [\ell_{\mathtt{IN}} \mapsto S'_i, \ell_a \mapsto S'_{C0}, \ell_b \mapsto S'_{D0}, \ell_{\mathtt{OUT}} \mapsto S'_o] \\ S_C \curlywedge_\delta S_o, S_B \qquad S_i, S_D \curlyvee_\alpha S_A \qquad S'_C \curlywedge_\delta S'_o, S'_B \qquad S'_i, S'_D \curlyvee_\alpha S'_A \\ R_A \uplus [\ell_{\mathtt{IN}} \mapsto S_A, \ell_{\mathtt{OUT}} \mapsto (S_{C0}, S_C)] \xrightarrow[t_A]{P_A} R'_A \uplus [\ell_{\mathtt{IN}} \mapsto S'_A, \ell_{\mathtt{OUT}} \mapsto (S'_{C0}, S'_C)] \quad t_A \leq t \\ R_B \uplus [\ell_{\mathtt{IN}} \mapsto S_B, \ell_{\mathtt{OUT}} \mapsto (S_{D0}, S_D)] \xrightarrow[t_B]{P_B} R'_B \uplus [\ell_{\mathtt{IN}} \mapsto S'_B, \ell_{\mathtt{OUT}} \mapsto (S'_{D0}, S'_D)] \quad t_B \leq t \end{array}}{R \xrightarrow[t]{P_A \circlearrowright^{\ell_a, \ell_b}_{\alpha,\delta} P_B} R'}\ \text{[D-Circle]}$$

Figure 4. Operational Semantics



(a) Chain (b) Diamond (c) Circle

Figure 5. Operational Semantics Illustrations

Operational Semantics Figure 4 defines operational semantics, with relation $R \xrightarrow[t]{P} R'$ denoting that the runtime of program P transitions from R to R' in time t. To model stream rates explicitly, we need to (1) "count the beans," *i.e.*, the number of data items on the input/output streams, and (2) be aware of time. Since parallelism affects how time is accounted for, we elect to explicitly consider the impact of parallelism for every expression. (In contrast, standard operational semantics for concurrent languages typically employs one single "context" rule to capture non-determinism.)

The reduction relation is reflexive and transitive. [D-Filter] relies on a pre-defined Π to obtain the execution time of each filter. The reduction takes n_i data items from the "tail" of the input stream, applies function F to it, and places n_o number of data items on the "head" of the output stream. The rest of the rules are illustrated in Figure 5, where streams are illustrated as arrows and labeled with metavariables appearing in rules. By convention, each time we identify a pre-reduction semantic element with a symbol, we use the same symbol with an apostrophe to indicate its corresponding element after reduction.

Before we look into the details the last 3 rules, there are two important high-level observations. First, the data flow dependency of the two sub-programs P_A and P_B – be it in a chain, diamond, or circle – does not prevent parallelism. The reduction time for each rule is only bound by the longer reduction of P_A and P_B. In particular, we wish to stress that even in the case of chain composition — which is sometimes termed as "serial composition" in existing literature — parallelism still exists between the two sub-programs. Second, the reduction system does not require synchronization over any sub-programs. For all three composition forms, a non-reflexive reduction can happen to one sub-program independently, given the other sub-program takes a [D-Reflex] step. In other words, sub-programs of a stream program – including all filters – can operate asynchronously, and no predefined schedules [16] are required.

In [D-Chain], ℓ represents the stream in between P_A and P_B, *i.e.*, both the output stream of P_A and the input stream of P_B. In [D-Diamond], $\curlywedgedownarrow$ allows one stream to be "viewed" as two, whereas $\curlyveedownarrow$ allows two streams to be "viewed" as one. This idea of "views" is inspired by lens [27], as demonstrated in Figure 5(b)

As illustrated in Figure 5(c), a circle composition intuitively looks like a "reverse" diamond composition, in that there is a "join" at input, and a "split" at output. The input stream (S_i) and the output stream of the feedback sub-program (S_D) is "joined" to form the input stream of the forward sub-program, whose output stream (S_C) is "split" to the output stream (S_o) and the input stream of the feedback sub-program (S_B). The thorny issue is after reduction, the additional data items produced by P_A and P_B need to be properly represented. Unlike [D-Diamond], we cannot further "lens" them because that would lead to the next iteration of loop reduction. To address this, we introduce an imaginary stream between P_A and the lensed stream, labeled ℓ_A, and represented by S_{AA} in Figure 5(c) — and use this stream "buffer" for the additional data produced by P_A reduction. The same scheme is used for treating the post-reduction output of P_B. The stream labels of the two additional introduced streams, S_{AA} and S_{BB}, are the two labels associated with the circle construct, *i.e.*, ℓ_a and ℓ_b.

Properties The operational semantics enjoys several simple properties, which we state now. The proofs for all lemmas and theorems throughout this paper can be found in [28].

Lemma 1 (Stream Count Preservation)**.** If $R \xrightarrow[t]{P} R'$, then $\mathtt{dom}\,(R) = \mathtt{dom}\,(R') = \mathtt{slabels}\,(P) \cup \{\ell_{\mathtt{IN}}, \ell_{\mathtt{OUT}}\}$.

Lemma 2 (Monotonicity of Input and Output Streams)**.** If $R \xrightarrow[t]{P} R'$, then $|R(\ell_{\mathtt{IN}})| \geq |R'(\ell_{\mathtt{IN}})|$, and $|R(\ell_{\mathtt{OUT}})| \leq |R'(\ell_{\mathtt{OUT}})|$.

Stream Rates Our operational semantics is friendly for calculating stream rates. First, let us formalize the notion of how fast the size of a stream changes:

Definition 1 (Stream Rates)**.** Given a reduction from R to R' over time t, the rate for stream ℓ is defined by function $\mathtt{rchange}()$:

$$\mathtt{rchange}(R, R', t, \ell) \stackrel{\text{def}}{=} \frac{\mathtt{abs}\,(|R'(\ell)| - |R(\ell)|)}{t}$$

where unary operator $\mathtt{abs}\,()$ computes the absolute value.

For instance during a reduction of 8 seconds, if the size of a stream increases from 25 to 45, then the stream rate is $\frac{|45-25|}{8} = 2.5$ data items per second. Stream rates range over non-negative floating point numbers, $\mathbb{FLOAT}+$, and from this point on, we will use metavariable r to represent a stream rate.

Observe that according to Lemma 2, the input stream of a program through a reduction is monotonically decreasing, whereas the output stream of a program through a reduction is monotonically increasing. Hence we define:

Definition 2 (Input/Output Stream Rates)**.** Given a reduction from R to R' over time t, we define:

$$\begin{aligned} \mathtt{rti}\,(R, R', t) &\stackrel{\text{def}}{=} \mathtt{rchange}(R, R', t, \ell_{\mathtt{IN}}) \\ \mathtt{rto}\,(R, R', t) &\stackrel{\text{def}}{=} \mathtt{rchange}(R, R', t, \ell_{\mathtt{OUT}}) \end{aligned}$$

Bootstrapping A technical detail for a program with circle compositions is that one needs to *prime* the loop. For instance, the `anneal` circle composition in Figure 2 cannot start until the output stream of `randomJump` contains 99 items. In practice, most languages allow programmers to specify the initialization data items to "prime" the loop. To model this, we say R is a *primer* of P iff $\mathtt{dom}\,(R)$ subsumes $\ell_{\mathtt{IN}}$ and the smallest set of ℓ where $P_A \circlearrowright^{\ell', \ell}_{\alpha, \delta} P_B$ is a sub-program of P, $\alpha = \langle n; n' \rangle$ and $|R(\ell)| = n'$. Here, stream label ℓ' intuitively identifies the output stream of the forward sub-program (S_{C0} in Figure 5(c)), and ℓ for the output stream of the feedback sub-program (S_{D0} in Figure 5(c)).

Definition 3 (Bootstrapping Runtime)**.** Given program P, and primer R_0, function $\mathtt{init}\,(P, R_0)$ computes the initial runtime of P, defined as the smallest R satisfying the following conditions: $R_0 \subseteq R$, $R(\ell_{\mathtt{OUT}}) = \emptyset$, and $R(\ell) = \emptyset$ for any $\ell \in \mathtt{slabels}\,(P) \wedge \ell \notin \mathtt{dom}\,(R_0)$.

4. Rate Types

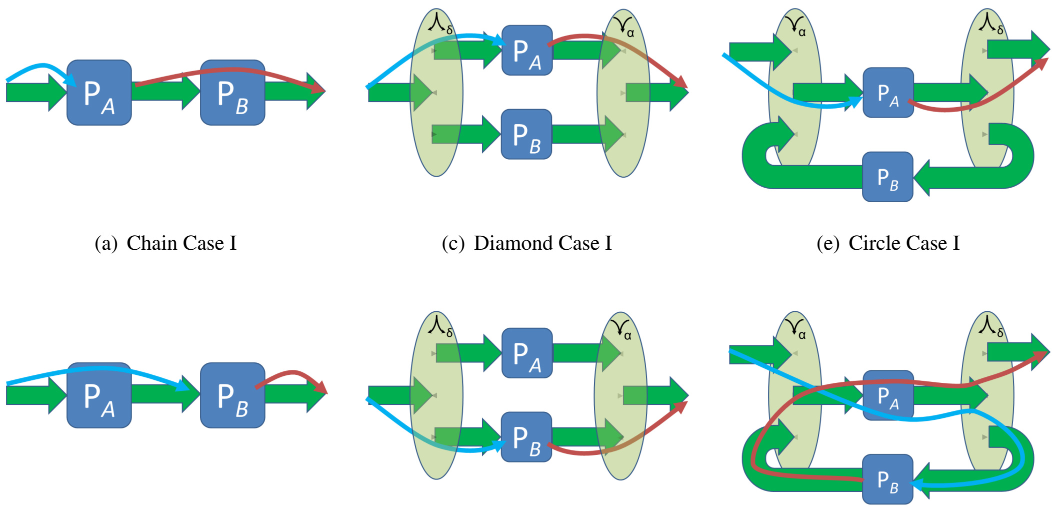
In this section, we describe our type system. We first present a type checking algorithm, followed by a type inference

$$\frac{\vdash_{\mathrm{t}} P : \tau' \qquad \tau' <: \tau}{\vdash_{\mathrm{t}} P : \tau}\;[\text{T-Sub}] \qquad \frac{\vdash_{\mathrm{t}} P_A : \langle \theta_a ; \nu_a \rangle \qquad \vdash_{\mathrm{t}} P_B : \langle \theta_b ; \nu_b \rangle}{\vdash_{\mathrm{t}} P_A \rhd^{\ell} P_B : \langle \theta_a \times \theta_b ; \mathtt{min}\,(\nu_a \times \theta_b, \nu_b) \rangle}\;[\text{T-Chain}]$$

$$\frac{}{\vdash_{\mathrm{t}} F^L[n_i, n_o] : \langle n_o/n_i ; n_o/\Pi(L) \rangle}\;[\text{T-Filter}] \qquad \frac{\begin{array}{cc} \vdash_{\mathrm{t}} P_A : \langle \theta_a ; \nu_a \rangle & \vdash_{\mathrm{t}} P_B : \langle \theta_b ; \nu_b \rangle \\ \theta'_a = \curlywedge^1(\delta) \times \theta_a \times \Upsilon^1(\alpha) & \nu'_a = \nu_a \times \Upsilon^1(\alpha) \\ \theta'_b = \curlywedge^2(\delta) \times \theta_b \times \Upsilon^2(\alpha) & \nu'_b = \nu_b \times \Upsilon^2(\alpha) \end{array}}{\vdash_{\mathrm{t}} P_A \diamond_{\delta,\alpha} P_B : \langle \mathtt{min}\,(\theta'_a, \theta'_b) ; \mathtt{min}\,(\nu'_a, \nu'_b) \rangle}\;[\text{T-Diamond}]$$

$$\frac{\theta_2 \le \theta_1 \qquad \nu_2 \le \nu_1}{\langle \theta_1 ; \nu_1 \rangle <: \langle \theta_2 ; \nu_2 \rangle}\;[\text{Sub}] \qquad \frac{\begin{array}{c} \vdash_{\mathrm{t}} P_A : \langle \theta_a ; \nu_a \rangle \qquad \vdash_{\mathrm{t}} P_B : \langle \theta_b ; \nu_b \rangle \\ \theta'_a = \mathtt{min}\left(\Upsilon^1(\alpha), \theta'_b \times \Upsilon^2(\alpha)\right) \times \theta_a \\ \theta'_b = \theta'_a \times \curlywedge^2(\delta) \times \theta_b \\ \nu'_a = \mathtt{min}\left(\nu_a, \nu'_b \times \Upsilon^2(\alpha) \times \theta_a\right) \\ \nu'_b = \mathtt{min}\left(\nu_b, \nu'_a \times \curlywedge^2(\delta) \times \theta_b\right) \end{array}}{\vdash_{\mathrm{t}} P_A \circlearrowleft^{\ell',\ell}_{\alpha,\delta} P_B : \langle \theta'_a \times \curlywedge^1(\delta) ; \nu'_a \times \curlywedge^1(\delta) \rangle}\;[\text{T-Circle}]$$

Figure 6. Rate Type Checking Rules



(a) Chain Case I (c) Diamond Case I (e) Circle Case I

(b) Chain Case II (d) Diamond Case II (f) Circle Case II

Figure 7. Reasoning about Throughput Ratios and Natural Rates (For throughput ratio reasoning, follow both the blue/lighter arrows and the red/darker arrows. For natural rate reasoning, follow the red/darker arrows.)

algorithm proven to be sound and complete relative to the former. Types in our system are defined as follows:

τ	$::=$	$\langle \theta ; \nu \rangle$	*stream rate type*
θ	$\in$	$\mathbb{TR} \subseteq \mathbb{FLOAT}+$	*throughput ratio*
ν	$\in$	$\mathbb{FLOAT}+$	*natural rate*

The *throughput ratio*, θ, statically characterizes the ratio of the output stream rate over the input stream rate. The *natural rate*, ν, statically approximates the output stream rate when the program can "naturally" produce output, *i.e.*, with no limitation on the input stream rate.

The type form here reveals a fundamental phenomenon of stream rate control: the input stream rate and the output stream rate can be correlated by a ratio θ, but the correlation only holds when the input stream rate is low enough such that its corresponding output stream rate does not reach ν. Just as we explained in Section 1, each filter execution takes time, and each filter instance can only take "one firing at a time." The combined effect is that a program simply cannot — in practice or in theory — produce output streams at an unlimited rate.

4.1 Type Checking

Judgment $\vdash_{\mathrm{t}} P : \tau$ denotes P has type τ. This is directly related to two questions in Section 2. Question **Q1** attempts to determine whether a program P can sustain the production of n_2 data items per second when its input is fed with n_1 items per second. That question is tantamount to finding out whether a derivation exists for $\vdash_{\mathrm{t}} P : \langle n_2/n_1 ; n_2 \rangle$. Question **Q2** — determining whether the output stream can sustain rate n with no limitation on the input stream rate —

can be answered by finding out whether a derivation exists for $\vdash_{\mathrm{t}} P : \langle\theta; n\rangle$ for some θ.

The typing rules are summarized in Figure 6. Simple functions $\curlywedge^1(\delta)$, $\curlywedge^2(\delta)$, $\Upsilon^1(\alpha)$ and $\Upsilon^2(\alpha)$ can be informally viewed as computing a form of "normalized" distribution and aggregation factors. They are defined as:

$$\begin{array}{lclclcl} \curlywedge^1(\delta) & \overset{\text{def}}{=} & \frac{n}{n+n'} & \quad & \Upsilon^1(\alpha) & \overset{\text{def}}{=} & \frac{n+n'}{n} \\ \curlywedge^2(\delta) & \overset{\text{def}}{=} & \frac{n'}{n+n'} & \quad & \Upsilon^2(\alpha) & \overset{\text{def}}{=} & \frac{n+n'}{n'} \\ \multicolumn{3}{c}{\text{where } \delta = \langle n, n'\rangle} & & \multicolumn{3}{c}{\text{where } \alpha = \langle n, n'\rangle} \end{array}$$

[T-Sub] introduces subtyping. The $<:$ relation in turn is defined in [Sub] rule. Intuitively, if a program can sustain a throughput ratio of 0.4, it can sustain throughput ratio 0.3. In addition, if a program is known to be capable of producing as much as 300 items a second, it can output 200 items a second.

In [T-Filter], the throughput ratio of a filter is simply the ratio between the number of data items placed on the output stream through one filter firing and that of data items consumed by the same firing. Following the "one-firing-at-a-time" execution strategy, the natural rate for a filter is achieved when it runs "non-stop": the filter produces every n_0 items for its execution time $\Pi(L)$. As a result, the natural rate of the filter is $n_0/\Pi(L)$.

As revealed by [T-Chain], the throughput ratio of a chain composition is the *multiplication* of the throughput ratios of the two chaining sub-programs. Consider an example where the throughput ratio of P_A is 3 and that of P_B is 2 — meaning P_A outputs 3 items for each 1 item on the input where as P_B outputs 2 times for each 1 item on its input — the composition program indeed has 6 items on the output for each item on the input.

There are two cases for reasoning about the natural rate for chain composition, illustrated in Figure 7(a) and Figure 7(b), respectively. In the first case, the rate of the input stream to P_A (*i.e.*, the rate for feeding data to P_A) is higher than what P_A can consume following the "one-firing-at-a-time" strategy. Since we know the rate for the output stream of P_A given unlimited input rate is its natural rate ν_A, the output stream rate of P_B — and hence also the output stream rate of the entire program — is no higher than $\nu_A \times \theta_b$. In the second case, Figure 7(b) shows the rate of feeding the input stream of P_B is high enough that the output stream of P_B reaches its natural rate, ν_b. In this case, ν_b becomes the natural rate for the entire program. Combining the two cases, the natural rate of the program should be the minimal of the two, computed by the standard binary function $\texttt{min}\,()$.

In the [T-Diamond] rule, observe that throughput ratio can be viewed as the "normalized" output stream rate relative to the input stream rate, through the two possible paths from input to output. Figures 7(c) and 7(d) demonstrate these cases. Consider Figures 7(c) for example. For simplicity, let us consider the input stream rate of the composed program be 1. Thanks to the simple operators we defined earlier, the input stream rate for P_A is thus $\curlywedge^1(\delta)$, and the corresponding output stream rate of P_A is $\curlywedge^1(\delta) \times \theta_a$. If the output stream rate of P_A determines the output stream rate of the entire program, the throughput ratio thus would be $\curlywedge^1(\delta) \times \theta_a \times \Upsilon^1(\alpha)$. On the other hand, if the output stream rate of P_B determines the output stream rate of the entire program, the throughput ratio would be $\curlywedge^2(\delta) \times \theta_b \times \Upsilon^2(\alpha)$. Overall, the throughput ratio is the minimum of the two. The reasoning for the natural rate is simpler. The key insight is that the natural rate of the program — the upper bound of output stream — depends on the natural rates of P_A and P_B.

To see how [T-Circle] works, we first focus on the reasoning of natural rates. First let us consider how the rate at the output stream of P_A and the rate at the output stream of P_B are bounded. Let the two be ν'_a and ν'_b respectively. Observe that if we can determine ν'_a, the natural rate of the entire program is simply $\nu'_a \times \curlywedge^1(\delta)$. To determine ν'_a, the general philosophy we used for [T-Chain] reasoning can still be applied, with two cases: (1) Figure 7(e): when P_A is fed with an input stream whose rate is so high that its output stream is already the natural rate of P_A, then $\nu'_a = \nu_a$; (2) Figure 7(f): otherwise, ν'_a is determined by the input stream rate of P_A, which in turn is determined by the natural rate of P_B, *i.e.*, $\nu'_b \times \Upsilon^2(\alpha) \times \theta_a$. In the more general case, ν'_a and ν'_b are mutually dependent. Let us consider an iterative scheme where we compute ν'_a and ν'_b in iterations, with superscript k on the left to indicate the number of iterations, then:

$$\begin{aligned} {}^{(k+1)}\nu'_a &= \texttt{min}\left(\nu_a, {}^{(k)}\nu'_b \times \Upsilon^2(\alpha) \times \theta_a\right) \\ {}^{(k+1)}\nu'_b &= \texttt{min}\left(\nu_b, {}^{(k)}\nu'_a \times \curlywedge^2(\delta) \times \theta_b\right) \end{aligned}$$

The convergence of such iterations has been well-studied in control theory as the stability of feedback loop. More general solutions would determine the existence of — and if so compute — the fix point. [T-Circle] adopts a simple scheme, requiring convergence without iteration. From a type system perspective, this implies we may conservatively reject programs whose natural rate may stabilize after iterations. Nonetheless, the simple rule here sufficiently demonstrates our core philosophy: our type system is stability-aware, and programs with unstable circle compositions should be rejected.

The throughput ratio reasoning of [T-Circle] follows similar logic. Here θ'_a intuitively captures the throughput ratio between the output stream of P_A and the input stream of the entire program, and θ'_b intuitively captures the throughput ratio between the output stream of P_B and the input stream of the entire program. The key insight is that circle composition resembles a "reverse diamond composition," an analogy we used while introducing the operational semantics. As a result, the particular assumptions related to throughput ratio reasoning in [T-Circle] (those involving θ'_a and θ'_b in the rule) bear strong resemblance to their counterparts in [T-Diamond].

Meta-Theories RATE TYPES correctly captures the dynamic behavior as defined by the operational semantics: specifically, both the throughput ratio and natural rate we reason about statically is a faithful approximation of dynamic stream rates:

Theorem 1 (Static/Dynamic Correspondence from an Initial State)**.** Given a program P

- If $R = \mathtt{init}(P, R_0)$ for some R_0 and $R \xrightarrow[t]{P} R'$, then $\vdash_{\mathrm{t}} P : \langle r_1/r_2; r_1\rangle$ where $\mathtt{rto}(R, R', t) = r_1$ and $\mathtt{rti}(R, R', t) = r_2$.
- If $\vdash_{\mathrm{t}} P : \langle\theta; \nu\rangle$, then for any small real numbers, ϵ_1, ϵ_2, there exist some R_0 and some reduction $R \xrightarrow[t]{P} R'$ such that $\theta - r_1/r_2 \leq \epsilon_1$ and $\nu - r_1 \leq \epsilon_2$ where $R = \mathtt{init}(P, R_0)$ and $r_1 = \mathtt{rto}(R, R', t)$ and where $r_2 = \mathtt{rti}(R, R', t)$.

This theorem relates program dynamic behaviors and typing. The first part of the theorem is analogous to the type system completeness property, whereas the second part is analogous to the soundness property.

The first part of the theorem says that if an execution exhibits a particular input/output data rate, we can use the observed rates to compute the throughput ratio and natural rate, and the program should typecheck given the computed type. The importance of this part of the theorem is its contrapositive: if we cannot type a program given a throughput ratio and natural rate, then no matter how many times the program is executed (*e.g.*, through testing), the same throughput ratio and natural rate will not occur at run time.

The second part of the theorem says if a program is indeed typable, then at least one execution sequence exists to exhibit a throughput ratio and natural rate "close enough" to the ones used for typing. The definition indeed says types are the *limitation* of the observed executions. Readers might wonder why we cannot always find an execution sequence whose throughput ratio and natural rate are *equal* to the ones used by the type system. The root cause lies with the beginning steps of the stream program execution: *it takes steps for a stream program to produce the first data item on the output.* Our theorem says, given the execution sequence is long enough – also implying the input data stream contains enough data items — the (detrimental) effect on output stream rate during the initial execution stage will be minimized, so that the throughput ratio and natural rate can be (nearly) reached.

Theorem 1 is a strong result, but it requires relating a runtime state with the initial state. Will the same theorem hold for two arbitrary runtime states? The answer is negative. Observe that each stream program runtime may contain "intermediate streams," *i.e.*, streams that are not identified by $\ell_{\mathtt{IN}}$ and $\ell_{\mathtt{OUT}}$. For instance, in a simple program that only involves chaining two filters together, there is an intermediate stream connecting the two filters. Such intermediate streams may "buffer" data, potentially leading to localized "bursty" behaviors. We now state a stronger result saying that the throughput ratio and natural rate are effective in characterizing arbitrary reduction steps as well, as long as a *steadfast* condition is met:

Theorem 2 (Static/Dynamic Correspondence over Arbitrary Reduction Steps)**.** Given a program P

- If $R \xrightarrow[t]{P} R'$ then $\vdash_{\mathrm{t}} P : \langle r_1/r_2; r_1\rangle$ where $r_1 = \mathtt{rto}(R, R', t)$, and $r_2 = \mathtt{rti}(R, R', t)$, and for any $\ell \in \mathtt{dom}(R) - \{\ell_{\mathtt{IN}}, \ell_{\mathtt{OUT}}\}$, $\mathtt{rchange}(R, R', t, \ell) = 0$
- If $\vdash_{\mathrm{t}} P : \langle\theta; \nu\rangle$, then for any small ϵ_1, ϵ_2, there exists a reduction $R \xrightarrow[t]{P} R'$ such that $\theta - r_1/r_2 \leq \epsilon_1$ and $\nu - r_1 \leq \epsilon_2$ where $\mathtt{rto}(R, R', t) = r_1$ and $\mathtt{rti}(R, R', t) = r_2$, and for any $\ell \in \mathtt{dom}(R) - \{\ell_{\mathtt{IN}}, \ell_{\mathtt{OUT}}\}$, $\mathtt{rchange}(R, R', t, \ell) = 0$.

Here we call the $\mathtt{rchange}()$ assumption in the theorem the *steadfast* condition. The theorem says that a reduction sequence beginning at *any* reduction step observes the throughput ratio and natural rate reasoned about by our type system, as long as the size of each intermediate stream at the starting step is the same as its size at the end of the reduction sequence. Since $\xrightarrow[t]{P}$ is transitive, this theorem does not require every small step maintain steadfastness: it only requires the end state is "steadfast" relative to the beginning step. In other words, the theorem is tolerant of temporary "bursty" behaviors during the reduction(s) from R to R'.

4.2 Rate Type Inference

In this section, we define a constraint-based type inference for RATE TYPES. The key element is *throughput ratio type variable* $p \in \mathbb{TVAR}$ and *natural rate type variable* $q \in \mathbb{NVAR}$, the type variable counterparts of θ and ν. Each element in the set is either an equality constraint ($e \doteq e$) or an inequality one ($e \preceq e$) over expressions formed by type variables and arithmetic expressions over them, such as multiplication ($\bullet$). To avoid confusion, we use different symbols for syntactic elements in constraints and those in predicates, whose pairwise relationships should be obvious: $\doteq$ and $=$, $\preceq$ and $\leq$, $\bullet$ and $\times$. We define a *solution*, σ, as a mapping from type variables to throughput ratios and natural rates. We use predicate $\sigma \Downarrow \Sigma$ to indicate that σ is a solution to Σ. Formally, the predicate holds if every constraint is a tautology for a set identical to Σ, except that every occurrence of p is substituted with $\sigma(p)$, and q with $\sigma(q)$.

Type inference rules are given in Figure 8. Judgment $\vdash_i P : \langle p; q\rangle\backslash\Sigma$ says program P is inferred to have throughput ratio represented by type variable p and natural rate represented by type variable q under constraint Σ. The rules have a one-to-one correspondence with the type checking rules we introduced in Figure 6. Indeed, the close relationship between the two can be formally established:

$$\frac{p, q \text{ fresh}}{\vdash_i F^L[n_i, n_o] : \langle p; q\rangle \backslash \{p \preceq n_o/n_i, q \preceq n_o/\Pi(L)\}}\ \text{[I-Filter]}$$

$$\frac{\vdash_i P_A : \langle p_a; q_a\rangle\backslash\Sigma_A \quad \vdash_i P_B : \langle p_b; q_b\rangle\backslash\Sigma_B \quad p, q \text{ fresh}}{\vdash_i (P_A \rhd^{\ell} P_B) : \langle p; q\rangle\backslash\Sigma_A \cup \Sigma_B \cup \{p \preceq p_a \bullet p_b, q \preceq q_a \bullet p_b,\ q \preceq q_b\}}\ \text{[I-Chain]}$$

$$\frac{\begin{array}{c}\vdash_i P_A : \langle p_a; q_a\rangle\backslash\Sigma_A \qquad \vdash_i P_B : \langle p_b; q_b\rangle\backslash\Sigma_B \qquad p, q \text{ fresh}\\ \Sigma_1 = \left\{p \preceq p_a \bullet \Upsilon^1(\alpha) \times \text{⅄}^1(\delta), p \preceq p_b \bullet \Upsilon^2(\alpha) \times \text{⅄}^2(\delta)\right\} \quad \Sigma_2 = \left\{q \preceq q_a \bullet \Upsilon^1(\alpha), q \preceq q_b \bullet \Upsilon^2(\alpha)\right\}\end{array}}{\vdash_i (P_A \diamond_{\delta,\alpha} P_B) : \langle p; q\rangle\backslash\Sigma_A \cup \Sigma_B \cup \Sigma_1 \cup \Sigma_2}\ \text{[I-Diamond]}$$

$$\frac{\begin{array}{c}\vdash_i P_A : \langle p_a; q_a\rangle\backslash\Sigma_A \qquad \vdash_i P_B : \langle p_b; q_b\rangle\backslash\Sigma_B \qquad p, q, p'_a, q'_a, p'_b, q'_b \text{ fresh}\\ \Sigma_0 = \left\{p \preceq p'_a \bullet \text{⅄}^1(\delta), q \preceq q'_a \bullet \text{⅄}^1(\delta)\right\}\\ \Sigma_1 = \left\{p'_a \preceq p_a \bullet \Upsilon^1(\alpha), p'_a \preceq p'_b \bullet p_a \bullet \Upsilon^2(\alpha)\right\} \quad \Sigma_2 = \{p'_b \preceq p'_a \bullet p_b \bullet \text{⅄}^2(\delta)\}\\ \Sigma_3 = \{q'_a \preceq q_a, q'_a \preceq q'_b \bullet p_a \bullet \Upsilon^2(\alpha)\} \quad \Sigma_4 = \{q'_b \preceq q_b, q'_b \preceq q'_a \bullet p_b \bullet \text{⅄}^2(\delta)\}\end{array}}{\vdash_i (P_A \circlearrowleft^{\ell',\ell}_{\alpha,\delta} P_B) : \langle p; q\rangle\backslash\Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4 \cup \Sigma_A \cup \Sigma_B}\ \text{[I-Circle]}$$

Figure 8. Rate Type Inference Rules

Theorem 3 (Soundness of Inference)**.** If $\vdash_i P : \langle p; q\rangle\backslash\Sigma$ and $\sigma \Downarrow \Sigma$ then $\vdash_t P : \langle\sigma(p), \sigma(q)\rangle$.

Theorem 4 (Completeness of Inference)**.** If $\vdash_t P : \langle\theta, \nu\rangle$, then $\exists\sigma$ such that $\sigma(p) = \theta, \sigma(q) = \nu$ and $\sigma \Downarrow \Sigma$, where $\vdash_i P : \langle p; q\rangle\backslash\Sigma$.

A (trivial) solution clearly exists for the constraints produced by the inference: solving all p's and q's to 0. What is more interesting is whether the "best" solution exists: this is the existence of principal typing, a property our type inference algorithm enjoys:

Theorem 5 (Principal Typing)**.** For any P such that $\vdash_i P : \langle p; q\rangle\backslash\Sigma$, there exists a unique σ such that $\sigma \Downarrow \Sigma$, and for any $\sigma' \Downarrow \Sigma$, $\langle\sigma(p); \sigma(q)\rangle <: \langle\sigma'(p); \sigma'(q)\rangle$. We further call $\langle\sigma(p); \sigma(q)\rangle$ the *principal type* of P.

This property has important consequence to stream rate reasoning. Recall in the previous section, subtyping $<:$ is defined by comparing the values of throughput ratios and natural rates. What the theorem here tells us is that there exists the "highest" throughput ratio and natural rate for every program. Given the theorem above, computing the principal type is simple: for any program P, and $\vdash_i P : \langle p; q\rangle\backslash\Sigma$, the prinipal type can be computed by maximizing p and q over constraints Σ.

The two questions — **Q3** and **Q4** — can be easily answered given we can compute principal type for the program. Observe that the principal type provides the highest throughput ratio and natural rate for the program. Let it be $\langle\theta; \nu\rangle$. The answer to Question **Q3** — the minimal input stream rate given the output stream is expected to produce n items per second — is simply n/θ if $n \leq \nu$. The answer to question **Q4** — the expected output stream rate given the input stream rate is n items per second — is either $n \times \theta$ or ν, whichever is less.

5. Experimental Validation

5.1 Implementation

We have implemented RATE TYPES on top of StreamIt version 2.1.1 [18]. Our primary extension lies upon the implementation of the type inference algorithm itself, *i.e.*, calculating the throughput ratio and natural rate of a stream program. In addition, our implementation includes the following components:

1. a profiler to measure the elapsed time consumed by the work function of a filter (to prepare Π in the formalism)
2. a data rate management module for measuring and controlling the input stream rate, and measuring the output stream rate
3. a modification from a file I/O model to a memory-mapped I/O model, in which file pages are populated and pre-faulted before measurements start in order to minimize the impact of file I/O on input or output stream rates

A compiler optimization StreamIt performs is *partitioning*. A stream program typically has many programmer-level filters which StreamIt can partition into a fixed number of work-balanced subgraphs, and mechanically transform ("fuse") each sub-graph into a single filter. Each partition can then be deployed to a physical thread. In order to preserve the multi-threading model adopted by StreamIt, we choose to apply our type inference on the post-partitioned stream program. We modified StreamIt to enable profiling on either pre-partition (programmer level) or post-partitioning (core level) filters. We also ensure each partition is appropriately deployed and executed on a unique core. Our experiments consider two partitioning scenarios: 8 partitions/cores and 16 partitions/cores.

5.2 Benchmarks and Platforms

Our selection of benchmarks include both micro-benchmarks and a number of existing StreamIt programs [29]. For micro-benchmarking, we focused on demonstrating the effectiveness of reasoning on the four basic stream graph configurations:

- `TRIV-Filter`: a single filter that pops a single value for each firing, performs a (parameterized amount of) calculation, and pops a single result
- `TRIV-Chain`: two `TRIV-Filter`'s in a chain composition
- `TRIV-Diamond`: two `TRIV-Filter`'s in a diamond composition
- `TRIV-Circle`: two `TRIV-Filter`'s in a circle composition

The five real-world StreamIt programs we selected are:

- `bitonic`: an implementation of Batcher's bitonic sort network for sorting power-of-2 sized key sets
- `dct`: an implementation of a two-dimensional 8 x 8 inverse Discrete Cosine Transfer, which transforms a 16 x 16 signal from frequency domain to signal domain
- `fft`: a streamed implementation of a Fast Fourier Transform
- `fm`: a simulation of an FM radio with a multi-band equalizer
- `vocoder`: an implementation of a the speech filter analysis portion of a source-filter model. The analysis includes Fourier analysis, a low-pass filter, a bank of band-pass filters, and a pitch detector

We made minor modifications to these benchmarks, primarily to ensure each benchmark reads input data from a file, and writes output data to a file. We equipped our file reader with the capability to provide a constant number of data items per second, based on a parameter, to provide a controllable stream input rate. Our file writer measures the number of data items written, and the elapsed time from the first to the last write so we can calculate the average stream graph output rate.

For each benchmark, we perform experiments and report results running on two different platforms. The first was an 8-core AMD FX-8150 processor (Bulldozer micro-architecture) running Debian 3.2.46-1 Linux (kernel 3.2.0-4-amd64) and 16GB memory. The second was a 32-core AMD Opteron 6378 processor (Piledriver micro-architecture) running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64) with 64GB memory.

5.3 Experimental Methodologies

Overall, our experiments for each benchmark are designed to demonstrate the relationship between (1) the throughput ratio bound and natural rate bound as computed by our implemented type inference algorithm, and (2) the output stream rates measured at various different input stream rates. For example, the "throughput ratio bound" line and the "natural rate bound" line in Figure 1 we showed at the beginning of the paper are plotted by applying the type inference algorithm on TRIV-Filter. The "measured rate" line in the same Figure are computed by measuring output stream rates at different input stream rates. We now detail several practical issues in the experimentation process.

Measurement Point Selection The first issue is to determine the range of input stream rates we wish to measure. The *maximum measurement input rate* we benchmark is heuristically determined to be the lower of the following two:

- Twice the computed natural rate bound divided by the computed throughput ratio bound
- Twice the measured *fastest practical output rate* divided by the computed throughput ratio bound, where the fastest practical output rate is measured through a test run where no restriction is placed on the input stream

We choose 200 measurement points decrementing from the maximum measurement input rate down to zero in 200 equal steps. In some cases, benchmarks may not reach the maximum measurement input rate due to system-level restrictions. In these cases, we selected the highest reachable input rate as maximum measurement input rates.

When given very low input stream rates, the execution of some benchmarks may take a long time. We choose to stop measurements on a specific benchmark when the execution time exceeds 5 minutes. Even though we sometimes dropped the lowest one or two measurement points in some of our benchmarks, it had no impact on the trends demonstrated in our results.

Batch Size Selection In a stream program execution, each (post-partition) filter runs in a separate thread, deployed on a separate core. The streams flowing in between filters are thus inter-thread data transfers across cores, via a buffer. This leads to some practical issues: first, there is overhead for data movement from one core to another, and second, there is a need for synchronization for shared buffer access. To reduce the overhead from data movement and synchronization, StreamIt performs *batching* at data transfer time: an "upstream" filter places a "batch" of data (instead of just one element) in the shared buffer, before it becomes available for consumption by the "downstream" filter.

The effects of this implementation on our experiments are twofold. On one hand, the effect is similar to the "bursty behaviors" we described for Theorem 2. Given a long execution where the number of processed data far exceeds the size of the batch, the "bursty" behavior from batching should have little effect on the average stream rate, and should not affect our reasoning. On the other hand, the idealized theoretical model does not consider the data transfer and synchronization overhead which will occur in real-world implementations. The selection of batch size in fact has non-trivial effect on this overhead.

Larger batch sizes (*i.e.*, more data per movement) reduces synchronization overhead, but its bursty nature results in downstream idle time, waiting for a full batch. In some cases, especially for stream graphs with circle constructs, a very large batch size can delay execution of downstream filters to the point where the entire stream graph stalls. Large batch sizes also have the effect of reducing parallel activity. Downstream filters must wait for a full batch before starting execution. A smaller batch size resolves these problems, but introduces more synchronization overhead, and hence impacts the ability to achieve theoretical stream rates. We tuned the batch size for our experiments. All experimental results were produced with a batch size of 100 data items.

Overhead and Compensation There are several practical considerations which can cause the measured rates to deviate from the theoretical throughput ratio bound and natural rate bound, including the following:

- The idealized model assumes zero time for data transfers across filters. It further assumes there is no overhead associated with checking whether there are sufficient data available to invoke a filter.
- The idealized model assumes infinite sized stream buffers, with no overhead for synchronization. In practice, buffer size is fixed, resulting in occasional "back pressure" where a filter must wait for available stream buffer space in order to push a new data item. There is also locking and unlocking overhead for synchronizing the transfer of data batches.
- There may also be a limit on the fastest rate that an output stream can be consumed, caused by the mechanics of the output stream I/O. In our experiments, the output stream is consumed by memory mapped file output, so is limited by memory transfer rates.

The impact of these effects are most prominent on filters which run very fast. In order to compensate for these effects, we increase the elapsed times for "fast" filters linearly, by a constant *compensation ratio*. Heuristically, we consider a filter as "fast" if its execution time (per firing) is less than 2 microseconds.

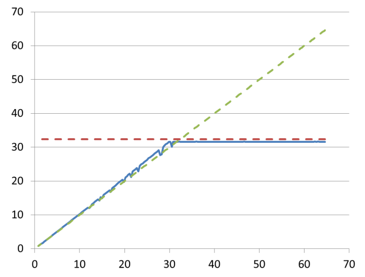
We determine the compensation ratio as follows. We first find whether the "fast" filter happens to be a "limiting" filter: we consider a filter as limiting if changes in its elapsed time have direct impact on the output stream rate of the entire program. We find limiting filters by re-calculating the theoretical output rate under the assumption that the filter in question has a higher natural rate and determining if such a change would lead the type inference algorithm to compute a different output rate. Second, for each filter that is both "fast" and "limiting," we plot an (x, y) point in a graph where x is the natural rate (computed through the inverse of the profiled elapsed time), and y is the observed data production/-consumption rate, obtained through monitoring the changes in low-level data buffers, both in units of "1/microsecond." Third, a linear regression is performed to determine the constant compensation ratio. As expected, the compensation ratio differs from platform to platform, depending factors such as inter-core latency. Our linear regression resulted in the fitting functions $y = x \times 0.686 + 0.028$ on the 8-core machine, and $y = x \times 1.6734 + 0.045$ on the 32-core machine. Our compensation only affects filters that are both fast and limiting.

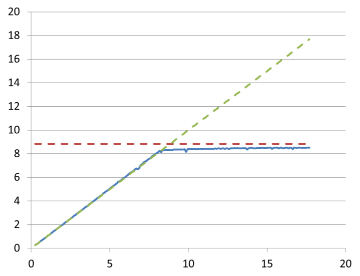
5.4 Experimental Results

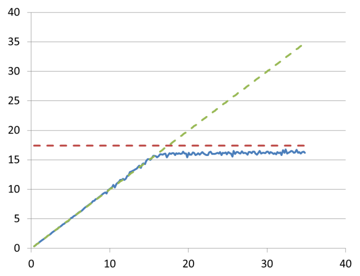
Detailed experimental results are in Figure 9. The results of micro-benchmarks are demonstrated in Figures 9(a)(b)(c)(d), and the results for StreamIt benchmarks are in the rest of the Figures. Overall, we believe the results experimentally confirm the effectiveness of our reasoning framework.

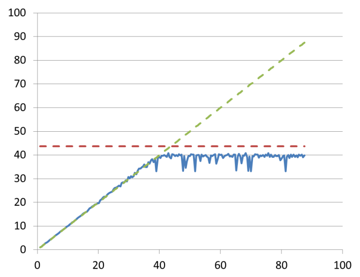
Microbenchmarking — Figures 9(a)(b)(c)(d) — shows very close resemblance between the theoretical result and the measured result: observe that the (blue) solid line lies almost exactly on the boundaries carved out by the two dotted lines. Since micro-benchmarks all have less than 8 filters, the results on both the 8-core machine and the 32-core machine are similar, and we only present the 8-core version. Relatively speaking, TRIV-Circle (and to some extent TRIV-Diamond) experiences more frequent fluctuations, as shown in Figures 9(c)(d). This phenomenon also recurs in real-world StreamIt benchmarks with the same composition operators. Recall that both forms of compositions involve forking and joining of streams. We speculate the fluctuation is related to the behaviors of forking and joining, especially in the presence of batching.

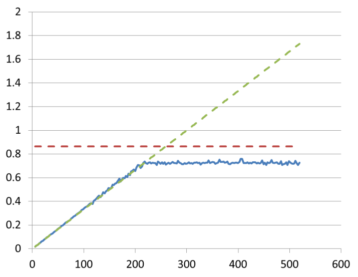
For StreamIt benchmarks, Figures 9(e)(f)(g)(h)(i) show the results when the benchmarks are partitioned into 8 partitions and run on the 8-core machine, whereas Figures 9(j)(k)(l)(m)(n) are their counterparts while being partitioned into 16 partitions and run on the 32-core machine. Even though the 32-core machine has a higher level of parallelism, the 8-core machine runs at a higher frequency (3.6 GhZ) than the 32-core machine (2.4 GhZ), so the output stream rates were typically higher for the 8-core versions. In most cases — the exceptions are perhaps the two `vocoder` benchmark results — the measured rates still closely fit the theoretical results from type inference. For the `fm` benchmark, its 8-core execution is unable to consume more than 25,000 items per second due to I/O restrictions, so we are unable to select the measurement points far to the right on the X-axis, as demonstrated in Figure 9(h). We discussed

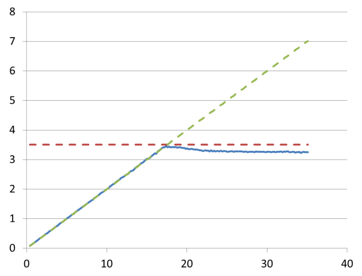

(a) TRIV-Filter

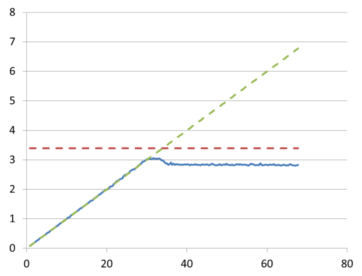

(b) TRIV-Chain

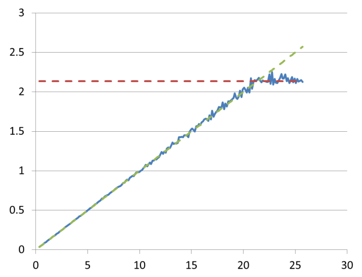

(c) TRIV-Diamond

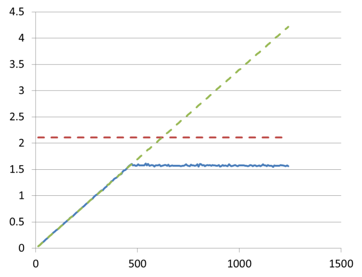

(d) TRIV-Circle

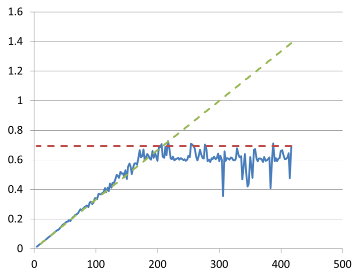

(e) bitonic - 8 filter

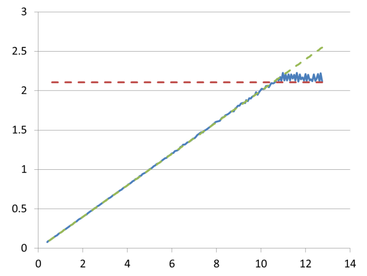

(f) dct - 8 filter

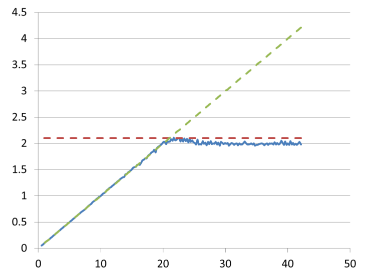

(g) fft - 8 filter

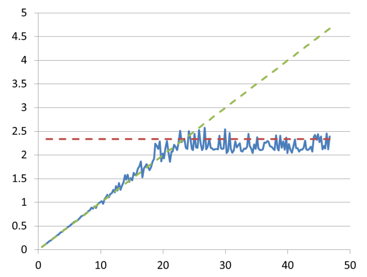

(h) fm - 8 filter

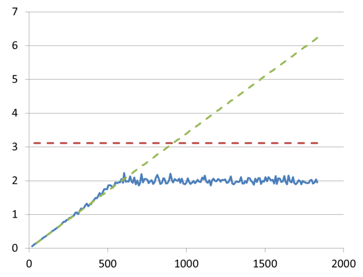

(i) vocoder - 8 filter


(j) bitonic - 16 filter


(k) dct - 16 filter


(l) fft - 16 filter


(m) fm - 16 filter


(n) vocoder - 16 filter

▬ Measured Rate
╌ Natural Rate Bound
╌ Throughput Ratio Bound

Figure 9. Benchmark Results (X axis: input stream rate in 1000 items/second; Y axis: output stream rate in 1000 items/second)

this issue in the "measurement point selection" subsection. Note that this issue goes away for its 16-core execution in Figure 9(m). The converse is true for the `dct` benchmark, which runs at all input rates for the 8-core execution, but is limited for the 16-core execution.

The least satisfying results are from the `vocoder` benchmark, in Figures 9(i)(n). We investigated the runtime of this benchmark, and found that the filter that limits the overall output rate is being fed by a very high volume of input stream data as compared with the other benchmarks. On the 16-core version, this filter has 7.5M data items per second available, but can only consume 2.3M data items per second. The computed natural rate says it should be able to consume 2.9M data items per second. We surmise that the high stream data rates incur a higher batching and synchronization costs than expected.

6. Generality and Limitations

In this section, we place RATE TYPES in a broader context by evaluating the potential opportunities and limitations of building a similar type system on top of several data-oriented programming languages other than our current implementation. Our intent is to examine the generality of RATE TYPES by probing how syntactic, semantic, and implementation choices made by RATE TYPES impact its applicability.

Table 1 summarizes our investigation, where each row characterizes a representative data-oriented programming language based on columns that delineate syntactic or semantic design choices of RATE TYPES. These design choices are:

- *Synchronous Input* - characterizes the filter firing mechanism where filter execution is triggered by the existence of sufficient input data items for an instance of filter execution.
- *Static Items per Firing* - identifies the design feature in which the number of data items a filter will consume and produce for each instance of execution is statically known.
- *Non-Blocking Memory Access* - indicates whether memory access within a filter execution is block-free, *i.e.*, without waits or locks.
- *Graph Composition* - explores whether the *Chain*, *Diamond*, *Circle* operators that RATE TYPES supports are sufficient to capture the stream graph composition mechanisms associated with the language.

The languages we investigated, including several languages not traditionally branded as "stream" languages, are selected from a variety of sources from recent literature. The *Category* column groups these languages into categories. The "stream" category is closest to our context, where high-volume, high-throughput, potentially infinite data flow through program-defined data processing units and data paths. The "signal" category is historically used in contexts where continuous signals flow through program-defined processing units and paths. Both stream languages and signal languages belong to the broader category of dataflow languages — those characterized by the invocation of functional units based on the presence of input data, rather than an imperative control flow. In dataflow languages, data flow through program-defined data processing units and data paths, even though data items might be finite, and their volume and throughput may or may not be high. Toward the end of Section 2, we described why signal or dataflow languages may potentially benefit from RATE TYPES.

We next describe the design choices in more detail in order to assess their impact on the generality of RATE TYPES.

Impacts of Semantic Choices on Generality Two basic requirements of synchronous dataflow (SDF) languages are *Synchronous Input* and *Static Items per Firing*. Many languages we studied have not been categorized in the literature as SDF languages, but as Table 1 shows, most support these two basic semantic requirements.

In most cases, data processing units (*i.e.*, filters in RATE TYPES) are modeled as functions, and data read from the input stream are modeled as function parameters. Furthermore, these languages have no capability to access input data streams other than through parameters. The RATE TYPES *Synchronous Input* requirement is trivially satisfied by a combination of these two factors. The requirement of *Static Items per Firing* is also satisfied thanks to the function-based filter design. Most languages align the per-firing input data items and the output data items with function parameters and return values, whose sizes are statically known through function signatures.

The notable exception is StreamFlex, where input/output streams are first-class values (called *channels*) within the scope of filters. Data stream read is a first-class expression in StreamFlex, and hence in the most general interpretation of StreamFlex, a programmer can both determine when data should be read from the input stream, and how many data items can be read. In addition, StreamFlex allows programmers to circumvent the default synchronous input filter firing mechanism by overriding a method called *trigger*. RATE TYPES in its current form cannot support this flexible design. It can be applied in limited cases where the default trigger mechanism is used, and static analysis can determine the number of channel reads/writes per-firing (*e.g.*, when the data stream read/write are not in recursions or loops). The examples used in the Streamflex paper appear to fit into these cases.

The majority of the reasoning ability of RATE TYPES — such as the ability to determine the overall data throughput when the characteristics of (base case) filters are known — remains intact even when *Static Items per Firing* does not hold. The primary limitation in a scenario like StreamFlex

Language	Category	Synchronous Input	Static Items per Firing	Non-Blocking Memory Access	Graph Composition
StreamIt [18]	stream	Yes	Yes	Yes, except teleporting	Chain/Diamond/Circle
Aurora [3]	stream	Yes	Yes	Yes	Chain/Diamond
Borealis [30]	stream	Yes	Yes	Yes	Chain/Diamond
STREAM [31]	stream	Yes	Yes	Yes	Chain/Diamond
StreamFlex [22]	stream	No	No	Yes	declarative channels
Elm [5]	signal	Yes	Yes	Yes	function application
FlapJax [32]	signal	Yes	Yes	Yes	function application
Pig Latin [33]	dataflow	Yes	Yes	Yes	Chain/Diamond
Eon/Flux [7]	dataflow	Yes	Yes	Yes	Chain/Diamond
FlumeJava [25]	dataflow	Yes	Yes	Yes	Chain/Diamond
Bamboo [34]	dataflow	Yes	Yes	transactional locking	typestate-based

Table 1. RATE TYPES Generality and Limitations

lies in *when* the characteristics of the filters — the leaf nodes in a type derivation — can be known. In the future, we are interested in developing a dynamic variant of RATE TYPES that uses lightweight dynamic constraint solving for adaptive stream rate control. We speculate the dynamic variant may be more suitable for coping with flexible models where the number of input/output items per firing fluctuates at run time.

The *Non-Blocking Memory Access* column describes the memory access semantics of each language in the presence of concurrency. RATE TYPES does not place restrictions *per se* on memory access, but blocking during filter execution may impact a practical aspect of RATE TYPES: the stability of profiling results. Most data-oriented programming models — especially those geared towards high data throughput — either prevent or minimize blocking at the language level, or caution against their common use. The former approach is taken by both languages with a functional core (such as Elm and FlapJax) and languages such as StreamFlex. Indeed, a key design goal of StreamFlex is to implement non-blocking semantics during filter execution, achieved through a combination of techniques including an ownership type system to reason about memory disjointness, a non-blocking bounded channel design, and preemptible atomic regions for Java compatibility. Other languages minimize blocking, such as StreamIt, which defers most blocking needs to an uncommon message teleporting mechanism.

The exception here is Bamboo, a language where data processing units (called *tasks*) can access global memory and the language enforces atomicity through locks. Bamboo acquires all locks required for transactional task execution at the beginning of the task execution. In that sense, non-blocking memory access is still maintained in Bamboo, good news for profiling stability. The challenge of applying RATE TYPES to Bamboo however is that task firing may not be solely dependent on input data availability, but also lock acquisition. Bamboo has a data-oriented programming model, but its focus is on parallelism support and task scheduling, not stream-like data processing.

Impacts of Syntactic Choices on Generality The *Graph Composition* column addresses the syntactic choices used to create the stream graph common to stream/signal/dataflow programs: a directed graph in which processing units are nodes and data streams are edges between nodes.

A significant number of languages we studied only allow acyclic dataflow/stream graphs, a design choice on par with languages that support *Chain* and *Diamond* combinators. The scope RATE TYPES considers is more expressive, because it also considers feedbacks. Indeed, the concrete grammar of different languages to construct dataflow/stream graphs can be diverse, sometimes through algebraic combinators like what RATE TYPES uses, or through conventional node / edge representations, or simply through a sequence of assignments (such as in Pig Latin). The difference, however, is largely stylistic, in that though the combinator-based expressions are more friendly for formalization, different syntactic styles can be transformed from one to another statically.

In terms of the dataflow graph topology, the flexible languages are StreamFlex, Elm, FlapJax, and Bamboo. In StreamFlex, a filter can declare the channels it interacts with, and the topology of the underlying stream graph may involve cycles. Interestingly, since the channels in StreamFlex must be explicitly declared for each filter, the stream graph can be statically inferred through a trivial program analysis. We speculate RATE TYPES can be applied when the static stream graph (which may or may not contain cycles) is known. Elm and FlapJax follow the tradition of FRP languages, where function composition – when two functions which take a stream as an argument and produce a stream as a result are composed — can be viewed analogously as a form of *Chain*, and the typing rule for *Chain* of RATE TYPES can be built analogously for function composition expressions. Notably, most signal languages come up with well-defined combinators, which match well with the syntactic choices of RATE TYPES. For example, `mergeE` in FlapJax is similar to the joining of *Diamond*.

Bamboo adopts a syntax conceptually related to tuple spaces, where (in the view of programmers) data live in the global space. A Bamboo programmer defines *tasks*, each of which specifies the types of data (streams) it reads from the global space as input and the conditions associated with that stream to fire the task. Once fired, the task processes the data item and places the processed item back to the global space. Since all streams are strongly typed and the conditions are declarative similar to typestates, we speculate the task/data dependency explicitly declared in Bamboo programs may help induce the stream graph. Interestingly, Bamboo's compiler infers a *combined state transition graph* for optimization purposes. This data structure explicitly expresses the data flows between tasks, which appears to be a refined form of stream graphs.

An orthogonal issue is the expressiveness of the stream manipulation operators themselves. For instance, FlumeJava supports the `groupByKeys` operator to compute equivalence classes for a finite set of data items, and Pig Latin supports operators to compute the cross product of multiple finite streams, such as the `COGROUP` operator. To investigate the applicability of RATE TYPES in these cases, there are two separate issues: (1) whether such operators can be encoded through the primitives currently in RATE TYPES; (2) whether the program analysis enabled by RATE TYPES can precisely predict performance over the encodings of such operators. We believe (1) can be answered positively (see Appendix), but the answer to (2) relies on how faithful the encodings are to the internal implementation of these operators. For practical adoption of RATE TYPES to these languages, the more direct route is to consider these operators as language primitives, a direction open for further research. This incompleteness should come as no surprise: RATE TYPES considers the more general case of data processing over dataflow graphs, whereas languages such as FlumeJava and Pig Latin further refine/restrict the domain by focusing on database programming with finite-sized tables.

Finally, even though unseen in formalisms, practical languages often allow multiple output streams, and sometimes multiple input streams for increased programmability. In these cases, RATE TYPES can be extended in a predictable manner to construct constraints for each of the input/output streams independently. In the inference algorithm, each input/output stream should be represented by a separate rate type variable.

Impacts of Implementation Choices on Generality Our implementation base, StreamIt, is a relatively optimized streaming infrastructure. It should be pointed out however, the effectiveness of RATE TYPES reasoning in practice – such as predicting the throughput of a stream program – is largely orthogonal to whether the base infrastructure is highly optimized or not. The key insight here is that RATE TYPES reasoning can be effective as long as the overhead introduced by the base stream infrastructure can be well accounted for. Stream processing overhead can be divided into two categories: a small fixed overhead required per firing, and an often larger overhead proportional to the idle time associated with a filter, such as the time required for polling. The overhead proportional to idle time does not affect stream rates because it consumes idle time, not the time required to produce output data. The remaining fixed, per-firing overhead will reduce the output rate only when the per-firing overhead far exceeds that due to the idle time of a filter. As long as the per-firing overhead is sufficiently compensated in the inference, RATE TYPES will guide real-world systems to produce experimental results with accuracy. Indeed, even in a relatively optimized system such as StreamIt, the overhead still needs to be compensated, in a fashion we described in Section 5, and we believe similar overhead compensation strategies can be constructed for other streaming language systems, no matter what level of optimization is present.

Our implementation of RATE TYPES on top of the StreamIt infrastructure interacts with three important features of StreamIt, namely profiling, partitioning, and stream batching. Among them, only StreamIt's profiling feature is a "helper" to our implementation, and the other two features in fact complicated our implementation as we described in Section 5. Take partitioning for instance, the feature that refactors the stream graph so that the number of filters matches the number of cores available on the hardware. Since our operational semantics considers each filter as a separate thread, we have to take StreamIt's partitioning into consideration and work with its post-partition graph. The StreamIt stream batching feature required significant tuning for RATE TYPES, as discussed in the *Batch Size Selection* paragraph in section 5. We used StreamIt's profiler only after significant modification, and surmise that similar modifications could enable similar capabilities in other languages.

7. Related Work

Stream languages have their root in dataflow languages. Earlier examples include Lucid [35], LUSTRE [36], and Ptolemy [21], with well-known foundations such as Kahn networks [37]. Among them, the operational semantics of RATE TYPES is closest to the synchronous dataflow (SDF) design of Ptolemy, which later influenced the design of StreamIt, the system our implementation is built upon. A classic static reasoning result on SDF is static scheduling, by Lee and Messerschmitt [16]. The goal of static scheduling is to compute a *schedule*, *e.g.*, AAB to mean filter A should be fired twice for every filter B execution. This algorithm is influential in the design of stream languages. For instance, StreamIt adopts a variant of the algorithm, called steady-state scheduling [23]. Our type system is orthogonal to static scheduling, both in goals and in methodologies. RATE TYPES finds the best possible output rate, independent of the firing order of the filters. A good schedule will come close to achieving the best possible rate, but even the best possible schedule cannot exceed that limit. RATE TYPES is a type-based compositional reasoning framework, whereas Lee and Messerschmit is based on graph theory through a notion called a *topology matrix*.

Clock calculus [17] is based on a restrictive form of SDF, where different sub-programs (filters) of a stream program are synchronized. Their type system infers a "clock" to synchronize filters. RATE TYPES does not require any synchronization between sub-programs. Furthermore, unlike clock calculus where filter execution time is treated as "unit length," RATE TYPES illuminates the relationship between filter execution time and stream rates. Thiele and Stoimenov [38] studied performance analysis in the presence of cycles in SDF. They do not define a dynamic semantics to formally relate the runtime and the analysis, and their approach is not type-based.

Some type systems exist to reason about non-quantitative aspects of stream/dataflow/signal programs. StreamFlex [22] has an ownership type system aimed at enforcing memory safety, especially non-shared memory access from different filters. A dependent type system [39] was designed for FRP to enforce productivity: a liveness property to guarantee the program continues to deliver output. Krishnaswami *et al.* [40] introduced a linear type system to bound resource usage (especially space) in higher-order FRP. Suenaga *et al.* [11] designed a type system in the Hoare-style on top of a stream language core as an example to demonstrate the expressiveness of their discrete-continuous transfer framework for verification. Elm [5] as a FRP-family language has a type system to outlaw higher-order signals. None of the related work above reasons about the rate of data processing.

For control-flow languages, there is a large body of work on qualitative and quantitative reasoning of performance-related properties, *e.g.*, WCET [12], cost semantics [13], resource usage analysis [14], amortized resource analysis [15], and Energy Types [41]. RATE TYPES focuses on the data rates of data-flow programming models.

Our prior work Green Streams [42] achieves energy efficiency by DVFS through a program analysis on balancing data rates. The analysis is not type-based, and with a different goal. The two systems share the high-level observation in analyzing stream programs: data rates can be abstracted.

8. Conclusion

This paper describes a novel type system for performance reasoning over stream programs, focusing on a highly dynamic aspect – stream rate control. Our experimental results show the reasoning framework can effectively predict stream program behaviors in terms of data throughput.

The proofs of theorems and lemmas described in the paper can be found in the accompanying technical report [28].

Acknowledgments We thank the anonymous reviewers for their useful suggestions, and Amer Diwan and Jens Palsberg for insightful discussions. This work is partially supported by NSF CCF-1054515. Any opinions expressed here are from the authors and do not necessarily reflect the views of the sponsors.

References

[1] Thies, W., Amarasinghe, S.: An empirical characterization of stream programs and its implications for language and compiler design. In: PACT '10. (2010) 365–376

[2] Nvidia: Compute unified device architecture programming guide. NVIDIA: Santa Clara, CA (2007)

[3] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring streams - a new class of data management applications. In: VLDB. (2002) 215–226

[4] Krishnaswami, N.R., Benton, N.: A semantic model for graphical user interfaces. In: ICFP '11. (2011) 45–57

[5] Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for GUIs. In: PLDI'13. (June 2013)

[6] Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP '97. (1997) 263–273

[7] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M.D., Berger, E.D.: Eon: a language and runtime system for perpetual systems. In: SenSys '07. (2007) 161–174

[8] Monsanto, C., Foster, N., Harrison, R., Walker, D.: A compiler and run-time system for network programming languages. In: POPL '12. (2012) 217–230

[9] Soulé, R., Hirzel, M., Grimm, R., Gedik, B., Andrade, H., Kumar, V., Wu, K.L.: A universal calculus for stream processing languages. In: ESOP'10. (2010) 507–528

[10] Botinčan, M., Babić, D.: Sigma*: symbolic learning of input-output specifications. In: POPL '13. (2013) 443–456

[11] Suenaga, K., Sekine, H., Hasuo, I.: Hyperstream processing systems: nonstandard modeling of continuous-time signals. In: POPL '13. (2013) 417–430

[12] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem – overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3) (May 2008) 36:1–36:53

[13] Blelloch, G.E., Greiner, J.: A provable time and space efficient implementation of nesl. In: ICFP '96. (1996) 213–225

[14] Igarashi, A., Kobayashi, N.: Resource usage analysis. In: POPL '02. (2002) 331–342

[15] Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: POPL '11. (2011) 357–370

[16] Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1) (January 1987) 24–35

[17] Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-synchronous kahn networks: a relaxed

model of synchrony for real-time systems. In: POPL '06. (2006) 180–193

[18] Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: CC'02. (2002) 179–196

[19] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220** (1983) 671–680

[20] Agha, G.: ACTORS : A model of Concurrent computations in Distributed Systems. MITP, Cambridge, Mass. (1990)

[21] Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9) (Sept 1987) 1235–1245

[22] Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: StreamFlex: high-throughput stream programming in Java. In: OOPSLA '07. (2007) 211–228

[23] Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS'06. (2006)

[24] Furr, M., An, J.h.D., Foster, J.S.: Profile-guided static typing for dynamic scripting languages. In: OOPSLA '09. (2009) 283–300

[25] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. In: PLDI '10. (2010) 363–375

[26] Morton, K., Balazinska, M., Grossman, D.: ParaTimer: a progress indicator for MapReduce DAGs. In: SIGMOD '10. (2010) 507–518

[27] Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL '08. (2008) 407–419

[28] Bartenstein, T., Liu, Y.D.: Rate Types for Stream Programs - Technical Report. `https://www.cs.binghamton.edu/~tbarten1/RateTypesForStreamPrograms_TechReport.pdf`, (2014)

[29] Gordon, M.I.: Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (May 2010)

[30] Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: CIDR. Volume 5. (2005) 277–289

[31] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system, CIDR (2003)

[32] Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. In: OOPSLA '09. (2009) 1–20

[33] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: A not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08, New York, NY, USA, ACM (2008) 1099–1110

[34] Zhou, J., Demsky, B.: Bamboo: a data-centric, object-oriented approach to many-core software. In: PLDI'10, ACM (2010) 388–399

[35] Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Commun. ACM **20**(7) (July 1977)

[36] Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: POPL '87. (1987) 178–188

[37] Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, J.L., ed.: Information processing, Stockholm, Sweden, North Holland, Amsterdam (Aug 1974) 471–475

[38] Thiele, L., Stoimenov, N.: Modular Performance Analysis of Cyclic Dataflow Graphs. In: Proceedings of the Seventh ACM International Conference on Embedded Software. EMSOFT '09, New York, NY, USA, ACM (2009) 127–136

[39] Sculthorpe, N., Nilsson, H.: Safe functional reactive programming through dependent types. In: ICFP '09. (2009) 23–34

[40] Krishnaswami, N.R., Benton, N., Hoffmann, J.: Higher-order functional reactive programming in bounded space. In: POPL '12. (2012) 45–58

[41] Cohen, M., Zhu, H.S., Emgin, S.E., Liu, Y.D.: Energy types. In: OOPSLA '12. (October 2012)

[42] Bartenstein, T., Liu, Y.D.: Green streams for data-intensive software. In: ICSE'13. (May 2013)

Appendix

A. Encoding

First, a k-way fork-join of programs $P_1, \ldots, P_k$ with distribution factor $\delta = \langle n_1; \ldots; n_k \rangle$ and aggregation factor $\alpha = \langle n'_1; \ldots; n'_k \rangle$ can be encoded as:

$$P_1 \diamond_{\delta_1,\alpha_1} (P_2 \diamond_{\delta_2,\alpha_2} (\ldots (P_{k-1} \diamond_{\delta_{n-1},\alpha_{n-1}} P_k)))$$

where $\delta_i = \langle n_i; n_{i+1} + \ldots + n_k \rangle$ and $\alpha_i = \langle n'_i; n'_{i+1} + \ldots + n'_k \rangle$ for i = $1..n-1$.

Round-robin is not the only way the input stream of a split-join can be divided. Another useful pattern is to duplicate every input stream element, and feed each duplicate to the input stream of the two sub-programs (say P_1 and P_2) participating the split-join. This can be encoded as $F^L[1,2] \rhd (P_1 \diamond_{\langle 1;1 \rangle,\alpha} P_2)$ where F is function that takes $[x]$ and returns $[x, x]$ with $\Pi(L) = 0$.

Similarly, the output stream of a split-join does not need to be aggregated through round-robin either. A useful pattern is to aggregate the two output streams of the two sub-programs (say P_1 and P_2) participating the split-join through some binary operators. For example, one may wish to only put the greater value of each pair of output elements of P_1 and P_2 to the output stream of the split-join. This can be encoded as $(P_1 \diamond_{\delta,\langle 1;1 \rangle} P_2) \rhd F^L[2,1]$ where F represents the binary operator. For the example above, F is the function that takes $[x, y]$ and returns $[\mathtt{max}(x, y)]$ where `max` () is the standard maximum operator.

In a streaming database query language, a database table *join* operation typically consists of a cross-product of a finite window over two input streams. Such a cross product can be modeled with a round-robin join, followed by a filter which pops a full window's worth of input data, and then pushes the cross-product of that window.

Last, our formal system idealizes the data transfer across buffers. Consider the chain composition $P_1 \rhd P_2$ for instance. A real-world implementation would place a buffer between the output stream of P_1 and the input stream of P_2. Such buffer read/write may take time. The time of buffer read-/write can accounted for by encoding $P_1 \rhd F^L[1,1] \rhd P_2$ where F is the identity function, and $\Pi(L)$ is the time needed for buffer access.