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 — a critical performance characteristic previously addressed by numerous experimental approaches but few foundational efforts. Even though 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 sound over a time-aware and parallelism-aware operational semantics. The strong soundness result tolerates arbitrary schedules, and does not require any synchronization between stream filters. We further demonstrate the applications of RATE TYPES in energy-efficient computing and CPU allocation on multi-core architectures.

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. Stream programming — together with its close relatives of signal programming and (more generally) dataflow programming — is 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].

In this paper, we develop a novel theoretical foundation to reason about the *data rate* aspect of stream programming. 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 upper bound of the output stream rate of a stream program regardless of its input stream rate.

Our key insight is that both throughput ratio and natural rate are closely related to the *program structure*. 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. At the core of this exploration is *quantitative reasoning* about performance-related properties, an active area of research (*e.g.* [12–16]). RATE TYPES follows a less explored path by focusing on quantitative reasoning over *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 [17] nor inter-filter synchronization [18].

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 happens to unify many interesting practical algorithms in stream rate control.

To demonstrate the applications of RATE TYPES and bring it closer to real-world computing, we further extend RATE TYPES to illustrate its usefulness in energy management and CPU allocation on multi-core architectures. The relationship between energy consumption and performance, and that of CPU allocation and performance, have long been explored in experimental research. Our approach is a first step to formally illuminate this complex landscape that involves performance (data throughput in our case), CPU energy consumption, CPU core allocation, and program structure.

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 type soundness property.
- 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 applies the type system to assist energy optimization, in a setting where stream filters may be executed on CPUs whose frequencies are dynamically adjustable.
- It applies the type system to assist CPU allocation optimization, where multiple instances of the same filter may execute in parallel to support data parallelism.

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

```
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
16 xy→xy filter getNeighbors() {
17     work push 9 pop 1 {
18         xy p = pop();
19         push(p);
20         push({p.x+1, p.y, p.ben});
21         push({p.x, p.y+1, p.ben});
22         ...
23 }}
24 xy→xy splitjoin computeProfits() {
25     split roundrobin(1,1);
26     add getProfit;
27     add getProfit;
28     join roundrobin(1,1);
29 }
30 xy→xy filter getProfit() {
31     work push 1 pop 1 {
32         xy loc =pop();
33         loc.ben= ...; // look up profit
34         push(xy);
35 }}
36 xy→xy filter evalNeighbors() {
37     work push 1 pop 9 {
38         xy p1 = pop(), p2=pop(), ...
39                 p9 = pop();
40         xy best = ...; // best of 9
41         push(best);
42 }}
43 xy→xy filter randomJump() {
44     work push 1 pop 1 {
45         xy p = pop();
46         xy rand = ...;
47         if (...) push(p)
48         else push(rand);
49 }}
```

Fig. 1. A StreamIt Program for Simulated Annealing

variety of programming languages. Here we choose StreamIt [19] as the host language, and discuss other applicable languages at the end of the section.

Stream Programming Figure 1 is a stream program for *simulated annealing* [20], a classic optimization algorithm that probabilistically finds globally optimal solutions through 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 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 43).

The base processing unit of a stream program is a *filter*, like `getNeighbors` in Line 16, whose body is a function labeled with keyword `work`. A filter execution instance takes in a 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 18-22, 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.

With sufficient data items on their respective input streams, different filters — such as a `getNeighbors` and an `evalNeighbors` — can execute in parallel. Similar to other concurrency models such as actors [21], a filter execution instance abides by a "one firing at a time" rule: the evaluation of each function application must be completed before the a 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 24 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 `randomJump` 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 high 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: What is the upper bound for a program's data production, 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** though type inference. It further demonstrates the relationship among these questions in a unified, provably sound framework.

We further extend RATE TYPES to establish its relationship with the settings of CPU frequencies, and its relationship with the number of CPU cores available for data parallelism. Surprisingly, these expressive features only requires minimal extensions to the framework core. We are able to formally capture how performance and energy are linked, and formally demonstrate how performance is impacted by CPU allocation to support data parallelism. Specifically, we offer theoretical answers to two questions actively under investigation by experimental research:

Q5: Given an expected data production rate, what is the minimal CPU frequency for each filter execution instance executed on CPUs that support Dynamic Voltage and Frequency Scaling (DVFS) [22]? As CPU frequency and energy consumption is correlated, a solution along these lines is tantamount to improving energy efficiency without performance degradation.

Q6: Given an expected data production rate, and if we relax our framework to allow multiple instances of the same filter to be executed in parallel to support data parallelism, what is the fewest number of parallel execution instances for each filter – hence the fewest CPU cores – to achieve the expected data production rate?

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 we conventionally consider as "static." We consider it reasonable because (a) filter behaviors are much more predictable than full-fledged programs, thanks to the non-shared memory model and its lack of side effects often called for in real-world stream languages [19, 23]; (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 [24] rely on it; (d) real-world software development is iterative. Profiling-guided typing is not new [25]. What 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. More broadly, the framework may be applied 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 composition frameworks such as FlumeJava [26] and ParaTimer [27], where single MapReduce-like units are composed together in expressive ways.

3 Syntax and Dynamic Semantics

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

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

The four forms of a program P are a filter, a chain composition, a diamond composition, and a circle composition, in that order. Each filter is defined as a filter function body F, together with a unique filter (program) label $L \in \mathbb{FLAB}$. Each filter is further declared with two natural numbers: n_i for the number of items to be consumed at a time, and n_o for the number of items to be produced at a time. The two numbers correspond to the `pop` and `push` declarations in Figure 1. 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. 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` and the `join` in the example respectively. Each chain expression is associated with a distinct *stream label*, $\ell \in \mathbb{SLAB}$, and the circle expression is associated with 2. They are only used for formal development, to be explained later.

In Appendix A.1, 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. Given a program P, we use $\texttt{flabels}\,(P)$ to enumerate all filter labels, and $\texttt{slabels}\,(P)$ to enumerate all stream labels.

The program in Figure 1 can be represented by our syntax as $P_{\texttt{anneal}}$ where

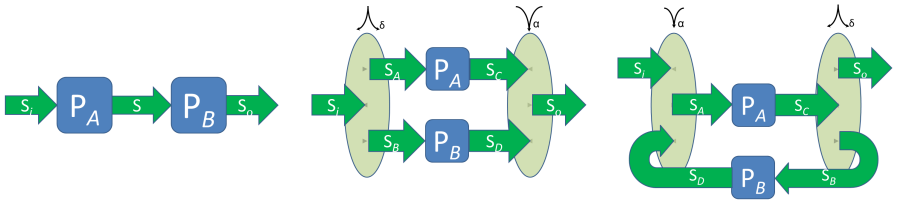
$$
\begin{aligned}
P_{\texttt{anneal}} &= P_{\texttt{checkNeighbors}} \circlearrowright^{\ell_0,\ell_1}_{\langle 1;99\rangle,\langle 1;99\rangle} P_{\texttt{randomJump}} \\
P_{\texttt{checkNeighbors}} &= (P_{\texttt{getNeighbors}} \rhd^{\ell_2} P_{\texttt{computeProfits}}) \rhd^{\ell_3} P_{\texttt{evalNeighbors}}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

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 $|[X_1, \ldots, X_n]| \stackrel{\text{def}}{=} n$ and use comma for sequence concatenation, *i.e.* $[X_1, \ldots, X_n], [Y_1, \ldots, Y_{n'}] \stackrel{\text{def}}{=} [X_1, \ldots, X_n, Y_1, \ldots Y_{n'}]$. 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 $\texttt{dom}\,(M) \stackrel{\text{def}}{=} \{X_1, \ldots, X_n\}$ and $\texttt{ran}\,(M) \stackrel{\text{def}}{=} \{Y_1, \ldots, Y_n\}$ 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.

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

$$
\begin{array}{lr}
R ::= \overline{\ell \mapsto S} & \textit{program runtime} \\
S ::= \overline{d} & \textit{stream} \\
\ell \in \mathbb{SLABEL} \cup \{\ell_{\texttt{IN}}, \ell_{\texttt{OUT}}\} & \textit{stream label} \\
t \in \mathbb{TIME} \subseteq \mathbb{REAL}+ & \textit{time} \\
\mathit{\Pi} \in \mathbb{FLAB} \mapsto \mathbb{TIME} & \textit{filter time mapping}
\end{array}
$$

The runtime, R, consists of a mapping sequence from stream labels to streams. A stream is defined as a list of data items where each data item $d \in \mathbb{DATA}$. 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 $\mathit{\Pi}$ maps each filter (label) to its execution time.



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

Fig. 2. Stream Runtime Semantics Illustration

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. Some examples should demonstrate the functionality of these predicates, whose predictable formal definitions appear in Appendix A.2:

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

Operational Semantics Figure 3 defines operational semantics, with relation $\vdash_d R \xrightarrow[t]{P} R'$ 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 illusrated in Figure 2, where streams are green arrows and labeled with metavariables appearing in rules. By convention, we identify a pre-reduction value with a symbol, and use the same symbol with an apostrophe to indicate the value after reduction. 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], $\curlywedge$ allows one stream to be "viewed" as two, whereas $\curlyvee$ allows two streams to be "viewed" as one. This idea of "views" is inspired by lens [28]. As illustrated in Fig. 2(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 is "split" to the output stream (S_o) and the input stream of the feedback sub-program (S_B). We postpone its formal definition to Appendix A.3.

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

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

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

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

Fig. 3. Operational Semantics (see Appendix A.3 for circle composition)

There are two important 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. Second, the reduction system does not require synchronization over any sub-programs. For all three composition forms, a reduction can happen if one 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 [17] are required.

Properties The operational semantics enjoys several simple properties:

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

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

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 Size Change Rates). *Given a reduction from* R *to* R' *over time* t, *the rate for stream* ℓ *is defined by function* `rchange`$()$*:*

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

where unary operator `abs` $()$ *computes the absolute value.*

From this point on, we use metavariable r to represent stream size change rates. $r \in \mathbb{FLOAT}+$. Observe that according to Lem. 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. We define:

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

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

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 1 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 $\texttt{dom}\,(R)$ is the smallest set of ℓ where $P_A \circlearrowleft_{\alpha,\delta}^{\ell',\ell} P_B$ is a sub-program of P, $\alpha = \langle n; n' \rangle$ and $|R(\ell)| = n'$. Here, stream label ℓ' intuitively identifies the input stream of the forward sub-program (P_A), and ℓ for the output stream of the feedback sub-program (P_B).

Definition 3 (Bootstrapping Runtime). *Given program* P, *input stream* S_0, *and primer* R_0, *function* `init` (P, S_0, R_0) *computes the initial runtime of* P, *defined as the smallest* R *satisfying the following conditions:* $R(\ell_{\texttt{IN}}) = S_0$, $R(\ell_{\texttt{OUT}}) = \emptyset$, $R_0 \subseteq R$, *and* $R(\ell) = \emptyset$ *for any* $\ell \in \texttt{slabels}\,(P) \cap \overline{\texttt{dom}\,(R_0)}$.

4 Rate Types

Types in our system are defined as follows:

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

$$\frac{\vdash_t P : \tau' \qquad \tau' <: \tau}{\vdash_t P : \tau}\ \text{[T-Sub]} \qquad \frac{\begin{array}{c}\vdash_t P_a : \langle \theta_a ; \nu_a \rangle \qquad \vdash_t P_b : \langle \theta_b ; \nu_b \rangle \\ \theta = \theta_a \times \theta_b \qquad \nu = \mathtt{min}\,(\nu_a \times \theta_b, \nu_b)\end{array}}{\vdash_t P_A \rhd^{\ell} P_B : \langle \theta ; \nu \rangle}\ \text{[T-Chain]}$$

$$\frac{\theta = n_o / n_i \qquad \nu = n_o / \Pi(L)}{\vdash_t F^L[n_i, n_o] : \langle \theta ; \nu \rangle}\ \text{[T-Filter]} \qquad \frac{\begin{array}{c}\vdash_t P_a : \langle \theta_a ; \nu_a \rangle \qquad \vdash_t P_b : \langle \theta_b ; \nu_b \rangle \\ \theta'_a = \curlywedge^1(\delta) \times \theta_a \times \curlyvee^1(\alpha) \\ \theta'_b = \curlywedge^2(\delta) \times \theta_b \times \curlyvee^2(\alpha) \\ \nu'_a = \nu_a \times \curlyvee^1(\alpha) \qquad \nu'_b = \nu_b \times \curlyvee^2(\alpha) \\ \theta = \mathtt{min}\,(\theta'_a, \theta'_b) \qquad \nu = \mathtt{min}\,(\nu'_a, \nu'_b)\end{array}}{\vdash_t P_a \diamond_{\delta,\alpha} P_b : \langle \theta ; \nu \rangle}\ \text{[T-Diamond]}$$

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

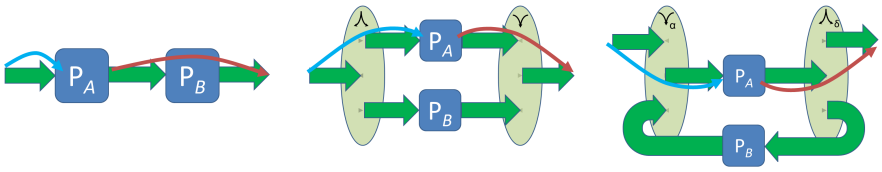
Fig. 4. Rate Type Checking Rules

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 with no limitation on the input stream rate. In other words, it characterizes the upper bound of the output stream rate.

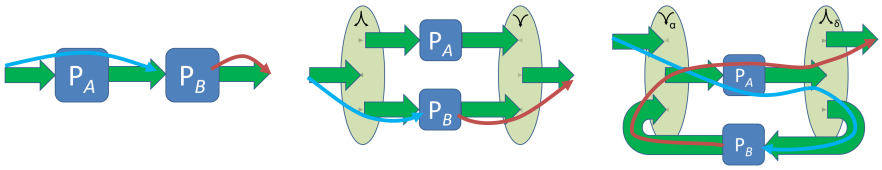
The type form here reveals a fundamental phenomenon of stream rate control: the input stream rate and the output stream rate can be correlated by a ratio θ, but the correlation only holds when the output stream rate does not reach its upper bound ν. To gain intuition on the upper bound aspect of stream control, the key insight is that *it takes time to process data*: each filter execution takes time, and each filter instance can only take "one firing at a time." The combinational effect is that a program simply cannot — in practice or in theory — produce output streams at an unlimited rate.

4.1 Type Checking

Judgment $\vdash_t P : \tau$ 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_1 data items per second when its input is fed with n_2 items per second. That question is tantamount to finding out whether a derivation exists for $\vdash_t P : \langle n_1/n_2 ; n_1 \rangle$. Question **Q2** — determining the upper bound of the output rate ν — can be answered by finding out whether a derivation exists for $\vdash_t P : \langle \theta ; \nu \rangle$ for some θ.



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



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

Fig. 5. Reasoning about Throughput Ratios and Natural Rates (For natural rate reasoning, follow the red lines; For throughput ratio reasoning, follow both the blue and the red lines.)

The typing rules are summarized in Figure 4. Simple functions $\curlywedge^1(\delta)$, $\curlywedge^2(\delta)$, $\curlyvee^1(\alpha)$ and $\curlyvee^2(\alpha)$ can be informally viewed as computing a form of "normalized" distribution and aggregation factors. They are defined as:

$$\curlywedge^1(\delta) \stackrel{\text{def}}{=} \frac{n}{n+n'} \qquad \curlyvee^1(\alpha) \stackrel{\text{def}}{=} \frac{n+n'}{n}$$
$$\curlywedge^2(\delta) \stackrel{\text{def}}{=} \frac{n'}{n+n'} \text{ where } \delta = \langle n, n' \rangle \qquad \curlyvee^2(\alpha) \stackrel{\text{def}}{=} \frac{n+n'}{n'} \text{ where } \alpha = \langle n, n' \rangle$$

[T-Sub] introduces subtyping. 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 function application and that of data items consumed by the same application. Following the "one-firing-at-a-time" execution strategy, the upper bound rate for a filter to produce data items is it runs "non-stop": the filter produces every n_0 items for its execution time $\Pi(L)$.

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. There are two cases: (1) Figure 5(a): when P_B is fed with an input stream whose rate is high enough that the output stream of P_B reaches its natural rate ν_b, then ν_b should also be the upper bound for the entire composition program. (2) Figure 5(b): otherwise, the upper bound of the entire program is determined by the rate of the input stream of P_B, which is, in this case, the output stream rate of P_A. Since we know the upper bound of that rate is ν_A, the output stream rate of P_B — and hence also the output stream rate of the entire

program — is no higher than $\nu_A \times \theta_b$. The natural rate of the entire program should be the minimal of the two, computed by the standard binary function $\texttt{min}\,()$.

In [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 5(c) and 5(d) demonstrate this case.

To see how [T-Circle] works, first let the upper bound of stream rate at the point of P_A's output be ν'_a and let the upper bound of stream rate at the point of P_B's output be ν'_b. 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 5(e): when P_A is fed with an input stream whose rate is so high that its output stream is already the upper bound, then $\nu'_a = \nu_a$; (2) Figure 5(f): otherwise, ν'_a is determined by the input stream rate of P_A, which in turn is determined by the upper bound of the output rate of P_B, *i.e.* $\nu'_b \times \curlyvee^2(\alpha) \times \theta_a$. 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 indicates the number of iterations, then:

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

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.

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 the dynamic stream rates:

Theorem 3 (Type Soundness). Given a program, P, and the initial runtime $R = \texttt{init}(P, S_0, R_0)$ for some S_0 and R_0, and if $\vdash_d R \xrightarrow[t]{P} R'$ and $\texttt{rto}\,(R, R', t) = r_1$ and $\texttt{rti}\,(R', R, t) = r_2$, then $\vdash_t P : \langle r_1/r_2; r_1 \rangle$.

This theorem relates program dynamic behaviors and typing. One drawback is that it requires relating a runtime state with the initial state. In other words, the rates being computed are averaged through the entire program execution. Will the same theorem hold for two arbitrary runtime states (*i.e.* if we removed the requirement of starting from the initial state)? The answer unfortunately is no. The reason is that each stream program runtime may contain "intermediate streams", *i.e.* streams that are not identified by $\ell_{\texttt{IN}}$ and $\ell_{\texttt{OUT}}$. 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 *sustainability* condition is met:

Theorem 4 (Type Soundness over Arbitrary Reduction Steps). Given program P and $\vdash_d R \xrightarrow[t]{P} R'$ where $\texttt{rto}(R, R', t) = r_1$, $\texttt{rti}(R', R, t) = r_2$, and for any $\ell \in \texttt{dom}(R) - \{\ell_{\texttt{IN}}, \ell_{\texttt{OUT}}\}$, $\texttt{rchange}(R, R', t, \ell) = 0$ then $\vdash_t P : \langle r_1/r_2; r_1 \rangle$.

Here we call the $\texttt{rchange}()$ predicate in the theorem the *sustainability* condition. The theorem says that a reduction sequence starting at *any* state 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 state 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 sustainability: it only requires the end state is sustainable relative to the beginning state. In other words, the theorem is tolerant of temporary "bursty" behaviors during the reduction(s) from R to R'.

In pratice, we expect Theorem 3 and Theorem 4 are useful in complementary scenarios. Theorem 3 is more appropriate for characterizing short-running programs, where sustainability may never be achieved before the execution completes. In that scenario, the theorem unconditionally says performance reasoning is effective in modeling runtime stream rates, regardless of sustainability. Theorem 4 says for long-running programs, we can still capture the rate behavior of a program execution, say, from its 68^{th} minute to its 95^{th} minute. The theorem further tolerates bursty behaviors, say in the 77^{th} minute, as long as the 95^{th} minute execution snapshot conforms to the sustainability condition relative to the 68^{th} minute snapshot. This is sufficient to characterize long-running programs that 1) become sustainable after an initial duration of time (but may be followed by intermittent bursty behaviors); 2) or demonstrate periodic behaviors. In the latter case, the theorem is useful when R and R' are states on the reduction sequence with full periods in between.

Furthermore, Theorems 3 and 4 may be combined to characterize the same program execution, with the former addressing the beginning (potentially unstable) stage, and the latter for the rest.

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, including multiplication ($\bullet$) and computing the minimal value of the two ($\texttt{minn}$). 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 $\times$, $\texttt{minn}$ and $\texttt{min}$. We further 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 $\sigma(q)$.

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

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

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

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

Fig. 6. Rate Type Inference Rules

Type inference rules are given in Figure 6. 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 4. Indeed, the close relationship between the two can be formally established:

Theorem 5 (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 6 (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 7 (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.

We last introduce an effective way to compute the principal type. First, let us define a new judgment $\vdash_{ip} P : \langle p; q\rangle \backslash \Sigma$. The typing rules for this judgment is identical to that of $\vdash_i$, except that every occurrence of $\preceq$ is replaced with $\doteq$. The following theorem is an effective decision procedure for computing principal types for $\vdash_i$:

Theorem 8 (Effective Principal Type Computation). For any P such that $\vdash_{ip} P : \langle p; q\rangle \backslash \Sigma$ and $\sigma \Downarrow \Sigma$ holds, then $\langle \sigma(p); \sigma(q)\rangle$ is the principal type of P.

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 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 Applications

5.1 RATE TYPES for Energy Management

RATE TYPES, with minor modifications as noted below, provides opportunities for energy optimization thanks to its refined support for performance reasoning on stream rates.

The vast majority of CPUs today are equipped with DVFS, which enables dynamic modification of the operational frequency and supply voltage. They typically support a limited predefined set of frequencies, which we model as an ordered set, $\langle \mathbb{FREQ}; <\rangle$, where each element, $fq \in \mathbb{FREQ} \subseteq \mathbb{FLOAT}+$ is a frequency (in Hertz) supported by the hardware. Let us make a few simple assumptions for this short description of the application of RATE TYPES to stream rate control in a DVFS-enabled setting. First, we assume each filter runs on its own DVFS-manageable CPU core. Second, we assume the execution time of a piece of code — such as a filter function — is proportional to the CPU frequency at which the code (the filter) runs. The key to connect DVFS with our framework is a simple fact: DVFS affects the execution time of filters. By speeding up or slowing down the frequency for a specific filter, we can modify its execution time (and natural rate). We formalize this model by extending the filter time mapping Π to Π^E, the filter execution time at a given frequency, defined as:

$$\Pi^E(L, fq) = \Pi(L) \times \frac{\mathtt{max}\,(\mathbb{FREQ})}{fq}$$

where we use the original Π to keep the execution time when a filter is executed at the highest frequency $\mathtt{max}\,(\mathbb{FREQ})$, and compute the execution time of a filter L on a CPU at frequency fq with $\Pi^E(L, fq)$.

DVFS is long known to be an effective energy management strategy (*e.g.* [22, 29–31]). CPU frequency downscaling (and its corresponding effect on voltage adjustment) can significantly reduce the power consumption of CPUs. If the CPU frequency of an execution is scaled down without degrading performance, the overall energy consumption is reduced. For our discussion here, we choose a very simple energy model. We use notation $\texttt{energy}\,(L,fq)$ to denote the energy consumption of filter L on frequency fq. We leave its definition abstract, and only axiomatically define that $\texttt{energy}\,(L,fq_1) < \texttt{energy}\,(L,fq_2)$ iff $fq_1 < fq_2$.

We can now investigate into **Q5** posted in Section 2: identifying the highest possible output rate using the least possible energy. The key insight here is Π^E introduces us a way to directly relate CPU frequencies to the type system. If we can introduce *frequency variables*, $fv \in \mathbb{FREQVAR}$, which are variables whose solutions are frequencies, and use them to construct constraints related to filter execution time, the solution of the constraints produced by the type inference algorithm may provide direct answers to frequency settings.

We first extend our type constraint definitions, as follows:

$$\begin{array}{lr}
\Sigma^E ::= \overline{c^E} & \textit{E constraints} \\
c^E ::= e^E \preceq e^E \mid e^E \doteq e^E & \textit{E constraint} \\
e^E ::= p \mid q \mid fv \mid \texttt{minn}\left(e^E, e^E\right) \mid e^E \bullet e^E & \textit{E expr} \\
\sigma^E ::= \overline{p \mapsto \theta} \cup \overline{q \mapsto \nu} \cup \overline{fv \mapsto fq} & \textit{E solution}
\end{array}$$

Then, we introduce a new system of type inference $\vdash_{ie}$. Judgment $\vdash_{ie} P : \langle p; q\rangle \backslash \Sigma^E$ is identical to $\vdash_i$, except all metavariables should have the predictable E- superscript, and the rule for filter inference is updated as:

$$\frac{\Sigma^E = \left\{p \preceq n_o/n_i, q \preceq \frac{n_o \bullet \Upsilon^E(L)}{\Pi(L) \times \texttt{max}(\mathbb{FREQ})}\right\}}{\vdash_{ie} F^L[n_i, n_o] : \langle p; q\rangle \backslash \Sigma^E} \; [I^E\text{-Filter}]$$

where the simple bijective mapping $\Upsilon^E \in \mathbb{FLAB} \mapsto \mathbb{FREQVAR}$ allows us to find out the frequency variable name used for a particular filter L where $\texttt{dom}\left(\Upsilon^E\right) = \texttt{flabels}\,(P)$. Note that the second constraint in Σ^E is very similar to its counterpart $q \preceq n_0/\Pi(L)$ in [I-Filter], except that the execution time is $\Pi(L) \times \frac{\texttt{max}(\mathbb{FREQ})}{\Upsilon^E(L)}$ instead of $\Pi(L)$.

Next, let us define an ordering relation to "rank" constraint solutions with regard to DVFS settings. $\sigma^E \leq^E \sigma'^E$ iff $\forall fv \in \texttt{dom}\left(\sigma^E\right), \sigma^E(fv) \leq \sigma'^E(fv)$. Finally, we can state our theorem on finding the optimal settings for DVFS without any performance degradation:

Theorem 9 (DVFS-Optimal Solution). For a program P where $\vdash_{ie} P : \langle p; q\rangle \backslash \Sigma^E$, the GLB of $\langle \{\sigma^E | (\sigma^E \Downarrow \Sigma^E \cup \{p \doteq \theta, q \doteq \nu\})\}; \leq^E\rangle$ exists where $\langle \theta; \nu\rangle$ is the principal type of P according to $\vdash_i$. We call the GLB the DVFS-optimal solution for P.

This important theorem comes with some subtleties. First, observe that the type inference algorithm $\vdash_i$ uses Π, which in this setting keeps the execution time of each filter when it runs on the highest possible CPU frequency. Second, the principal type of P says that θ and ν is the highest possible throughput ratio and natural rate. Combining

the two observations, θ and ν are indeed the best possible performance for P even when the hardware is considered: all CPUs are running on the highest frequency. As a result, any solution to $\Sigma^E \cup \{p \doteq \theta, q \doteq \nu\}$ would be a solution without performance degradation. The GLB computation yields a performance-preserving solution with the lowest setting of CPU frequencies.

Corollary 1 (Energy Optimality). *If σ^E is the DVFS-optimal solution for P, $\sigma_0^E \Downarrow \Sigma^E$ where $\vdash_{ie} P : \langle p; q \rangle \backslash \Sigma^E$ for some p and q,*

$$\sum_{L \in \texttt{flabels}(P)} \texttt{energy}\left(L, \sigma^E(\Upsilon^E(L))\right) \leq \sum_{L \in \texttt{flabels}(P)} \texttt{energy}\left(L, \sigma_0^E(\Upsilon^E(L))\right)$$

5.2 RATE TYPES for CPU Resource Allocation

Our discussion so far has been following the "one-firing-at-a-time" assumption for filter executions. As we have shown, this is one of the performance-limiting factors of stream programs. In this section, we allow a single filter to be assigned to multiple CPU cores to achieve data parallelism. We assume that every filter can be replicated, and there are unlimited number of cores. If the time required to execute a filter once is defined by $\Pi(L)$, we can multiply the output rate for that filter by $k \in \mathbb{NAT}+$ when running the filter k times on k different cores. A new CPU resource allocation problem arises: assuming we know the output rate we wish to achieve for a stream program, how can we allocate as few CPU cores for sustaining this output rate as possible?

Similar to our energy-motivated extension, the key insight is that data-parallel CPU allocation again affects filter execution time: allocating 5 CPU cores for a filter can be modeled by reducing the execution time of that filter by 5 times. We again formalize this model by extending the filter time mapping Π to Π^A, the filter execution time at a given number of CPU allocation, defined as: $\Pi^A(L,k) = \Pi(L)/k$ In other words, we use the original Π to keep the execution time when the filter is executed without multiple CPU core allocation, and compute the execution time of a filter L on a CPU at particular allocation k with $\Pi^E(L,k)$.

We posit another extension to our rate inference structures to manage the allocation of cores to filters.

$$\begin{array}{lll}
\Sigma^A ::= \overline{c^A} & & \textit{A constraints} \\
c^A ::= e^A \preceq e^A \mid e^A \doteq e^A & & \textit{A constraint} \\
e^A ::= p \mid q \mid kv \mid \texttt{minn}\left(e^A, e^A\right) \mid e^A \bullet e^A & & \textit{A expr} \\
\sigma^A ::= \overline{p \mapsto \theta} \cup \overline{q \mapsto \nu} \cup \overline{kv \mapsto k} & & \textit{A solution}
\end{array}$$

As we did in the previous sub-section, let us introduce a new system of type inference $\vdash_{ia}$. Judgment $\vdash_{ia} P : \langle p; q \rangle \backslash \Sigma^A$ is identical to $\vdash_i$, except all metavariables should have the predictable A- superscript, and the rule for filter inference is updated as:

$$\frac{\Sigma^A = \left\{p \preceq n_o/n_i, q \preceq \Upsilon^A(L) \bullet n_o/\Pi(L)\right\}}{\vdash_{ia} F^L[n_i, n_o] : \langle p; q \rangle \backslash \Sigma^A} \; [I^A\text{-Filter}]$$

where the simple bijective mapping $\Upsilon^A \in \mathbb{FLAB} \mapsto \mathbb{KVAR}$ allows us to find out the allocation variable name, $kv \in \mathbb{KVAR}$ used for a particular filter L.

Next, we define an ordering relation to "rank" constraint solutions w.r.t. CPU allocation. $\sigma^A \leq^A \sigma'^A$ iff $\forall kv \in \mathtt{dom}\left(\sigma^A\right), \sigma^A(kv) \leq \sigma'^A(kv)$. Finally, we can state our theorem on finding the minimal CPU allocation without any performance degradation:

Theorem 10 (Allocation-Optimal Solution). Given a program P such that $\vdash_{ia} P : \langle p; q\rangle \backslash \Sigma^A$, then the GLB of $\langle \{\sigma^A | (\sigma^A \Downarrow \Sigma^A \cup \{\nu \leq q\})\}; \leq^A \rangle$ exists, where ν is the requested output rate. We call the GLB the allocation-optimal solution for P.

This theorem directly answers **Q6** in Section 2. Optimally, each filter (of label L) is allocated with $\sigma^A(\Upsilon^A(L))$ number of cores, where σ^A is the allocation-optimal solution for P. The optimal solution by definition minimizes the CPU core allocation for every filter, and therefore minimize the overall CPU allocation as well.

6 Related Work

Quantitative reasoning about stream programs is uncommon. One established direction that loosely fits into the category is static scheduling of synchronous data flows (SDF) (*e.g.* [17]). For example, a schedule of AAB is computed when filters A and B are chained together and B pops 2 items for each firing. StreamIt implemented an enriched variant of the static scheduling, steady-state scheduling [24]. The goal of RATE TYPES – stream rate reasoning – is orthogonal to static scheduling. Our system can reason about stream rates with arbitrary schedules. Clock calculus [18] was designed as a more 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.

Some type systems have been proposed for non-quantitative aspect of stream reasoning. StreamFlex [23] has an ownership type system aimed at enforcing memory safety, especially non-shared memory access from different filters. A dependent type system [32] was designed for FRP to enforce productivity: a liveness property to guarantee the program continues to deliver output. Krishnaswami et. al. [33] 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. Dynamic semantics of stream/signal/dataflow programs is well understood. Historically significant systems such as Kahn networks, Ptolemy, LUSTRE, and classic FRP all have solid foundations, with numerous formalizations (*e.g.*, [34, 35, 33, 9, 36]). Within this backdrop, ours is designed for friendly accounting of data rates.

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 bound certification [14], resource usage analysis [15], amortized resource

analysis [16], and Energy Types [30]. RATE TYPES focuses on (data rates of) data-flow programming models, and their impact on energy consumption and CPU allocation.

In experimental research, data throughput is a critical metric to evaluate the performance of stream systems. An active research area is to minimize energy consumption with least performance penalty (*e.g.*[29]). These solutions do not often consider program structures, and usually measure data rate as the *effect* of their approaches, whereas RATE TYPES directly reasons about it, and makes rate control the *cause* of optimizing energy and CPU allocation. Our prior work, Green Streams [31], achieves energy efficiency by DVFS through a (non-type-based) program analysis.

7 Conclusion

This paper describes a novel type system for performance reasoning over stream programs, focusing on a highly dynamic aspect – stream rate control. The framework can potentially be applied to energy management and CPU resource allocation. The proofs of theorems and lemmas described in the paper can be found in the technical report. [1]

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

[1] `http://www.cs.binghamton.edu/~tbarten1/ESOP14_RateTypes_TechReport.pdf`

13. Blelloch, G.E., Greiner, J.: A provable time and space efficient implementation of nesl. In: ICFP '96. (1996) 213–225
14. Crary, K., Weirich, S.: Resource bound certification. In: POPL '00. (2000) 184–198
15. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: POPL '02. (2002) 331–342
16. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: POPL '11. (2011) 357–370
17. 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
18. 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
19. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: CC'02. (2002) 179–196
20. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220** (1983) 671–680
21. Agha, G.: ACTORS : A model of Concurrent computations in Distributed Systems. MITP, Cambridge, Mass. (1990)
22. Pering, T., Burd, T., Brodersen, R.: The simulation and evaluation of dynamic voltage scaling algorithms. In: ISLPED. (1998) 76–81
23. Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: StreamFlex: high-throughput stream programming in Java. In: OOPSLA '07. (2007) 211–228
24. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS'06. (2006)
25. Furr, M., An, J.h.D., Foster, J.S.: Profile-guided static typing for dynamic scripting languages. In: OOPSLA '09. (2009) 283–300
26. 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
27. Morton, K., Balazinska, M., Grossman, D.: ParaTimer: a progress indicator for MapReduce DAGs. In: SIGMOD '10. (2010) 507–518
28. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL '08. (2008) 407–419
29. Rangasamy, A., Srikant, Y.N.: Evaluation of dynamic voltage and frequency scaling for stream programs. In: Proceedings of the 8th ACM International Conference on Computing Frontiers (CF). (2011) 40:1–40:10
30. Cohen, M., Zhu, H.S., Emgin, S.E., Liu, Y.D.: Energy types. In: OOPSLA '12. (October 2012)
31. Bartenstein, T., Liu, Y.D.: Green streams for data-intensive software. In: ICSE'13. (May 2013)
32. Sculthorpe, N., Nilsson, H.: Safe functional reactive programming through dependent types. In: ICFP '09. (2009) 23–34
33. Krishnaswami, N.R., Benton, N., Hoffmann, J.: Higher-order functional reactive programming in bounded space. In: POPL '12. (2012) 45–58
34. Wan, Z., Taha, W., Hudak, P.: Real-time FRP. In: ICFP '01. (2001) 146–156
35. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: PLDI '00. (2000) 242–252
36. Selinger, P.: A survey of graphical languages for monoidal categories. In: New Structures for Physics. Springer Berlin/Heidelberg (2011) 289–355

A Appendix

A.1 Encodings

First, a k-way fork-join of programs $P_1, \ldots, P_k$ with distribution factor $\langle n_1; \ldots; n_k \rangle$ and aggregation factor $\langle n'_1; \ldots; n'_k \rangle$ 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 where 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 $[x, x]$ and $\Pi(L) = 0$.

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 and $\Pi(L) = 0$. For the example above, F is the function that takes $[x, y]$ and returns $[\texttt{max}(x, y)]$ where $\texttt{max}\,()$ is the standard maximum operator.

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 be taken into account through encoding $P_1 \rhd F^L[1,1] \rhd P_2$ where F is the identity function, and $\Pi(L)$ is time needed for buffer access.

A.2 Stream Assembly and Dissembly

$$\frac{|S| < (n + n')}{S \curlywedge_{\langle n;n' \rangle} \emptyset; \emptyset} \qquad \frac{\begin{array}{c} S_1 = \begin{bmatrix} d_{a_1}, \ldots, d_{a_n}, \\ d_{b_1}, \ldots d_{b_{n'}}, \end{bmatrix} \\ S_{A1} = [d_{a_1}, \ldots, d_{a_n}] \\ S_{B1} = [d_{b_1}, d_{b_2}, \ldots, d_{b_{n'}}] \\ S_2 \curlywedge_{\langle n,n' \rangle} S_{A2}, S_{B2} \end{array}}{S_1, S_2 \curlywedge_{\langle n;n' \rangle} S_{A1}, S_{A2}; S_{B1}, S_{B2}}$$

$$\frac{(|S_A| < n) \text{ OR } (|S_B| < n')}{S_A; S_B \curlyvee_{\langle n;n' \rangle} \emptyset} \qquad \frac{\begin{array}{c} S_{A1} = [d_{a_1}, d_{a_2}, \ldots, d_{a_n}] \\ S_{B1} = [d_{b_1}, d_{b_2}, \ldots, d_{b_{n'}}] \\ S_1 = [d_{a_1}, d_{a_2}, \ldots, d_{a_n}, d_{b_1}, d_{b_2}, \ldots, d_{b_{n'}}] \\ S_{A2}, S_{B2} \curlyvee_{\langle n;n' \rangle} S_2 \end{array}}{S_{A1}, S_{A2}; S_{B1}, S_{B2} \curlyvee_{\langle n;n' \rangle} S_1, S_2}$$

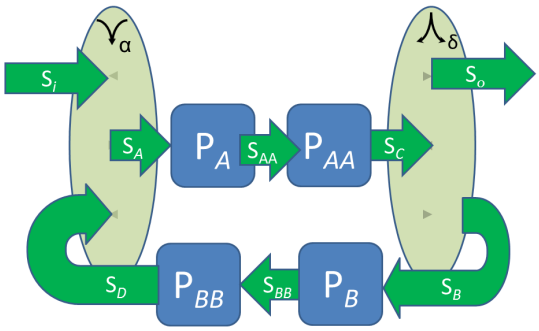


Fig. 7. Circle Composition Semantics Illustration

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

Fig. 8. Circle Composition Operational Semantics

A.3 Circle Composition Operational Semantics

The operational semantics of a circle composition, the [D-Circle] rule, is defined in Figure 10. As illustrated in Figure 9, the key insight for understanding [D-Circle] is a circle composition is (roughly) a reverse diamond composition: it assembles upon input and dissembles upon output. The former combines the program input stream (*i.e.* S_i) with the output stream of the feedback (*i.e.* S_D) to form the input stream of sub-program P_A (*i.e.* S_A), and the latter divides the output stream of P_A (*i.e.* S_C) into the program output stream (*i.e.* S_o) and the input stream of sub-program P_B (*i.e.* 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 chain P_A with an imaginary filter — P_{AA} in Figure 9 — and use the shared stream in between the two (*i.e.* S_{AA}) as the "buffer" for the additional data produced by P_A reduction. P_{AA} intuitively is an "identity" filter that places every input data item to the output as is. Predicate $\texttt{identity}\,(P)$ holds iff $P = F^L[1,1]$ for some L and F is the identity function. The same scheme is used for treating the post-reduction output of P_B. The identifiers of the two additional introduced streams — S_{AA} and S_{BB} — are the two program labels associated with the circle construct.