Rate Types for Stream Programs

Thomas W. Bartenstein
SUNY Binghamton
tbarten1@binghamton.edu

Yu David Liu
SUNY Binghamton
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 centers around static 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. Our work is built upon a key insight that, even though streams are fundamentally dynamic, two essential concepts of stream rate control — throughput ratio and natural rate — are intimately related to the program structure itself. We define a type system to reason about these two concepts, and formally prove type soundness over a time-aware and parallelism-aware operational semantics. We further demonstrate the usefulness of our formal framework in applications such as 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 [28, 32]. A stream program is composed of data-processing units connected in intricate “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 brings data transformation and data flow to the forefront of the programming experience, attractive for developing data-intensive applications. Furthermore, stream programs are easily parallelizable, known to yield scalable high-performance on CPUs [14], GPUs [33], and FPGAs [2]. Stream programming — together with its close relatives of data-flow models — is successful in scientific computing [30], graphics [22], databases [7], GUI design [11, 18], robotics [12], sensor networks [26], and network switches [20]. Its growing popularity also generates significant interest in developing theoretical foundations for stream programming [6, 27, 29].

In this paper, we develop a novel theoretical foundation to reason about the data aspect of stream programming. The centerpiece of this work is Rate Types, the first type system we know of 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 upper bound of the output stream rate regardless of its input stream rate.

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 runtime. 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 performance reasoning of data-flow programming models. Quantitative reasoning of performance-related properties is an active area of research in the programming language community (e.g. [4, 10, 15, 16, 36]), but the vast majority of existing efforts focus on control-flow-centric programming models. Rate Types is one of the first formal systems to demonstrate the importance and viability of quantitative reasoning over data-flow models.

In particular, 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 (throughput ratio and natural rate in our system) characteristic of program structures. (2) Type systems have strong support for modularity, which in our case spearheads a flavor of compositional performance reasoning. Imagine the output stream of the “decoder” is being fed to another stream program, say an “interpolator.” Rate Types allows the stream rate characteristics of ‘decoder’ and ‘interpolator’ to be reasoned about independently, and determines whether the two can be composed to meet particular stream rate expectations. (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 further demonstrate the applications of our theoretical framework and bring it closer to real-world computing, we extend Rate Types to illustrate how the framework can be used for energy management and CPU allocation on multi-core architectures. The relationship between energy consumption and performance, and that of scheduling and performance, have long been actively explored in experimental research. Our approach is a first step to formally illuminate this complex landscape that involves program structure, performance (stream throughput), CPU energy consumption, and scheduling.

This paper makes the following contributions:

- It develops a novel type system to reason about throughput ratio and natural rate of stream programs, and formally establishes
the correlation between the stream rates reasoned about by the type system and those manifest at run time – as a type soundness property. The run-time behaviors of stream programs are defined via a time-aware and parallelism-aware operational semantics friendly for stream rate computation.

• It defines a type inference to infer the throughput ratio and the natural rate. The inference is sound and complete relative to the type checking algorithm, and further enjoys principal typing, which translates as 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 processing units may be executed on CPUs whose frequencies are dynamically adjustable.

• It applies the type system to assist CPU allocation optimization, in a setting where multiple instances of the same stream processing units may be scheduled at the same time to support data parallelism.

2. Stream Programming and Reasoning

We now overview 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 [32] 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 [17], a classic optimization algorithm that probabilistically finds globally optimal solutions through randomized locally optimal search. The goal of this oversimplified example is to find the coordinate with the best profit from a large 2D space of coordinates. Given a number of “seed” coordinates as the input stream, this program fragment (entry at anneal in Line 2) takes each input coordinate, checks its 8 neighbors in the 2D space, and picks one with best benefit from the neighbors and itself (checkNeighbors in Line 8). The best coordinate is thus fed back for the next round of space search. This neighborhood-based strategy may “trap” the search to local optimality but not the global one. The program thus is equipped with a “jump” strategy to allow some coordinates to be randomly mutated (randomJump in Line 39).

The base-case processing unit of a stream program is a filter, such as getNeighbors in Line 13, whose main body is a function labeled with keyword work. A filter execution instance (i.e., one function application) takes in a finite number of data items from the input stream (through push) and places a finite number of data items to the output stream (through push). For instance in Lines 15-19, the filter places 9 coordinates on the output stream for each coordinate it reads from the input stream. A filter execution instance can only be launched when there are enough data items on the input stream.

Assuming there are enough data items on their individual input streams, different filters — such as getNeighbors and an evalNeighbors — can execute in parallel. In the most simple conceptual model, the number of parallel units for a stream program is equal to the number of (inlined) filter uses, e.g., 2 getProfit execution instances and 1 execution instance for each of the other filters. Similar to other concurrency models such as actors [1], a filter execution instance takes “one firing at a time”: an application to its function body must be completed before the filter can take more data items off the input stream and start another function application. This last requirement can be relaxed when one considers data-parallel scheduling, a topic we will cover shortly.

The expressiveness of stream languages mostly comes from their rich support for combining filters. To stay neutral to the terminology of host languages, we name the 3 most commonly used combinators as follows:

• Chain (pipeline in StreamIt): a combinator to connect the output stream of one sub-program to the input stream of another. For example, checkNeighbors in Line 8 “chains” together the output stream of getNeighbors and the input stream of computeProfits.

• Diamond (splitJoin in StreamIt): a combinator to dissemble and assemble data streams. For example, computeProfits in Line 21 says that the data items on the input stream will be alternately 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 2 says that for every 100 coordinates produced by checkNeighbors, 99 are fed back for randomJump processing. The post-processing coordinates are fed to checkNeighbor again. Every time 99 coordinates are fed back, 1 more new coordinate (additional “seed” coordinates) will be admitted for annealing.
Stream Reasoning  For stream applications such as simulated annealing, a large number of calculations are involved, so high performance is often a matter of necessity. 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 \( n_1 \) data items per second when its input is fed with \( n_2 \) items per second?

Q2: What is the upper bound for a program’s data production, given unlimited rates for input data?

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 sound framework.

To model real-world computing in a more faithful manner, we further extend our framework to establish its relationship with the settings of CPU frequencies, and its relationship with filter scheduling. Surprisingly, these expressive features only require 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) [23]? 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 to support data parallelism, what is the fewest number of parallel execution instances for each filter — 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 \( \mathbb{int} \) 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; (b) formal systems to reason about individual filter behaviors exist [6]; (c) Experimentally, filter execution time is known to be stable through profiling. Core optimizations of the StreamIt compiler [14] rely on it; (d) real-world software development is iterative. Profiling-guided typing is not new [13]. What matters is to help programmers reason about performance at some point during the software life cycle.

**Applicability**  RATE TYPES are primarily designed for expressive and general-purpose stream languages [28, 32]. 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. [26]), where determining the lowest sensing rate possible is relevant; (b) signal languages such as FRP [12]. 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 [8] and ParaTimer [21], where single MapReduce-like units are composed together in expressive ways.

### Syntax and Dynamic Semantics

#### Abstract Syntax

The syntax of our formal language is defined in Figure 2. A program \( P \) is either a filter \((F_t^{\ell}[i_1, n_1])\), a chain composition \((P \circ \ell \gamma P')\), or a circle composition \((P \circ \circ \circ \circ \circ \circ \circ \gamma P')\). Each filter is defined with a unique filter (program) label \( L \), declared with the number of items to be consumed at a time \( n_1 \) (the \texttt{pop} declaration in Figure 1), the number of items to be produced at a time \( n_2 \) (the \texttt{push} declaration), and the filter function body \( F \) (the \texttt{work} function). For each filter, we further require \( F \) to be an element of \( \mathbb{DATA}^+ \rightarrow \mathbb{DATA}^+ \) where \( \mathbb{DATA} \) is the set of data items. Each chain construct is also associated with a distinct program label, whereas the circle construct is associated with two. Their purpose will be explained shortly. Given a program \( P \), we use \texttt{label1} to enumerate all filter labels, and \texttt{label2} to enumerate all other labels. As we shall see, the labels in \texttt{label1} will be used as identifiers of streams at run time, which we call stream (program) labels. Metaversary \( \delta \) and \( \alpha \) represent the distribution factor and the aggregation factor respectively. They correspond to the tuples succeeding the \texttt{split roundrobin} and the join \texttt{roundrobin} in the example respectively.

The program in Figure 1 can be represented by our syntax as:

\[
P_{\text{anneal}} \triangleq P_{\text{checkNeighbors}} \diamond \langle \ell, 1 \rangle P_{\text{computeProfits}} \diamond \langle \alpha, 1 \rangle P_{\text{randomJump}}
\]

\[
P_{\text{checkNeighbors}} \triangleq P_{\text{getNeighbors}} \diamond \langle \ell, 2 \rangle P_{\text{pavalNeighbours}}
\]

\[
P_{\text{getNeighbors}} \triangleq F_{\text{p1}}[1, 9]
\]

\[
P_{\text{computeProfits}} \triangleq F_{\text{p2}}[1, 1] \diamond \langle \ell, 1 \rangle F_{\text{p3}}[1, 1]
\]

\[
P_{\text{pavalNeighbours}} \triangleq F_{\text{p4}}[9, 1]
\]

\[
P_{\text{randomJump}} \triangleq F_{\text{p5}}[1, 1]
\]

In the rest of the paper, we use notation \([X_1, \ldots, X_n]\) to represent a sequence with elements \(X_1, \ldots, X_n\), or simply \(X\) when sequence length does not matter. Furthermore, we define \(\{X_1, \ldots, X_n\} \defas X\) and use comma for sequence concatenation, i.e. \([X_1, \ldots, X_n, Y_1, \ldots, Y_m] \defas [X_1, \ldots, X_n, Y_1, \ldots, Y_m]\). We call a sequence in the form of \([X_1 \mapsto Y_1, \ldots, X_n \mapsto Y_n]\) a mapping sequence if \(X_1, \ldots, X_n\) 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 \(\text{dom}(M) \defas \{X_1, \ldots, X_n\}\) and \(\text{ran}(M) \defas \{Y_1, \ldots, Y_n\}\) and...

<table>
<thead>
<tr>
<th>(P)</th>
<th>(F_t^{\ell}[i_1, n_1])</th>
<th>(P \circ \ell \gamma P')</th>
<th>(P \circ \circ \circ \circ \circ \circ \circ \gamma P')</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta)</td>
<td>(\langle n; n'\rangle)</td>
<td>(\text{distribution factor})</td>
<td></td>
</tr>
<tr>
<td>(\alpha)</td>
<td>(\langle n; n'\rangle)</td>
<td>(\text{aggregation factor})</td>
<td></td>
</tr>
<tr>
<td>(L)</td>
<td>(\mathbb{FLAB})</td>
<td>(\text{(filter) program label})</td>
<td></td>
</tr>
<tr>
<td>(\ell)</td>
<td>(\mathbb{SLAB})</td>
<td>(\text{(stream) program label})</td>
<td></td>
</tr>
<tr>
<td>(n)</td>
<td>(\mathbb{NAT}^+)</td>
<td>(\text{natural number})</td>
<td></td>
</tr>
</tbody>
</table>
\[ R \ ::= \ell \mapsto S \quad \text{program runtime} \]
\[ S \ ::= \overline{a} \quad \text{stream} \]
\[ t \in \text{TIME} \subseteq \text{REAL} \cap \text{NONNEG} \quad \text{time} \]
\[ d \in \text{DATA} \subseteq \text{FLOAT} \quad \text{data item} \]
\[ \Pi \in \text{FLAB} \mapsto \text{TIME} \quad \text{filter time mapping} \]

**Figure 3.** Program Runtime Definition

\[
(n + n') \times u \leq |S| < (n + n') \times (u + 1) \\
S = [d_{a1}, \ldots, d_{an}, d_{b1}, \ldots, d_{bn}, \\
d_{a_{n+1}}, \ldots, d_{a_{2n}}, d_{b_{n+1}}, \ldots, d_{b_{2n}}, \\
\vdots, \\
d_{a_{(u-1)n+n'+1}}, \ldots, d_{a_{un+n'}, d_{b_{(u-1)n+n'+1}}, \ldots, d_{b_{un+n'}}] \\
\]

\[
S_{\delta(n,n')} [d_{a1}, d_{a2}, \ldots, d_{a_{un+n'}}, d_{b1}, d_{b2}, \ldots, d_{b_{un+n'}}] \\
(n + n') \times u \leq |S| < (n + n') \times (u + 1) \\
S = [d_{a1}, \ldots, d_{an}, d_{b1}, \ldots, d_{bn}, \\
\vdots, \\
d_{a_{(u-1)n+n'+1}}, \ldots, d_{a_{un+n'}, d_{b_{(u-1)n+n'+1}}, \ldots, d_{b_{un+n'}}] \\
\]

\[ \overline{\delta} \in S \Rightarrow \overline{\delta} \in S \]

**Figure 4.** Stream Disassembly and Assembly

use notation \( M(X_i) \) to refer \( Y_i \) for any \( 1 \leq i \leq n \). Binary operator \( \sqcup \) is defined as \( M_1 \sqcup M_2 = M_1 \sqcup M_2 \) iff \( \text{dom}(M_1) \cap \text{dom}(M_2) = \emptyset \).

The operator is otherwise undefined.

**Stream Program Runtime** As defined in Figure 3, the runtime state of a stream program, \( R \), consists of a mapping sequence from stream labels to streams (\( S \)). Two built-in identifiers \( \ell_{\text{IN}} \) and \( \ell_{\text{OUT}} \) are used to facilitate the semantics development. Given a program \( P \), \( \ell_{\text{IN}} \) and \( \ell_{\text{OUT}} \) roughly coincide with the intention of identifying the input stream and the output stream of \( P \).

Each stream is a sequence of data items. For simplicity, we only consider the case where all data items are of \( \text{FLOAT} \) type. Supporting programs with data flows of different types has been routinely formalized (such as numerous FRP languages), and is orthogonal to our interest of reasoning about rates here.

Ternary predicate \( S \downarrow \delta \) \( S_A, S_B \) holds when \( S \) can be dissembled to \( S_A \) and \( S_B \) according to distribution factor \( \delta \), whereas \( S_A, S_B \uparrow\alpha \) \( S \) holds when \( S_A \) and \( S_B \) can be assembled together as \( S \) according to aggregation factor \( \alpha \). Their definitions appear in Figure 4. The round-robin nature resonates in both predicates. For example, if \( S = [1, 2, 3, 4, 5, 6], S_A = [1, 4], S_B = [2, 3, 5, 6], \) both \( S \downarrow_{\{1, 2\}} S_A, S_B \) and \( S_A, S_B \uparrow_{\{1, 2\}} S \) hold. However, that it is not possible to encode one of these operators with the other because of the requirement that each round of the round-robin must be complete. The following examples should illuminate this point:

\[
[1, 2, 3, 4, 5, 6, 7] \downarrow_{\{1, 2\}} [1, 4], [2, 3, 5, 6] \\
[1, 4, 7], [2, 3, 5, 6] \uparrow_{\{1, 2\}} [1, 2, 3, 4, 5, 6] 
\]

**Operational Semantics** The operational semantics of our language is defined in Figure 6. The ultimate goal of this semantics is to account for stream rates, and this leads to several distinct traits of our rules. First, we need to “count the beans”. The number of data items on the input stream, the number of data items on the output streams, and in particular, how they change, need to be carefully accounted for. Second, the semantic system needs to be time-aware, because as we shall see, the execution time of individual filter execution instances affects stream rates; Third, 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 semantics is defined as big-step reductions. Relation \( \vdash \text{d} \)

\[ R \vdash_{\ell} R' \] says the runtime of program \( P \) transitions from \( R \) to \( R' \) in time \( t \). Metavariable \( t \) represents a non-negative (\( \text{NONNEG} \)) floating point number. The relation is reflexive and transitive, as indicated by [D-Reflex] and [D-Trans]. A graphical representation of the rest of the rules is provided in Figure 5, where streams are explicitly represented with green arrows and properly labeled with metavariables used in individual rules.

[D-Filter] models the behavior of the filter. In each execution, it takes off \( n_t \) data items (\( S_{\text{IN}} \)) from the tail of the input stream — as represented by \( \ell_{\text{IN}} \) — applies function \( F \) to it, and places \( n_o \) number of data items \( S_{\text{OUT}} \) on the head of the output stream — as represented by \( \ell_{\text{OUT}} \).

We rely on a pre-defined mapping function \( \Pi \) to map each filter to its execution time. Recall in Section 2, we described this execution time as the base case assumption of our type system. The \( \Pi \) mapping here will reappear in the type system.

In [D-Chain], the most important observation is that the data flow dependency of the two sub-programs \( P_A \) and \( P_B \) does not prevent parallelism. The connection of \( P_A \) and \( P_B \) is that they share one stream — before reduction and after reduction — and that stream is identified by the program label \( \ell \) in the rule. The two parallel reductions start with the same shared stream data (\( S_1, S_1 \)) — as the output stream of \( P_A \) and as the input stream of \( P_B \). After reduction, the shared stream needs to be composed by removing the data items already consumed by \( P_B \) (i.e. \( S_1 \)) and adding the data items further produced by \( P_A \) (i.e. \( S_3 \)). Thanks to parallelism, the [D-Chain] reduction time is only bound by the longer reduction of \( P_A \) and \( P_B \). Furthermore, observe that [D-Chain] further subsumes the case when only one of the \( P_A \) and \( P_B \) reduces, because of [D-Reflex].
Similar to [D-Chain], [D-Diamond] also enables parallel reductions of the two sub-programs \( P_A \) and \( P_B \). Its key behavior is to dissemble the input stream into two , feeding each to the respective input stream of the two sub-programs, and assemble the two output streams of the two sub-programs as the output stream of the entire program. Predictably, predicates \( \lambda \) and \( \gamma \) are used. Each appears twice to account for both pre-reduction and post-reduction. Another intuitive way to interpret the roles of the two predicates is that \( \lambda \) allows one stream to be “viewed” as two, whereas \( \gamma \) allows two streams to be “viewed” as one. This is the fundamental reason why our stream runtime does not need to maintain an extra “lens” them because that would lead to the next iteration.

As illustrated in Figure 8, the key insight for understanding [D-Circle] is a circle composition is (roughly) a reverse diamond composition: it assembles upon 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. \( S_{A1} \), \( S_{A1} \)), and the latter divides the output stream of \( P_A \) (i.e. \( S_{C1} \)) into the program output stream (i.e. \( S_o \)) and the input stream of sub-program \( P_B \) (i.e. \( S_{B1} \), \( S_{B1} \)). 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 — \( \delta_{P_A} \) in Figure 8 — 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. \( \delta_{P_A} \) intuitively is an “identity” filter that places every input data item to the output as is. Predicate \( \delta_{P_A} \) holds iff \( P = F^L[1,1] \) for some \( L \) and \( F \) is the identity func-

---

**Figure 6. Operational Semantics**

\[
\begin{align*}
\text{D-Reflex} & \quad \vdash A \rightarrow B \quad \text{ID}\quad \text{R}\quad \text{R}' \quad \text{t} + t_2 \leq t \\
\text{D-Trans} & \quad \vdash A \rightarrow B \quad \text{ID}\quad \text{R}\quad \text{R}' \quad \quad \text{t} + t_2 \leq t \\
\text{D-Filter} & \quad \vdash A \rightarrow B \quad \text{ID}\quad \text{R}\quad \text{R}' \quad \quad \text{t} + t_2 \leq t \\
\text{D-Chain} & \quad \vdash A \rightarrow B \quad \text{ID}\quad \text{R}\quad \text{R}' \quad \quad \text{t} + t_2 \leq t \\
\text{D-Diamond} & \quad \vdash A \rightarrow B \quad \text{ID}\quad \text{R}\quad \text{R}' \quad \quad \text{t} + t_2 \leq t \\
\text{D-Circle} & \quad \vdash A \rightarrow B \quad \text{ID}\quad \text{R}\quad \text{R}' \quad \quad \text{t} + t_2 \leq t \\
\end{align*}
\]
tion. 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.

Properties and Bootstrapping The operational semantics enjoys several simple properties, which we state now. All proofs can be found in the accompanying technical report.

Lemma 1 (Stream Count Preservation). If $\vdash_d R \xrightarrow{t} R'$, then $\text{dom}(R) = \text{dom}(R') = \text{slables}(P) \cup \{\ell_{IN}, \ell_{OUT}\}$.

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

A technical question for bootstrapping a program with feedback loops is that one needs to prime the loop. For instance, the anneal circle construct in Figure 1 cannot “jump start” until the output stream of $\text{randomJump}$ starts with 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 $\text{dom}(R)$ is the smallest set of $\ell$ where $P_1 \cap \alpha, \delta, P_2$ is a sub-program of $P$. $\alpha = \langle \ell; n \rangle$ and $|R(\ell)| = n$.

Definition 1 (Bootstrapping Runtime). Given a stream program $P$, an input stream $S_0$, and a primer $R_0$ of $P$, we say its initial runtime — denoted as $\text{init}(P, S_0, R_0)$ — as the smallest $R$ satisfying the following conditions:

- $R(\ell_{IN}) = S_0$.
- $R(\ell_{OUT}) = \emptyset$.
- $R_0 \subseteq R$.
- $R(\ell) = \emptyset$ for any $\ell \in \text{slables}(P) \cap \text{dom}(R_0)$.

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

Definition 2 (Stream Size Change Rates). Given a reduction from $R$ to $R'$ over time $t$, the rate for stream $\ell$ is defined by function $rchange()$:

$$rchange(R, R', t, \ell) \triangleq \frac{\text{abs}(|R'(\ell)| - |R(\ell)|)}{t}$$

where unary operator $\text{abs}()$ 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 size change rate is $(45 - 25)/8 = 2.5$ data items per second. From this point on, we use metavariable $r$ to represent stream size change rates. $r \in \text{FLOAT} \cap \text{NONNEG}$. Now 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. Thus, our intuitive notion of “input stream rate” happens to be aligned with the stream size change rate for the program input stream, and the “output stream rate” is aligned with the stream size change rate for the program output stream. We define:

Definition 3 (Input/Output Stream Rates). Given a reduction from $R$ to $R'$ over time $t$, we define:

$$\text{rti}(R, R', t) \triangleq rchange(R, R', t, \ell_{IN})$$
$$\text{rto}(R, R', t) \triangleq rchange(R, R', t, \ell_{OUT})$$

Encodings Before moving on to the static system, we discuss some commonly used stream programming idioms encodable by our core calculus. First, a k-way fork-join of programs $P_1, \ldots,
\[ \tau ::= (\theta; \nu) \quad \text{stream rate type} \]
\[ \theta \in \text{TR} \subseteq \text{FLOAT} \cap \text{NONNEG} \quad \text{throughput ratio} \]
\[ \nu \in \text{FLOAT} \cap \text{NONNEG} \quad \text{natural rate} \]

**Figure 9. Type Elements**

\[ P_k \] with distribution factor \( \langle n_1; \ldots; n_k \rangle \) and aggregation factor \( \langle n'_1; \ldots; n'_k \rangle \) can be encoded as:
\[ P_1 \circ_1 \circ_2 \ldots \circ_{n-1} \circ_{n-1} \ldots \circ_{n-1} (P_1 \circ_1 \circ_2 \ldots (P_{k-1} \circ_{n-1} \circ_{n-1} \ldots \circ_{n-1} P_k))) \]
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 \( i = 1, \ldots, n - 1 \).

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 \( F^L[1, 2] \circ_1 \circ_2 \ldots \circ_{n-1} \circ_{n-1} \ldots \circ_{n-1} P_k \) where \( F \) is a function that takes \( \langle x \rangle \) and returns \( \langle x \rangle \) 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 \( (P_1 \circ_1 \circ_2 \ldots \circ_{n-1} \circ_{n-1} \ldots \circ_{n-1} P_k) \circ [1, 2, 1] \) where \( F \) represents the binary operator and \( \Pi(L) = 0 \). The example above, \( F \) is the function that takes \( \langle x, y \rangle \) and returns \( \max(x, y) \) where \( \max(x, y) \) is the standard maximum operator.

Last, our formal system idealizes the data transfer across buffers. Consider the chain composition \( P_1 \circ_1 \circ_2 \) 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 \( P_1 \circ_1 \circ_2 \ldots \circ_{n-1} \circ_{n-1} \ldots \circ_{n-1} P_k \) where \( F \) is the identity function, and \( \Pi(L) = 0 \) is time needed for buffer access.

**4. Rate Types**

In this section, we describe our type system. We first present a type checking algorithm, followed by a type inference algorithm proven to be sound and complete relative to the former.

**Types** Types in our system are tuples in the form of \( \langle \theta; \nu \rangle \), described in Figure 9. The throughput ratio, \( \theta \), statically characterizes the ratio of the output stream rate over the input stream rate. When the input stream rate is zero (no change) we define the throughput ratio as zero if the output stream rate is zero, infinite otherwise. The natural rate, \( \nu \), 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 adopted by our type system reveals a fundamental phenomenon of stream rate control: the input stream rate and the output stream rate can be correlated by a ratio \( \theta \), but the correlation only holds when the output stream rate does not reach its upper bound \( \nu \). The correlation between the input stream rate and the output stream rate should come as no surprise. To gain intuition on the upper bound aspect of stream control, the key insight is that it takes time to process data. According to our informal discussion in Section 2 and our formal operational semantics, 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 unlimited rate.

**Figure 10. Rate Type Checking**

\[ \vdash \tau: \tau' < \tau \quad \text{[T-Sub]} \]
\[ \vdash \nu: \tau \quad \text{[T-Filter]} \]
\[ \vdash \theta: \theta \circ_1 \circ_2 \ldots \circ_{n-1} \circ_{n-1} \ldots \circ_{n-1} \nu: \tau \quad \text{[T-Chain]} \]
\[ \vdash \nu: \nu' \quad \text{[T-Diamond]} \]
\[ \vdash \theta: \theta \quad \text{[Sub]} \]

**Figure 11. Distribution/Aggregation Fractions**

\[ \lambda^1(\delta) \triangleq \frac{n}{n + n'} \quad \lambda^2(\delta) \triangleq \frac{n}{n + n'} \]
\[ \gamma^1(\alpha) \triangleq \frac{n}{n + n'} \quad \gamma^2(\alpha) \triangleq \frac{n}{n + n'} \]

**Type Checking** We use judgment \( \vdash \tau: \tau \) to denote program \( P \) has type \( \tau \). The typing rules are summarized in Figure 10. Several simple functions used in the rules — namely \( \lambda^1(\delta), \lambda^2(\delta), \gamma^1(\alpha) \) and \( \gamma^2(\alpha) \) — are defined in Figure 11. They can be informally viewed as computing a form of “normalized” distribution and aggregation factors.

Recall the two questions we raised 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 \( \vdash \tau: (n_1, n_2) \). Question Q2 — determining the upper bound of the output rate \( \nu \) — can be answered by finding out whether a derivation exists for \( \vdash \tau: \langle \theta; \nu \rangle \) for some \( \theta \).

[T-Sub] introduces subtyping where the subtyping relation is defined in [Sub]. 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 practice, this intuitive definition also coincides with the intentions of stream control: it would be counter-intuitive to answer Q1 with “no” (i.e. a type error) simply because the throughput ratio given is 0.3, whereas the highest possible throughput ratio is 0.4.
The throughputs in the two cases: (1) Figure 12(a): when sub-program \( P_B \) is fed with an input stream whose rate is high enough that the output stream of \( P_B \) reaches its natural rate \( \nu_B \), then \( \nu_B \) should also be the upper bound for the entire composition program. (2) Figure 12(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 \( \nu_A \), the output stream rate of \( P_B \) — and hence also the output stream rate of the entire program — is no higher than \( \nu_A \times \theta_a \). The natural rate of the entire program should be the minimal of the two, computed by the standard binary function \( \min(\cdot) \).

To understand the throughput ratio of a diamond composition defined in [T-Diamond], observe that throughput ratio can be viewed as the "normalized" output stream rate relative to the input stream rate. Let us consider the input stream rate be 1. Thanks to

\[
\frac{\nu_B}{\nu_A} = \min(\nu_A, \nu_B \times \theta_a)
\]

\[
\frac{\nu_B}{\nu_B} = \min(\nu_B, \nu_B \times \theta_a)
\]
Studying the convergence of such iterations is a well-known problem (such as in control theory) as the stability of feedback. More general solutions would determine the existence of — and if so compute — the fix point. Not to complicate our system with a full-fledged (yet standard) control-theoretic formulation, we choose to adopt a simple scheme to require convergence without iteration. From a type system perspective, this implies we may conservatively reject programs whose natural rate may stabilize after iterations, but we think the simplification sufficiently demonstrates our core philosophy here: our type system is stability-aware, and programs with unstable circle compositions should be rejected. The throughput ratio reasoning of [T-Circle] follows a very similar route. Other than the aspect of cyclic dependency, circle composition shares many traits with diamond composition, which should help readers understand the predicates involving \( \theta \) and \( \theta'_0 \) in the rule.

**Meta-Theories** We now demonstrate the type system does correctly capture 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 1** (Type Soundness). Given program \( P \) and the initial runtime \( R = \text{init}(P, S_0, R_0) \) for some \( S_0 \) and \( R_0 \), and if \( R \xrightarrow{d} R' \xrightarrow{i} R' \) and \( \text{rtto}(R, R', \ell) = r_1 \) and \( \text{rtti}(R', R, \ell) = r_2 \), then 
\[
\vdash \exists \ell : \langle r_1/r_2; r_1 \rangle.
\]

The theorem above corresponds program dynamic behaviors and typing. One drawback is it requires to relate 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 relating to the initial state)? The answer unfortunately is no. The root reason is that each stream program runtime may contain “intermediate streams”, i.e. streams that are not identified by \( \ell_\text{thr} \) and \( \ell_\text{gatt} \). 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 reasoned about by the static system is effective in characterizing arbitrary reduction steps too, as long as a stability condition is met:

**Theorem 2** (Type Soundness over Arbitrary Reduction Steps). Given program \( P \) and \( R \xrightarrow{d} R' \xrightarrow{i} R' \) where \( \text{rtto}(R, R', \ell) = r_1 \) and \( \text{rtti}(R', R, \ell) = r_2 \), and for any \( \ell \in \text{dom}(R) \) — \( \{\ell_\text{thr}, \ell_\text{gatt}\} \), \( \text{rchange}(R, R', \ell, \ell) = 0 \) then \( \vdash \exists \ell : \langle r_1/r_2; r_1 \rangle \).

Here we call the \( \text{rchange}() \) predicate in the theorem the stability condition. The theorem says two arbitrary states on the reduction sequence can be related — with the reduction(s) observing the throughput ratio and natural rate reasoned about by our type system — as long as the size of each intermediate stream before the reduction(s) is the same as one after the reduction(s). Note that the reduction relation \( \xrightarrow{d} \) is transitive (see the [D-Trans] rule), so this theorem does not require every small step maintain stability: it only requires the end state is stable 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 practice, we expect Theorem 1 and Theorem 2 are useful in complementary scenarios. Theorem 1 is more appropriate for characterizing short-running programs, where stability 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 stability. Theorem 2 says for long-running programs, we can still capture the rate behavior of a program execution, say, from its 68th minute to its 95th minute. The theorem further tolerates bursty behaviors, say in the 77th minute, as long as the 95th minute execution snapshot conforms to the stability condition relative to the 68th minute snapshot. This is sufficient to characterize long-running programs that 1) stabilize after an initial duration of time (but may be followed by intermittent bursty behaviors); 2) or demonstrate period 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 1 and 2 may be combined to characterize the same program execution, with the former addressing the beginning (potentially unstable) stage, and the latter for the rest.

### 5. Rate Type Inference

In this section, we define a constraint-based type inference for **Rate Types**. Figure 13 defines the related syntactical elements. The key element is \( \text{throughput ratio type variable} \ p \) and \( \text{natural rate type variable} \ q \), the type variable counterparts of \( \theta \) and \( \nu \). Each element in the set is either an equality constraint \( (e \equiv e) \) 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 \( (\text{minn}) \). To avoid confusion, we choose to use different symbols for syntactic elements.

**Figure 13. Type Constraints**

\[
\begin{align*}
\Sigma & := \tau & \text{constraints} \\
c & ::= e \preceq e | e \equiv e & \text{constraint expression} \\
e & ::= p | q | \text{minn}(c, e) | e \bullet e & \text{throughput ratio type variable} \\
\sigma & ::= p \leadsto \theta \cup q \leadsto \nu & \text{natural rate type variable} \\
p & \in TVAR & \text{p, q fresh} \\
q & \in TVAR & \text{p, q fresh} \\
\end{align*}
\]

\[
\begin{align*}
\vdash & \text{P}[\ell][n_\mathit{in}, n_\mathit{out}] : (p,q) \langle p \leq n_\mathit{in} / n_\mathit{out} \rangle & \text{[I-Filter]} \\
\vdash & \text{P} : (p,q) \langle p \leq q \rangle & \text{[I-Chain]} \\
\vdash & \text{P} : (p,q) \langle p \leq \text{minn}(q, p) \rangle & \text{[I-Diamond]} \\
\vdash & \text{P} : (p,q) \langle q \leq \text{minn}(p, q) \rangle & \text{[I-Circle]} \\
\end{align*}
\]
in constraints and those in predicates, whose pairwise relationships should be obvious: \(=\) and \(\leq\) and \(\leq\), and \(\times\)., \(\text{min}\) and \(\text{min}\).

We further define a function \(\sigma\) as a mapping from type variables to throughput ratios and natural rates. We use predicate \(\vartheta \subseteq \Sigma\) to indicate that \(\vartheta\) is a solution to \(\Sigma\). Formally, the predicate holds if every constraint is a tautology for a set identical to \(\Sigma\), except that every occurrence of \(p\) is substituted with \(\sigma(p)\), and every occurrence of \(q\) is substituted with \(\sigma(q)\).

Type inference rules are given in Figure 14. Judgment \(\vdash_{\Gamma} P : \langle p; q \rangle \Sigma\) says program \(P\) is inferred to have throughput ratio represented by type variable \(p\) and natural rate represented by type variable \(q\) under constraint \(\Sigma\). The rules have a one-to-one correspondence with the type checking rules we introduced in Figure 10. Indeed, the close relationship between the two can be formally established:

**Theorem 3** (Soundness of Inference). If \(\vdash_{\Gamma} P : \langle p; q \rangle \Sigma\) and \(\vartheta \vartriangleleft \Sigma\) then \(\vdash_{\Gamma} P : (\sigma(p), \sigma(q))\).

**Theorem 4** (Completeness of Inference). If \(\vdash_{\Gamma} P : (\vartheta, \nu)\), then \(\exists \sigma\) such that \(\sigma(p) = \vartheta, \sigma(q) = \nu\) and \(\vartheta \vartriangleleft \Sigma\), where \(\vdash_{\Gamma} P : \langle p; q \rangle \Sigma\).

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 \(\vdash_{\Gamma} P : \langle p; q \rangle \Sigma\), there exists a unique \(\sigma\) such that \(\vartheta \vartriangleleft \Sigma\), and for any \(\vartheta \vartriangleleft \Sigma\), \((\sigma(p); \sigma(q))\) \(\lll (\sigma'(p); \sigma'(q))\). We further call \((\sigma(p); \sigma(q))\) the principal type of \(P\).

This property has important consequence to stream rate reasoning. Recall in the previous section, subtyping \(\lll\) 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 \(\vdash_{\Gamma}^P P : \langle p; q \rangle \Sigma\). The typing rules for this judgment are identical to that of \(\vdash_{\Gamma}\), except that every occurrence of \(\leq\) is replaced with \(\leq\). The following theorem is an effective decision procedure for computing principal types for \(\vdash_{\Gamma}^P\):

**Theorem 6** (Effective Principal Type Computation). For any \(P\) such that \(\vdash_{\Gamma}^P P : \langle p; q \rangle \Sigma\) and \(\vartheta \vartriangleleft \Sigma\) holds, then \((\sigma(p); \sigma(q))\) 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 \((\vartheta; \nu)\). The answer to Question \(Q3\) — the minimal input stream rate given the output stream is expected to produce \(n\) items per second — is simply \(n/\vartheta\) 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 \(n \times \vartheta\) or \(\nu\), whichever is less.

### 6. Applications

#### 6.1 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. Related definitions are provided in Fig. 15.

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 pre-defined set of frequencies, which we model as an ordered set, \(\{\text{FREQ}; <\)

<table>
<thead>
<tr>
<th>(\Sigma^E)</th>
<th>(\Sigma^E)</th>
<th>(E) constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Sigma^E)</td>
<td>(\Sigma^E)</td>
<td>(E) constraints</td>
</tr>
<tr>
<td>(\Sigma^E)</td>
<td>(\Sigma^E)</td>
<td>(E) expr</td>
</tr>
<tr>
<td>(\Sigma^E)</td>
<td>(\Sigma^E)</td>
<td>(E) solution</td>
</tr>
</tbody>
</table>

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

DVFS is long known to be an effective energy management strategy (e.g.,\([3, 9, 23, 24]\)). 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 \(\text{energy}(L, f_q)\) to denote the energy consumption of filter \(L\) on frequency \(f_q\). We leave its definition abstract, and only axiomatically define that \(\text{energy}(L, f_{q_1}) < \text{energy}(L, f_{q_2})\) if \(f_{q_1} < f_{q_2}\).

We can now investigate into \(Q5\) posted in Section 2: identifying the highest possible output rate using the least possible energy. We achieve this by extending our type inference. The key insight here is \(\Pi^E\) introduces us a way to directly relate CPU frequencies to the type system. If we can introduce frequency variables — the 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.

Let us introduce a new system of type inference \(\vdash_{\text{ic}}\). Judgment \(\vdash_{\text{ic}} P : \langle p; q \rangle \Sigma^E\) is identical to \(\vdash_{\Gamma}\), except all metavariables should have the predictable E-superscript, and the rule for filter inference is updated as:

\[
\Sigma^E = \left\{ f \leq n_0/n_q, q \leq n_0 \Pi(L) \frac{\Pi(L)}{\text{FREQ}} \right\} \quad [\text{I-Filt}]
\]

where the simple bijective mapping \(\text{FREQ}\) allows us to find out the frequency variable name used for a particular filter \(L\) and \(\text{dom}(\text{FREQ}) = \{\text{FREQ} : \text{float} \cap \text{nonneg}\}\). Note that the second constraint in \(\Sigma^E\) is very similar to its counterpart \(q \leq n_0 / \Pi(L)\) in \([\text{I-Filt}]\), except that the execution time is \(\Pi(L) \times \max(\text{FREQ}) / \gamma\) instead of \(\Pi(L)\)
Next, let us define an ordering relation to “rank” constraint solutions with regard to DVFS settings. \( \sigma^E \leq \sigma^E \) if and only if \( \forall \nu \in \text{dom}(\sigma^E \cup \nu) \) \( \sigma^E(\nu) \leq \sigma^E(\nu) \). Finally, we can state our theorem on finding the optimal settings for DVFS without any performance degradation:

**Theorem 7** (DVFS-Optimal Solution). For a program \( \Gamma \in P \) where \( \Gamma \text{.}P : \langle p; q \rangle \cup E \), the GLB of \( \langle (\sigma^E | (\sigma^E \cup \nu) \cup \{ p = \theta, q = \nu \}) \rangle \) exists where \( \theta, \nu \) is the principal type of \( P \) according to \( \Gamma \). We call the GLB the DVFS-optimal solution for \( P \).

This important theorem comes with some subtleties. First, observe that the type inference algorithm \( \Gamma \text{.} \) uses \( \Pi \), 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 \( \theta \) and \( \nu \) is the highest possible throughput ratio and natural rate. Combining the two observations, \( \theta \) and \( \nu \) 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 \( \{ \} \cup \{ \theta = \theta, \nu = \nu \} \) 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 \( \sigma^E \) is the DVFS-optimal solution for \( P \), then \( \forall \nu \in \text{dom}(\sigma^E \cup \nu) \) \( \sigma^E(\nu) \leq \sigma^E(\nu) \) for some \( \nu \) and \( q \).

\[
\sum_{L \in \text{labels}(P)} \text{energy}(L, \sigma^E(\Gamma^E(L))) \leq \sum_{L \in \text{labels}(P)} \text{energy}(L, \sigma^E(\Gamma^E(L)))
\]

### 6.2 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 \( k \) 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?

Related definitions of our extension is provided in Fig. 16. 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 \( \Pi \) to \( \Pi^A \), the filter execution time at a given number of CPU allocation, defined as:

\[
\forall \Pi^A(L, k) = \Pi(L)/k
\]

In other words, we use the original \( \Pi \) 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 \( \Pi^A(L, k) \).

We posit another extension to our rate inference rules to manage the allocation of cores to filters. As we did in the previous subsection, let us introduce a new system of type inference \( \Gamma \text{.} \). Judgment \( \Gamma \text{.} P : \langle p; q \rangle \cup E \) is identical to \( \Gamma \text{.} \), except all metavariables should have the predictable A-superscript, and the rule for filter inference is updated as:

\[
\begin{align*}
\Sigma^A & \equiv \langle p \leq n_o/k \rangle \\
\sigma^A & \equiv \{ p \leq n_o/k \} \cup \{ p = \theta, q = \nu \}
\end{align*}
\]

The simple bijective mapping \( \Sigma^A \) allows us to find out the allocation variable name used for a particular filter \( L \) and \( \text{dom}(\Sigma^A) = \text{labels}(P) \).

Next, let us define an ordering relation to “rank” constraint solutions with regard to CPU allocation. \( \sigma^A \leq A \sigma^A \) if and only if \( \forall \nu \in \text{dom}(\sigma^A \cup \nu) \) \( \sigma^A(\nu) \leq \sigma^A(\nu) \). Finally, we can state our theorem on finding the minimal CPU allocation without any performance degradation:

**Theorem 8** (Allocation-Optimal Solution). For a program \( P \) where \( \Gamma \text{.} P : \langle p; q \rangle \cup E \), then the GLB of \( \langle (\sigma^A | (\sigma^A \cup \nu) \cup \{ p = \theta, q = \nu \}) \rangle \) exists where \( \nu \) is the requested output rate. We call the GLB the allocation-optimal solution for \( P \).

The theorem here provides a direct answer to Q6 in Section 2. Here the allocation-optimal solution will allocate each filter (of label \( L \)) with \( \sigma^A(\Gamma^A(L)) \) number of cores, where \( \sigma^A \) 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.

### 7. Related Work

In this section, we discuss related work, with a focus on techniques designed for stream/language/dataflow languages. Due to space limitation, we do not offer a comprehensive survey of the broader scope which would have included historically significant systems such as Kahn process networks, synchronous data flows (Ptolemy), Lustre, and classic FRP.

**Stream Type System Support** StreamFlex [28] is a Java-based stream programming framework. Its ownership type system is aimed at enforcing memory safety, especially non-shared memory access from different filters. A dependent type system [25] was designed for FRP to enforce productivity: a liveness property to guarantee the program continues to deliver output. Krishnaswami et al. [19] introduced a linear type system to bound resource usage (especially space) in higher-order FRP. Suenaga et al. [29] 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 [11] is a FRP-family language designed for GUI design. The type system of Elm is designed to outlaw higher-order signals, a construct that may lead to inefficiencies for GUI programming. None of the related work above reasons about the rate of data processing.

**Stream Semantics** Brooklet [27] is a calculus designed as an intermediate language into which diverse flavors of stream languages can be translated. A denotational semantics [31] was defined during the early development of StreamIt [32]. It is common to define semantics for signal languages and dataflow languages, either operational (e.g. [35]) or denotational (e.g. [34]) or both (e.g. [19]). Our operational semantics focuses on precise accounting of data rates.
Reasoning about Performance There is a large body of work on reasoning about performance-related properties of programs. A very incomplete list includes WCET [36], cost semantics [4], resource bound certification [10], resource usage analysis [16], amortized resource analysis [15], and Energy Types [9]. A common thread of these systems is to reason about resource consumption of programs in control-flow languages. Our system offers a new perspective of quantitative performance reasoning by focusing on data rates of data-flow programming models, and their impact on energy consumption and CPU allocation.

Experimental Approaches In experimental research, data throughput is a critical metric to evaluate the performance of stream systems. On multi-core CPUs, the problem is often considered as a form of load balancing, such as stream graph partitioning and steady-state scheduling [14]. An active research area is to minimize energy consumption with least performance penalty (e.g., [24]). 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 [3] achieves energy efficiency by DVFS through a program analysis. Our discussion in Sec. 6.1 can be conceptually viewed as a type system interpretation of their analysis.

8. 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.

References