Logical Methods in Computer Science
Vol. 12(3:11)2016, pp. 1–38
www.lmcs-online.org

Submitted Dec. 30, 2015
Published Sep. 22, 2016

DATA OPTIMIZATIONS FOR CONSTRAINT AUTOMATA

SUNG-SHIK T.Q. JONGMANS,[a] AND FARHAD ARBAB,[b]

[a] Open University of the Netherlands, Radboud University Nijmegen, the Netherlands
e-mail address: ssj@ou.nl

[b] Centrum Wiskunde & Informatica, Leiden University, the Netherlands
e-mail address: farhad@cwi.nl

Abstract. Constraint automata (CA) constitute a coordination model based on finite automata on infinite words. Originally introduced for *modeling* of coordinators, an interesting new application of CAs is *implementing* coordinators (i.e., compiling CAs into executable code). Such an approach guarantees correctness-by-construction and can even yield code that outperforms hand-crafted code. The extent to which these two potential advantages materialize depends on the smartness of CA-compilers and the existence of proofs of their correctness.

Every transition in a CA is labeled by a "data constraint" that specifies an atomic data-flow between coordinated processes as a first-order formula. At run-time, compiler-generated code must handle data constraints as efficiently as possible. In this paper, we present, and prove the correctness of two optimization techniques for CA-compilers related to handling of data constraints: a reduction to eliminate redundant variables and a translation from (declarative) data constraints to (imperative) data commands expressed in a small sequential language. Through experiments, we show that these optimization techniques can have a positive impact on performance of generated executable code.

1. Introduction

Context. In the early 2000s, hardware manufacturers shifted their attention from manufacturing faster—yet purely sequential—unicore processors to manufacturing slower—yet increasingly parallel—multicore processors. In the wake of this shift, *concurrent programming* became essential for writing scalable programs on general hardware. Conceptually, concurrent programs consist of *processes*, which implement modules of sequential computation, and *protocols*, which implement the rules of concurrent interaction that processes must abide by. As programmers have been writing sequential code for decades, programming processes poses no new fundamental challenges. What *is* new—and notoriously difficult—is programming protocols.

In ongoing work, we study an approach to concurrent programming based on syntactic separation of processes from protocols. In this approach, programmers write their processes

2012 ACM CCS: [**Theory of computation**]: Models of computation—Concurrency; Semantics and reasoning—Program semantics.

Key words and phrases: protocols, constraint automata, Reo, compilation, optimization, performance.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-12(3:11)2016
© S.-S.T.Q. Jongmans and F. Arbab
Creative Commons

in a *general-purpose language* (GPL), while they write their protocols in a complementary *domain-specific language* (DSL). Paraphrasing the definition of DSLs by Van Deursen et al. [vDKV00], a DSL for protocols "is a programming language that offers, through appropriate notations and abstractions, expressive power focused on, and [..] restricted to, [programming protocols]." In developing DSLs for protocols, we draw inspiration from existing *coordination models* and *languages*, which typically provide high-level constructs and abstractions that more easily compose into correct—relative to programmers' intentions—protocol code than do lower-level synchronization mechanisms (e.g., locks or semaphores). Significant as their software engineering advantages may be, however, performance is an important concern too. A crucial step toward adoption of coordination models and languages for programming protocols is, therefore, the development of compilers capable of generating efficient lower-level protocol implementations from high-level protocol specifications.

Our current work focuses on developing compilation technology for *constraint automata* (CA) [BSAR06, Jon16a], a coordination model based on finite automata on infinite words, originating from research on the coordination language Reo [Arb04, Arb11]. Every CA models (the behavior of) a *coordinator* that enforces a protocol among coordinated processes. Structurally, a CA consists of a finite set of states, a finite set of transitions, a set of directed *ports*, and a set of local *memory cells*. States model the internal configurations of a coordinator; transitions model a coordinator's atomic coordination steps. Ports constitute the interface between a coordinator and its coordinated processes, the latter of which can perform blocking I/O-operations on their coordinator's ports: a coordinator's *input ports* admit `put` operations, while its *output ports* admit `get` operations. Memory cells model a coordinator's internal buffers to temporarily store data in. Different from classical automata, transition labels of CAs consist of two elements: a set of ports, called a *synchronization constraint*, and a logical formula over ports and memory cells, called a *data constraint*. A synchronization constraint specifies which ports need an I/O-operation for its transition to fire (i.e., those ports synchronize in that transition and their pending I/O-operations complete), while a data constraint specifies which particular data those I/O-operations may involve. Every CA, then, constrains *when* I/O-operations may complete on *which* ports.

Problem. Briefly, our current CA-to-Java compiler translates passive data structures for CAs into (re)active "coordinator threads". A coordinator thread is, effectively, a state machine whose transitions correspond one-to-one to transitions in a CA. Essentially, then, compiler-generated coordinator threads simulate CAs by firing their transitions, continuously monitoring run-time data structures for their ports.[1]

To actually fire a transition, a coordinator thread must first check both that transition's synchronization constraint and its data constraint. The check for the synchronization constraint ensures that all ports involved in the transition have a pending I/O-operation (and are thus ready to participate in the transition); the check for the data constraint subsequently ensures that those pending I/O-operations can result in admissible data-flows.

[1]One needs to overcome a number of serious issues before this approach can yield practically useful code. Most significantly, these issues include exponential explosion of the number of states or transitions of CA, and *oversequentialization* (i.e., the situation where coordinator threads unnecessarily prevent concurrent execution of independent activities) or *overparallelization* (i.e., the situation where the synchronization necessary for parallel execution of multiple coordinator threads dominates execution time, to the extent that concurrency causes slowdown instead of speedup) of generated code. We have already reported our work on these issues along with promising results elsewhere [JA16, JHA14, JSA15].

Checking synchronization constraints is relatively cheap. Checking data constraints, in contrast, requires calls to a constraint solver. When using a general-purpose constraint solver, as we currently do, such calls inflict high run-time overhead. This overhead has a severe impact on the overall performance of programs, because coordinator threads execute purely serially. As such, checking data constraints can become a serious sequential bottleneck to an entire program (e.g., whenever all other threads depend on the firing of a transition to make progress).

Contribution. In this paper, we present two techniques to optimize the performance of checking data constraints. The first technique reduces the size of data constraints at compile-time, to reduce the complexity of (number of variables involved in) constraint solving at run-time. The second technique translates data constraints into small pieces of imperative code (in a sequential language with assignment and guarded failure statements) at compile-time, to replace expensive calls to a general-purpose constraint solver at run-time. We prove that both our techniques are correct. Such correctness proofs are important, because they ensure that our compilation approach guarantees *correctness-by-construction* (e.g., model-checking results obtained for pre-optimized CA also hold for their generated, optimized implementations). We evaluate our techniques in a number of experiments using their implementation in our current CA-to-Java compiler.

In Section 2, we present preliminaries on data constraints and CAs. In Section 3, we discuss our first optimization technique; in Section 4, we discuss our second. In Section 5, we report on an experimental evaluation of our two optimization techniques. Section 6 concludes this paper. Appendix A contains proof sketches; full, detailed proofs appear in [Jon16b] (referenced more specifically in Appendix A). A preliminary version of this paper, in which we report only on the optimization technique presented in Section 4, appeared in the proceedings of COORDINATION 2015 [JA15].

2. Preliminaries

Data Constraints. In this subsection, we present a first-order calculus of data constraints. In the next subsection, we label transitions in CAs with objects from this calculus. We start by defining elementary notions of data, ports, and memory cells.

Definition 2.1 (data)**.** A datum is an unstructured object. $\mathbb{D}$ denotes the possibly infinite set of all data, ranged over by d.

Definition 2.2 (empty datum)**.** `nil` is an unstructured object such that $\texttt{nil} \notin \mathbb{D}$.

Definition 2.3 (ports)**.** A port is an unstructured object. $\mathbb{P}$ denotes the set of all ports, ranged over by p. $2^{\mathbb{P}}$ denotes the set of all sets of ports, ranged over by P.

Definition 2.4 (memory cells)**.** A memory cell is an unstructured object. $\mathbb{M}$ denotes the set of all memory cells, ranged over by m. $2^{\mathbb{M}}$ denotes the set of all sets of memory cells, ranged over by M.

The exact content of $\mathbb{D}$ depends on the context of its use and formally does not matter. Henceforth, we write elements of $\mathbb{P}$ in capitalized lower case sans-serif (e.g., A, B, C, In_1, Out_2), while we write elements of $\mathbb{D}$ in lower case monospace (e.g., `1`, `3.14`, `true`, `"foo"`).

Although data flow through ports always in a certain direction, we do not yet distinguish input ports from output ports; this comes later.

Out of ports and memory cells, we construct *data variables*, which serve as the variables in our calculus. Every data variable designates a datum. For instance, ports can hold data (to exchange), so every port serves as a data variable in the calculus. Similarly, memory cells can hold data, but the meaning of "to hold" differs in this case. Ports hold data only for exchange *during* a coordination step (i.e., transiently, in passing). In contrast, memory cells hold data also *before* and *after* a coordination step. Consequently, in the context of data variables, a memory cell before a coordination step and the same memory cell after that step have different identities. After all, the content of the memory cell may have changed in between. Therefore—inspired by notation from Petri nets [Rei85]—for every memory cell m, both ${}^\bullet m$ and $m^\bullet$ serve as data variables: ${}^\bullet m$ refers to the datum in m before a coordination step, while $m^\bullet$ refers to the datum in m after that coordination step. We abbreviate sets $\{{}^\bullet m \mid m \in M\}$ and $\{m^\bullet \mid m \in M\}$ as ${}^\bullet M$ and $M^\bullet$.

Definition 2.5 (data variables)**.** A data variable is an object x generated by the following grammar:

$$x ::= p \mid {}^\bullet m \mid m^\bullet$$

$\mathbb{X}$ denotes the set of all data variables. $2^{\mathbb{X}}$ denotes the set of all sets of data variables, ranged over by X.

We subsequently assign meaning to data variables with *data assignments*.

Definition 2.6 (data assignments)**.** A data assignment is a partial function from data variables to data. $\mathbb{A}\textsc{ssignm} = \mathbb{X} \rightharpoonup \mathbb{D}$ denotes the set of all data assignments, ranged over by σ. $2^{\mathbb{A}\textsc{ssignm}}$ denotes the set of all sets of data assignments, ranged over by Σ.

Essentially, a data assignment σ comprehensively models a coordination step involving the ports and memory cells in $\text{Dom}(\sigma)$ and the data in $\text{Img}(\sigma)$. As coordinators have only finitely many ports and memory cells in practice, we stipulate that the domain of every data assignment is finite, too. The same holds for their support.

We proceed by defining *data functions* and *data relations*, which serve as the functions and predicates in our calculus. Together, data, data functions, and data relations constitute our set of extralogicals. To avoid excessive machinery—but at the cost of formal imprecision—we do not distinguish extralogical symbols from their interpretation as data, data functions, and data relations.

Definition 2.7 (data functions)**.** A data function is a function from tuples of data to data. $\mathbb{F} = \bigcup\{\mathbb{D}^k \rightarrow \mathbb{D} \mid k > 0\}$ denotes the set of all data functions, ranged over by f.

Definition 2.8 (data relations)**.** A data relation is a relation on tuples of data. $\mathbb{R} = \bigcup\{2^{\mathbb{D}^k} \mid k > 0\}$ denotes the set of all data relations, ranged over by R.

Henceforth, we write elements of $\mathbb{F}$ in camel case monospace (e.g., `divByThree`, `inc`), while we write elements of $\mathbb{R}$ in captitalized camel case monospace (e.g., `Odd`, `SmallerThan`).

Out of data variables, data, and data functions, we construct *data terms*, which serve as the terms in our calculus. Every data term represents a datum.

Definition 2.9 (data terms)**.** A data term is an object t generated by the following grammar:

$$t ::= x \mid d \mid f(t_1, \ldots, t_{k \geq 1})$$

$\mathbb{T}\text{ERM}$ denotes the set of all data terms. $2^{\mathbb{T}\text{ERM}}$ denotes the set of all sets of data terms, ranged over by T.

Henceforth, let $<_{\mathbb{T}\text{ERM}}$ denote some strict total order on $\mathbb{T}\text{ERM}$.[2]

Given a data assignment whose domain includes at least the data variables in a data term t, we can *evaluate* t to a datum. (To evaluate t, additionally, every data function application in t must have the right number of inputs: the *arity* of a data function and its number of inputs must match. Henceforth, we tacitly assume that this always holds true.)

Definition 2.10 (evaluation)**.** $\mathsf{eval} : \mathbb{A}\text{SSIGNM} \times \mathbb{T}\text{ERM} \to \mathbb{D} \cup \{\mathtt{nil}\}$ denotes the function defined by the following equations:

$$\mathsf{eval}_\sigma(x) = \begin{cases} \sigma(x) & \textbf{if } x \in \mathrm{Dom}(\sigma) \\ \mathtt{nil} & \textbf{otherwise} \end{cases}$$

$$\mathsf{eval}_\sigma(d) = d$$

$$\mathsf{eval}_\sigma(f(t_1, \ldots, t_k)) = \begin{cases} f(\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) & \textbf{if } \begin{bmatrix} \mathsf{eval}_\sigma(t_1) \neq \mathtt{nil} \\ \textbf{and } \cdots \textbf{ and} \\ \mathsf{eval}_\sigma(t_k) \neq \mathtt{nil} \end{bmatrix} \\ \mathtt{nil} & \textbf{otherwise} \end{cases}$$

Out of data terms, data relations, and data variables, we construct data constraints.

Definition 2.11 (data constraints)**.** A data constraint is an object φ generated by the following grammar:

$$\begin{array}{lll} a ::= \bot \mid \top \mid t_1 = t_2 \mid R(t_1, \ldots, t_{k \geq 1}) & & \text{(data atoms)} \\ \ell ::= a \mid \neg a & & \text{(data literals)} \\ \varphi ::= \exists x.\varphi \mid \ell_1 \wedge \cdots \wedge \ell_{k \geq 1} & & \text{(data constraints)} \end{array}$$

$\mathbb{DC}$ denotes the set of all data constraints. $2^{\mathbb{DC}}$ denotes the set of all sets of data constraints, ranged over by Φ.

Henceforth, let $<_{\mathbb{DC}}$ denote a strict total order on $\mathbb{DC}$, and let $\bigwedge\Phi$ denote the *unique* multiary conjunction of the data constraints in Φ under $<_{\mathbb{DC}}$. Also, for a data constraint $\varphi = \exists x_1.\cdots\exists x_l.(\ell_1 \wedge \cdots \wedge \ell_k)$, call $\ell_1 \wedge \cdots \wedge \ell_k$ the *kernel* of φ, and let $\mathsf{Liter}(\varphi) = \{\ell_1, \ldots, \ell_k\}$.

Every data constraint characterizes a set of data assignments through an *entailment relation*. This entailment relation, thus, formalizes the semantics of data constraints. Let $\varphi[t/x]$ denote data constraint φ with data term t substituted for every occurrence of data variable x (in a capture-free way).

Definition 2.12 (entailment)**.** $\models \subseteq \mathbb{A}\text{SSIGNM} \times \mathbb{DC}$ denotes the smallest relation induced by the rules in Figure 1.

Contradiction, tautology, and (multiary) conjunction have standard semantics [Rau10]. Negation $\neg a$ means that, despite all free variables in a having a value, a does not hold true; the extra condition on the free variables in a ensures the *monotonicity* of entailment (i.e., $\sigma|_X \models \varphi$ implies $\sigma \models \varphi$, for all X, φ). Data atom $t_1 = t_2$ means that t_1 and t_2 evaluate to the same datum. Typical examples include $p_1 = p_2$ (i.e., the same datum passes through ports p_1 and p_2), $p = m^\bullet$ (i.e., the datum that passes through port p enters the buffer

[2]It does not matter what this strict total order exactly looks like, so long as we have some way of selecting the least element of any set of terms. We use this property in Definition 3.3.

$$\frac{}{\sigma \models \top} \quad (2.1) \qquad \frac{\mathsf{Free}(a) \subseteq \mathrm{Dom}(\sigma) \ \textbf{ and } \ \sigma \not\models a}{\sigma \models \neg a} \quad (2.2)$$

$$\frac{\mathsf{eval}_\sigma(t_1) = \mathsf{eval}_\sigma(t_2) \neq \mathtt{nil}}{\sigma \models t_1 \texttt{=} t_2} \quad (2.3) \qquad \frac{\sigma \models \phi[d/x] \ \textbf{ for some } \ d}{\sigma \models \exists x.\phi} \quad (2.4)$$

$$\frac{(\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) \in R}{\sigma \models R(t_1, \ldots, t_k)} \quad (2.5) \qquad \frac{\sigma \models \phi_1 \ \textbf{ and } \ \cdots \ \textbf{ and } \ \sigma \models \phi_k}{\sigma \models \phi_1 \wedge \cdots \wedge \phi_k} \quad (2.6)$$

FIGURE 1. Addendum to Definition 2.12

modeled by memory cell m), and $p \texttt{=} {}^\bullet m$ (i.e., the datum in the buffer modeled by memory cell m exits that buffer and passes through port p). Tautology $\top$ means that it does not matter which data flow through which ports.

Henceforth, let $\Rightarrow$ and $\equiv$ denote the implication relation and the equivalence relation on data constraints, derived from $\models$ in the usual way [Rau10]. Furthermore, let $\mathsf{Variabl}(\varphi)$ denote the set of data variables in φ, and let $\mathsf{Free}(\varphi)$ denote its set of *free* data variables.

Constraint Automata. We proceed by formally defining a CA $\boldsymbol{a}$, which models a coordinator, as a tuple consisting of a set of states Q, a triple of three sets of ports $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}})$, a set of memory cells M, a transition relation $\longrightarrow$, and an initial state q^0. The set P^{all} contains all ports monitored and controlled by $\boldsymbol{a}$, while P^{in} and P^{out} contain only its input ports and its output ports. Although P^{all} contains the union of P^{in} and P^{out}, the converse not necessarily holds true: beside input and output ports, P^{all} may contain also *internal ports*. If a CA has internal ports, we call it a *composite*; otherwise, we call it a *primitive*.

Definition 2.13 (states)**.** A state represents a configuration of a coordinator. $\mathbb{Q}$ denotes the set of all states, ranged over by q. $2^{\mathbb{Q}}$ denotes the set of all sets of states, ranged over by Q.

Definition 2.14 (constraint automata)**.** A constraint automaton is a tuple:

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, q^0)$$

where:

- $Q \subseteq \mathbb{Q}$ (states)
- $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) \in 2^{\mathbb{P}} \times 2^{\mathbb{P}} \times 2^{\mathbb{P}}$ such that: (ports)

$$P^{\text{in}}, P^{\text{out}} \subseteq P^{\text{all}} \ \textbf{ and } \ P^{\text{in}} \cap P^{\text{out}} = \emptyset$$

- $M \subseteq \mathbb{M}$ (memory cells)
- $\longrightarrow \subseteq Q \times 2^{P^{\text{all}}} \times \mathbb{DC} \times Q$ such that: (transitions)

$$\left[q \xrightarrow{P,\varphi} q' \ \textbf{ implies } \ \mathsf{Free}(\varphi) \subseteq P \cup {}^\bullet M \cup M^\bullet\right] \ \textbf{ for all } \ q, q', P, \varphi$$

- $q^0 \in Q$ (initial state)

$\mathbb{A}$UTOM denotes the set of all constraint automata, ranged over by $\boldsymbol{a}$.

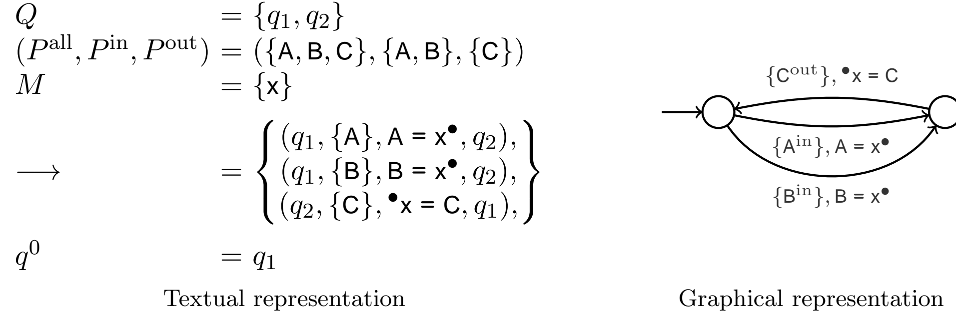

FIGURE 2. Example CA for a producers/consumer coordinator

The requirement $\mathsf{Free}(\varphi) \subseteq P \cup {}^\bullet M \cup M^\bullet$ means that the effect of a transition remains local to its own scope: a transition cannot affect, or be affected by, ports outside its synchronization constraint and memory cells outside its CA. Henceforth, let $\mathsf{Dc}(\boldsymbol{a})$ denote the set of data constraints that occur on the transitions of a CA $\boldsymbol{a}$ (not to be confused with $\mathbb{DC}$, which denotes the set of all data constraints; see Definition 2.11).

Figure 2 shows an example of a CA. In graphical representations of CAs, we annotate ports in synchronization constraints with superscripts "in" and "out" to indicate their direction; internal ports have no such annotation. The CA in Figure 2 models a producers/ consumer coordinator with two input ports A and B (each shared with a different producer, presumably) and an output port C (shared with the consumer). Initially, a `put` by the producer on A can complete, causing that producer to offer a datum into internal buffer x (modeled by data constraint $\mathsf{A} = \mathsf{x}^\bullet$). Alternatively, a `put` by the other producer on B can similarly complete. Subsequently, only a `get` by the consumer on C can complete, causing the consumer to accept the datum previously stored in x. This coordinator, thus, enforces asynchronous, unordered, reliable communication from two producers to a consumer.

The precise definitions of *language acceptance* and *bisimulation* for CAs do not matter in this paper. Likewise, the precise definitions of *behavioral equivalence* (based on language acceptance) and *behavioral congruence* (based on bisimulation), such that behavioral congruence implies behavioral equivalence, do not matter. These definitions appear elsewhere [Jon16a]. The only result about the behavior of CAs that matters in this paper is the following intuitive proposition. Let $\simeq$ denote behavioral congruence, and let $\boldsymbol{a}[\varphi'/\varphi]$ denote CA $\boldsymbol{a}$ with data constraint φ' substituted for every occurrence of data constraint φ.

Proposition 2.15 (Lemma 43 in [Jon16b, Appendix C.4])**.** $\varphi \equiv \varphi'$ **implies** $\boldsymbol{a} \simeq \boldsymbol{a}[\varphi'/\varphi]$

This proposition means that we can freely replace every data constraint in a CA with an equivalent data constraint in a behavior-neutral way. This proposition plays a key role in the correctness proofs of the two optimization techniques presented in the rest of this paper.

Instead of defining CAs directly, in practice, we construct them *compositionally* using two binary operations [BSAR06, Jon16a]: *join*, denoted by $\otimes$, and *hide*, denoted by $\ominus$. Join performs parallel composition: it "glues" together two CAs on their shared ports, after which those shared ports become internal. Essentially, whenever two CAs have joined, if a transition in one of those CAs involves shared ports, that transition can fire only synchronously with a transition in the other CA that involves *exactly the same* shared ports (i.e., at any time, the CAs must agree on firing transitions involving their shared ports).

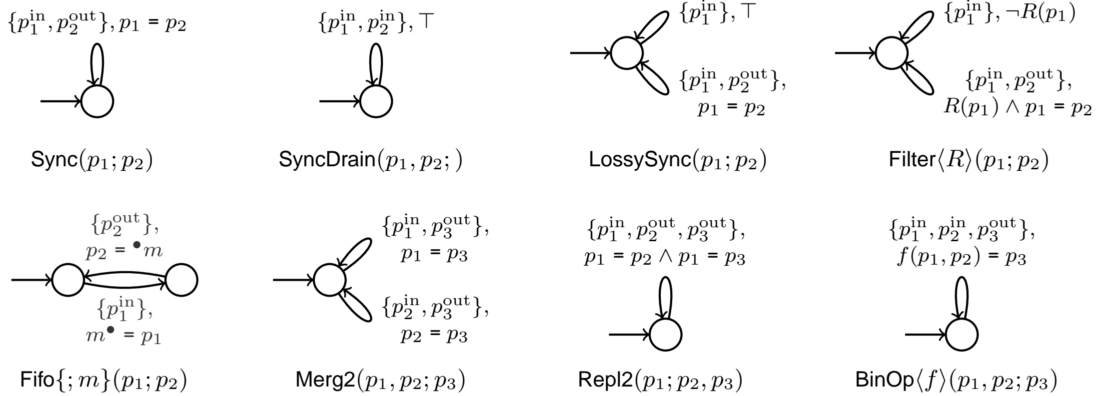

FIGURE 3. Eight primitives

Sync$(p_1; p_2)$	Infinitely often atomically $\big[$accepts a datum d on its input port p_1, then offers d on its output port $p_2\big]$.
SyncDrain$(p_1, p_2;)$	Infinitely often atomically $\big[$accepts data d_1 and d_2 on its input ports p_1 and p_2, then loses d_1 and $d_2\big]$.
LossySync$(p_1; p_2)$	Infinitely often either atomically $\big[$accepts a datum d on its input port p_1, then offers d on its output port $p_2\big]$ or atomically $\big[$accepts a datum d on p_1, then loses $d\big]$.
Filter$\langle R \rangle(p_1; p_2)$	Infinitely often either atomically $\big[$accepts a datum d on its input port p_1, then establishes that d satisfies data relation R, then offers d on its output port $p_2\big]$ or atomically $\big[$accepts a datum d on p_1, then establishes that d violates R, then loses $d\big]$.
Fifo$\{; m\}(p_1; p_2)$	Infinitely often first atomically $\big[$accepts a datum d on its input port p_1, then stores d in its memory cell $m\big]$ and subsequently atomically $\big[$loads d from m, then offers d on its output port $p_2\big]$.
Merg2$(p_1, p_2; p_3)$	Infinitely often atomically $\big[$accepts a datum d either on its input port p_1 or on its input port p_2, then offers d on its output port $p_3\big]$.
Repl2$(p_1; p_2, p_3)$	Infinitely often atomically $\big[$accepts a datum d on its input port p_1, then offers d on its output ports p_2 and $p_3\big]$.
BinOp$\langle f \rangle(p_1, p_2; p_3)$	Infinitely often atomically $\big[$accepts data d_1 and d_2 on its input ports p_1 and p_2, then applies data function f to d_1 and d_2, then offers $f(d_1, d_2)$ on its output port $p_3\big]$.

FIGURE 4. Data-flow behavior of the primitives in Figure 3

Hide performs port abstraction: it "cuts" a port out from a CA. Typically, we use hide to remove internal ports from the definition of a CA, as such ports do not directly contribute to its observable behavior (i.e., processes cannot perform I/O-operations on internal ports).

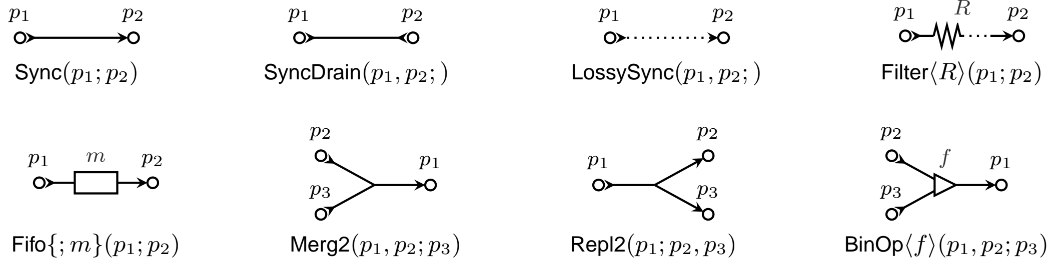

FIGURE 5. Digraphs for the primitives in Figure 3

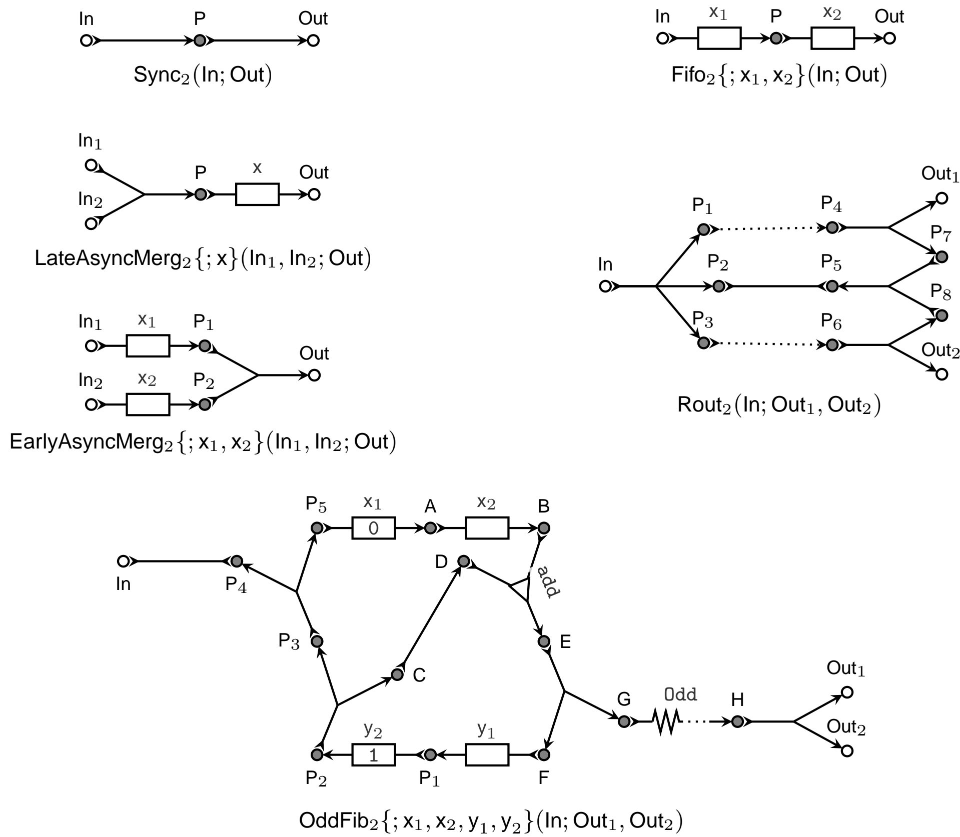

FIGURE 6. Digraphs for six example composites

To compositionally construct a CA, then, we first join a number of "small" primitive CAs into a "large" composite CA. Second, we hide all internal ports from this large CA to make its definition more concise (without losing essential information). Figure 3 shows a number of common primitive CAs; Figure 4 explains their behavior in terms of data-flows between their ports. In these figures, every CA has a signature formatted as follows:

$$\mathit{name}\langle\mathit{extralogicals}\rangle\{\mathit{internal\ ports}\ ;\ \mathit{memory\ cells}\}(\mathit{input\ ports}\ ;\ \mathit{output\ ports})$$

Instead of writing explicit $\otimes/\ominus$-expressions to construct CAs, in practice, we often draw them in a graphical, more intuitive syntax, based on the coordination language Reo [Arb04,

Arb11].[3] Essentially, in this syntax, we draw a (hyper)digraph, where every vertex denotes a port, and where every (hyper)arc denotes a CA consisting of the ports denoted by its connected vertices. By convention, every vertex has degree 1 (for input and output ports) or 2 (for internal ports). The $\otimes/\ominus$-expression denoted by a digraph, then, is the join of (the denotations of) its arcs, and the hide of (the denotations of) its vertices of degree 2. Intuitively, every transition in the (evaluated) $\otimes/\ominus$-expression for a digraph corresponds to an atomic flow of data along the arcs in that digraph. Figure 5 shows digraphs for the primitives in Figure 3; Figure 6 shows digraphs for example composites.

In Figure 6, Sync_2 models the same coordinator as a single Sync: it enforces a standard synchronous channel protocol between a producer and a consumer. Fifo_2 models a coordinator between a producer and a consumer that enforces a standard (order-preserving) asynchronous channel protocol with a buffer of capacity 2. $\mathsf{LateAsyncMerg}_2$ is (a behaviorally congruent CA to) the CA in Figure 2. $\mathsf{EarlyAsyncMerg}_2$ models a coordinator between two producers and one consumer, as $\mathsf{LateAsyncMerg}_2$. The difference between the two is that with $\mathsf{EarlyAsyncMerg}_2$, every producer has its own buffer, which results in significantly different behavior (as producers no longer need to wait for each other before their `put`s can complete). Rout_2 models a coordinator between one producer and two consumers that enforces a symmetric protocol to Merg2: infinitely often, it atomically $\big[$accepts a datum on its input port, then offers it on one of its output ports$\big]$. Finally, OddFib_2 models a coordinator between two producers and one consumer. Whenever the i-th `put` by the producer completes, one of two things happens. If the i-th Fibonacci number is even, the datum `put` by the producer is lost, and no interaction occurs between the producer and the two consumers. If the i-th Fibonacci number is odd, in contrast, a `get` by each of the two consumers *must* complete at the same time (i.e., atomically, i.e., synchronously). In this case, specifically, the datum `put` by the producer is lost, while the consumers `get` the i-th Fibonacci number. This coordinator, thus, enforces synchronous, unreliable (in the sense just described) communication from a producer to two consumers.

The primitives in Figure 5 were introduced by Arbab [Arb04], except BinOp, which was introduced by Jongmans [Jon16a] (BinOp is, however, a generalization of primitive Join, which was introduced by Kokash and Arbab [KA09]). LateAsyncMerg and EarlyAsyncMerg in Figure 6 are probably folklore; these two names were first used by Jongmans [Jon16a]. OddFib is based on Arbab's Fibonacci [Arb05]. Rout was introduced by Arbab [Arb05].

3. Optimization I: Eliminate (Instead of Hide)

Motivating Example. To illustrate the need for our first technique to optimize the performance of checking data constraints, presented in this section, we start with a motivating example. Recall the Sync primitive in Figure 3. Sync has a special property: it acts as a kind of algebraic identity of join and hide, in the following sense. Let $\boldsymbol{a}[p'/p]$ denote CA $\boldsymbol{a}$ with port p' substituted for every occurrence of port p. Let $\boldsymbol{a}$ range over the set of all CAs that (i) have an input port p_2 *and* (ii) in which port p_1 does not occur. Then:

$$(\mathsf{Sync}(p_1; p_2) \otimes \boldsymbol{a}) \ominus p_2 \simeq \boldsymbol{a}[p_1/p_2]$$

[3] Other syntaxes for CAs beside Reo exist. For instance, we know how to translate UML sequence/activity diagrams and BPMN to CAs [AKM08, CKA10, MAB11]. Connector algebras of Bliudze and Sifakis [BS10] also have a straightforward interpretation in terms of CAs, so offering an interesting alternative syntax [DJAB15].

In words, $(\mathsf{Sync}(p_1;p_2) \otimes \boldsymbol{a}) \ominus p_2$ and $\boldsymbol{a}$ are behaviorally congruent modulo substitution of p_1 for p_2. Generally, we can "prefix" (i.e., join on its input ports) or "suffix" (i.e., join on its output ports) any number of Syncs to a CA without affecting—in the sense just described—that CA's behavior. Given this property, it seems not unreasonable to assume that compiler-generated code for a single Sync has the same performance as a chain of 64 Syncs. Slightly more formally, if $\sim$ means "has the same performance", one may expect:

$$\mathsf{Sync}(p_1;p_{65}) \sim (\mathsf{Sync}(p_1;p_2) \otimes \cdots \otimes \mathsf{Sync}(p_{64};p_{65})) \ominus p_2 \ominus \cdots \ominus p_{64}$$

Our compiler-generated code, however, violates this equation: a single Sync fires 27 million transitions in four minutes, whereas the chain of 64 Syncs fires only nine million transitions.

To understand this phenomenon, we first present the definition of hide [BSAR06, Jon16a]:

Definition 3.1 (hide). $\ominus : \mathbb{A}\text{UTOM} \times \mathbb{P} \to \mathbb{A}\text{UTOM}$ denotes the function defined by the following equation:

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, q^0) \ominus p = (Q, (P^{\text{all}} \setminus \{p\}, P^{\text{in}} \setminus \{p\}, P^{\text{out}} \setminus \{p\}), M, \longrightarrow_\ominus, q^0)$$

where $\longrightarrow_\ominus$ denotes the smallest relation induced by the following rule:

$$\frac{q \xrightarrow{P,\phi} q'}{q \xrightarrow{P\setminus\{p\},\exists p.\phi}_\ominus q'} \tag{3.1}$$

In words, hide removes a port both from sets $P^{\text{all}}, P^{\text{in}}, P^{\text{out}}$ and from every transition. (Because $P^{\text{in}}, P^{\text{out}} \subseteq P^{\text{all}}$ by Definition 2.14, we need to remove p not only from P^{in} and P^{out} but also from P^{all}.) But whereas hide removes ports from synchronization constraints *syntactically*—effectively making those constraints smaller—it removes ports from data constraints only *semantically*. Indeed, $\ominus$ does not reduce the size of data constraints (in terms of the number of data variables, data literals, and existential quantifications) but, in fact and in contrast, makes data constraints larger by enveloping them in existential quantifications: the transition in the single Sync has just $p_1 = p_{65}$ as its data constraint, whereas the corresponding transition in the chain of 64 Syncs has $\exists p_{64}.\cdots\exists p_2.(p_1 = p_2 \wedge \cdots \wedge p_{64} = p_{65})$. Clearly, although the two data constraint expressions are semantically (logically) equivalent, checking the latter data constraint expression requires more resources than the former.

Below, we develop a variant of hide, called *eliminate*, that, when applied 63 times to the chain of 64 Syncs, yields the same data constraint as the one in the single Sync. The key idea is to mechanically simplify data constraint expressions using the equivalence $\exists p.(p = t \wedge \phi) \equiv \phi[t/p]$, if $p \notin \mathsf{Free}(t)$, whenever this equivalence becomes applicable after hiding. In the previous example, for instance, we can use this equivalence to simplify $\exists p_{64}.\cdots\exists p_3.\exists p_2.(p_1 = p_2 \wedge p_2 = p_3 \wedge \cdots \wedge p_{64} = p_{65})$ to $\exists p_{64}.\cdots\exists p_3.(p_1 = p_3 \wedge \cdots \wedge p_{64} = p_{65})$. We can subsequently repeat this process until we indeed arrive at the expression $p_1 = p_{65}$, as desired.

Eliminate. First, we need to introduce the concept of *determinants* of free data variables in data constraints. For a data constraint φ and one of its free data variables $x \in \mathsf{Free}(\varphi)$, the set of determinants of x consists of those terms that *precisely* determine the datum $\sigma(x)$ assigned to x in any data assignment σ that satisfies φ (i.e., $\sigma \models \varphi$). "Precisely" here means that a determinant neither overspecifies nor underspecifies $\sigma(x)$. Thus, if a set of determinants contains multiple data terms, each of those data terms evaluates to the same

datum under σ. Determinants furthermore determine $\sigma(x)$ independent of x itself: no determinant of x has x among its free data variables (i.e., determinants have no recursion).

Definition 3.2 (determinants)**.** $\mathsf{Determ} : \mathbb{X} \times \mathbb{DC} \to 2^{\mathbb{T}\textsc{erm}}$ denotes the function defined by the following equations:

$$\begin{array}{ll}
\mathsf{Determ}_x(\top), \mathsf{Determ}_x(\bot) & = \emptyset \\
\mathsf{Determ}_x(t_1 \texttt{ = } t_2) & = \begin{cases} \{t_2\} & \textbf{if } \big[t_1 = x \textbf{ and } x \notin \mathsf{Variabl}(t_2)\big] \\ \{t_1\} & \textbf{if } \big[t_2 = x \textbf{ and } x \notin \mathsf{Variabl}(t_1)\big] \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{Determ}_x(R(t_1, \ldots, t_k)) & = \emptyset \\
\mathsf{Determ}_x(\neg a) & = \emptyset \\
\mathsf{Determ}_x(\ell_1 \wedge \cdots \wedge \ell_k) & = \mathsf{Determ}_x(\ell_1) \cup \cdots \cup \mathsf{Determ}_x(\ell_k) \\
\mathsf{Determ}_x(\exists x'.\varphi') & = \begin{cases} \mathsf{Determ}_x(\varphi) & \textbf{if } x \neq x' \\ \emptyset & \textbf{otherwise} \end{cases}
\end{array}$$

For instance, consider the following data constraint:

$$\varphi_{\text{eg}} = {}^\bullet\mathsf{x}_2 \texttt{ = } \mathsf{B} \wedge \mathsf{C} \texttt{ = } \mathsf{D} \wedge \mathsf{add}(\mathsf{B}, \mathsf{D}) \texttt{ = } \mathsf{E} \wedge \mathsf{E} \texttt{ = } \mathsf{F} \wedge \mathsf{E} \texttt{ = } \mathsf{G} \wedge \neg\mathsf{Odd}(\mathsf{G})$$

(This data constraint appears in the $\otimes/\ominus$-expression denoted by the digraph for OddFib in Figure 6.) The free data variables in φ_{eg} have the following determinants:

$$\begin{array}{ll}
\mathsf{Determ}_{^\bullet\mathsf{x}}(\varphi_{\text{eg}}) = \{\mathsf{B}\} & \mathsf{Determ}_{\mathsf{E}}(\varphi_{\text{eg}}) = \{\mathsf{add}(\mathsf{B}, \mathsf{D}), \mathsf{F}, \mathsf{G}\} \\
\mathsf{Determ}_{\mathsf{B}}(\varphi_{\text{eg}}) = \{^\bullet\mathsf{x}\} & \mathsf{Determ}_{\mathsf{F}}(\varphi_{\text{eg}}) = \{\mathsf{E}\} \\
\mathsf{Determ}_{\mathsf{C}}(\varphi_{\text{eg}}) = \{\mathsf{D}\} & \mathsf{Determ}_{\mathsf{G}}(\varphi_{\text{eg}}) = \{\mathsf{E}\} \\
\mathsf{Determ}_{\mathsf{D}}(\varphi_{\text{eg}}) = \{\mathsf{C}\} &
\end{array}$$

Next, let $\boldsymbol{a}$ denote a CA, and let φ denote one of its data constraints. Suppose that we hide x from $\boldsymbol{a}$ with $\ominus$. By Definition 3.1 of $\ominus$, the transition(s) of $\boldsymbol{a}$ previously labeled by φ are now labeled with $\exists x.\varphi$. However, if x has determinants, instead of enveloping φ in an existential quantification as $\ominus$ does, we can alternatively perform a syntactic substitution of one of those determinants for x. We formalize such a substitution as follows.

Definition 3.3 (syntactic existential quantification)**.** $\mathsf{exists} : \mathbb{X} \times \mathbb{DC} \to \mathbb{DC}$ denotes the function defined by the following equation:

$$\mathsf{exists}_x(\varphi) = \begin{cases} \varphi[t/x] & \textbf{if } \big[\mathsf{Determ}_x(\varphi) \neq \emptyset \textbf{ and } t = \min(\mathsf{Determ}_x(\varphi))\big] \\ \exists x.\varphi & \textbf{otherwise} \end{cases}$$

In this definition, function $\min(\cdot)$ takes the least element in $\mathsf{Determ}_x(\varphi)$, under the global order on data terms $<_{\mathbb{T}\textsc{erm}}$, to ensure that exists always produces the same output under

the same input. The following equations exemplify the (nested) application of exists on φ_{eg}.

$$\begin{aligned}
&\mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\mathsf{exists}_{\mathsf{B}}(\varphi_{\mathrm{eg}}))))\\
= {}&\mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\mathsf{exists}_{\mathsf{B}}(\\
&\quad {}^{\bullet}\mathtt{x} = \mathtt{B} \wedge \mathtt{C} = \mathtt{D} \wedge \mathtt{add}(\mathtt{B},\mathtt{D}) = \mathtt{E} \wedge \mathtt{E} = \mathtt{F} \wedge \mathtt{E} = \mathtt{G} \wedge \neg\mathtt{Odd}(\mathtt{G})))))\\
= {}&\mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\\
&\quad {}^{\bullet}\mathtt{x} = {}^{\bullet}\mathtt{x} \wedge \mathtt{C} = \mathtt{D} \wedge \mathtt{add}({}^{\bullet}\mathtt{x},\mathtt{D}) = \mathtt{E} \wedge \mathtt{E} = \mathtt{F} \wedge \mathtt{E} = \mathtt{G} \wedge \neg\mathtt{Odd}(\mathtt{G}))))\\
= {}&\mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\\
&\quad {}^{\bullet}\mathtt{x} = {}^{\bullet}\mathtt{x} \wedge \mathtt{C} = \mathtt{C} \wedge \mathtt{add}({}^{\bullet}\mathtt{x},\mathtt{C}) = \mathtt{E} \wedge \mathtt{E} = \mathtt{F} \wedge \mathtt{E} = \mathtt{G} \wedge \neg\mathtt{Odd}(\mathtt{G})))\\
= {}&\mathsf{exists}_{\mathsf{G}}(\\
&\quad {}^{\bullet}\mathtt{x} = {}^{\bullet}\mathtt{x} \wedge \mathtt{C} = \mathtt{C} \wedge \mathtt{add}({}^{\bullet}\mathtt{x},\mathtt{C}) = \mathtt{F} \wedge \mathtt{F} = \mathtt{F} \wedge \mathtt{F} = \mathtt{G} \wedge \neg\mathtt{Odd}(\mathtt{G}))\\
= {}&\quad {}^{\bullet}\mathtt{x} = {}^{\bullet}\mathtt{x} \wedge \mathtt{C} = \mathtt{C} \wedge \mathtt{add}({}^{\bullet}\mathtt{x},\mathtt{C}) = \mathtt{F} \wedge \mathtt{F} = \mathtt{F} \wedge \mathtt{F} = \mathtt{F} \wedge \neg\mathtt{Odd}(\mathtt{F})
\end{aligned}$$

We define eleminate in terms of exists.

Definition 3.4 (eliminate)**.** $\circleddash : \mathbb{A}\mathrm{UTOM} \times \mathbb{P} \to \mathbb{A}\mathrm{UTOM}$ denotes the function defined by the following equation:

$$(Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow, q^0) \circleddash p = (Q, (P^{\mathrm{all}} \setminus \{p\}, P^{\mathrm{in}} \setminus \{p\}, P^{\mathrm{out}} \setminus \{p\}), M, \longrightarrow_{\circleddash}, q^0)$$

where $\longrightarrow_{\circleddash}$ denotes the smallest relation induced by the following rule:

$$\frac{q \xrightarrow{P,\varphi} q'}{q \xrightarrow{P\setminus\{p\},\mathsf{exists}_p(\varphi)}_{\circleddash} q'} \tag{3.2}$$

In the previous definition, we use exists to remove ports from data constraints. Although Definition 3.3 of exists also allows for removing data variables for memory cells, we do not pursue such elimination in this paper.

Correctness and Effectiveness. We conclude this section by establishing the correctness and effectiveness of eliminate. We consider eliminate correct if it yields a CA behaviorally congruent to the CA that hide yields. Before formulating this as a theorem, the following lemma first states the equivalence of existential quantification and exists.

Lemma 3.5. $\exists x.\varphi \equiv \mathsf{exists}_x(\varphi)$

From Proposition 2.15 and Lemma 3.5, we conclude the following correctness theorem.

Theorem 3.6. $\boldsymbol{a} \ominus p \simeq \boldsymbol{a} \circleddash p$

We consider eliminate effective if, after eliminating a port p from a CA $\boldsymbol{a}$, that port no longer occurs in any of that CA's data constraint expressions. Generally, however, such unconditional effectiveness does not hold true: if $\boldsymbol{a}$ has a data constraint φ in which p occurs, but p has no determinants in φ, eliminate has nothing to replace p with. In that case, $\mathsf{exists}_p(\varphi) = \exists p.(\varphi)$, and consequently, eliminate does not have its intended (simplifying) effect. Eliminate *does* satisfy a weaker—but useful—form of effectiveness, though. To formulate this as a theorem, we first define a function that computes *ever-determined ports*. We call a port p ever-determined in a CA $\boldsymbol{a}$ iff both p occurs in $\boldsymbol{a}$ and every data constraint in $\boldsymbol{a}$ has a determinant for p.

Definition 3.7 (ever-determined ports)**.** $\mathsf{Edp} : \mathbb{A}\text{UTOM} \to 2^{\mathbb{P}}$ denotes the function defined by the following equation:

$$\mathsf{Edp}(\boldsymbol{a}) = \{p \mid [\begin{bmatrix} p \in \mathsf{Variabl}(\varphi) \\ \textbf{and}\ \ \varphi \in \mathsf{Dc}(\boldsymbol{a}) \end{bmatrix}\ \textbf{implies}\ \ \mathsf{Determ}_p(\varphi) \neq \emptyset]\ \ \textbf{for all}\ \ \varphi\}$$

For instance, p_1, p_2, and p_3 all qualify as ever-determined in Merg2 in Figure 3. To understand the ever-determinedness of p_1, observe that p_1 occurs in the data constraint on the top transition in Merg2 and that p_1 has a determinant in that data constraint (namely p_3); because p_1 does not occur in the data constraint on the bottom transition in Merg2, p_1 indeed qualifies as ever-determined. A similar explanation applies to p_2. To understand the ever-determinedness of p_3, observe that p_3 occurs in the data constraint on both transitions in Merg2 and that p_3 has a determinant in both these data constraints (namely p_1 and p_2). Consequently, also p_3 qualifies as ever-determined. In contrast, p_1 in members of Filter in Figure 3 does *not* qualify as ever-determined, because p_1 occurs in the data constraint on the top transition in Filter but does not have a single determinant in that data constraint.

The following theorem states the effectiveness of eliminate, conditional on ever-determinedness: after eliminating an ever-determined port from a CA, that port no long occurs in any of that CA's data constraints.

Theorem 3.8. $p \in \mathsf{Edp}(\boldsymbol{a})$ **implies** $p \notin \{x \mid \varphi \in \mathsf{Dc}(\boldsymbol{a} \ominus p)$ **and** $x \in \mathsf{Variabl}(\varphi)\}$

"Effectiveness" refers to a rather theoretical property; it says nothing yet about the impact of applying $\ominus$ in practice. In Section 5, we study this impact through a number of experiments; in this section, we only revisit our motivating example. By using $\ominus$ instead of $\ominus$, and after removing $t = t$ literals (each of which trivially equates to $\top$), we get exactly the same data constraint in the chain of 64 Syncs as in the single Sync. Consequently, the compiler-generated code for the chain of 64 Syncs has the same performance as compiler-generated code for the single Sync (which corresponds to a 3× speedup relative to unoptimized code generated with hide instead of eliminate).

4. Optimization II: Commandify (Instead of Seek)

Data Commands. In the previous section, we presented a first technique to optimize the performance of checking data constraints. In this section, we present a second technique to further optimize the performance of such checks and, in particular, the expensive constraint solver calls involved. Essentially, this new technique comprises the generation of a little, dedicated constraint solver for every data constraint at compile-time. At run-time, then, instead of calling a general-purpose constraint solver to check a data constraint, the compiler-generated coordinator thread for a CA calls a more efficient constraint solver generated specifically for that data constraint. First, in this subsection, we describe a basic sequential language (syntax, semantics, proof system) in which to express such dedicated constraint solvers; in the next subsections, we present the process of their generation.

General-purpose techniques for constraint solving—an NP-complete problem for finite domains—inflict not only a solving overhead proportional to the size of a data constraint but also a constant overhead for preparing, making, and processing the result of every call to a full-fledged solver. Although we generally cannot escape using such techniques for checking arbitrary data constraints, a better alternative exists for many data constraints in practice. The crucial observation is that the data constraints in *all* CAs that we know of

in the literature really constitute declarative specifications of a relatively straightforward imperative program. What we need to do, then, is develop a technique for statically translating such a data constraint φ, off-line at compile-time, into a small imperative program that computes a data assignment σ such that $\sigma \models \varphi$, without resorting to general-purpose constraint solving. We call such a small program a *data command* and the translation from data constraints to data commands *commandification*. Essentially, we formalize and automate what programmers do when they write an imperative implementation of a declarative specification expressed as a data constraint. After presenting our technique, we make the class of data constraints currently supported by commandification precise.

Definition 4.1 (data commands)**.** A data command is an object generated by the following grammar:

$$\pi ::= \texttt{skip} \mid x \texttt{ := } t \mid \varphi \texttt{ -> } \pi \mid \pi \texttt{ ; } \pi \mid \varepsilon \qquad \text{(data commands)}$$

$\mathbb{C}\textsc{omm}$ denotes the set of all data commands.

In the previous definition, ε denotes the empty data command, $x \texttt{ := } t$ denotes an *assignment*, and $\varphi \texttt{ -> } \pi$ denotes a *failure statement*.[4] Henceforth, we often write "value of x" instead of "the datum assigned to x".

We define an operational semantics for data commands based on an operational semantics for a sequential language by Apt et al. [AdBO09]. As data commands are supposed to solve data constraints, we model the *data state* that a data command executes in with either a function from data variables to data—a data assignment—or the distinguished object fail, which models abnormal termination. A *data configuration*, then, consists of a data command and a data state to execute that data command in.

Definition 4.2 (abnormal termination)**.** fail is an unstructured object such that fail $\notin$ $\mathbb{A}\textsc{ssignm}$.

Definition 4.3 (data configurations)**.** A data configuration is a pair (π, ς) where:
- $\pi \in \mathbb{C}\textsc{omm}$ (data command)
- $\varsigma \in \mathbb{A}\textsc{ssignm} \cup \{\mathsf{fail}\}$ (data state)

$\mathbb{C}\textsc{onf}$ denotes the set of all data configurations.

A *transition system* on configurations formalizes their evolution in time.

Definition 4.4 (transition system on data configurations)**.** $\Longrightarrow \subseteq \mathbb{C}\textsc{onf} \times \mathbb{C}\textsc{onf}$ denotes the smallest relation induced by the rules in Figure 7.

Note that $\varphi \texttt{ -> } \pi$ indeed denotes a failure statement rather than a *conditional statement*: if the current data state violates the *guard* φ, execution abnormally terminates.

Through the transition system in Definition 4.4, we associate two different semantics with data commands. The *partial correctness semantics* of a data command π under a set of *initial* data states Σ consists of all the *final* data states Σ' to which any of those initial states may evolve through execution of π. Notably, this partial correctness semantics ignores abnormal termination. In contrast, the *total correctness semantics* of π under Σ consists not only of Σ' but, if at least one execution abnormally terminates, also of fail.

[4] The term "failure statement" may be confusing. As shortly formalized in Definition 4.4, it refers to a special conditional statement that fails in case the first alternative cannot be selected. By calling such statements "failure statements", we follow Apt et al. [AdBO09], which strongly influenced this section.

$$\frac{}{(\texttt{skip}, \sigma) \Longrightarrow (\varepsilon, \sigma)} \quad (4.1) \qquad \frac{}{(x \texttt{ := } t, \sigma) \Longrightarrow (\varepsilon, \sigma[x \mapsto \mathsf{eval}_\sigma(t)])} \quad (4.2)$$

$$\frac{\sigma \models \varphi}{(\varphi \texttt{ -> } \pi, \sigma) \Longrightarrow (\pi, \sigma)} \quad (4.3) \qquad \frac{\sigma \not\models \varphi}{(\varphi \texttt{ -> } \pi, \sigma) \Longrightarrow (\varepsilon, \mathsf{fail})} \quad (4.4)$$

$$\frac{(\pi, \sigma) \Longrightarrow (\pi', \sigma') \ \textbf{ and } \ \pi' \neq \varepsilon}{(\pi \texttt{ ; } \pi'', \sigma) \Longrightarrow (\pi' \texttt{ ; } \pi'', \sigma')} \quad (4.5) \qquad \frac{(\pi, \sigma) \Longrightarrow (\varepsilon, \sigma')}{(\pi \texttt{ ; } \pi'', \sigma) \Longrightarrow (\pi'', \sigma')} \quad (4.6)$$

FIGURE 7. Addendum to Definition 4.4

Definition 4.5 (correctness semantics of data commands)**.** Final, respectively, $\mathsf{Final}_{\mathrm{fail}}$ denote the functions $\mathbb{C}\mathrm{OMM} \times 2^{\mathbb{A}\mathrm{SSIGNM}} \to 2^{\mathbb{A}\mathrm{SSIGNM} \cup \{\mathsf{fail}\}}$ defined by the following equations:

$$\begin{aligned} \mathsf{Final}(\pi, \Sigma) &= \{\sigma' \mid \sigma \in \Sigma \ \textbf{ and } \ (\pi, \sigma) \Longrightarrow^* (\varepsilon, \sigma')\} \\ \mathsf{Final}_{\mathrm{fail}}(\pi, \Sigma) &= \mathsf{Final}(\pi, \Sigma) \cup \{\mathsf{fail} \mid \sigma \in \Sigma \ \textbf{ and } \ (\pi, \sigma) \Longrightarrow^* (\pi', \mathsf{fail})\} \end{aligned}$$

Apt et al. showed that all programs from a superset of the set of all data commands execute deterministically [AdBO09]. Consequently, also data commands execute deterministically.

Proposition 4.6 (Lemma 3.1 in [AdBO09, Section 3.2])**.**
- $|\mathsf{Final}(\pi, \{\sigma\})| \leq 1$
- $|\mathsf{Final}_{\mathrm{fail}}(\pi, \{\sigma\})| = 1$

To prove the correctness of commandification, we use Hoare logic [Hoa69], where *triples* of the form $\{\varphi\} \ \pi \ \{\varphi'\}$ play a central role. In such a triple, *precondition* φ characterizes the set of initial data states, π denotes the data command to execute on those states, and *postcondition* φ' characterizes the set of final data states after executing π.

Definition 4.7 (triples)**.** $\mathbb{T}\mathrm{RIPL} = \mathbb{DC} \times \mathbb{C}\mathrm{OMM} \times \mathbb{DC}$ denotes the set of all triples, typically denoted by $\{\varphi\} \ \pi \ \{\varphi'\}$.

Let $[\![\varphi]\!]$ denote the set of data states that satisfy φ (i.e., the data assignments characterized by φ). We interpret triples in two senses: that of partial correctness and that of total correctness. In the former case, a triple $\{\varphi\} \ \pi \ \{\varphi'\}$ holds true iff every final data state to which an initial data state characterized by φ can evolve under π satisfies φ'; in the latter case, additionally, execution of π does not abnormally terminate.

Definition 4.8 (interpretation of triples)**.** $\models_{\mathrm{part}}, \models_{\mathrm{tot}} \subseteq \mathbb{T}\mathrm{RIPL}$ denote the smallest relations induced by the following rules:

$$\frac{\mathsf{Final}(\pi, [\![\varphi]\!]) \subseteq [\![\varphi']\!]}{\models_{\mathrm{part}} \{\varphi\} \ \pi \ \{\varphi'\}} \quad (4.7) \qquad \frac{\mathsf{Final}_{\mathrm{fail}}(\pi, [\![\varphi]\!]) \subseteq [\![\varphi']\!]}{\models_{\mathrm{tot}} \{\varphi\} \ \pi \ \{\varphi'\}} \quad (4.8)$$

To prove properties of data commands, we use the following sound *proof systems* for partial and total correctness, adopted from Apt et al. with some minor cosmetic changes [AdBO09].

Definition 4.9 (proof systems of triples)**.** $\vdash_{\mathrm{part}}, \vdash_{\mathrm{tot}} \subseteq \mathbb{T}\mathrm{RIPL}$ denote the smallest relations induced by the rules in Figure 8.

$$\frac{}{\vdash_{\text{part}} \{\varphi\}\ \texttt{skip}\ \{\varphi\}} \quad (4.9) \qquad \frac{}{\vdash_{\text{tot}} \{\varphi\}\ \texttt{skip}\ \{\varphi\}} \quad (4.10)$$

$$\frac{}{\vdash_{\text{part}} \{\varphi[t/x]\}\ x \texttt{ := } t\ \{\varphi\}} \quad (4.11) \qquad \frac{}{\vdash_{\text{tot}} \{\varphi[t/x]\}\ x \texttt{ := } t\ \{\varphi\}} \quad (4.12)$$

$$\frac{\vdash_{\text{part}} \{\varphi_1\}\ \pi_1\ \{\varphi\} \\ \textbf{and } \vdash_{\text{part}} \{\varphi\}\ \pi_2\ \{\varphi_2\}}{\vdash_{\text{part}} \{\varphi_1\}\ \pi_1 \texttt{ ; } \pi_2\ \{\varphi_2\}} \quad (4.13) \qquad \frac{\vdash_{\text{tot}} \{\varphi_1\}\ \pi_1\ \{\varphi\} \\ \textbf{and } \vdash_{\text{tot}} \{\varphi\}\ \pi_2\ \{\varphi_2\}}{\vdash_{\text{tot}} \{\varphi_1\}\ \pi_1 \texttt{ ; } \pi_2\ \{\varphi_2\}} \quad (4.14)$$

$$\frac{\vdash_{\text{part}} \{\varphi_1'\}\ \pi\ \{\varphi_2'\} \\ \textbf{and } \varphi_1 \Rightarrow \varphi_1' \textbf{ and } \varphi_2' \Rightarrow \varphi_2}{\vdash_{\text{part}} \{\varphi_1\}\ \pi\ \{\varphi_2\}} \quad (4.15) \qquad \frac{\vdash_{\text{tot}} \{\varphi_1'\}\ \pi\ \{\varphi_2'\} \\ \textbf{and } \varphi_1 \Rightarrow \varphi_1' \textbf{ and } \varphi_2' \Rightarrow \varphi_2}{\vdash_{\text{tot}} \{\varphi_1\}\ \pi\ \{\varphi_2\}} \quad (4.16)$$

$$\frac{\vdash_{\text{part}} \{\varphi \wedge \ell\}\ \pi\ \{\varphi'\}}{\vdash_{\text{part}} \{\varphi\}\ \ell \texttt{ -> } P\ \{\varphi'\}} \quad (4.17) \qquad \frac{\vdash_{\text{tot}} \{\varphi\}\ \pi\ \{\varphi'\} \textbf{ and } \varphi \Rightarrow \ell}{\vdash_{\text{tot}} \{\varphi\}\ \ell \texttt{ -> } \pi\ \{\varphi'\}} \quad (4.18)$$

$$\frac{\vdash_{\text{part}} \{\varphi\}\ \pi\ \{\varphi_1\} \textbf{ and } \vdash_{\text{tot}} \{\varphi\}\ \pi\ \{\varphi_2\}}{\vdash_{\text{tot}} \{\varphi\}\ \pi\ \{\varphi_1 \wedge \varphi_2\}} \quad (4.19)$$

FIGURE 8. Addendum to Definition 4.9

Proposition 4.10 (Theorem 3.6 in [AdBO09, Section 3.7])**.**

- $\vdash_{\text{part}} \{\varphi\}\ \pi\ \{\varphi'\}$ **implies** $\models_{\text{part}} \{\varphi\}\ \pi\ \{\varphi'\}$
- $\vdash_{\text{tot}} \{\varphi\}\ \pi\ \{\varphi'\}$ **implies** $\models_{\text{tot}} \{\varphi\}\ \pi\ \{\varphi'\}$

Note that the first four rules for $\vdash_{\text{part}}$ and the first four rules for $\vdash_{\text{tot}}$ have the same premise/consequence. We use $\vdash_{\text{part}}$ to prove the *soundness* of commandification; We use $\vdash_{\text{tot}}$ to prove commandification's *completeness.*

Commandification (without Cycles). At run-time, to check if a transition (q, P, φ, q') can fire, a compiler-generated coordinator thread first checks every port in P for *readiness.* For instance, every (data structure for an) input port should have a pending `put`. Subsequently, the coordinator thread checks whether a data state σ exists that (i) satisfies φ and (ii) subsumes an *initial data state* σ_{init} (i.e., $\sigma_{\text{init}} \subseteq \sigma$). If so, we call σ a *solution* of φ under σ_{init}. The domain of σ_{init} contains all *uncontrollable data variables* in φ: the input ports in P (intersected with $\mathsf{Free}(\varphi)$) and $^\bullet m$ for every memory cell m in the CA (also intersected with $\mathsf{Free}(\varphi)$). More precisely, σ_{init} maps every input port p in $\mathsf{Free}(\varphi)$ to the particular datum *forced* to pass through p by the process thread on the other side of p (i.e., the datum involved in p's pending `put`), while σ_{init} maps every $^\bullet m$ in $\mathsf{Free}(\varphi)$ to the datum that currently resides in m. Thus, before the coordinator thread invokes a constraint solver for φ, it already fixes values for all uncontrollable data variables in φ; when subsequently invoked, a constraint solver may, in search of a solution for φ under σ_{init}, select values only

for data variables outside σ_{init}'s domain. Slightly more formally:

$$\sigma_{\text{init}} = \left\{ p \mapsto d \;\middle|\; \begin{array}{l} [\text{the } \texttt{put} \text{ pending on input port } p \text{ involves datum } d] \\ \textbf{and } \; p \in \mathsf{Free}(\varphi) \end{array} \right\}$$
$$\cup \left\{ {}^{\bullet}m \mapsto d \;\middle|\; \begin{array}{l} [\text{memory cell } m \text{ currently contains datum } d] \\ \textbf{and } \; {}^{\bullet}m \in \mathsf{Free}(\varphi) \end{array} \right\}$$

With commandification, instead of invoking a constraint solver, the coordinator thread executes a compiler-generated data command for φ on σ_{init}, thereby gradually extending σ_{init} to a full solution. This compiler-generated data command essentially works as an efficient, small, dedicated constraint solver for φ.

To translate a data constraint of the form $\ell_1 \wedge \cdots \wedge \ell_k$, we construct a data command that (i) enforces as many data literals of the form t_1 = t_2 as possible with assignment statements and (ii) checks all remaining data literals with failure statements. We call data literals of the form t_1 = t_2 *data equalities*. To examplify such commandification, recall data constraint φ_{eg} on page 12. In this data constraint, let C denote an input port and let x denote a memory cell. In that case, the set of uncontrollable data variables in φ_{eg} consists of C and •x. Now, φ_{eg} has six correct commandifications:

```
π1 = B := •x ;         π2 = B := •x ;         π3 = B := •x ;
     D := C ;               D := C ;               D := C ;
     E := add(B, D) ;       E := add(B, D) ;       E := add(B, D) ;
     F := E ;               G := E                 G := E
     G := E                 F := E ;               ¬Odd(G) -> skip ;
     ¬Odd(G) -> skip ;      ¬Odd(G) -> skip ;      F := E ;

π4 = D := C ;          π5 = D := C ;          π6 = D := C ;
     B := •x ;              B := •x ;              B := •x ;
     E := add(B, D) ;       E := add(B, D) ;       E := add(B, D) ;
     F := E ;               G := E                 G := E
     G := E                 F := E ;               ¬Odd(G) -> skip ;
     ¬Odd(G) -> skip ;      ¬Odd(G) -> skip ;      F := E ;
```

We stipulate the same precondition for each of these data commands, namely that •x and C have a non-`nil` value (later formalized as data literals •x = •x and C = C). This precondition models that the execution of these data commands should always start on an initial data state over the uncontrollable data variables •x and C. Under this precondition, if a coordinator thread executes π_1, it first assigns the values of •x and C to B and D. Subsequently, it assigns the evaluation of add(B, D) to E. Next, it assigns the value of E to F and G. Finally, it checks ¬Odd(G) with a failure statement. Data commands π_2 and π_3 differ from data command π_1 only in the order of the last three steps; data commands π_4, π_5 and π_6 differ from π_1, π_2 and π_3 only in the order of the first two steps. If execution of π_i on σ_{init} successfully terminates, the resulting final data state σ satisfies φ_{eg}. We call this *soundness*. Moreover, if a σ' exists such that $\sigma' \models \varphi_{\text{eg}}$ and $\sigma_{\text{init}} \subseteq \sigma'$, execution of π_i successfully terminates. We call this *completeness*.

Generally, soundness and completeness crucially depend on the order in which assignments and failure statements follow each other in π. For instance, changing the order of G := E and ¬Odd(G) -> skip in the previous example yields a data command whose execution always fails (because G does not have a value yet on evaluating the guard of the

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \\ \textbf{and } x \in \mathsf{Variabl}(\ell)}{x = t \sqsubseteq \ell} \quad (4.20) \qquad \frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \\ \textbf{and } [\ell \neq x' = t' \textbf{ for all } x', t']}{x = t \sqsubseteq \ell} \quad (4.21)$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \textbf{ and } \ell_2 \sqsubseteq \ell_3 \textbf{ and } \ell_2 \notin \{\ell_1, \ell_3\}}{\ell_1 \sqsubseteq \ell_3} \quad (4.22)$$

Figure 9. Addendum to Definition 4.12

failure statement). Such a trivially sound but incomplete data constraint serves no purpose. As another complication, not every data equality can become an assignment. In a first class of cases, neither the left-hand side nor the right-hand side of a data equality matches data variable x. For instance, We *must* translate $\texttt{add}(\mathsf{B}, \mathsf{D}) = \texttt{mult}(\mathsf{B}, \mathsf{D})$ into a failure statement, because we clearly cannot assign either of its two operands to the other. In a second class of cases, multiple data equalities in a data constraint have a left-hand side or a right-hand side that matches the same data variable x. For instance, we can translate only one data equality in $\mathsf{E} = \texttt{add}(\mathsf{B}, \mathsf{D}) \wedge \mathsf{E} = \texttt{mult}(\mathsf{B}, \mathsf{D})$ into an assignment, after which we *must* translate the other one into a failure statement, to avoid conflicting assignments to E.

To deal with these complications, we define a *precedence relation* on the data literals in a data constraint that formalizes their dependencies. Recall from Definition 2.11 that every data constraint consists of a conjunctive kernel of data literals, enveloped with existential quantifications. First, for technical convenience, we introduce a function that extends $\mathsf{Liter}(\varphi)$ (i.e., the data literals in the kernel of φ) with "symmetric data equalities".

Definition 4.11 (=-symmetric closure)**.** $\mathsf{Liter}^{=} : \mathbb{DC} \to 2^{\mathbb{DC}}$ denotes the function defined by the following equation:

$$\mathsf{Liter}^{=}(\varphi) = \mathsf{Liter}(\varphi) \cup \{t_2 = t_1 \mid t_1 = t_2 \in \mathsf{Liter}(\varphi)\}$$

Obviously, because $t_1 = t_2 \equiv t_2 = t_1$, we have $\bigwedge \mathsf{Liter}(\varphi) \equiv \bigwedge \mathsf{Liter}^{=}(\varphi)$ for all φ.

Definition 4.12 (precedence I)**.** $\sqsubseteq : \mathbb{DC} \to 2^{\mathbb{DC} \times \mathbb{DC}}$ denotes the function defined by the following equation:

$$\sqsubseteq(\varphi) = \sqsubseteq$$

where $\sqsubseteq$ denotes the smallest relation induced by the rules in Figure 9.

We usually write $\sqsubseteq_\varphi$ instead of $\sqsubseteq(\varphi)$ and use $\sqsubseteq_\varphi$ as an infix relation. In words, $x = t \sqsubseteq_\varphi \ell$ means that the assignment $x := t$ precedes the commandification of ℓ (i.e., ℓ depends on x). Rule 4.20 deals with the previously discussed first class of data-equalities-that-cannot-become-assignments, by imposing precedence only on data literals of the form $x = t$; shortly, we comment on the second class of data-equalities-that-cannot-become-assignments. Rule 4.21 conveniently ensures that every $x = t$ precedes all differently shaped data literals. Strictly speaking, we do not need this rule, but it simplifies some notation and proofs later on.

For the sake of argument—generally, this does *not* hold true—suppose that a precedence relation $\sqsubseteq_\varphi$ denotes a *strict partial order* on $\mathsf{Liter}^{=}(\varphi)$. In that case, we can *linearize* $\sqsubseteq_\varphi$ to a strict total order $<$ (i.e., embedding $\sqsubseteq_\varphi$ into $<$ such that $\sqsubseteq_\varphi \subseteq <$) with a topological sort on the digraph $(\mathsf{Liter}^{=}(\varphi), \sqsubseteq_\varphi)$ [Kah62, Knu97]. Intuitively, such a linearization gives us an order in which we can translate data literals in $\mathsf{Liter}^{=}(\varphi)$ to data commands in a sound and

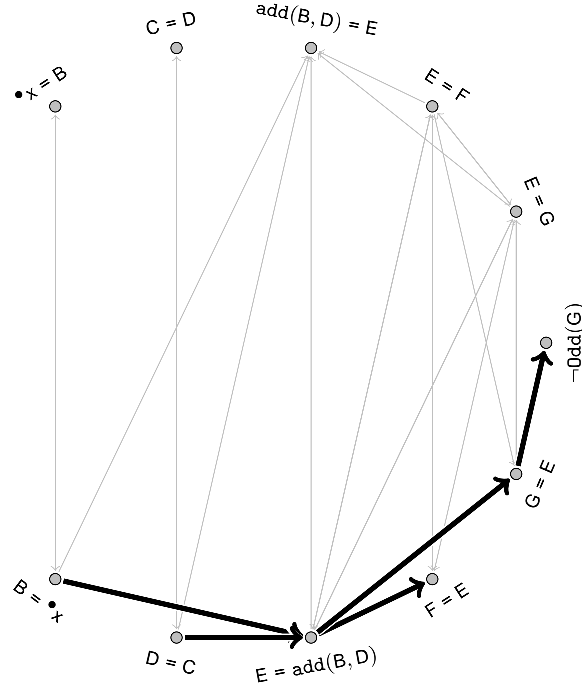

FIGURE 10. Digraph for precedence relation $\sqsubseteq_{\varphi_{\mathrm{eg}}}$ (without loop arcs and without arcs induced by Rule 4.21, to avoid further clutter). An arc (ℓ, ℓ') corresponds to $\ell \sqsubseteq_{\varphi_{\mathrm{eg}}} \ell'$. Arcs between the same data vertices, but in different directions, lie on top of each other. Bold arcs represent a fragment of the strict partial order extracted from $\sqsubseteq_{\varphi_{\mathrm{eg}}}$.

complete way. Shortly, we give an algorithm for doing so and indeed prove its correctness. Problematically, however, $\sqsubseteq_{\varphi}$ generally does not denote a strict partial order: generally, it violates asymmetry and irreflexivity (i.e., graph-theoretically, it contains many cycles). For instance, Figure 10 shows the digraph $(\mathsf{Liter}^{=}(\varphi_{\mathrm{eg}}), \sqsubseteq_{\varphi_{\mathrm{eg}}})$, which indeed contains cycles. For now, we defer this issue to the next subsection, because it forms a concern orthogonal to the commandification algorithm and its correctness. Until then, we simply assume the existence of a procedure for extracting a strict partial order from $\sqsubseteq_{\varphi}$, represented by bold arcs in Figure 10.

Algorithm 1 translates a data constraint φ, a set of data variables X, and a binary relation on data literals $<$ to a data command π. It **require**s the following on its input. First, $<$ should denote a strict total order on the =-symmetric closure of φ's data literals. Let n denote *a*—not necessarily *the*—number of data equalities in $\mathsf{Liter}^{=}(\varphi)$, and let m denote the number of remaining data literals in $\mathsf{Liter}^{=}(\varphi)$. Then, $\ell_1, \ldots, \ell_{n+m}$ denote the data literals in $\mathsf{Liter}^{=}(\varphi)$ such that (i) their indices respect $<$ and (ii) every ℓ_i denotes $x_i = t_i$ for $1 \leq i \leq n$. Next, for every data variable in a data literal in $\mathsf{Liter}^{=}(\varphi)$, but outside the set of uncontrollable data variables X, a data equality $x_i = t_i$ should exist. Otherwise, such a data variable can get a value only through search—exactly what commandification tries to avoid—and not through assignment; *underspecified* data constraints fundamentally lie outside the scope of commandification in general and Algorithm 1 in particular. Finally, if

Algorithm 1 Algorithm for translating a data constraint φ, a set of data variables X, and a binary relation on data literals $<$ to a data command π

Require: $<$ denotes a strict total order on $\mathsf{Liter}^{=}(\varphi)$
and $\mathsf{Liter}^{=}(\varphi) = \{\ell_1, \ldots, \ell_{n+m}\}$
and $\ell_1 < \cdots < \ell_n < \ell_{n+1} < \cdots < \ell_{n+m}$
and $\ell_1 = x_1 = t_1$ **and** $\cdots$ **and** $\ell_n = x_n = t_n$
and $\mathsf{Variabl}(\varphi) \setminus X \subseteq \{x_1, \ldots, x_n\}$
and $\left[\left[\begin{array}{l}[x = t \in \mathsf{Liter}^{=}(\varphi) \textbf{ and } x' \in \mathsf{Variabl}(t)] \textbf{ implies} \\ \quad [x' \in X \textbf{ or } [x' = t' < x = t \textbf{ for some } t']]\end{array}\right] \textbf{ for all } x, x', t\right]$

function COMMANDIFY($\varphi, X, <$)
    $\pi :=$ `skip`
    $i := 1$
    while $i \leq n$ **do**
        if $x_i \in X \cup \{x_1, \ldots, x_{i-1}\}$ **then**
            $\pi := (\pi$ `;` $x_i = t_i$ `->` `skip`$)$
        else
            $\pi := (\pi$ `;` x_i `:=` $t_i)$
        fi
        $i := i + 1$
    od
    while $i \leq n + m$ **do**
        $\pi := (\pi$ `;` ℓ_i `->` `skip`$)$
        $i := i + 1$
    od
    return π
end function

Ensure: $\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_{n+m}\}$
and $\left[\left[\begin{array}{l}\qquad \sigma \models \ell_1 \wedge \cdots \wedge \ell_{n+m} \textbf{ implies} \\ \vdash_{\text{tot}} \{\bigwedge\{x = \sigma(x) \mid x \in X\}\} \\ \qquad \pi \\ \qquad \{\bigwedge\{x = \sigma(x) \mid x \in X \cup \{x_1, \ldots, x_n\}\}\}\end{array}\right] \textbf{ for all } \sigma\right]$

a term t in a data equality $x = t$ depends on a variable x', a data equality $x' = t'$ should precede $x = t$ under $<$. The rules in Definition 4.12 induce precedence relations for which all these **require**ments hold true, except that those precedence relations do not necessarily denote strict partial orders and, hence, may not admit linearization. Consequently, the precedence relations in Definition 4.12 may not yield strict total orders as **require**d by Algorithm 1. We address this issue in the next subsection.

Assuming satisfaction of its **require**ments, Algorithm 1 works as follows. It first loops over the first n (according to $<$) $x_i = t_i$ data literals. If an assignment for x_i already exists in the data command under construction π, Algorithm 1 translates $x_i = t_i$ to a failure statement; otherwise, it translates $x_i = t_i$ to an assignment. This approach resolves issues

$$\frac{\ell_1 \sqsubseteq_\varphi \ell_2}{\ell_1 \sqsubseteq \ell_2} \quad (4.23) \qquad \frac{\ell \in \mathsf{Liter}^{=}(\varphi) \text{ and } \mathsf{Variabl}(\ell) \subseteq X}{\bigstar \sqsubseteq \ell} \quad (4.24) \qquad \frac{x = t \in \mathsf{Liter}^{=} \text{ and } \mathsf{Variabl}(t) \subseteq X}{\bigstar \sqsubseteq x = t} \quad (4.25)$$

FIGURE 11. Addendum to Definition 4.14

with the previously discussed second class of equalities-that-cannot-become-assignments. After the first loop, the algorithm uses a second loop to translate the remaining m data literals to failure statements. The algorithm runs in time linear in $n+m$, and it terminates.

Upon termination, Algorithm 1 **ensure**s the soundness (first conjunct) and completeness of π (second conjunct). Note that we use a different proof system for soundness (partial correctness, $\vdash_{\text{part}}$) than for completeness (total correctness, $\vdash_{\text{tot}}$).

Theorem 4.13. *Algorithm 1 is correct.*

Algorithm 1 has the minor issue that it may produce more failure statements than strictly necessary. For instance, if we run Algorithm 1 on the total order extracted from $\sqsubseteq_{\varphi_{\text{eg}}}$ in Figure 10, we get both the assignment `D := C` and the unnecessary failure statement `C = D -> skip`. After all, the digraph contains both `D = C` and `C = D`, one of which we added while computing $\mathsf{Liter}^{=}(\varphi_{\text{eg}})$ to account for the symmetry of `=`. Generally, such symmetric data literals result either in one assignment and one failure statement or in two failure statements; one can easily prove that symmetric data literals never result in two assignments. In both cases, one can safely remove one of the failure statements, because successful termination of the remaining statement already accounts for the removed failure statement.

Commandification (with Cycles). Algorithm 1 **require**s that $<$ denotes a strict total order. Precedence relations in Definition 4.12 of $\sqsubseteq$, however, do not yield such orders: graph-theoretically, they may contain cycles. In this subsection, we present a solution for this problem. We start by extending the previous precedence relations with a unique least element, $\bigstar$, and by making dependencies of data literals on uncontrollable data variables explicit. In the following definition, let X denote a set of such variables.

Definition 4.14 (precedence II). $\sqsubseteq : \mathbb{DC} \times 2^{\mathbb{X}} \to 2^{(\mathbb{DC} \cup \{\bigstar\}) \times \mathbb{DC}}$ denotes the function defined by the following equation:

$$\sqsubseteq(\varphi, X) = \sqsubseteq$$

where $\sqsubseteq$ denotes the smallest relation induced by the rules in Figure 11.

We usually write $\sqsubseteq_\varphi^X$ instead of $\sqsubseteq(\varphi, X)$ and use $\sqsubseteq_\varphi^X$ as an infix relation. The two new rules state that data literals in which only uncontrollable data variables occur "depend" on $\bigstar$.

Relation $\sqsubseteq_\varphi^X$ denotes a strict partial order if its digraph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}, \sqsubseteq_\varphi^X)$ defines a $\bigstar$-*arborescence*: a digraph consisting of $n-1$ arcs such that a path exists from $\bigstar$ to each of its n vertices [KV08]. Equivalently, in a $\bigstar$-arborescence, $\bigstar$ has no incoming arcs, every other vertex has exactly one incoming arc, and the arcs form no cycles [KV08]. The first formulation seems more intuitive here: every path from $\bigstar$ to some data literal ℓ represents an order in which Algorithm 1 should translate the data literals on that path to ensure

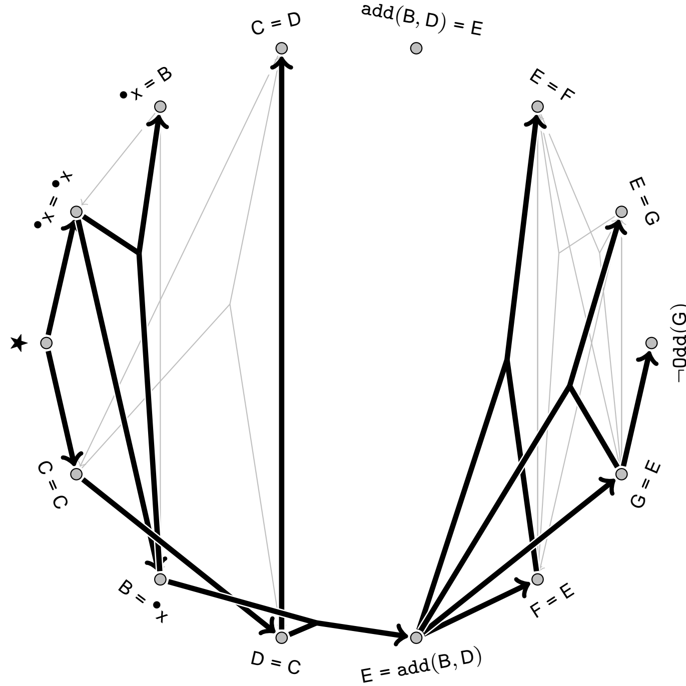

FIGURE 12. B-graph corresponding to the digraph in Figure 10 (without loop B-arcs and without three-tailed B-arcs, to avoid further clutter). An arc (ℓ, ℓ') corresponds to $\ell \sqsubseteq_{\varphi_{\mathrm{eg}}} \ell'$. Bold arcs represent an arborescence.

the correctness of the translation of ℓ. The second formulation simplifies observing that arborescences correspond to strict partial orders.

A naive approach to extract a strict partial order from $\sqsubseteq_{\varphi}^{X}$ consists of computing a $\bigstar$-arborescence of the digraph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}, \sqsubseteq_{\varphi}^{X})$. Even if such a $\bigstar$-arborescence exists, however, this approach does not work as expected if $\mathsf{Liter}^{=}(\varphi)$ contains a data literal $x = t$ where t has more than one data variable. For instance, by definition, every arborescence of the digraph in Figure 10 has only one incoming arc for $\mathsf{E} = \mathsf{add}(\mathsf{B}, \mathsf{D})$, even though assignments to *both* B *and* D must precede an assignment to E. Because these dependencies exist as two separate arcs, no arborescence can capture them. To solve this, we must somehow represent the dependencies of $\mathsf{E} = \mathsf{add}(\mathsf{B}, \mathsf{D})$ with a single incoming arc. We can do so by allowing arcs to have multiple tails, one for every data variable. In that case, we can replace the two separate incoming arcs of $\mathsf{E} = \mathsf{add}(\mathsf{B}, \mathsf{D})$ with a single two-tailed incoming arc as in Figure 12. The two tails make explicit that to evaluate add, we need values for both its arguments: multiple tails represent a conjunction of dependencies of a data literal.

By combining single-tailed arcs into multiple-tailed arcs, we effectively transform the digraphs considered so far into B-*graphs*, a special kind of hypergraph with only B-*arcs* (i.e., *b*ackward hyperarcs, i.e., hyperarcs with exactly one head) [GLPN93]. Generally, we cannot derive such B-graphs from precedence relations as in Definition 4.14: their richer structure makes B-graphs more expressive—they convey strictly more information—than digraphs. In contrast, we can easily transform a B-graph into a precedence relation by splitting B-arcs into single-tailed arcs in the obvious way. Deriving precedence relations from more

$$\frac{\ell \in \mathsf{Liter}^{=}(\varphi) \text{ and } \mathsf{Variabl}(\ell) = \{x_1, \ldots, x_k\} \text{ and } x_1 = t_1, \ldots, x_k = t_k \in \mathsf{Liter}^{=}(\varphi) \cup \{\hat{x} = \hat{x} \mid \hat{x} \in X\}}{\{x_1 = t_1, \ldots, x_k = t_k\} \blacktriangleleft \ell} \quad (4.26)$$

$$\frac{x = t \in \mathsf{Liter}^{=}(\varphi) \text{ and } \mathsf{Variabl}(t) = \{x_1, \ldots, x_k\} \text{ and } x_1 = t_1, \ldots, x_k = t_k \in \mathsf{Liter}^{=}(\varphi) \cup \{\hat{x} = \hat{x} \mid \hat{x} \in X\}}{\{x_1 = t_1, \ldots, x_k = t_k\} \blacktriangleleft x = t} \quad (4.27)$$

$$\frac{x \in X}{\bigstar \blacktriangleleft x = x} \quad (4.28)$$

FIGURE 13. Addendum to Definition 4.15

expressive B-graphs therefore constitutes a correct way of obtaining strict total orders that satisfy the **require**ments of Algorithm 1; doing so just eliminates irrelevant information.

Thus, we propose the following. Instead of formalizing dependencies among data literals in a set $\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}$ directly as a precedence relation, we first formalize those dependencies as a B-graph. If the resulting B-graph defines a $\bigstar$-arborescence, we can directly extract a cycle-free precedence relation $\sqsubset$. Otherwise, we compute a $\bigstar$-arborescence of the resulting B-graph and extract a cycle-free precedence relation $\sqsubset$ afterward. Either way, $\sqsubset$ denotes a strict partial order whose linearization satisfies the **require**ments in Algorithm 1.

Definition 4.15 (B-precedence)**.** $\blacktriangleleft : \mathbb{DC} \times 2^{\mathbb{X}} \to 2^{(2^{\mathbb{DC}} \cup \{\bigstar\}) \times \mathbb{DC}}$ denotes the function defined by the following equation:

$$\blacktriangleleft(\varphi, X) = \blacktriangleleft$$

where $\blacktriangleleft$ denotes the smallest relation induced by the rules in Figure 13.

We usually write $\blacktriangleleft_{\varphi}^{X}$ instead of $\blacktriangleleft(\varphi, X)$ and use $\blacktriangleleft_{\varphi}^{X}$ as an infix relation. Rule 4.26 generalizes Rule 4.20 in Definition 4.12, by joining sets of dependencies of a data literal in a single B-arc. Rule 4.27 states that $x = t$ does not necessarily depend on x—as implied by Rule 4.26—but only on the free variables in t (i.e., we can derive a value for x from values of the data variables in t). Note that through Rules 4.26 and 4.27, we extend the previous domain $\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}$ with *semantically insignificant* data equalities of the form $x = x$, each of which we relate to $\bigstar$ with Rule 4.28. We do this only for the technical convenience of treating both uncontrollable data variables in X (which may have no data equalities in $\mathsf{Liter}^{=}(\varphi)$) and the other variables (which must have data equalities) in a uniform way. For instance, Figure 12 shows the B-graph for data constraint φ_{eg}.

Generally, in a B-graph, data literals can have multiple incoming B-arcs, which represents a disjunction of conjunctions of dependencies. Importantly, as long as Algorithm 1 respects the dependencies represented by *one* incoming B-arc, the other incoming B-arcs do not matter. An arborescence, which contains one incoming B-arc for every data literal, therefore preserves enough dependencies. Shortly, Theorem 4.17 makes this more precise.

$$\frac{\ell_1 \in \mathsf{Liter}^{=}(\varphi) \cap L \ \textbf{ and } \ L \lhd^X_\varphi \ell_2}{\ell_1 \sqsubset \ell_2} \quad (4.29) \qquad \frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{ and } \ \big[\ell \neq x' = t' \ \textbf{ for all } \ x', t'\big]}{x = t \sqsubset \ell} \quad (4.30)$$

$$\frac{\ell_1 \sqsubset \ell_2 \ \textbf{ and } \ \ell_2 \sqsubset \ell_3 \ \textbf{ and } \ \ell_2 \notin \{\ell_1, \ell_3\}}{\ell_1 \sqsubset \ell_3} \quad (4.31)$$

FIGURE 14. Addendum to Definition 4.16

We can straightforwardly compute an arborescence of a B-graph

$$(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}, \blacktriangleleft^X_\varphi)$$

with an exploration algorithm reminiscent of breadth-first search. First, let $\lhd \subseteq \blacktriangleleft^X_\varphi$ denote the aborescence under computation, and let $L_{\text{done}} \subseteq \mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}$ denote the set of vertices (i.e., data literals) already explored; initially, $\lhd = \emptyset$ and $L_{\text{done}} = \{\bigstar\}$. Now, given some L_{done}, compute a set of vertices L_{next} connected only to vertices in L_{done} by a B-arc in $\blacktriangleleft^X_\varphi$. Then, for every vertex in L_{next}, add an incoming B-arc to $\lhd$.[5] Afterward, add L_{next} to L_{done}. Repeat this process until L_{next} becomes empty. Once that happens, either $\lhd$ contains an arborescence (if $L_{\text{done}} = L$) or no arborescence exists. This computation runs in linear time, in the size of the B-graph. See also Footnote 5. Henceforth, let $\lhd^X_\varphi$ denote the final arborescence so computed; if no arborescence exists, we stipulate $\lhd^X_\varphi = \emptyset$.

Definition 4.16 (precedence III)**.** $\sqsubset : \mathbb{DC} \times 2^{\mathbb{X}} \to \mathbb{DC} \times \mathbb{DC}$ denotes the function defined by the following equation:

$$\sqsubset(\varphi, X) = \sqsubset$$

where $\sqsubset$ denotes the smallest relation induced by the rules in Figure 14.

We usually write $\sqsubset^X_\varphi$ instead of $\sqsubset(\varphi, X)$. Rules 4.30 and 4.31 have the same premise/consequence as Rules 4.21 and 4.22; Rule 4.29 straightforwardly splits B-arcs into single-tailed arcs. For instance, the bold arcs in Figure 10 represent a fragment of the precedence relation so derived from the arborescence in Figure 12.

For every $\sqsubset^X_\varphi$ induced from a nonempty $\bigstar$-arborescence (i.e., $\lhd^X_\varphi \neq \emptyset$), let $<^X_\varphi$ denote its linearization. The following theorem states that this linearization satisfies the **require**ments of Algorithm 1.

Theorem 4.17.

$$\lhd^X_\varphi \neq \emptyset \ \textbf{ implies } \ \big[(\varphi, X, <^X_\varphi) \text{ satisfies the } \textbf{\textit{require}}\text{ments of Algorithm 1}\big]$$

[5] If a vertex ℓ in L_{next} has multiple incoming B-arcs, the choice among them matters not: the choice remains local, because every B-arc has only one head (i.e., adding an ℓ-headed B-arc to $\lhd$ cannot cause another vertex to get multiple incoming B-arcs, which would invalidate the arborescence). General hypergraphs, whose hyperarcs can have multiple heads, violate this property (i.e., the choice of which hyperarc to add becomes global instead of local). As a result, and in stark constrast to B-graphs, one cannot compute arborescences of general hypergraphs—an NP-complete problem [Woe92]—in polynomial time (if P $\neq$ NP).

If the B-graph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}, \blacktriangleleft_{\varphi}^{X})$ neither defines nor contains a $\bigstar$-arborescence, no B-graph equivalent of a path [AFF01] exists from $\bigstar$ to at least one vertex ℓ. In that case, the other vertices fail to resolve at least one of ℓ's dependencies. This occurs, for instance, when ℓ depends on x, but the B-graph contains no $x = t$ vertex. As another example, consider a recursive data equality $x = t$ with $x \in \mathsf{Variabl}(t)$: unless another data equality $x = t'$ with $t \neq t'$ exists, every incoming B-arc in its B-graph loops onto itself. Consequently, no arborescence exists. In practice, such cases inherently require constraint solving techniques with backtracking to find a value for x. Nonexistence of a $\bigstar$-arborescence thus signals a hard limit to the applicability of Algorithm 1 (although mixed techniques of translating some parts of a data constraint to a data command at compile-time and leaving other parts to a constraint solver at run-time seem worthwhile to explore; we leave this possibility for future work). Thus, the set of data constraints to which we can apply Algorithm 1 contains those (i) whose B-graph has a $\bigstar$-arborescence, which guarantees linearizability of the induced precedence, and (ii) that satisfy also the rest of the **require**ments in Algorithm 1.

Commandify. To introduce data commands in CAs, we introduce *commandify* as a unary operation on CAs. First, because we want to avoid ad-hoc modifications to Definitions 2.11 and 2.14 (of data constraints and CAs), we present an encoding of data commands as data relations. In the following definition, let φ denote a data constraint in a CA, let X denote the set of uncontrollable data variables in φ, and let $x_1, \ldots, x_k$ denote the free data variables in φ, ordered by $<_{\mathbb{T}\text{ERM}}$. Then, data relation R, which encodes the commandification π of φ, holds true of a data tuple $(d_1, \ldots, d_k)$ iff execution of π on an initial data state (over the variables in X) successfully terminates on a data state σ that maps every x_i to d_i.

Definition 4.18 (data commands as data relations)**.** $\mathsf{comm} : \mathbb{DC} \times 2^{\mathbb{X}} \to \mathbb{DC}$ denotes the function defined by the following equation:

$$\mathsf{comm}(\varphi, X) = \begin{cases} R(x_1, \ldots, x_k) \ \ \textbf{if} & \begin{bmatrix} \mathsf{Free}(\varphi) = \{x_1, \ldots, x_k\} \\ \textbf{and} \ \ x_1 <_{\mathbb{T}\text{ERM}} \cdots <_{\mathbb{T}\text{ERM}} x_k \\ \textbf{and} \ \ \lhd_{\varphi}^{X} \neq \emptyset \\ \textbf{and} \ \ X \subseteq \mathsf{Free}(\varphi) \end{bmatrix} \\ \varphi & \textbf{otherwise} \end{cases}$$

where R denotes the smallest relation induced by the following rule:

$$\frac{\begin{array}{l} \pi = \textsc{Algorithm1}(\varphi, X, \sqsubset_{\varphi}^{X}) \\ \textbf{and} \ \ \sigma \in \mathsf{Final}(\pi, [\![\bigwedge \{x = x \mid x \in X\}]\!]) \\ \textbf{and} \ \ \sigma(x_1), \ldots, \sigma(x_k) \in \mathbb{D} \end{array}}{(\sigma(x_1), \ldots, \sigma(x_k)) \in R} \tag{4.32}$$

Note that σ in Rule 4.32 may map also data variables outside $\mathsf{Free}(\varphi)$. This happens, for instance, with data constraints with existential quantifiers. The data commands for such data constraints explicitly assign values to quantified data variables, even though those variables do not qualify as free. Because $\{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ contains the free data variables in φ, however, the additional data variables mapped by σ cannot affect the truth of φ (by monotonicity of entailment).

We define commandification in CAs in terms of comm.

Definition 4.19 (commandify). $(\!|\cdot|\!) : \mathbb{A}\text{UTOM} \to \mathbb{A}\text{UTOM}$ denotes the function defined by the following equation:

$$(\!|(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, q^0)|\!) = (Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, (\!|\longrightarrow|\!), q^0, \mu^0)$$

where $(\!|\longrightarrow|\!)$ denotes the smallest relation induced by the following rules:

$$\frac{q \xrightarrow{P,\varphi} q' \ \textbf{ and } \ X^{\text{init}} = P^{\text{in}} \cup {}^{\bullet}M}{q \ (\!|\xrightarrow{P,\mathsf{comm}(\varphi,\mathsf{Free}(\varphi)\cap X^{\text{init}})}|\!) \ q'} \tag{4.33}$$

Correctness and Effectiveness. We conclude this section by establishing the correctness and effectiveness of commandify. We consider commandify correct if it yields a behaviorally congruent CA to the original one. Before formulating this as a theorem, the following lemma first states the equivalence of a data constraint and its commandification.

Lemma 4.20. $\varphi \equiv \mathsf{comm}(\varphi, X)$

From Proposition 2.15 and Lemma 4.20, we conclude the following correctness theorem.

Theorem 4.21. $\boldsymbol{a} \simeq (\!|\boldsymbol{a}|\!)$

We consider commandify effective if, after commandifying a CA $\boldsymbol{a}$, every data constraint in the resulting CA either encodes a data command as in Definition 4.18 or has no data variables in it (in which case a compiler can statically check that data constraint). Generally, however, such unconditional effectiveness does not hold true. After all, if the B-graph for a data constraint φ in $\boldsymbol{a}$ has no $\bigstar$-arborescence, we have no strict precedence relation to run Algorithm 1 with. In that case, $\mathsf{comm}(\varphi, X) = \varphi$, and consequently, commandify does not have its intended effect. Fortunately, commandify does satisfy a weaker—but useful—form of effectiveness. To formulate this as a theorem, we first define a relation that holds true of *arborescent* CAs. We consider a CA arborescent if the B-graph for each of its data constraints has a $\bigstar$-arborescence.

Definition 4.22 (arborescentness). $\clubsuit \subseteq \mathbb{A}\text{UTOM}$ denotes the smallest relation induced by the following rule:

$$\frac{\left[\varphi \in \mathsf{Dc}(\boldsymbol{a}) \ \textbf{ implies } \ \lhd_{\varphi}^{X} \neq \emptyset\right] \ \textbf{ for all } \ \varphi}{\clubsuit \, \boldsymbol{a}} \tag{4.34}$$

The following theorem states the effectiveness of commandify, conditional on arborescentness: after commandifying an arborescent CA $\boldsymbol{a}$, every data constraint in the resulting CA encodes a data command as a data relation (as in Definition 4.18). Let R range over the set of data relations defined in Definition 4.18 of comm.

Theorem 4.23. $\clubsuit\, \boldsymbol{a}$ **implies** $\mathsf{Dc}((\!|\boldsymbol{a}|\!)) \subseteq \{R(x_1, \ldots, x_k) \mid \textbf{true}\}$

Discussion. The constraint programming community has already observed that, for constraint solving, "if domain specific methods are available they should be applied *instead* [sic] of the general methods" [Apt09a]. Commandification pushes this piece of conventional wisdom to an extreme: essentially, every data command generated for a data constraint φ by Algorithm 1 constitutes a small, dedicated constraint solver capable of solving only φ. Nevertheless, execution of data commands bears similarities with *constraint propagation* techniques, in particular with *forward checking* [BMFL02]. Generally, constraint propagation aims to reduce the search space of a constraint satisfaction problem by transforming it into an equivalent "simpler" one, where variables have smaller domains, or where constraints refer to fewer variables. With forward checking, whenever a variable x gets a value d, a constraint solver removes values from the domains of all subsequent variables that, given d, violate a constraint. In the case of an equality $x = x'$, for instance, forward checking reduces the domain of x' to the singleton $\{d\}$ after an assignment of d to x. Commandification implicitly uses that same property of equality, but instead of explicitly representing the domain of a variable and the reduction of this domain to a singleton at run-time, commandification already turns the equality into an assignment at compile-time.

Commandification may also remind one of classical *Gaussian eliminination* for solving systems of linear equations over the reals [Apt09b]: there too, one orders variables and substitutes values/expressions for variables in other expressions. Data constraints, however, have a significantly different structure from real numbers, which makes solving data constraints directly via Gaussian elimination at least not obvious.

Before we did the work presented in this paper, Clarke et al. already worked on purely constraint-based implementations of protocols [CPLA11]. Essentially, Clarke et al. specify not only the transition labels of an automaton as boolean constraints but also its state space and transition relation. In recent work, Proença and Clarke developed a variant of compile-time *predicate abstraction* to improve performance [PC13a]. They also used this technique to allow a form of interaction between a constraint solver and its environment during constraint solving [PC13b]. The work of Proença and Clarke resembles our work in the sense that we all try to "simplify" constraints at compile-time. We see also differences, though: (i) commandification fully avoids constraint solving and (ii) we adopted a richer language of data constraints in this paper. For instance, Proença and Clarke have only unary functions in their language, which would have avoided our need for B-graphs.

5. Experiments

Setup. We implemented our two optimization techniques as extensions to our existing CA-to-Java compiler, a plug-in for the Eclipse IDE. This plug-in is an integrated part of a larger toolset, which also consists of an editor that supports the graphical syntax for CAs presented in Section 2, through a drag-and-drop interface. To evaluate the impact of our optimization techniques in practice, then, we performed a number of experiments with their implementation, the results of which we present in this section.

We divided our experiments into two categories. The first category consists of experiments involving compiler-generated coordinator threads in isolation. These experiments are "pure" in the sense that we measure only the performance of the compiler-generated code, without "polluting" these measurements with delays caused by process threads. The second category consists of experiments involving compiler-generated coordinator threads

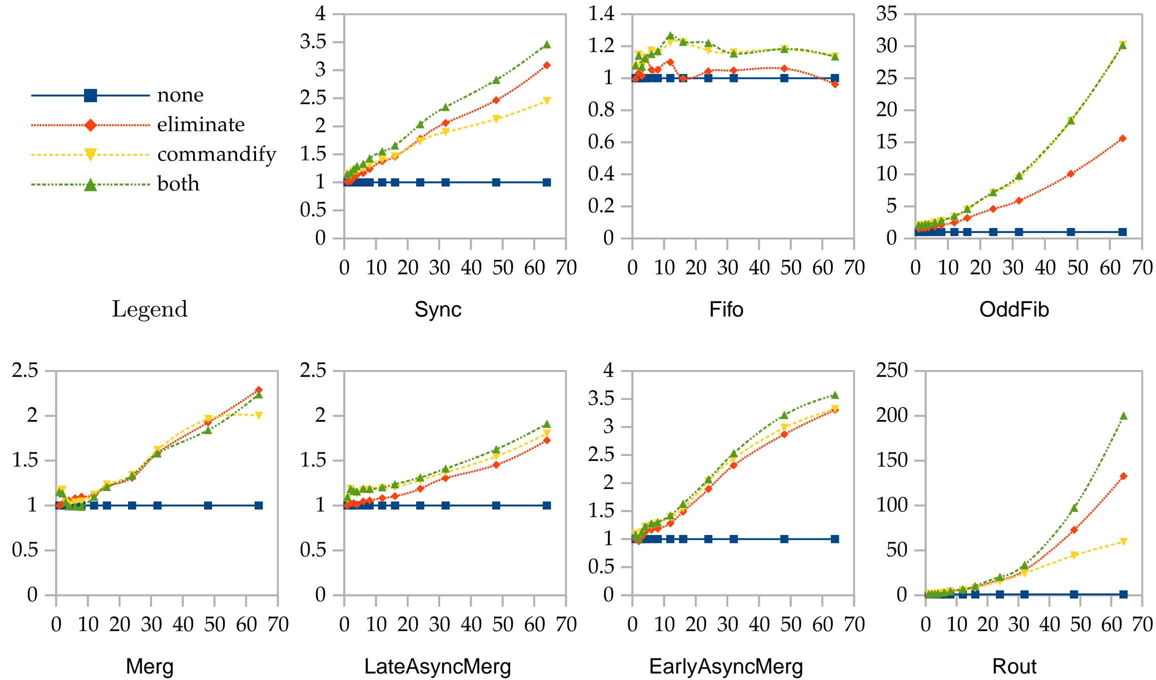

FIGURE 15. Experimental results for seven sets of CAs in isolation: speedups, on the y-axis, of compiler-generated code optimized with eliminate, commandify, or both, relative to unoptimized code, as a function of the number of processes, on the x-axis

in the context of full programs. These experiments allow us to observe the impact of our optimization techniques on the performance of full programs.

We ran each of our experiments five times on a machine with 24 cores (two Intel E5-2690V3 processors in two sockets), without Hyper-Threading and without Turbo Boost (i.e., with a static clock frequency), and averaged our measurements afterward.

Category I. To study the performance of compiler-generated coordinator threads in isolation, we selected seven sets of CAs for experimentation, whose elements differ in the value of $k \in \{1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64\}$: Sync_k, Fifo_k, OddFib_k, Merg_k, $\mathsf{LateAsyncMerg}_k$, $\mathsf{EarlyAsyncMerg}_k$, and Rout_k. In total, thus, we generated code for 96 CAs, yielding 96 experiments. Application of our optimization techniques did not add any measurable compilation overhead. Each of these CAs, except the Merg_k CAs, is the k-parametric generalization of a CA denoted by a digraph in Figure 6; every Merg_k CA is the k-parametric generalization of Merg2 in Figure 3. For Sync_k/Fifo_k, parameter k controls the number of Syncs/Fifos in the chain. For Merg_k, $\mathsf{LateAsyncMerg}_k$, and $\mathsf{EarlyAsyncMerg}_k$, parameter k controls the number of producers. For OddFib_k and Rout_k, parameter k controls the number of consumers. See Section 2 for a brief description of the behavior of these CAs for $k = 2$.

In each run of an experiment, we measured the number of completed transitions in four minutes after warming up the Java virtual machine for thirty seconds. To measure the performance of only the compiler-generated code, we used "empty" producers and consumers, which essentially execute `while (true) put(...)` and `while (true) get(...)`.

Figure 15 shows our experimental results. The figure shows that, individually, our two optimization techniques are already very effective. When we apply both optimization techniques simultaneously, in many cases (Sync_k, $\mathsf{LateAsyncMerg}_k$, $\mathsf{EarlyAsyncMerg}_k$, and Rout_k), performance is further improved, but the improvement is not the sum of the individual improvements. The reason is that after applying one of the techniques, there is "less room" for the other technique to make further improvement: there is only so much that can be optimized in checking data constraints, and each of our two techniques individually seems to already make a significant step toward an optimum. Still, as Figure 15 shows, it is useful to apply both techniques, especially since they do not appear to negatively influence each other.

Category II. To study the performance of compiler-generated coordinator threads in the context of full programs, we adapted the NAS *Parallel Benchmarks* NPB [BBB+91], a popular suite to evaluate parallel performance with. The NPB suite specifies eight benchmarks—five computational kernels and three realistic applications—derived from computational fluid dynamics programs; for each of these benchmarks, to standardize comparisons, the NPB suite specifies four classes of problem sizes (class W, class A, class B, class C).

We compared the Java reference implementation of NPB with a CA-based implementation. The Java reference implementation, developed by Frumkin et al. [FSJY03], contains a Java program for seven of NPB's eight benchmarks; one kernel benchmark is missing. Each of these programs consists of a *master* process and a number of *worker* processes. The master and its workers interact with each other under a classical master/workers protocol (i.e., the master distributes work among its workers; the workers inform their master once their work is done). Frumkin et al. programmed this protocol using monitors.

We took the Java reference implementation of NPB as the basis for our CA-based implementation. First, we removed all instances of the master/workers protocol from the seven programs. Then, we added ports and `put`/`get`. Separately, we drew the master/workers protocol in our graphical syntax for CAs. Subsequently, we compiled our specification for $k \in \{2, 4, 8, 16, 32, 64\}$ workers (unless a combination of benchmark+class supported only fewer workers), and let our compiler automatically integrate the hand-written code (for masters/workers) with its own compiler-generated code. Application of our optimization techniques did not add any measurable compilation overhead.

Figures 16 and 17 show our experimental results. These results, in contrast to the results in Figure 15, look messy and are hard to derive a meaningful conclusion from: in some cases, using both optimizations results in the best performance, but in other cases, using only one of the optimizations results in the best performance, and in yet a few other cases, using *no* optimization actually results in the best performance.

The reason for these results, so we found out, has to do with hardware cache performance: it turns out that the memory footprint of our compiler-generated code seriously impacts numbers of cache misses, a phenomenon that did not yet manifest when we ran our compiler-generated code in isolation. As we have not yet optimized compiler-generated code for memory usage, a reasonable assumption is that code with a large memory footprint results in more cache misses. However, things are even more subtle than that: due to the way the Java virtual machine allocates memory, so we found out, a *larger* memory footprint may in fact result in *fewer* cache misses. We admit that we do not yet understand the impact of the memory footprint of our compiler-generated code on the execution-time performance of the code sufficiently well enough to appropriately account for this impact in our

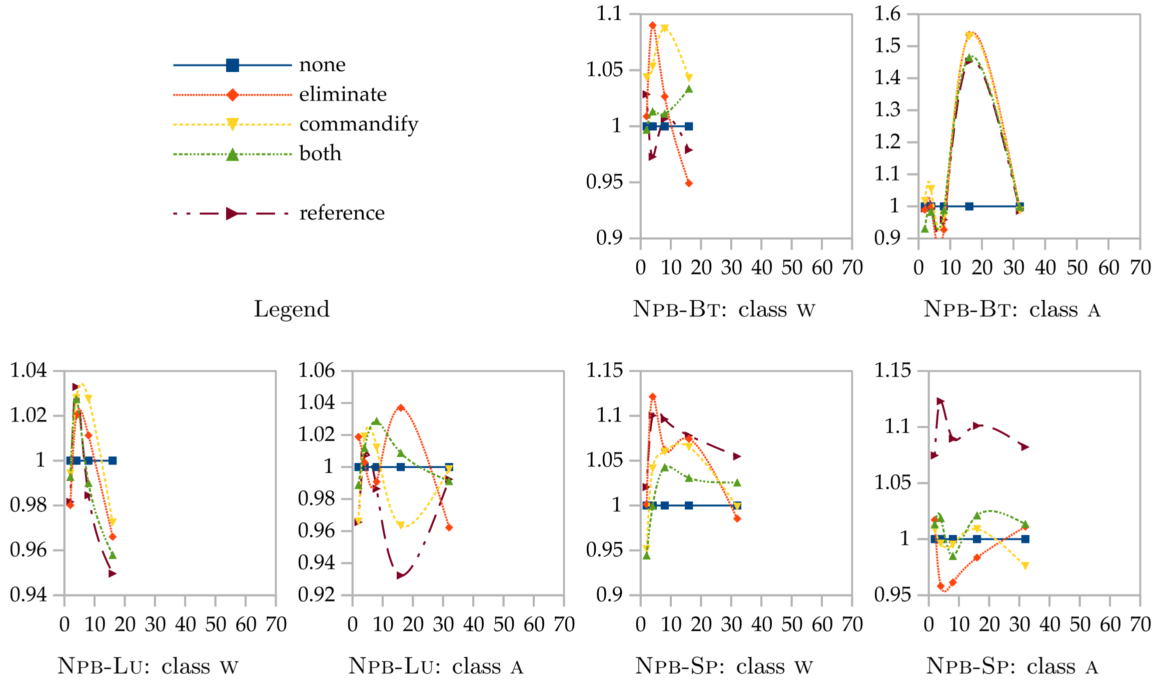

FIGURE 16. Experimental results for three NPB applications: speedups (y-axis) of compiler-generated code optimized with eliminate, commandify, or both, and of reference code by Frumkin et al., relative to unoptimized compiler-generated code, as a function of the number of processes (x-axis)

optimization schemes. This investigation constitutes an important piece of our future work. We consider the revelation of this underdeveloped aspect of our compilation technology as a significant contribution of this paper.

6. Conclusion

We presented, and established the correctness of, two techniques to optimize the performance of checking data constraints. The first technique, called “eliminate” and formalized as operation $\ominus$, reduces the size of data constraints at compile-time, to reduce the complexity of constraint solving at run-time. The second technique, called “commandify” and formalized as operation $(\!|\cdot|\!)$, translates data constraints into small pieces of imperative code at compile-time, to replace expensive calls to a general-purpose constraint solver at run-time. Finding satisfying assignments for data constraints resembles a game of hide-and-seek, played by our compiler-generated code at run-time with the aid of a constraint solver. This game was reasonable when our CA compilation technology was still in its infancy, but no longer as this technology matures.

Although the experiments in which we evaluated compiler-generated code in isolation show that eliminate and commandify indeed have a positive impact on performance, the experiments in which we evaluated compiler-generated code in the context of full programs remain inconclusive because of seemingly erratic hardware cache behavior. Here lies an important next research step: we need to better understand the impact of memory footprints

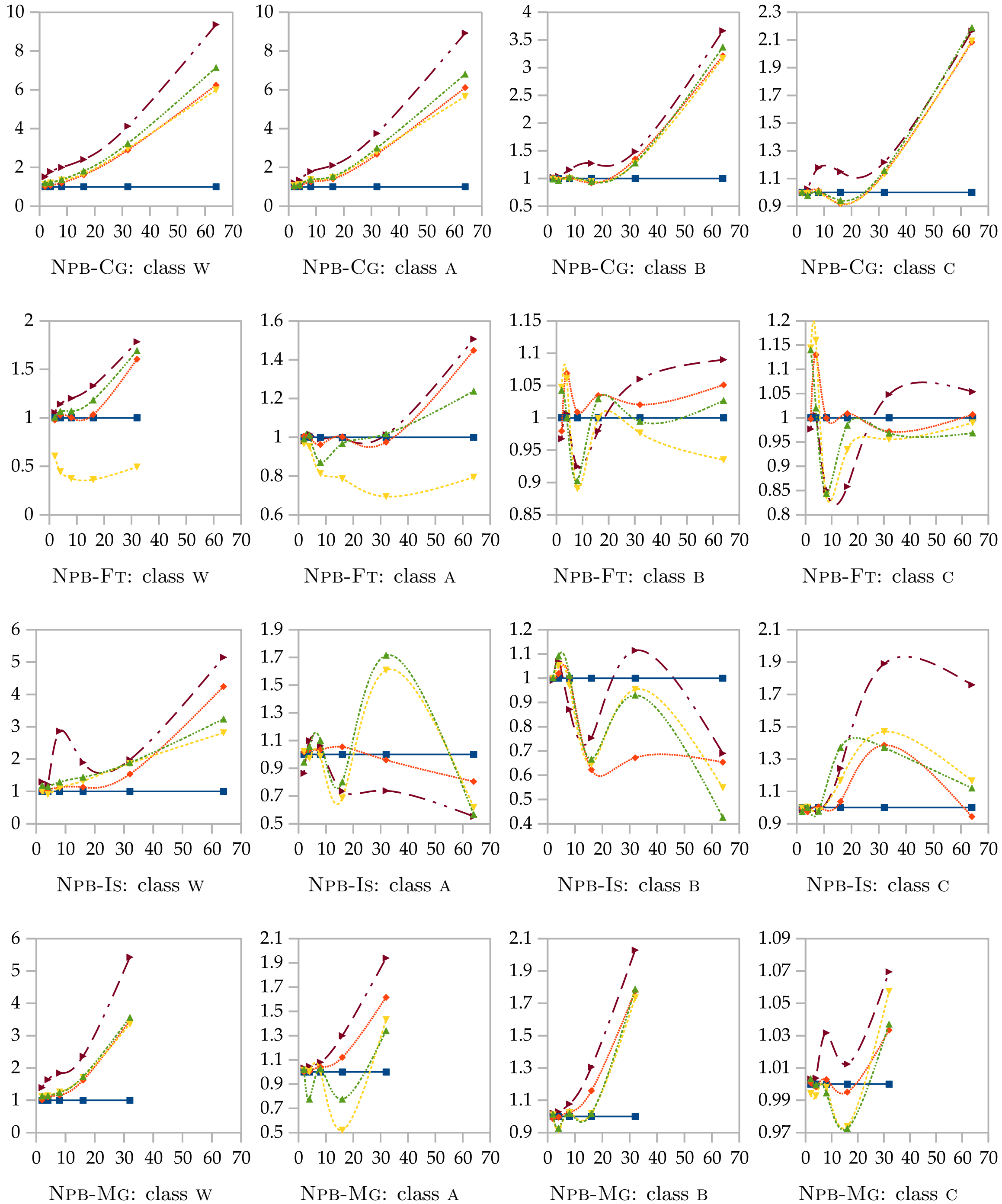

FIGURE 17. Experimental results for four NPB kernels: speedups, on the y-axis, of compiler-generated code optimized with eliminate, commandify, or both, and of reference code by Frumkin et al., relative to unoptimized compiler-generated code, as a function of the number of processes, on the x-axis. See Figure 16 for a legend.

of compiler-generated code. So far, including in this paper, we have focused our attention exclusively on compilation techniques for optimizing "algorithmic" aspects of compiler-generated code (i.e., minimizing the number of computation steps necessary to, for instance, check data constraints). Our experimental results in this paper show that we need to start considering memory too.

Another interesting piece of future work involves comparing our compilation technology for constraint automata, including the optimization techniques presented in this paper, with compilation technology for other coordination models and languages. One interesting candidate is Bip. In recent work [DJAB15], we already performed a theoretical study on the relation between (the formal semantics of) Reo and Bip. A natural next step in this line of work consists of a practical comparison of these models (including not only performance of their generated code, but also such software engineering qualities as programmability, maintainability, reusability, and so on).

References

[AdBO09] Krzysztof Apt, Frank de Boer, and Ernst-Rüdiger Olderog. While Programs. In *Verification of Sequential and Concurrent Programs*, Texts in Computer Science, chapter 3, pages 55–126. Springer, 3rd edition, 2009.

[AFF01] Giorgio Ausiello, Paolo Franciosa, and Daniele Frigioni. Directed Hypergraphs: Problems, Algorithmic Results, and a Novel Decremental Approach. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science (Proceedings of ICTCS 2001)*, volume 2202 of *LNCS*, pages 312–328. Springer, 2001.

[AKM08] Farhad Arbab, Natallia Kokash, and Sun Meng. Towards Using Reo for Compliance-Aware Business Process Modeling. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation (Proceedings of ISoLA 2008)*, volume 17 of *CCIS*, pages 108–123. Springer, 2008.

[Apt09a] Krzysztof Apt. Introduction. In *Principles of Constraint Programming*, chapter 1, pages 1–7. Cambridge University Press, 2nd edition, 2009.

[Apt09b] Krzysztof Apt. Some Complete Constraint Solvers. In *Principles of Constraint Programming*, chapter 4, pages 82–134. Cambridge University Press, 2nd edition, 2009.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[Arb05] Farhad Arbab. Abstract Behavior Types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1–3):3–52, 2005.

[Arb11] Farhad Arbab. Puff, The Magic Protocol. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems (Talcott Festschrift)*, volume 7000 of *LNCS*, pages 169–206. Springer, 2011.

[BBB+91] David Bailey, Eric Barszcz, John Barton, David Browning, Russell Carter, Leonardo Dagum, Rod Fatoohi, Paul Frederickson, Thomas Lasinski, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[BMFL02] Christian Bessière, Pedro Meseguer, Eugene Freuder, and Javier Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141(1–2):205–224, 2002.

[BS10] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, 2010.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[CKA10] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. A Unified Toolset for Business Process Model Formalization. In Barbora Buhnova and Jens Happe, editors, *Preproceedings of FESCA 2010*, pages 147–156, 2010.

[CPLA11] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76(8):681–710, 2011.

[DJAB15] Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, and Simon Bliudze. Relating BIP and Reo. In Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente, and Hugo-Torres Vieira, editors, *Proceedings of ICE 2015*, volume 189 of *EPTCS*, pages 3–20. CoRR, 2015.

[FSJY03] Michael Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Performance and Scalability of the NAS Parallel Benchmarks in Java. In Jack Dongarra, Yves Robert, David Walker, Josep Torrellas, and John Mellor-Crummey, editors, *Proceedings of IPDPS 2003*, pages 139–44. IEEE, 2003.

[GLPN93] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2–3):177–201, 1993.

[Hoa69] Tony Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JA15] Sung-Shik Jongmans and Farhad Arbab. Take Command of Your Constraints! In Tom Holvoet and Mirko Viroli, editors, *Coordination Models and Languages (Proceedings of COORDINATION 2015)*, volume 9037 of *LNCS*, pages 117–132. Springer, 2015.

[JA16] Sung-Shik Jongmans and Farhad Arbab. Global consensus through local synchronization: A formal basis for partially-distributed coordination. *Science of Computer Programming*, 115–116:199–224, 2016.

[JHA14] Sung-Shik Jongmans, Sean Halle, and Farhad Arbab. Automata-based Optimization of Interaction Protocols for Scalable Multicore Platforms. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages (Proceedings of COORDINATION 2014)*, volume 8459 of *LNCS*, pages 65–82. Springer, 2014.

[Jon16a] Sung-Shik Jongmans. *Automata-Theoretic Protocol Programming*. PhD thesis, Universiteit Leiden, 2016.

[Jon16b] Sung-Shik Jongmans. Automata-Theoretic Protocol Programming (With Proofs). Technical Report FM-1601, Centrum Wiskunde & Informatica, 2016.

[JSA15] Sung-Shik Jongmans, Francesco Santini, and Farhad Arbab. Partially-Distributed Coordination with Reo and Constraint Automata. *Service Oriented Computing and Applications*, 9(3):311–339, 2015.

[KA09] Natallia Kokash and Farhad Arbab. Formal Behavioral Modeling and Compliance Analysis for Service-Oriented Systems. In Frank de Boer, Marcello Bonsangue, and Eric Madelaine, editors, *Formal Methods for Components and Objects (Proceedings of FMCO 2008)*, volume 5751 of *LNCS*, pages 21–41. Springer, 2009.

[Kah62] Arthur Kahn. Topological Sorting in Large Networks. *Communications of the ACM*, 5(11):558–562, 1962.

[Knu97] Donald Knuth. Information Structures. In *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 232–465. Addison-Wesley, 3rd edition, 1997.

[KV08] Bernhard Korte and Jens Vygen. Spanning Trees and Arborescences. In *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*, chapter 6, pages 127–150. Springer, 4th edition, 2008.

[MAB11] Sun Meng, Farhad Arbab, and Christel Baier. Synthesis of Reo circuits from scenario-based interaction specifications. *Science of Computer Programming*, 76(8):651–680, 2011.

[PC13a] José Proença and Dave Clarke. Data Abstraction in Coordination Constraints. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing (Proceedings of FOCLASA 2013)*, volume 393 of *CCIS*, pages 159–173. Springer, 2013.

[PC13b] José Proença and Dave Clarke. Interactive Interaction Constraints. In Rocco de Nicola and Christine Julien, editors, *Coordination Models and Languages (Proceedings of COORDINATION 2013)*, volume 7890 of *LNCS*, pages 211–225. Springer, 2013.

[Rau10] Wolfgang Rautenberg. First-Order Logic. In *A Concise Introduction to Mathematical Logic*, Universitext, chapter 2, pages 41–90. Springer, 3rd edition, 2010.

[Rei85] Wolfgang Reisig. Introductory Examples and Basic Definitions. In *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*, chapter 1, pages 3–16. Springer, 1985.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[Woe92] Gerhard Woeginger. The complexity of finding arborescences in hypergraphs. *Information Processing Letters*, 44(3):161–164, 1992.

Appendix A. Proofs

Proof of Lemma 3.5. If p has no determinant in φ, we have $\mathsf{exists}_p(\varphi) = \exists p.\varphi$, and we are done (because $\equiv$ is reflexive).

Therefore, suppose that p has a determinant in φ, and let t denote the least such determinant under $<_{\mathbb{T}\text{ERM}}$ such that $\mathsf{exists}_p(\varphi) = \varphi[t/p]$. By the grammar of data constraints, φ must be of the form $\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k)$. Thus, we must show $\exists p.\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k) \equiv (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[t/p]$. To show this, without loss of generality, we assume $p, p_1, p_2, \ldots, p_l$ are all distinct (otherwise we can simply eliminate the quantifier of every duplicate variable). By the usual definitions of logical equivalence and entailment, we must show that $\sigma \models \exists p.\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k)$ implies $\sigma \models (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[t/p]$, for all σ, and vice versa.

Suppose $\sigma \models \exists p.\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k)$. By the usual semantics of $\exists$, this implies $\sigma \models (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[d/p]$ for some datum d. Because p is not bound by another $\exists$, we can expand also the other existential quantifications, and distribute the resulting substitutions over the conjunction, to get $\sigma \models \ell_i[d/p][d_1/p_1]\cdots[d_l/p_l]$ for every ℓ_i. Now, because t is a determinant of p, a literal $t = p$ (or, symmetrically, $p = t$) must exist among the ℓ_i literals. So, for that literal, we have $\sigma \models t[d/p][d_1/p_1]\cdots[d_l/p_l] = d$. A literal $t_1 = t_2$ holds under σ iff the evaluation of t_1 equals the evaluation of t_2. Hence, we know that the evaluation of $t[d/p][d_1/p_1]\cdots[d_l/p_l]$ equals d. From this, combined with the previous result $\sigma \models (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[d/p]$, we can establish $\sigma \models (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[t/p]$.

In the opposite direction, suppose $\sigma \models (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[t/p]$. We know that $\mathsf{eval}_\sigma(t) = d$ for some d, as before. In other words, there exists a d (namely $\mathsf{eval}_\sigma(t)$) such that $\sigma \models (\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k))[d/p]$. By the usual semantics of $\exists$, this implies $\sigma \models \exists p.\exists p_1.\cdots\exists p_l.(\ell_1 \wedge \cdots \wedge \ell_k)$.

A full, detailed proof appears as the proof of Lemma 16 in [Jon16b, Appendix D.3]. □

Proof of Theorem 3.6. Follows from Proposition 2.15 and Lemma 3.5.

A full, detailed proof appears as the proof of Theorem 14 in [Jon16b, Appendix D.3]. □

Proof of Theorem 3.8. Reasoning toward a contradiction, suppose that p still occurs in a data constraint φ in $\boldsymbol{a} \ominus p$. By the definition of $\ominus$, we have $\varphi = \mathsf{exists}_p(\varphi')$ for a data constraint φ' in $\boldsymbol{a}$. Because exists does not introduce new variables in data constraints, p must have occurred already in φ'. Because p is an ever-determined port of $\boldsymbol{a}$ by the premise of this theorem, by the definition of Edp, we know that p has a determinant t in φ'. Consequently, $\mathsf{exists}_p(\varphi') = \varphi'[t/p]$. Also, from the fact that p has a determinant in φ', we can derive that p is not bound by any of the existential quantifications inside φ'. Hence, p does not occur in $\varphi'[t/p]$. But then, p does not occur in $\mathsf{exists}_p(\varphi')$ either. Therefore, p does not occur in φ, which contradicts our intial assumption. Hence, p does not occur in any data constraint in $\boldsymbol{a} \ominus p$, which is the result stated in the consequence of this theorem.

A full, detailed proof appears as the proof of Theorem 15 in [Jon16b, Appendix D.3]. □

Proof of Theorem 4.13. To show the correctness of Algorithm 1 (henceforth "the algorithm"), we need to show that if its **require**ments are satisfied, upon termination, it **ensure**s both:

$$\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_i\}$$

and

$$\left[\begin{array}{l} \quad\quad \sigma \models \ell_1 \wedge \cdots \wedge \ell_{n+m} \ \textbf{implies} \\ \vdash_{\text{tot}} \{\bigwedge\{x = \sigma(x) \mid x \in X\}\} \\ \quad\quad \pi \\ \quad\quad \{\bigwedge\{x = \sigma(x) \mid x \in X \cup \{x_1, \ldots, x_n\}\}\} \end{array}\right] \textbf{ for all } \sigma$$

We call the former *soundness* and the latter *completeness* and prove their truth separately.

Soundness: We start by arguing that $\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_i\}$ holds after every iteration of the first loop. For $1 \leq i \leq n$, after doing an assignment $x_i := t_i$ in a data state σ, literal $\ell_i = x_i = t_i$ holds in σ if all variables in t_i have a non-`nil` value. (Otherwise, t_i evaluates to `nil`, which the definition of $\models$ forbids.)

Reasoning toward a contradiction, suppose that some variable y in t_i has a `nil` value. Then, because no assignment assigns `nil`, no $y := t$ assignment has occurred previously. But because $y \in \mathsf{Variabl}(t_i)$, either $\big[$a literal $y = t \in L$ exists that precedes $x_i = t_i\big]$ or $y \in X$ (by the **require**ments of the algorithm). In the former case, a $y := t$ assignment must have occurred previously, such that y in fact has a non-`nil` value (namely, the evaluation of t). In the latter case, by the precondition of the triple we are proving, we know that $\sigma \models y = y$ holds. By the definition of $\models$, this means that y has a non-`nil` value.

Thus, $\ell_i = x_i = t_i$ holds in σ after its update with $x_i := t_i$. By the precondition of the triple, we know that $x = x$ held for all $x \in X$ before updating σ. Additionally, suppose that the preceding literals $x_j = t_j$ (for $1 \leq j < i$) held before updating σ. Each of those literals can have become false only if the update overwrote an x or an x_j. In that case, $x_i \in X \cup \{x_1, \ldots, x_{i-1}\}$. But then, the algorithm did not translate $x_i = t_i$ to an assignment in the first place but to a failure statement $x_i = t_i$ `->` `skip`. If execution of this statement successfully terminates, obviously $x_i = t_i$ holds, and because it leaves σ unchanged, all preceding literals remain true. Note that the $\vdash_{\text{part}}$ proof rule for failure statements allows us to *assume* that the guard holds; we do not need to *establish* this yet (cf. completeness below, where we use $\vdash_{\text{tot}}$).

We can inductively repeat the reasoning in the previous paragraphs for all $1 \leq i \leq n$ to conclude that $\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_i\}$ holds after the first loop. The failure statements added in the second loop leave state σ unchanged, meaning that literals that held before executing those statements in σ remain true. Thus, if those statements successfully terminate,

$$\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_{n+m}\}$$

holds.

Completeness: Assume that $\sigma' \models \ell_1 \wedge \cdots \wedge \ell_{n+m}$ for some σ'. We start by arguing that $\vdash_{\text{tot}} \{\bigwedge\{x = \sigma'(x) \mid x \in X\}\}\ \pi\ \{\bigwedge\{x = \sigma'(x) \mid x \in X \cup \{x_1, \ldots, x_i\}\}\}$ holds after every iteration of the first loop. This means that the data state σ after executing π (starting from a data state where $\bigwedge\{x = \sigma'(x) \mid x \in X\}$ holds) maps every x_j (for $1 \leq j \leq i$) to the same value as σ' (i.e., $\sigma(x_j) = \sigma'(x_j)$). Let $1 \leq i \leq n$.

If $x_i \notin X \cup \{x_1, \ldots, x_{i-1}\}$, we know that $\ell_i = x_i \mathtt{=} t_i$ holds in σ after its update with $x_i \mathtt{:=} t_i$ (see soundness above). By our initial assumption, we also know that $\ell_i = x_i \mathtt{=} t_i$ holds in σ'. Thus, by the definition of $\models$, we conclude $\sigma(x_i) = \mathsf{eval}_\sigma(t_i)$ and $\sigma'(x_i) = \mathsf{eval}_{\sigma'}(t_i)$. Now, because a $y \mathtt{=} t$ literal precedes $x_i \mathtt{=} t_i$ for all $y \in \mathsf{Variabl}(t_i)$ (see soundness above), σ maps every such a y to the same value as σ' (i.e., $y = x_j$ for some $1 \leq j < i$). Consequently, $\mathsf{eval}_\sigma(t_i) = \mathsf{eval}_{\sigma'}(t_i)$. Combining this with the previous intermediate result, the following equation holds: $\sigma(x_i) = \mathsf{eval}_\sigma(t_i) = \mathsf{eval}_{\sigma'}(t_i) = \sigma'(x_i)$. Thus, $x_i \mathtt{=} \sigma'(x_i)$ holds in σ. As before (see soundness above), we can also establish that, for $x_j \in X \cup \{x_1, \ldots, x_{i-1}\}$, updating σ with $x_i \mathtt{:=} t_i$ does not make $x_j \mathtt{=} \sigma'(x_j)$ literals that held already before this update false. Thus, $\bigwedge\{x \mathtt{=} \sigma'(x) \mid x \in X \cup \{x_1, \ldots, x_i\}\}$ holds in σ.

If $x_i \in X \cup \{x_1, \ldots, x_{i-1}\}$, we can immediately conclude that $x_j \mathtt{=} \sigma'(x_j)$ held in σ for all $x_j \in X \cup \{x_1, \ldots, x_{i-1}\}$ already before executing the failure statement $x_i \mathtt{=} t_i$ `-> skip` added by the algorithm. To prove that this failure statement also successfully terminates, the $\vdash_{\text{tot}}$ proof rule for failure statements dictates that we must establish—instead of assume (cf. soundness above)—that the guard $x_i \mathtt{=} t_i$ holds in σ. This follows from the fact that $x_i \mathtt{=} t_i$ holds in σ' by our initial assumption, and because σ and σ' map all variables in $\ell_i = x_i \mathtt{=} t_i$ to the same values. To prove the latter, we can use a similar argument involving the precedence relation and its linearization as before (see soundness above).

We can inductively repeat the previous reasoning for all $1 \leq i \leq n$ to conclude that $\vdash_{\text{tot}} \{\bigwedge\{x \mathtt{=} \sigma'(x) \mid x \in X\}\}\ \pi\ \{\bigwedge\{x \mathtt{=} \sigma'(x) \mid x \in X \cup \{x_1, \ldots, x_n\}\}\}$ holds after the first loop. The failure statements added in the second loop leave σ unchanged, meaning that the $x_j \mathtt{=} \sigma'(x_j)$ literals that held already before executing those statements in σ, for $x_j \in X \cup \{x_1, \ldots, x_n\}$, remain true. In order to prove the successful termination of those failure statements, we can use a similar argument as for the failure statements added in the first loop: by our initial assumption, $\sigma' \models \ell_i$ for all $n + 1 \leq j \leq n + m$, and σ and σ' still map the same variables to the same values. Thus, $\vdash_{\text{tot}} \{\bigwedge\{x \mathtt{=} \sigma'(x) \mid x \in X\}\}\ \pi\ \{\bigwedge\{x \mathtt{=} \sigma'(x) \mid x \in X \cup \{x_1, \ldots, x_{n+m}\}\}\}$ holds also after the second loop.

A full, detailed proof appears as the proof of Theorem 18 in [Jon16b, Appendix D.4]. □

Proof of Theorem 4.17. Recall that the rules in Definition 4.12 of $\sqsubseteq$ (and, therefore, also the rules in Definition 4.14) induce precedence relations for which all **require**ments of Algorithm 1 (henceforth: "the algorithm") hold, except that those precedence relations do not necessarily denote strict partial orders. What we need to show here, then, is that $\sqsubset_\varphi^X$ is both a strict partial order and a "large enough" subset of $\sqsubseteq_\varphi^X$ to satisfy the algorithm's **require**ments. The theorem subsequently follows, as $<_\varphi^X$ is just the linearization of $\sqsubset_\varphi^X$.

The fact that $\sqsubset_\varphi^X$ is a strict partial order follows from $\lhd_\varphi^X$ forming an arborescence.

To show $\sqsubset_\varphi^X \subseteq \sqsubseteq_\varphi^X$, we need to consider the three rules in Definition 4.16 of $\sqsubset$. *First*, take any pair (ℓ, ℓ') such that $\ell \sqsubset_\varphi^X \ell'$ by Rule 4.29. Then, by the premise of that rule, $\{\ell_1, \ldots, \ell_k\} \lhd_\varphi^X \ell'$ such that $\ell = \ell_i$ for some $1 \leq i \leq k$. Because $\lhd_\varphi^X \subseteq \blacktriangleleft_\varphi^X$ (because the former is an arborescence of the latter), the premises of the rules in Definition 4.15 of $\blacktriangleleft$, subsequently guarantee after some manipulation that $\ell = \ell_i = x \mathtt{=} t$ for some x and t. Moreover, $x \in \mathsf{Variabl}(\ell')$. By Rule 4.20, we subsequently conclude that $\ell \sqsubseteq_\varphi^X \ell'$ holds.

Second, Rule 4.30 is identical to Rule 4.21, so any pair (ℓ, ℓ') in $\sqsubset_\varphi^X$ induced by the former is also induced in $\sqsubseteq_\varphi^X$ by the latter. *Third*, by induction, we can show the same result for pairs (ℓ, ℓ') such that $\ell \sqsubset_\varphi^X \ell'$ by Rule 4.31. Thus, $\sqsubset_\varphi^X \subseteq \sqsubseteq_\varphi^X$.

Finally, we must show that $\sqsubset_\varphi^X$ is "large enough" for it to satisfy the precondition of the algorithm. Informally, this means that arborescences do not exclude B-arcs in the B-graph that actually represent essential dependencies: for every free variable y that a literal $\ell \in L$ depends on, $\sqsubset_\varphi^X$ must contain at least one pair $(y = t, \ell)$ (for some t). To see that this holds, note that every B-arc entering a literal ℓ represents a complete set of dependencies of ℓ. If ℓ has multiple incoming B-arcs, this simply means that several ways exist to resolve ℓ's dependencies. In principle, however, keeping one of those options suffices for our purpose. Therefore, the single incoming B-arc that ℓ has in an arborescence represents enough dependencies of ℓ.

A full, detailed proof appears as the proof of Theorem 19 in [Jon16b, Appendix D.4]. □

Proof of Lemma 4.20. Follows from Theorems 4.13 and 4.17 and Definition 4.18.

A full, detailed proof appears as the proof of Lemma 18 in [Jon16b, Appendix D.4]. □

Proof of Theorem 4.21. Follows from Proposition 2.15 and Lemma 4.20.

A full, detailed proof appears as the proof of Theorem 20 in [Jon16b, Appendix D.4]. □

Proof of Theorem 4.23. To prove this theorem, by Definition 4.19 of $(\!|\cdot|\!)$, we need to show that for every data constraint φ in $\boldsymbol{a}$, the pair (φ, X) for $X = \mathsf{Free}(\varphi) \cap (P^{\text{in}} \cup {}^\bullet M)$ satisfies the four conditions in Definition 4.18 of comm. The first two conditions always hold. The third condition follows from $\clubsuit\, \boldsymbol{a}$: by Definition 4.22 of $\clubsuit$, every data constraint in $\boldsymbol{a}$ is arborescent. Finally, the fourth condition follows from set theory.

A full, detailed proof appears as the proof of Theorem 21 in [Jon16b, Appendix D.4]. □

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany