Efficient Coalgebraic Partition Refinement*†

Ulrich Dorsch[1], Stefan Milius[2], Lutz Schröder[3], and Thorsten Wißmann[4]

1 **Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany**
`Ulrich.Dorsch@fau.de`
2 **Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany**
`mail@stefan-milius.eu`, `http://orcid.org/0000-0002-2021-1644`
3 **Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany**
`Lutz.Schroeder@fau.de`
4 **Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany**
`Thorsten.Wissmann@fau.de`

Abstract

We present a generic partition refinement algorithm that quotients coalgebraic systems by behavioural equivalence, an important task in reactive verification; coalgebraic generality implies in particular that we cover not only classical relational systems but also various forms of weighted systems. Under assumptions on the type functor that allow representing its finite coalgebras in terms of nodes and edges, our algorithm runs in time $\mathcal{O}(m \cdot \log n)$ where n and m are the numbers of nodes and edges, respectively. Instances of our generic algorithm thus match the runtime of the best known algorithms for unlabelled transition systems, Markov chains, and deterministic automata (with fixed alphabets), and improve the best known algorithms for Segala systems.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases coalgebra, markov chains, partition refinement, transition systems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.32

1 Introduction

Minimization under bisimilarity is the task of identifying all states in a reactive system that exhibit the same behaviour. Minimization appears as a subtask in state space reduction (e.g. [5]) or non-interference checking [34]. The notion of bisimulation was first defined for relational systems [33, 24, 26]; it was later extended to other system types including probabilistic systems [23, 9] and weighted automata [6]. In fact, the importance of minimization under bisimilarity appears to increase with the complexity of the underlying system type. E.g., while in LTL model checking, minimization drastically reduces the state space but, depending on the application, does not necessarily lead to a speedup in the overall balance [11], in probabilistic model checking, minimization under strong bisimilarity does lead to substantial efficiency gains [19].

The algorithmics of minimization, often referred to as *partition refinement* or *lumping*, has received a fair amount of attention. Since bisimilarity is a greatest fixpoint, it is more or less immediate that it can be calculated in polynomial time by approximating this fixpoint from above following Kleene's fixpoint theorem. In the relational setting, Kanellakis and

* Full version with all proof details available at `http://arxiv.org/abs/1705.08362`.
† This work forms part of the DFG-funded project COAX (MI 717/5-1 and SCHR 1118/12-1).

Smolka [18] introduced an algorithm that in fact runs in time $\mathcal{O}(nm)$ where n is the number of nodes and m is the number of transitions. An even more efficient algorithm running in time $\mathcal{O}(m \log n)$ was later described by Paige and Tarjan [25]; this bound holds even if the number of action labels is not fixed [31]. Current algorithms typically apply further optimizations to the Paige-Tarjan algorithm, thus achieving better average-case behaviour but the same worst-case behaviour [10]. Probabilistic minimization has undergone a similarly dynamic development [3, 7, 36], and the best algorithms for minimization of Markov chains now have the same $\mathcal{O}(m \log n)$ run time as the relational Paige-Tarjan algorithm [15, 8, 32]. Using ideas from abstract interpretation, Ranzato and Tapparo [27] have developed a relational partition refinement algorithm that is generic over *notions of process equivalence*. As instances, they recover the classical Paige-Tarjan algorithm for strong bisimilarity and an algorithm for stuttering equivalence, and obtain new algorithms for simulation equivalence and for a new process equivalence.

In this paper we follow an orthogonal approach and provide a generic partition refinement algorithm that can be instantiated for many different *types* of systems (e.g. nondeterministic, probabilistic, weighted). We achieve this by methods of *universal coalgebra* [28]. That is, we encapsulate transition types of systems as endofunctors on sets (or a more general category), and model systems as coalgebras for a given type functor.

Our work proceeds on several levels of abstraction. On the most abstract level (Section 3) we work with coalgebras for a monomorphism-preserving endofunctor on a category with image factorizations. Here we present a quite general category-theoretic partition refinement algorithm, and we prove its correctness. The algorithm is parametrized over a select routine that determines which observations are used to split blocks of states; the corner case where all available observations are used yields known coalgebraic final chain algorithms, e.g. [22].

Next, we present an optimized version of our algorithm (Section 4) that needs more restrictive conditions to ensure correctness; specifically, we need to assume that the type endofunctor satisfies a condition we call *zippability* in order to allow for incremental computation of partitions. This property holds, e.g., for all polynomial endofunctors on sets and for the type functors of labelled and weighted transition systems, but not for all endofunctors of interest. In particular, zippable functors fail to be closed under composition, as exemplified by the double covariant powerset functor $\mathcal{PP}$ on sets, for which the optimized algorithm is in fact incorrect. However, it turns out that obstacles of this type can be removed by moving to multi-sorted coalgebras [29], so we do eventually obtain an efficient partition refinement algorithm for coalgebras of composite functors, including $\mathcal{PP}$-coalgebras as well as (probabilistic) Segala systems [30].

Finally, we analyse the run time of our algorithm (Section 5). To this end, we make our algorithm parametric in an abstract *refinement interface* to the type functor, which encapsulates the incremental calculation of partitions in the optimized version of the algorithm. We show that if the interface operations can be implemented in linear time, then the algorithm runs in time $\mathcal{O}(m \log n)$, where n is the number of states and m the number of 'edges' in a syntactic encoding of the input coalgebra. We thus recover the most efficient known algorithms for transition systems (Paige and Tarjan [25]) and for weighted systems (Valmari and Franceschinis [32]). Using the mentioned modularity results, we also obtain an $\mathcal{O}((m + n) \log(m + n))$ algorithm for Segala systems, to our knowledge a new result (more precisely, we improve an earlier bound established by Baier, Engelen, and Majster-Cederbaum [3], roughly speaking by letting only non-zero probabilistic edges enter into the time bound). The algorithm and its analysis apply also to generalized polynomial functors on sets; in particular, for the functor $2 \times (-)^A$, which models deterministic finite automata, we obtain the same complexity $\mathcal{O}(n \log n)$ as for Hopcroft's classical minimization algorithm for a fixed alphabet A [14, 21, 12].

2 Preliminaries

We assume that readers are familiar with basic category theory [2]. For the convenience of the reader we recall some concepts that are central for the categorical version of the algorithm.

▶ **Notation 2.1.** The terminal object is denoted by 1, with unique arrows $! : A \to 1$, and the product of objects A, B by $A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$. Given $f : D \to A$ and $g : D \to B$, the morphism induced by the universal property of the product $A \times B$ is denoted by $\langle f, g\rangle : D \to A \times B$. The *kernel* $\mathsf{ker}\, f$ of a morphism f is the pullback of f along itself. We write $\twoheadrightarrow$ for regular epimorphisms (i.e. coequalizers), and $\rightarrowtail$ for monomorphisms.

Kernels allow us to talk about equivalence relations in a category. In particular in Set, there is a bijection between kernels and equivalence relations in the usual sense: For a map $f : D \to A$, $\mathsf{ker}\, f = \{(x, y) \mid fx = fy\}$ is the equivalence relation induced by f. Generally, relations (i.e. jointly monic spans of morphisms) in a category are ordered by inclusion in the obvious way. We say that a kernel K is *finer* than a kernel K' if K is included in K'. We use intersection $\cap$ and union $\cup$ of kernels for meets and joins in the inclusion ordering on *relations* (not equivalence relations or kernels); in this notation, $\mathsf{ker}\langle f, g\rangle = \mathsf{ker}\, f \cap \mathsf{ker}\, g$. In Set, a map $f : D \to A$ factors through the partition $D/\mathsf{ker}\, f$ induced by its kernel, via the map $[-]_f : D \twoheadrightarrow D/\mathsf{ker}\, f$ taking equivalence classes

$$[x]_f := \{x' \in D \mid fx = fx'\} = \{x' \in D \mid (x, x') \in \mathsf{ker}\, f\}.$$

Well-definedness of functions on $D/\mathsf{ker}\, f$ is determined precisely by the universal property of $[-]_f$ as a coequalizer of $\mathsf{ker}\, f \rightrightarrows D$. In particular, f induces an injection $D/\mathsf{ker}\, f \rightarrowtail A$; together with $[-]_f$, this is the factorization of f into a regular epimorphism and a monomorphism. Categorically, this is captured by the following assumptions.

▶ **Assumption 2.2.** We assume throughout that $\mathcal{C}$ is a finitely complete category that has coequalizers and *image factorizations*, i.e. every morphism f has a factorization $f = m \cdot e$ as a regular epimorphism e followed by a monomorphism m. We call the codomain of e the *image* of f, and denote it by $D/\mathsf{ker}\, f$. Regular epis in $\mathcal{C}$ are closed under composition and right cancellation [2, Prop. 14.14].

▶ **Examples 2.3.** Examples of categories satisfying Assumption 2.2 abound. In particular, every regular category with coequalizers satisfies our assumptions. The category Set of sets and maps is, of course, regular. Every topos is regular, and so is every finitary variety, i.e. a category of algebras for a finitary signature satisfying given equational axioms (e.g. monoids, groups, vector spaces etc.). Posets and topological spaces fail to be regular but still satisfy our assumptions. If $\mathcal{C}$ is regular, so is the functor category $\mathcal{C}^{\mathcal{E}}$ for any category $\mathcal{E}$.

For a set $\mathcal{S}$ of sorts, the category $\mathsf{Set}^{\mathcal{S}}$ of $\mathcal{S}$-sorted sets has $\mathcal{S}$-tuples of sets as objects. We write $\chi_S : X \to 2$ for the characteristic function of a subset $S \subseteq X$, i.e. for $x \in X$ we have $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. We will also use a three-valued version:

▶ **Definition 2.4.** For $S \subseteq C \subseteq X$, define $\chi_S^C : X \to 3$ by $C \not\ni x \mapsto 0, C \setminus S \ni x \mapsto 1$, and $S \ni x \mapsto 2$. (This is essentially $\langle \chi_S, \chi_C\rangle : X \to 4$ without the impossible case $x \in S \setminus C$.)

Coalgebras. We briefly recall basic notions from coalgebra. For introductory texts, see [28, 17, 1, 16]. Given an endofunctor $H : \mathcal{C} \to \mathcal{C}$, a *coalgebra* is pair (C, c) where C is an object of $\mathcal{C}$ called the *carrier* and thought of as an object of *states*, and $c : C \to HC$ is a morphism called the *structure* of the coalgebra. Our leading examples are the following.

▶ **Example 2.5.**

1. Labelled transition systems with labels from a set A are coalgebras for the functor $HX = \mathcal{P}(A \times X)$ (and unlabelled transition systems are simply coalgebras for $\mathcal{P}$). Explicitly, a coalgebra $c : C \to HC$ assigns to each state x a set $c(x) \in \mathcal{P}(A \times X)$, and this represents the transition structure at x: x has an a-transition to y iff $(a, y) \in c(x)$.
2. Weighted transition systems with weights drawn from a commutative monoid are modelled as coalgebras as follows. For the given commutative monoid $(M, +, 0)$, we consider the *monoid-valued* functor $M^{(-)}$ on Set given for any map $h : X \to Y$ by

$$M^{(X)} = \{f : X \to M \mid f(x) \neq 0 \text{ for finitely many } x\}, \qquad M^{(h)}(f)(y) = \textstyle\sum_{hx=y} f(x).$$

   M-weighted transition systems are in bijective correspondence with coalgebras for $M^{(-)}$ [13] (and for M-weighted labelled transition systems one takes $(M^{(-)})^A$).
3. Probabilistic transition systems are modelled coalgebraically using the distribution functor $\mathcal{D}$. This is the subfunctor $\mathcal{D}X \subseteq \mathbb{R}_{\geq 0}^{(X)}$, where $\mathbb{R}_{\geq 0}$ is the monoid of addition on the non-negative reals, given by $\mathcal{D}X = \{f \in \mathbb{R}_{\geq 0}^{(X)} \mid \sum_{x \in X} f(x) = 1\}$.
4. The finite powerset functor $\mathcal{P}_\mathrm{f}$ is a monoid-valued functor for the Boolean monoid $\mathbb{B} = (2, \vee, 0)$. The *bag functor* $\mathcal{B}_\mathrm{f}$, which assigns to a set X the set of bags (i.e. finite multisets) on X, is the monoid-valued functor for the additive monoid of natural numbers.
5. Simple (resp. general) Segala systems [30] strictly alternate between non-deterministic and probabilistic transitions; they can be modeled as coalgebras for the set functor $\mathcal{P}_\mathrm{f}(A \times \mathcal{D}(-))$ (resp. $\mathcal{P}_\mathrm{f}\mathcal{D}(A \times -)$).

A *coalgebra morphism* from a coalgebra (C, c) to a coalgebra (D, d) is a morphism $h : C \to D$ such that $d \cdot h = Hh \cdot c$; intuitively, coalgebra morphisms preserve observable behaviour. Coalgebras and their morphisms form a category $\mathsf{Coalg}(H)$. The forgetful functor $\mathsf{Coalg}(H) \to \mathcal{C}$ creates all colimits, so $\mathsf{Coalg}(H)$ has all colimits that $\mathcal{C}$ has.

A *subcoalgebra* of a coalgebra (C, c) is represented by a coalgebra morphism $m : (C, c) \to (D, d)$ such that m is a monomorphism in $\mathcal{C}$. Likewise, a *quotient* of a coalgebra (C, c) is represented by a coalgebra morphism $q : (C, c) \to (D, d)$ carried by a regular epimorphism q of $\mathcal{C}$. If H preserves monomorphisms, then the image factorization structure on $\mathcal{C}$ lifts to coalgebras.

▶ **Definition 2.6.** A coalgebra is *simple* if it does not have any non-trivial quotients.

Equivalently, a coalgebra (C, c) is simple if every coalgebra morphism with domain (C, c) is carried by a monomorphism. Intuitively, in a simple coalgebra all states exhibiting the same observable behaviour are already identified. This paper is concerned with the design of algorithms for computing *the* simple quotient of a given coalgebra:

▶ **Lemma 2.7.** *The simple quotient of a coalgebra is unique (up to isomorphism).*

Intuitively speaking, two elements (possibly in different coalgebras) are called behaviourally equivalent if they can be identified by coalgebra morphisms. Hence, the simple quotient of a coalgebra is its quotient modulo behavioural equivalence. In our main examples, this means that we minimize w.r.t. standard bisimilarity-type equivalences.

▶ **Example 2.8.** Behavioural equivalence instantiates to various notions of bisimilarity:

1. Park-Milner bisimilarity on labelled transition systems;
2. weighted bisimilarity on weighted transition systems [20, Proposition 2];
3. stochastic bisimilarity on probabilistic transition systems [20];
4. Segala bisimilarity on simple and general Segala systems [4, Theorem 4.2].

3 A Categorical Algorithm for Behavioural Equivalence

We proceed to describe a categorical partition refinement algorithm that computes the simple quotient of a given coalgebra under fairly general assumptions.

▶ **Assumption 3.1.** Assume that H is an endofunctor on $\mathcal{C}$ that preserves monomorphisms.

Note that mono preservation is w.l.o.g. for $\mathcal{C} = \mathsf{Set}$. Roughly, for a given coalgebra $\xi : X \to HX$ in Set, a *partition refinement algorithm* maintains a quotient $q : X \twoheadrightarrow X/Q$ that distinguishes some (but possibly not all) states with different behaviour, and in fact, initially q typically identifies everything. The algorithm repeats the following steps:

1. Gather new information on which states should become separated by using $X \xrightarrow{\xi} HX \xrightarrow{Hq} HX/Q$, i.e., by identifying equivalence classes under q that contain states whose behaviour is observed to differ under one more step of the transition structure ξ.
2. Use parts of this information to refine q and repeat until q does not change any more.

One of the core ideas of the Paige-Tarjan partition refinement algorithm [25] is to not use all information immediately in the second step. Recall that the algorithm maintains two partitions Y and Z of the state set X of the given transition system; the elements of Y are called *subblocks* and the elements of Z are called *compound blocks*. The partition Y is a refinement of the partition Z. The key to the time efficiency of the algorithm is to select in each iteration a subblock that is at most half of the size of the compound block it belongs to. At the present high level of generality (which in particular does not know about sizes of objects), we encapsulate the subblock selection in a routine select, assumed as a parameter to our algorithm:

▶ **Definition 3.2.** A select routine is an operation that receives a chain of two regular epis $X \xrightarrow{y} Y \xrightarrow{z} Z$ and returns some morphism $k : Y \to K$ into some object K. We call Y the *subblocks* and Z the *compound blocks*.

The idea is that the morphism k throws away some of the information provided by the refinement Y. For example, in the Paige-Tarjan algorithm it models the selection of one compound block to be split in two parts, which then induce the further refinement of Y.

▶ **Example 3.3.**

1. In the classical Paige-Tarjan algorithm [25], i.e., for $\mathcal{C} = \mathsf{Set}$, one wants to find a proper subblock that is at most half of the size of the compound block it sits in. So let $S \in Y$ such that $2 \cdot |y^{-1}[\{S\}]| \leq |(zy)^{-1}[\{z(S)\}]|$. Here, $z(S)$ is the compound block containing S. Then we let $\mathsf{select}(z, y)$ be $k : Y \to 3$ given by $k(x) = 2$ if $x = S$, else $k(x) = 1$ if $z(x) = z(S)$, and $k(x) = 0$ otherwise; i.e. $k = \chi^{[S]_z}_{\{S\}}$ (Theorem 2.4). If Y and Z are encoded as partitions of X, then S and $C := z(S)$ are subsets of X and $k \cdot y = \chi^{C}_{S}$. If there is no such $S \in Y$, then z is bijective, i.e., there is no compound block from Z that needs to be refined. In this case, k does not matter and we simply put $k = ! : Y \to 1$.
2. One obvious choice for k is to take the identity on Y, so that *all* of the information present in Y is used for further refinement. We will discuss this in Theorem 3.
3. Two other, trivial, choices are $k = ! : Y \to 1$ and $k = z$. Since both of these choices provide no extra information, this will leave the partitions unchanged, see Theorem 3.12.

Given a select routine, the most general form of our partition refinement works as follows.

▶ **Algorithm 3.4.** Given a coalgebra $\xi : X \to HX$, we successively refine equivalence relations Q and P on X, maintaining the invariant that P is finer than Q. In each step, we take into account new information on the behaviour of states, represented by a map

$q : X \to K$, and accumulate this information in a map $\bar{q} : X \to \bar{K}$. To facilitate the analysis, these variables are indexed over loop iterations in the description. Initial values are

$$Q_0 = X \times X \qquad q_0 = ! : X \to 1 = K_0 \qquad P_0 = \ker(X \xrightarrow{\xi} HX \xrightarrow{H!} H1).$$

We then iterate the following steps while $P_i \neq Q_i$, for $i \geq 0$:

1. $X/P_i \overset{k_{i+1}}{\twoheadrightarrow} K_{i+1} := \mathsf{select}\big(X \twoheadrightarrow X/P_i \twoheadrightarrow X/Q_i\big)$, using that X/P_i is finer than X/Q_i
2. $q_{i+1} := X \longrightarrow\!\!\!\!\!\rightarrow X/P_i \xrightarrow{k_{i+1}} K_{i+1}$, $\bar{q}_{i+1} := \langle \bar{q}_i, q_{i+1} \rangle : X \longrightarrow \bar{K}_i \times K_{i+1}$
3. $Q_{i+1} := \ker \bar{q}_{i+1} \quad (= \ker\langle \bar{q}_i, q_{i+1}\rangle = \ker \bar{q}_i \cap \ker q_{i+1})$
4. $P_{i+1} := \ker\big(X \xrightarrow{\xi} HX \xrightarrow{H\bar{q}_{i+1}} H \prod_{j \leq i+1} K_j \big)$

Upon termination, the algorithm returns $X/P_i = X/Q_i$ as the simple quotient of (X, ξ).

▶ **Notation 3.5.** For spans $R \rightrightarrows X$, we will denote the canonical quotient by $\kappa_R : X \twoheadrightarrow X/R$.

We proceed to prove correctness, i.e. that the algorithm really does return the simple quotient of (X, ξ). We fix the notation in Algorithm 3.4 throughout. Since $\bar{q}$ accumulates more information in every step, it is clear that P and Q are really being successively refined:

▶ **Lemma 3.6.** *For every i, P_{i+1} is finer than P_i, Q_{i+1} is finer than Q_i, and P_i is finer than Q_{i+1}.*

$$\begin{array}{ccccccccccc} Q_0 & \longleftarrow & Q_1 & \longleftarrow & Q_2 & \leftarrow\!\cdots\!\prec & Q_{i+1} & \longleftarrow & Q_{i+2} & \leftarrow\cdots \\ \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow & \\ P_0 & \longleftarrow & P_1 & \leftarrow\!\cdots\!\prec & P_i & \longleftarrow & P_{i+1} & \leftarrow\cdots & & \end{array} \tag{3.1}$$

If we suppress the termination on $P_i = Q_i$ for a moment, then the algorithm thus computes equivalence relations refining each other. At each step, select decides which part of the information present in P_i but not in Q_i should be used to refine Q_i to Q_{i+1}.

▶ **Proposition 3.7.** *There exist morphisms $\xi/Q_i : X/P_i \to H(X/Q_i)$ for $i \geq 0$ (necessarily unique) such that* (3.2) *commutes.*

$$\begin{array}{ccc} X & \xrightarrow{\kappa_{P_i}} & X/P_i \\ {\scriptstyle\xi}\downarrow & & \vdots\,{\scriptstyle \xi/Q_i} \\ HX & \xrightarrow{H\kappa_{Q_i}} & H(X/Q_i) \end{array} \tag{3.2}$$

Upon termination the morphism ξ/Q_i yields the structure of a quotient coalgebra of ξ:

▶ **Corollary 3.8.** *If $P_i = Q_i$ then X/Q_i carries a unique coalgebra structure forming a quotient of $\xi : X \to HX$.*

This means intuitively that all states that are merged by the algorithm are actually behaviourally equivalent. The following property captures the converse:

▶ **Lemma 3.9.** *Let $h : (X, \xi) \to (D, d)$ be a quotient of (X, ξ). Then $\ker h$ is finer than both P_i and Q_i, for all $i \geq 0$.*

▶ **Theorem 3.10** (Correctness)**.** *If $P_i = Q_i$, then $\xi/Q_i : X/Q_i \to H\, X/Q_i$ is a simple coalgebra.*

▶ Remark. Most classical partition refinement algorithms are parametrized by an initial partition $\kappa_{\mathcal{I}} : X \twoheadrightarrow X/\mathcal{I}$. We start with the trivial partition $! : X \to 1$ because a non-trivial initial partition might split equivalent behaviours and then would invalidate Theorem 3.9. To accomodate an initial partition $X/\mathcal{I}$ coalgebraically, replace (X, ξ) with the coalgebra $\langle \xi, \kappa_{\mathcal{I}} \rangle$ for the functor $H(-) \times X/\mathcal{I}$ – indeed, already P_0 will then be finer than $\mathcal{I}$.

We look in more detail at two corner cases of the algorithm, where the select routine retains all available information, respectively none:

▶ Remark. Recall that H induces the *final sequence*:

$$1 \xleftarrow{!} H1 \xleftarrow{H!} H^2 1 \xleftarrow{H^2!} \cdots \xleftarrow{H^{i-1}!} H^i 1 \xleftarrow{H^i!} H^{i+1}1 \xleftarrow{H^{i+1}!} \cdots$$

Every coalgebra $\xi : X \to HX$ then induces a *canonical cone* $\xi^{(i)} : X \to H^i 1$ on the final sequence, defined inductively by $\xi^{(0)} = !$, $\xi^{(i+1)} = H\xi^{(i)} \cdot \xi$. The objects $H^n 1$ may be thought of as domains of n-step behaviour for H-coalgebras. If $\mathcal{C} = \mathsf{Set}$ and X is finite, then states x and y are behaviourally equivalent iff $\xi^{(i)}(x) = \xi^{(i)}(y)$ for all $i < \omega$ [35].

The vertical inclusions in (3.1) reflect that only some and not necessarily all of the information present in the relation P_i (resp. the quotient X/P_i) is used for further refinement. If indeed everything is used, i.e., we have $k_{i+1} := \mathsf{id}_{X/P_i}$, then these inclusions become isomorphisms and then our algorithm simply computes the kernels of the morphisms in the canonical cone, i.e. $Q_i = \ker \xi^{(i)}$.

That is, when select retains all available information, then Algorithm 3.4 just becomes a standard final chain algorithm (e.g. [22]). The other extreme is the following:

▶ **Definition 3.11.** We say that select *discards all new information at* $i+1$ if k_{i+1} factors through the morphism $X/P_i \twoheadrightarrow X/Q_i$ witnessing that P_i is finer than Q_i, see Theorem 3.6.

▶ **Lemma 3.12.** *The algorithm fails to progress in the* $i+1$*-th iteration, i.e.* $Q_{i+1} = Q_i$*, iff* select *discards all new information at* $i+1$.

▶ **Corollary 3.13.** *If* $\mathcal{C}$ *is (concrete over)* $\mathsf{Set}^{\mathcal{S}}$ *and* select *never discards all new information, then Algorithm 3.4 terminates and computes the simple quotient of a given finite coalgebra.*

Indeed, Proposition 3.7 shows that we obtain a chain of successively finer quotients of X, and by Theorem 3.12 this chain must finally converge (i.e. $P_i = Q_i$ will hold).

4 Incremental Partition Refinement

In the most generic version of the partition refinement algorithm (Algorithm 3.4), the partitions are recomputed from scratch in every step: In Step 4 of the algorithm, $P_{i+1} = \ker(H\langle \bar{q}_i, q_{i+1}\rangle \cdot \xi)$ is computed from the information $\bar{q}_i$ accumulated so far and the new information q_{i+1}, but in general one cannot exploit that the kernel of $\bar{q}_i$ has already been computed. We now present a refinement of the algorithm in which the partitions are computed incrementally, i.e. P_{i+1} is computed from P_i and q_{i+1}. This requires the type functor H to be *zippable* (Theorem 4.1). The algorithm will be further refined in the next section.

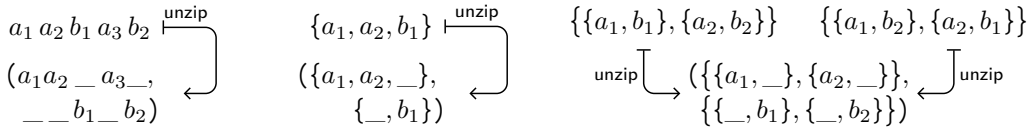
Note that in Step 3, Algorithm 3.4 computes a kernel $Q_{i+1} = \ker \bar{q}_{i+1} = \ker\langle \bar{q}_i, q_{i+1}\rangle$. In general, the kernel of a pair $\langle a, b\rangle : D \to A \times B$ is an intersection $\ker a \cap \ker b$. Hence, the partition for such a kernel can be computed in two steps:

1. Compute $D/\ker a$.
2. Refine every block in $D/\ker a$ with respect to $b : D \to B$.

Algorithm 3.4 can thus be implemented to keep track of the partition X/Q_i and then refine this partition by q_{i+1} in each iteration.

However, the same trick cannot be applied immediately to the computation of X/P_i, because of the functor H inside the computation of the kernel: $P_{i+1} = \ker(H\langle \bar{q}_i, q_{i+1}\rangle \cdot \xi)$. In the following, we will provide sufficient conditions for H, $a : D \to A$, $b : D \to B$ to satisfy

$$\ker H\langle a, b\rangle = \ker\langle Ha, Hb\rangle.$$



(a) $(-)^*$ is zippable. **(b)** $\mathcal{P}_\mathrm{f}$ is zippable. **(c)** $\mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}$ is not zippable.

Figure 1 Zippability of Set-Functors for sets $A = \{a_1, a_2, a_3\}$, $B = \{b_1, b_2\}$.

As soon as this holds for $a = \bar{q}_i, b = q_{i+1}$, we can optimize the algorithm by changing Step 4 to

$$P'_{i+1} := \mathsf{ker}\langle H\bar{q}_i \cdot \xi, Hq_{i+1} \cdot \xi\rangle. \tag{4.1}$$

▶ **Definition 4.1.** A functor H is *zippable* if the following morphism is a monomorphism:

$$\mathsf{unzip}_{H,A,B} : H(A + B) \xrightarrow{\langle H(A+!), H(!+B)\rangle} H(A + 1) \times H(1 + B)$$

Intuitively, if H is a functor on Set, we think of elements t of $H(A + B)$ as shallow terms with variables from $A + B$. Then zippability means that each t is uniquely determined by the two terms obtained by replacing A- and B-variables, respectively, by some placeholder $\_$, viz. the element of 1, as in the examples in Figure 1.

In the following, we work in the category $\mathcal{C} = \mathsf{Set}^{\mathcal{S}}$ of $\mathcal{S}$-sorted sets. However, most proofs are category-theoretic to clarify where sets are really needed and where the arguments are more generic.

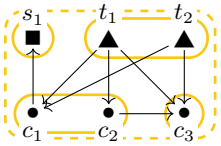
▶ **Example 4.2.** **1.** Constant functors $X \mapsto A$ are zippable: unzip is the diagonal $A \to A \times A$.
2. The identity functor is zippable since $A + B \xrightarrow{\langle A+!, !+B\rangle} (A + 1) \times (1 + B)$ is monic in $\mathsf{Set}^{\mathcal{S}}$.
3. From Lemma 4.3 it follows that every polynomial endofunctor is zippable.

▶ **Lemma 4.3.** *Zippable endofunctors are closed under products, coproducts and subfunctors.*

▶ **Lemma 4.4.** *If H has a componentwise monic natural transformation $H(X + Y) \rightarrowtail HX \times HY$, then H is zippable.*

▶ **Example 4.5.**
1. For every commutative monoid, the monoid-valued functor $M^{(-)}$ admits a natural isomorphism $M^{(X+Y)} \cong M^{(X)} \times M^{(Y)}$, and hence is zippable by Lemma 4.4.
2. As special cases of monoid-valued functors we obtain that the finite powerset functor $\mathcal{P}_\mathrm{f}$ and the bag functor $\mathcal{B}_\mathrm{f}$ are zippable.
3. The distribution functor $\mathcal{D}$ (see Example 2.5) is a subfunctor of the monoid-valued functor $M^{(-)}$ for M the additive monoid of real numbers, and hence is zippable by Item 1 and Lemma 4.3.
4. The previous examples together with the closure properties in Lemma 4.3 show that a number of functors of interest are zippable, e.g. $2 \times (-)^A$, $2 \times \mathcal{P}(-)^A$, $\mathcal{P}(A \times (-))$, $2 \times \big((-) + 1\big)^A$, and variants where $\mathcal{P}$ is replaced with $\mathcal{B}_\mathrm{f}$, $M^{(-)}$, or $\mathcal{D}$.

▶ **Example 4.6.** The finitary functor $\mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}$ fails to be zippable, as shown in Figure 1. First, this shows that zippable functors are not closed under quotients, since any finitary functor is a quotient of a polynomial, hence zippable, functor (recall that a Set-functor F is finitary if $FX = \bigcup\{Fi[Y] \mid i : Y \rightarrowtail X \text{ and } Y \text{ finite}\}$). Secondly, this shows that zippable functors are not closed under composition. One can extend the counterexample to a coalgebra to show that the optimization is incorrect for $\mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}$ and $\mathsf{select} = \chi^C_S$. We will remedy this later by making use of a second sort, i.e. by working in Set^2 (Theorem 4).



(a) Q_0, P_0 for $\bar{q}_0 = !$ **(b)** Q_1, P_1 for $\bar{q}_1 = \kappa_{P_0}$

Figure 2 Partitions of a coalgebra ξ for $H = \{\blacktriangle, \blacksquare, \bullet\} \times \mathcal{P}_\mathrm{f}(-)$. X/Q_i is indicated by dashed, X/P_i by solid lines.

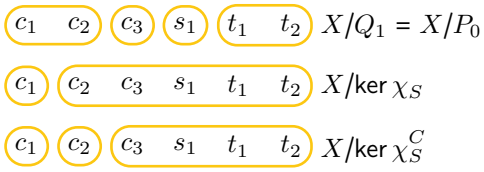


Figure 3 Grouping of elements when $S := \{c_1\}$ is chosen as the next subblock and $C := \{c_1, c_2\}$ as the compound block.

Additionally, we will need to enforce constraints on the select routine to arrive at the desired optimization (4.1). This is because in general, $\ker H\langle a, b\rangle$ differs from $\ker\langle Ha, Hb\rangle$ even for H zippable; e.g. for $H = \mathcal{P}$ and for π_1, π_2 denoting binary product projections, $\langle \mathcal{P}\pi_1, \mathcal{P}\pi_2\rangle$ in general fails to be injective although $\mathcal{P}\langle \pi_1, \pi_2\rangle = \mathcal{P}\mathsf{id} = \mathsf{id}$.

The next example illustrates this issue, and a related one: One might be tempted to implement splitting by a subblock S by $q_i = \chi_S$. While this approach is sufficient for systems with real-valued weights [32], it may in general let $\ker(H\langle \bar{q}_i, q_{i+1}\rangle \cdot \xi)$ and $\ker\langle H\bar{q}_i \cdot \xi, Hq_{i+1} \cdot \xi\rangle$ differ even for zippable H, thus rendering the algorithm incomplete:

▶ **Example 4.7.** Consider the coalgebra $\xi : X \to HX$ for the zippable functor $H = \{\blacktriangle, \blacksquare, \bullet\} \times \mathcal{P}_\mathrm{f}(-)$ illustrated in Figure 2 (essentially a Kripke model). The initial partition X/P_0 splits by shape and by $\mathcal{P}_\mathrm{f}!$, i.e. states with and without successors are split (Figure 2a). Now, suppose that select returns $k_1 := \mathsf{id}_{X/P_0}$, i.e. retains all information (cf. Remark 3), so that $Q_1 = P_0$ and P_1 puts c_1 and c_2 into different blocks (Figure 2b). We now analyse the next partition that arises when we split w.r.t. the subblock $S = \{c_1\}$ but not w.r.t. the rest $C \setminus S$ of the compound block $C = \{c_1, c_2\}$; in other words, we take $k_2 := \chi_{\{\{c_1\}\}} : X/P_1 \to 2$, making $q_2 = \chi_{\{c_1\}} : X \to 2$. Then, $H\langle \bar{q}_1, q_2\rangle \cdot \xi$ splits t_1 from t_2, because t_1 has a successor c_2 with $\bar{q}_1(c_2) = \{c_1, c_2\}$ and $q_2(c_2) = 0$ whereas t_2 has no such successor. However, t_1, t_2 fail to be split by $\langle H\bar{q}_1, Hq_2\rangle \cdot \xi$ because their successors do not differ when looking at successor blocks in X/Q_1 and $X/\ker \chi_S$ separately: both have $\{c_1, c_2\}$ and $\{c_3\}$ as successor blocks in X/Q_1 and $\{c_1\}$ and $X \setminus \{c_1\}$ as successors in $X/\ker \chi_S$. Formally:

$$H\bar{q}_1 \cdot \xi(t_1) = (\mathsf{id} \times \mathcal{P}_\mathrm{f}\kappa_{P_0}) \cdot \xi(t_1) = (\blacktriangle, \{\{c_1, c_2\}, \{c_3\}\}) = H\bar{q}_1 \cdot \xi(t_2)$$
$$Hq_2 \cdot \xi(t_1) = (\mathsf{id} \times \mathcal{P}_\mathrm{f}\chi_{\{c_1\}}) \cdot \xi(t_1) = (\blacktriangle, \{0, 1\}) = Hq_2 \cdot \xi(t_2)$$

So if we computed P_2 iteratively as in (4.1) for $q_2 = \chi_S$, then t_1 and t_2 would not be split, and we would reach the termination condition $P_2 = Q_2$ before all behaviourally inequivalent states have been separated.

Already Paige and Tarjan [25, Step 6 of the Algorithm] note that one additionally needs to split by $C \setminus S = \{c_3\}$, which is accomplished by splitting by $q_i = \chi_S^C$. This is formally captured by the condition we introduce next.

▶ **Definition 4.8.** A select routine *respects compound blocks* if whenever $k = \mathsf{select}(X \overset{y}{\twoheadrightarrow} Y \overset{z}{\twoheadrightarrow} Z)$ then the union $\ker k \cup \ker z$ is a kernel.

In $\mathsf{Set}^{\mathcal{S}}$, $\cup$ denotes the usual union of multi-sorted relations; and since reflexive and symmetric relations are closed under unions, the definition boils down to $\ker k \cup \ker z$ being transitive. We can rephrase the condition more explicitly, restricting to the single-sorted case for readability:

▶ **Lemma 4.9.** *For $a : D \to A$, $b : D \to B$ in* Set*, the following are equivalent:*

1. $\ker a \cup \ker b \rightrightarrows D$ *is a kernel.*
2. $\ker a \cup \ker b \rightrightarrows D$ *is the kernel of the pushout of a and b.*
3. *For all $x, y, z \in D$, $ax = ay$ and $by = bz$ implies $ax = ay = az$ or $bx = by = bz$.*
4. *For all $x \in D$, $[x]_a \subseteq [x]_b$ or $[x]_b \subseteq [x]_a$.*

The last item states that when going from a-equivalence classes to b-equivalence classes, the classes either merge or split, but do not merge with other classes and split at the same time. Note that in Figure 3, $Q_1 \cup \ker \chi_S$ fails to be transitive, while $Q_1 \cup \ker \chi_S^C$ is transitive.

▶ **Example 4.10.** All $\mathsf{select}\big(X \overset{y}{\twoheadrightarrow} Y \overset{z}{\twoheadrightarrow} Z\big)$ routines in Theorem 3.3 respect compound blocks.

▶ **Proposition 4.11.** *Let $a : D \to A$, $b : D \to B$ be a span such that $\ker a \cup \ker b$ is a kernel, and let $H :$ Set $\to \mathcal{D}$ be a zippable functor preserving monos. Then we have*

$$\ker\langle Ha, Hb\rangle = \ker H\langle a, b\rangle. \tag{4.2}$$

We thus obtain soundness of optimization (4.1); summing up:

▶ **Corollary 4.12.** *Suppose that H is a zippable endofunctor on* Set *and that* select *respects compound blocks and never discards all new information. Then Algorithm 3.4 with optimization* (4.1) *terminates and computes the simple quotient of a given finite H-coalgebra.*

▶ Remark. Like most results on set coalgebras, the above extends to multisorted sets by componentwise arguments, and this allows dealing with complex composite functors [29]. We restrict to a case with lightweight notation: Let F and G be zippable Set-functors, recalling from Theorem 4.6 that the composite FG need not itself be zippable, and let F be finitary. Then in lieu of FG-coalgebras, we can equivalently consider coalgebras for the endofunctor $H : (X, Y) \mapsto (FY, GX)$ on Set^2. In particular, a coalgebra $\xi : X \to FGX$ with finite carrier X induces, since F is finitary, a finite set $Y \subseteq_{\mathrm{f}} GX$, with inclusion y, such that $\xi = (X \xrightarrow{x} FY \xrightarrow{Fy} FGX)$, so we obtain a finite H-coalgebra $(x, y) : (X, Y) \to (FY, GX)$. Mutatis mutandis, Proposition 4.11 holds also for H, since kernels and pairs in Set^2 are computed componentwise, so we obtain a version of Theorem 4.12 for H. Explicitly, when computing the kernel of $H\bar{q}_{i+1} \cdot (x, y)$, we can use the optimization (4.1) in both sorts. The first component of the simple quotient of $((X, Y), (x, y))$ computed by the algorithm then yields the simple quotient of the original (X, ξ). Composites of more than two functors are treated similarly.

▶ **Example 4.13.** Applying this to the functors $F = \mathcal{P}_{\mathrm{f}}$, $G = A \times (-)$, and $H = \mathcal{D}$, we obtain simple (resp. general) Segala systems as coalgebras for FGH (resp. FHG). For simple Segala systems, the Set^3 functor is defined by $(X, Y, Z) \mapsto (FY, GZ, HX)$.

5 Efficient Calculation of Kernels

In Algorithm 3.4, it is left unspecified how the kernels are computed concretely. We proceed to define a more concrete algorithm based on a *refinement interface* of the functor. This interface is aimed at efficient implementation of the *refinement step* in the algorithm. Specifically from now on, we split along $\xi : X \to HY$ w.r.t. a subblock $S \subseteq C \in Y/Q$, and need to compute how the splitting of C into S and $C \setminus S$ within Y/Q affects the partition X/P.

The low complexity of Paige-Tarjan-style algorithms hinges on this refinement step running in time $\mathcal{O}(|\mathsf{pred}[S]|)$, where $\mathsf{pred}(y)$ denotes the set of predecessors of some $y \in Y$

in the given transition system. In order to speak about "predecessors" w.r.t. more general $\xi : X \to HY$, the refinement interface will provide an encoding of H-coalgebras as sets of states with successor states encoded as bags (implemented as lists up to ordering; recall that $\mathcal{B}_\mathrm{f}Z$ denotes the set of bags over Z) of A-labelled edges, where A is an appropriate label alphabet. Moreover, the interface will allow us to talk about the behaviour of elements of X w.r.t. the splitting of C into S and $C \setminus S$, looking only at points in S.

▶ **Definition 5.1.** A *refinement interface* for a Set-functor H is formed by a set A of *labels*, a set W of *weights* and functions

$$\begin{aligned} &\mathsf{b} : HY \to \mathcal{B}_\mathrm{f}(A \times Y), && \mathsf{init} : H1 \times \mathcal{B}_\mathrm{f}A \to W, \\ &w : \mathcal{P}_\mathrm{f}Y \to HY \to W, && \mathsf{update} : \mathcal{B}_\mathrm{f}A \times W \to W \times H3 \times W \end{aligned}$$

such that for every $S \subseteq C \subseteq Y$, the diagrams

$$\begin{array}{ccccc} HY & & \xrightarrow{w(Y)} & & \\ {\scriptstyle\langle H!,\, \mathcal{B}_\mathrm{f}\pi_1 \cdot \mathsf{b}\rangle}\downarrow & & & & \\ H1 \times \mathcal{B}_\mathrm{f}A & & \xrightarrow{\mathsf{init}} & & W \end{array} \qquad \begin{array}{ccccccc} HY & & & & \xrightarrow{\langle w(S), H\chi_S^C, w(C\setminus S)\rangle} & & \\ {\scriptstyle\langle \mathsf{b}, w(C)\rangle}\downarrow & & & & & & \\ \mathcal{B}_\mathrm{f}(A \times Y) \times W & & \xrightarrow{\mathsf{fil}_S \times W} & & \mathcal{B}_\mathrm{f}A \times W & \xrightarrow{\mathsf{update}} & W \times H3 \times W \end{array} \tag{5.1}$$

commute, where $\mathsf{fil}_S : \mathcal{B}_\mathrm{f}(A \times Y) \to \mathcal{B}_\mathrm{f}(A)$ is the filter function $\mathsf{fil}_S(f)(a) = \sum_{y \in S} f(a, y)$ for $S \subseteq Y$. The significance of the set $H3$ is that when using a set $S \subseteq C \subseteq X$ as a splitter, we want to split every block B in such a way that it becomes *compatible* with S and $C \setminus S$, i.e. we group the elements $s \in B$ by the value of $H\chi_S^C \cdot \xi(s) \in H3$. The set W depends on the functor. But in most cases $W = H2$ and $w(C) = H\chi_C : HY \to H2$ are sufficient.

In an implementation, we do not require a refinement interface to provide w explicitly, because the algorithm will compute the values of w incrementally using (5.1), and b need not be implemented because we assume the input coalgebra to be already *encoded* via b:

▶ **Definition 5.2.** Given an interface of H (Theorem 5.1), an *encoding* of a morphism $\xi : X \to HY$ is given by a set E and maps

$$\mathsf{graph} : E \to X \times A \times Y \qquad\qquad \mathsf{type} : X \to H1$$

such that $(\mathsf{b} \cdot \xi(x))(a, y) = |\{e \in E \mid \mathsf{graph}(e) = (x, a, y)\}|$, and with $\mathsf{type} = H! \cdot \xi$.

Intuitively, an encoding presents the morphism ξ as a graph with edge labels from A.

▶ **Lemma 5.3.** *Every morphism $\xi : X \to HY$ has a canonical encoding where E is the obvious set of edges of $\mathsf{b} \cdot \xi : X \to \mathcal{B}_\mathrm{f}(A \times Y)$. If X is finite, then so is E.*

▶ **Example 5.4.** In the following examples, we take $W = H2$ and $w(C) = H\chi_C : HY \to H2$. We use the helper function $\mathsf{val} := \langle H(=2), \mathsf{id}, H(=1)\rangle : H3 \to H2 \times H3 \times H2$, where $(=x) : 3 \to 2$ is the equality check for $x \in \{1, 2\}$, and in each case define $\mathsf{update} = \mathsf{val} \cdot \mathsf{up}$ for some function $\mathsf{up} : \mathcal{B}_\mathrm{f}A \times H2 \to H3$. We implicitly convert sets into bags.

1. For the monoid-valued functor $G^{(-)}$, for an Abelian group $(G, +, 0)$, we take labels $A = G$ and define $\mathsf{b}(f) = \{(f(y), y) \mid y \in Y, f(y) \neq 0\}$ (which is is finite because f is finitely supported). With $W = H2 = G \times G$, the weight $w(C) = H\chi_C : HY \to G \times G$ is the accumulated weight of $Y \setminus C$ and C. Then the remaining functions are

   $$\mathsf{init}(h_1, e) = (0, \Sigma e) \quad \text{and} \quad \mathsf{up}(e, (r, c)) = (r, c - \Sigma e, \Sigma e),$$

   where $\Sigma : \mathcal{B}_\mathrm{f}G \to G$ is the obvious summation map.

2. Similarly to the case $\mathbb{R}^{(-)}$, one has the following init and up functions for the distribution functor $\mathcal{D}$: put $\mathsf{init}(h_1, e) = (0, 1) \in \mathcal{D}2 \subset [0,1]^2$ and $\mathsf{up}(e, (r, c)) = (r, c - \Sigma e, \Sigma e)$ if the latter lies in $\mathcal{D}3$, and $(0, 0, 1)$ otherwise.
3. Similarly, one obtains a refinement interface for $\mathcal{B}_\mathrm{f} = \mathbb{N}^{(-)}$ adjusting the one for $\mathbb{Z}^{(-)}$; in fact, init remains unchanged and $\mathsf{up}(e, (r, c)) = (r, c - \Sigma e, \Sigma e)$ if the middle component is a natural number and $(0, 0, 0)$ otherwise.
4. Given a polynomial functor H_Σ for a signature Σ with bounded arity (i.e. there exists k such that every arity is at most k), the labels $A = \mathbb{N}$ encode the indices of the parameters:

$$\mathfrak{b}(\sigma(y_1, \dots, y_n)) = \{(1, y_1), \dots, (n, y_n)\} \quad \mathsf{init}(\sigma(0, \dots, 0), f) = \sigma(1, \dots, 1)$$
$$\mathsf{up}(I, \sigma(b_1, \dots, b_n)) = \sigma(b_1 + (1 \in I), \dots, b_i + (i \in I), \dots, b_n + (n \in I))$$

   Here $b_i + (i \in I)$ means $b_i + 1$ if $i \in I$ and b_i otherwise. Since I are the indices of the parameters in the subblock, $i \in I$ happens only if $b_i = 1$.

One example where $W = H2$ does not suffice is the powerset functor $\mathcal{P}$: Even if we know for a $t \in \mathcal{P}Y$ that it contains elements in $C \subseteq Y$, in $S \subseteq C$, and outside C (i.e. we know $\mathcal{P}\chi_S(t), \mathcal{P}\chi_C \in \mathcal{P}2$), we cannot determine whether there are any elements in $C \setminus S$ – but as seen in Theorem 4.7, we need to include this information.

▶ **Example 5.5.** The interface for the powerset functor needs to count the edges into blocks $C \supseteq S$ in order to know whether there are edges into $C \setminus S$, as described by Paige and Tarjan [25]. What happens formally is that first the interface for $\mathbb{N}^{(-)}$ is implemented for edge weights at most 1, and then the middle component of the result of update is adjusted. So $W = \mathbb{N}^{(2)}$, and the encoding $\mathfrak{b} : \mathcal{P}_\mathrm{f}Y \hookrightarrow \mathcal{B}_\mathrm{f}(1 \times Y)$ is the obvious inclusion. Then

$$\mathsf{init}(h_1, e) = (0, |e|) \qquad w(C)(M) = \mathcal{B}_\mathrm{f}\chi_C(M) = (|M \setminus C|, |C \cap M|)$$
$$\begin{aligned}\mathsf{update}(n, (c, r)) &= \langle \mathcal{B}_\mathrm{f}(=2),\ (\overset{?}{>}0)^3,\ \mathcal{B}_\mathrm{f}(=1)\rangle(r, c - |n|, |n|) \\ &= \big((r + c - |n|, |n|), (r \overset{?}{>} 0, c - |n| \overset{?}{>} 0, |n| \overset{?}{>} 0), (r + |n|, c - |n|)\big),\end{aligned}$$

where $x \overset{?}{>} 0$ is 0 if $x = 0$ and 1 otherwise.

▶ **Assumption 5.6.** From now on, assume a Set-functor H with a refinement interface such that init and update run in linear time and elements of $H3$ can be compared in constant time.

▶ **Example 5.7.** The refinement interfaces in Examples 5.4 and 5.5 satisfy Assumption 5.6.

▶ Remark. In the implementation, we encode the partitions X/P, Y/Q as doubly linked lists of the blocks they contain, and each block is in turn encoded as a doubly linked list of its elements. The elements $x \in X$ and $y \in Y$ each hold a pointer to the corresponding list entry in the blocks containing them. This allows removing elements from a block in $\mathcal{O}(1)$.

The algorithm maintains the following mutable data structures:

- An array $\mathsf{toSub} : X \to \mathcal{B}_\mathrm{f}E$, mapping $x \in X$ to its outgoing edges ending in the currently processed subblock.
- A pointer mapping edges to memory addresses: $\mathsf{lastW} : E \to \mathbb{N}$.
- A store of last values $\mathsf{deref} : \mathbb{N} \to W$.
- For each block B a list of markings $\mathsf{mark}_B \subseteq B \times \mathbb{N}$.

▶ **Notation 5.8.** In the following we write $e = x \xrightarrow{a} y$ in lieu of $\mathsf{graph}(e) = (x, a, y)$.

1: **for** $e \in E$, $e = x \xrightarrow{a} y$ **do**
2: add e to $\mathsf{toSub}[x]$ and $\mathsf{pred}[x]$.
3: **for** $x \in X$ **do**
4: $p_X :=$ new cell in deref containing $\mathsf{init}(\mathsf{type}[x], \mathcal{B}_\mathrm{f}(\pi_2 \cdot \mathsf{graph})(\mathsf{toSub}[x]))$
5: **for** $e \in \mathsf{toSub}[x]$ **do** $\mathsf{lastW}[e] = p_X$
6: $\mathsf{toSub}[x] := \varnothing$
7: $X/P :=$ group X by $\mathsf{type} : X \to H1$, $Y/Q := \{Y\}$.

Figure 4 The initialization procedure.

▶ **Definition 5.9** (Invariants). Our correctness proof below establishes the following properties that we call *the invariants*:

1. For all $x \in X, \mathsf{toSub}(x) = \varnothing$, i.e. toSub is empty everywhere.
2. For $e_i = x_i \xrightarrow{a_i} y_i, i \in \{1,2\}$, $\mathsf{lastW}(e_1) = \mathsf{lastW}(e_2) \iff x_1 = x_2$ and $[y_1]_{\kappa_Q} = [y_2]_{\kappa_Q}$.
3. For each $e = x \xrightarrow{a} y$, $C := [y]_{\kappa_Q} \in Y/Q$, $w(C, \xi(x)) = \mathsf{deref} \cdot \mathsf{lastW}(e)$.
4. For $x_1, x_2 \in B \in X/P$, $C \in Y/Q$, $(x_1, x_2) \in \ker(H\chi_C \cdot \xi)$.

In the following code listings, we use square brackets for array lookups and updates in order to emphasize they run in constant time. We assume that the functions $\mathsf{graph} : E \to X \times A \times Y$ and $\mathsf{type} : X \to H1$ are implemented as arrays. In the initialization step the predecessor array $\mathsf{pred} : Y \to \mathcal{P}_\mathrm{f} E$, $\mathsf{pred}(y) = \{e \in E \mid e = x \xrightarrow{a} y\}$ is computed. Sets and bags are implemented as lists. We only insert elements into sets not yet containing them.

We say that we *group* a finite set Z by $f : Z \to Z'$ to indicate that we compute $[-]_f$. This is done by sorting the elements of $z \in Z$ by a binary encoding of $f(z)$ using any $\mathcal{O}(|Z| \cdot \log |Z|)$ sorting algorithm, and then grouping elements with the same $f(z)$ into blocks. In order to keep the overall complexity for the grouping operations low enough, one needs to use a possible majority candidate during sorting, following Valmari and Franceschinis [32].

The algorithm computing the initial partition is listed in Figure 4.

▶ **Lemma 5.10.** *The initialization procedure runs in time $\mathcal{O}(|E| + |X| \cdot \log |X|)$ and makes the invariants true.*

The algorithm for one refinement step along a morphism $\xi : X \to HY$ is listed in Figure 5.

In the first part, all blocks $B \in X/P$ are collected that have an edge into S, together with $v_\varnothing \in H3$ which represents $H\chi_S^C \cdot \xi(x)$ for any $x \in B$ that has no edge into S. For each $x \in X$, $\mathsf{toSub}[x]$ collects the edges from x into S. The markings mark_B list those elements $x \in B$ that have an edge into S, together with a pointer to $w(C, x)$.

In the second part, each block B with an edge into S is refined w.r.t. $H\chi_S^C \cdot \xi$. First, for any $(x, p_C) \in \mathsf{mark}_B$, we compute $w(S, x)$, $v^x = H\chi_S^C \cdot \xi(x)$, and $w(C \setminus S, x)$ using update. Then, the weight of all edges $x \to C \setminus S$ is updated to $w(C \setminus S, x)$ and the weight of all edges $x \to S$ needs to be stored in a new cell containing $w(S, x)$. For all unmarked $x \in B$, we know that $H\chi_S^C \cdot \xi(x) = v_\varnothing$; so all x with $v^x = v_\varnothing$ stay in B. All other $x \in B$ are removed and distributed to new blocks w.r.t. v^x.

▶ **Theorem 5.11.** $\textsc{Split}(X/P, Y/Q, S \subseteq C \in Y/Q)$ *refines X/P by $H\chi_S^C \cdot \xi : X \to H3$.*

▶ **Lemma 5.12.** *After running* Split, *the invariants hold.*

▶ **Lemma 5.13.** *Lines $1 - 23$ in* Split *run in time $\mathcal{O}(\sum_{y \in S} |\mathsf{pred}(y)|)$.*

SPLIT$(X/P, Y/Q, S \subseteq C \in Y/Q)$

1: $\mathsf{M} := \emptyset \subseteq X/P \times H3$
2: **for** $y \in S, e \in \mathsf{pred}[y]$ **do**
3: $\quad x \xrightarrow{a} y := e$
4: $\quad B :=$ block with $x \in B \in X/P$
5: $\quad$ **if** mark_B is empty **then**
6: $\quad\quad w_C^x := \mathsf{deref} \cdot \mathsf{lastW}[e]$
7: $\quad\quad v_\emptyset := \pi_2 \cdot \mathsf{update}(\emptyset, w_C^x)$
8: $\quad\quad$ add $(B, v_\emptyset)$ to M
9: $\quad$ **if** $\mathsf{toSub}[x] = \emptyset$ **then**
10: $\quad\quad$ add $(x, \mathsf{lastW}[e])$ to mark_B
11: $\quad$ add e to $\mathsf{toSub}[x]$

(a) Collecting predecessor blocks

12: **for** $(B, v_\emptyset) \in \mathsf{M}$ **do**
13: $\quad B_{\neq\emptyset} := \emptyset \subseteq X \times H3$
14: $\quad$ **for** (x, p_C) in mark_B **do**
15: $\quad\quad \ell := \mathcal{B}_\mathrm{f}(\pi_2 \cdot \mathsf{graph})(\mathsf{toSub}[x])$
16: $\quad\quad (w_S^x, v^x, w_{C\setminus S}^x) := \mathsf{update}(\ell, \mathsf{deref}[p_C])$
17: $\quad\quad \mathsf{deref}[p_C] := w_{C\setminus S}^x$
18: $\quad\quad p_S :=$ new cell containing w_S^x
19: $\quad\quad$ **for** $e \in \mathsf{toSub}[x]$ **do** $\mathsf{lastW}[e] := p_S$
20: $\quad\quad \mathsf{toSub}[x] := \emptyset$
21: $\quad\quad$ **if** $v^x \neq v_\emptyset$ **then**
22: $\quad\quad\quad$ remove x from B
23: $\quad\quad\quad$ insert (x, v^x) into $B_{\neq\emptyset}$
24: $\quad B_1 \times \{v_1\}, \ldots, B_\ell \times \{v_\ell\} :=$
$\quad\quad$ group $B_{\neq\emptyset}$ by $\pi_2 : X \times H3 \to H3$
25: $\quad$ insert $B_1, \ldots, B_\ell :=$ into X/P

(b) Splitting predecessor blocks

Figure 5 Refining X/P w.r.t $\chi_C^S : Y \to 3$ and Y/Q along $\xi : X \to HY$.

▶ **Lemma 5.14.** *For $S_i \subseteq C_i \in Y/Q_i, 0 \leq i < k$, with $2 \cdot |S_i| \leq |C_i|$ and $Q_{i+1} = \ker\langle \kappa_{Q_i}, \chi_{S_i}^{C_i} \rangle$:*

1. *For each $y \in Y$, $|\{i < k \mid y \in S_i\}| \leq \log_2 |Y|$.*
2. SPLIT$(S_i \subseteq C_i \in Y/Q_i)$ *for all $0 \leq i < k$ takes at most $\mathcal{O}(|E| \cdot \log |Y|)$ time in total.*

Bringing Sections 3, 4, and 5 together, take a coalgebra $\xi : X \to HX$ for a zippable Set-functor H with a given refinement interface where init and update run in linear and comparison in constant time. Instantiate Algorithm 3.4 with the select routine from Theorem 3.3.1, making $q_{i+1} = k_{i+1} \cdot \kappa_{P_i} = \chi_{S_i}^{C_i}$ with $2 \cdot |S| \leq |C|$, $S_i, C_i \subseteq X$. Replace line 4 by

$$X/P_{i+1} = \text{SPLIT}(X/P_i, X/Q_i, S_i \subseteq C_i). \tag{4.1'}$$

By Theorem 5.11 this is equivalent to (4.1), and hence, by Theorem 4.12, to the original line 4, since $\chi_{S_i}^{C_i}$ respects compound blocks. By Lemmas 5.10 and 5.14.2, we have

▶ **Theorem 5.15.** *The above instance of Algorithm 3.4 computes the quotient modulo behavioural equivalence of a given coalgebra in time $\mathcal{O}((m+n) \cdot \log n)$, for $m = |E|$, $n = |X|$.*

▶ **Example 5.16.**

1. For $H = \mathcal{I} \times \mathcal{P}_\mathrm{f}$, we obtain the classical Paige-Tarjan algorithm [25] (with initial partition $\mathcal{I}$), with the same complexity $\mathcal{O}((m+n) \cdot \log n)$.
2. For $HX = \mathcal{I} \times \mathbb{R}^{(X)}$, we solve Markov chain lumping with an initial partition $\mathcal{I}$ in time $\mathcal{O}((m+n) \cdot \log n)$, like the best known algorithm (Valmari and Franceschinis [32]).
3. For infinite A, we need to decompose the functor $\mathcal{P}_\mathrm{f}(A \times (-))$ for labelled transition systems into $\mathcal{P}_\mathrm{f}$ and $A \times (-)$, and thus obtain run time $\mathcal{O}(m \cdot \log m)$, like in [10] but slower than Valmari's $\mathcal{O}(m \cdot \log n)$ [31]. For fixed finite A, we run in time $\mathcal{O}(m \log n)$.
4. Hopcroft's classical automata minimization [14] is obtained by $HX = 2 \times X^A$, with running time $\mathcal{O}(n \cdot \log n)$ for fixed alphabet A. For non-fixed A the best known complexity is $\mathcal{O}(|A| \cdot n \cdot \log n)$ [12, 21]. By using decomposition into $2 \times \mathcal{P}_\mathrm{f}$ and $\mathbb{N} \times (-)$ we obtain $\mathcal{O}(|A| \cdot n \cdot \log n + |A| \cdot n \cdot \log |A|)$.
5. We quotient simple (resp. general) Segala systems [30] by bisimilarity after decomposition into three sorts (cf. Theorem 4.13). The time bound $\mathcal{O}((n+m)\log(n+m))$ slightly improves on the previous bound [3].

6 Conclusions and Future Work

We have presented a generic algorithm that quotients coalgebras by behavioural equivalence. We have started from a category-theoretic procedure that works for every mono-preserving functor on a category with image factorizations, and have then developed an improved algorithm for *zippable* endofunctors on Set. Provided the given type functor can be equipped with an efficient implementation of a *refinement interface*, we have finally arrived at a concrete procedure that runs in time $\mathcal{O}((m+n)\log n)$ where m is the number of edges and n the number of nodes in a graph-based representation of the input coalgebra. We have shown that this instantiates to (minor variants of) several known efficient partition refinement algorithms: the classical Hopcroft algorithm [14] for minimization of DFAs, the Paige-Tarjan algorithm for unlabelled transition systems [25], and Valmari and Franceschinis's lumping algorithm for weighted transition systems [32]. Moreover, we obtain a new algorithm for simple Segala systems that is asymptotically faster than previous algorithms [3]. Coverage of Segala systems is based on modularity results in multi-sorted coalgebra [29].

It remains open whether our approach can be extended to, e.g., the monotone neighbourhood functor, which is not itself zippable and also does not have an obvious factorization into zippable functors. We do expect that our algorithm applies beyond weighted systems.

References

1 Jiří Adámek. Introduction to coalgebra. *Theory Appl. Categ.*, 14:157–199, 2005.
2 Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and Concrete Categories*. Wiley Interscience, 1990.
3 Christel Baier, Bettina Engelen, and Mila Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60:187–231, 2000.
4 Falk Bartels, Ana Sokolova, and Erik de Vink. A hierarchy of probabilistic system types. In *Coagebraic Methods in Computer Science, CMCS 2003*, volume 82 of *ENTCS*, pages 57 – 75. Elsevier, 2003.
5 Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1):74–86, 2005.
6 Peter Buchholz. Bisimulation relations for weighted automata. *Theoret. Comput. Sci.*, 393:109–123, 2008.
7 Stefano Cattani and Roberto Segala. Decision algorithms for probabilistic bisimulation. In *Concurrency Theory, CONCUR 2002*, volume 2421 of *LNCS*, pages 371–385. Springer, 2002.
8 Salem Derisavi, Holger Hermanns, and William Sanders. Optimal state-space lumping in markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003.
9 Josee Desharnais, Abbas Edalat, and Prakash Panangaden. Bisimulation for labelled markov processes. *Inf. Comput.*, 179(2):163–193, 2002.
10 Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.
11 Kathi Fisler and Moshe Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.
12 David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.
13 Heinz-Peter Gumm and Tobias Schröder. Monoid-labelled transition systems. In *Coalgebraic Methods in Computer Science, CMCS 2001*, volume 44 of *ENTCS*, pages 185–204, 2001.
14 John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

15 Dung Huynh and Lu Tian. On some equivalence relations for probabilistic processes. *Fund. Inform.*, 17:211–234, 1992.

16 Bart Jacobs. *Introduction to Coalgebras: Towards Mathematics of States and Observations.* Cambridge University Press, 2017.

17 Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bull. EATCS*, 62:222–259, 1997.

18 Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

19 Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *LNCS*, pages 87–101. Springer, 2007.

20 Bartek Klin. Structural operational semantics for weighted transition systems. In Jens Palsberg, editor, *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, volume 5700 of *LNCS*, pages 121–139. Springer, 2009.

21 Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.*, 250:333 – 363, 2001.

22 Barbara König and Sebastian Küpper. Generic partition refinement algorithms for coalgebras and an instantiation to weighted automata. In *Theoretical Computer Science, IFIP TCS 2014*, volume 8705 of *LNCS*, pages 311–325. Springer, 2014.

23 Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 94:1–28, 1991.

24 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

25 Robert Paige and Pobert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

26 David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

27 Francesco Ranzato and Francesco Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Inf. Comput.*, 206:620–651, 2008.

28 Jan Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.

29 Lutz Schröder and Dirk Pattinson. Modular algorithms for heterogeneous modal logics via multi-sorted coalgebra. *Math. Struct. Comput. Sci.*, 21(2):235–266, 2011.

30 Roberto Segala. *Modelling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusetts Institute of Technology, 1995.

31 Antti Valmari. Bisimilarity minimization in $\mathcal{O}(m \log n)$ time. In *Applications and Theory of Petri Nets, PETRI NETS 2009*, volume 5606 of *LNCS*, pages 123–142. Springer, 2009.

32 Antti Valmari and Giuliana Franceschinis. Simple $\mathcal{O}(m \log n)$ time Markov chain lumping. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010*, volume 6015 of *LNCS*, pages 38–52. Springer, 2010.

33 Johann van Benthem. *Modal Correspondence Theory.* PhD thesis, Universiteit van Amsterdam, 1977.

34 Ron van der Meyden and Chenyi Zhang. Algorithmic verification of noninterference properties. In *Views on Designing Complex Architectures, VODCA 2006*, volume 168 of *ENTCS*, pages 61–75. Elsevier, 2007.

35 James Worrell. On the final sequence of a finitary set functor. *Theor. Comput. Sci.*, 338:184–199, 2005.

36 Lijun Zhang, Holger Hermanns, Friedrich Eisenbrand, and David Jansen. Flow Faster: Efficient decision algorithms for probabilistic simulations. *Log. Meth. Comput. Sci.*, 4(4), 2008.