Future Generation Computer Systems 79 (2018) 653–668

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Automatically deriving cost models for structured parallel processes using hylomorphisms

David Castro *, Kevin Hammond, Susmit Sarkar, Yasir Alguwaifli

School of Computer Science, University of St Andrews, St Andrews, UK

HIGHLIGHTS

- An operational semantics of a queue language for streaming computations.
- An operational semantics of algorithmic skeletons using this queue language.
- Derive cost equations for algorithmic skeletons from the operational semantics.
- Cost equations for algorithmic skeletons are combined with sized types.

ARTICLE INFO

Article history:
Received 24 March 2017
Accepted 21 April 2017
Available online 9 May 2017

Keywords:
Operational semantics
Algorithmic skeletons
Cost models
Hylomorphisms

ABSTRACT

Structured parallelism using nested algorithmic skeletons can greatly ease the task of writing parallel software, since common, but hard-to-debug, problems such as *race conditions* are eliminated *by design*. However, choosing the best combination of algorithmic skeletons to yield good parallel speedups for a specific program on a specific parallel architecture is still a difficult problem. This paper uses the unifying notion of *hylomorphisms*, a general recursion pattern, to make it possible to reason about both the functional correctness properties and the extra-functional timing properties of structured parallel programs. We have previously used hylomorphisms to provide a denotational semantics for skeletons, and proved that a given parallel structure for a program satisfies functional correctness. This paper expands on this theme, providing a simple operational semantics for algorithmic skeletons and a cost semantics that can be automatically derived from that operational semantics. We prove that both semantics are sound with respect to our previously defined denotational semantics. This means that we can now automatically and statically choose a provably optimal parallel structure for a given program with respect to a cost model for a (class of) parallel architecture. By deriving an *automatic amortised analysis* from our cost model, we can also accurately predict parallel runtimes and speedups.

1. Introduction

In previous work [1], we have defined a type-based mechanism for reasoning about the safe introduction of parallelism using a structured parallel approach. Our approach allows us to extract parallel program structure as a type. Given a suitable model of a program's execution cost (e.g. in terms of its time performance), we can reason formally about performance improvements for alternative parallelisations, and so select a *provably optimal* parallel implementation. In this paper, we show how to derive appropriate cost models formally from the parallel structure of a program, using a new queue-based operational semantics (Fig. 1). This gives a completely formal system for reasoning about the performance of structured parallel programs. We combine related work on *algorithmic skeletons* and *recursion schemes*. Algorithmic skeletons [2] are parametric implementations of common patterns of parallel programming. Using a pattern/skeleton approach, the programmer can design and implement a parallel program in a top-down manner. For example, the programmer could first identify the parallel patterns that occur in a particular piece of software, then select the patterns that potentially lead to the best speedups, and finally select a suitable implementation for those patterns, as a composition of one or more algorithmic skeletons. This composition then exposes the *parallel structure* of the implementation. Developing an equational theory for easily, and automatically, changing the

* Corresponding author.
E-mail addresses: dc84@st-andrews.ac.uk (D. Castro), kevin@kevinhammond.net (K. Hammond), ss265@st-andrews.ac.uk (S. Sarkar), ya8@st-andrews.ac.uk (Y. Alguwaifli).

http://dx.doi.org/10.1016/j.future.2017.04.035
0167-739X/

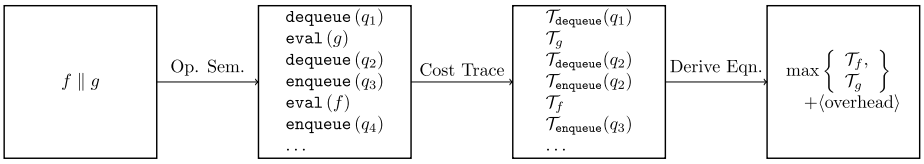


Fig. 1. Deriving cost equations from operational semantics.

parallel structure of a program has been the subject of research in the skeletons community for the last two decades [3–6]. Over a similar timeframe, in the functional programming community, a structured form of recursion has been explored, in the form of *patterns of recursion*, or *recursion schemes* [7]. Research on this topic has brought many improvements to equational reasoning in functional languages. One example is the application of the laws and properties of recursion schemes to the well-known *deforestation* optimisation [8,9].

There is an obvious connection between algorithmic skeletons and recursion schemes: algorithmic skeletons are essentially *higher order functions* that implement some common pattern of parallelism. This has already been exploited a number of times [3,10–12]. In [1], we expand on this connection, using the fact that a large number of recursion patterns can be represented as instances of a more general pattern, a *hylomorphism*. The basic idea is to provide a single unifying framework for reasoning both about program transformations and about parallel execution times. This unifying framework provides a type-level abstraction of the *program structure*, given as a combination of the *hylomorphisms* and *algorithmic skeletons* that are used to implement a particular program. This allows us to define a type-based mechanism for reasoning about the safe introduction of parallelism: specific combinations of hylomorphisms can be "replaced" by specific combinations of algorithmic skeletons. We provide strong static guarantees that the resulting program will be functionally equivalent to the original one. Given a suitable *model* for the cost of a parallel implementation (e.g. in terms of execution time), we can then use this type-based approach to reason formally about performance improvements for alternative parallelisations, and so to select a *provably optimal* parallel implementation.

1.1. Novel contributions

In this paper, we show how to formally derive appropriate cost models from the parallel structure of a program, using a new queue-based operational semantics. This gives a completely formal system for reasoning about the performance of structured parallel programs. The main novel contributions of the paper are:

- We define the operational semantics of a *queue language* that is powerful enough to describe the operational semantics of a number of algorithmic skeletons, and small and restrictive enough to facilitate reasoning about correctness and execution times (Section 4).
- We define the *operational semantics* of a number of key algorithmic skeletons using this queue language (Section 5).
- We derive a set of *cost equations* for a number of algorithmic skeletons from the operational semantics in a systematic way. We combine these cost equations with the notion of *sized types*, and sketch how this process can be automated (Section 6).

1.2. Motivating example

We illustrate our approach using the *image merge* example from [1]. The purpose of the `imgMerge` function is to `mark` and then `merge` pairs of images.

$$\texttt{imgMerge} \;:\; \mathsf{List}(\texttt{Img} \times \texttt{Img}) \rightarrow \mathsf{List}\ \texttt{Img}$$
$$\texttt{imgMerge} \;=\; \texttt{map}\ (\texttt{merge} \circ \texttt{mark})$$

This has many possible, semantically equivalent, parallel implementations.

$$\texttt{imgMerge}_1 \;=\; \mathsf{farm}\ n\ (\mathsf{fun}\ (\texttt{merge} \circ \texttt{mark}))$$
$$\texttt{imgMerge}_2 \;=\; \mathsf{farm}\ m\ (\mathsf{fun}\ \texttt{mark}) \parallel \mathsf{farm}\ n\ (\mathsf{fun}\ \texttt{merge})$$
$$\texttt{imgMerge}_3 \;=\; \mathsf{farm}\ n\ (\mathsf{fun}\ \texttt{mark}) \parallel \mathsf{fun}\ \texttt{merge}$$
$$\ldots$$

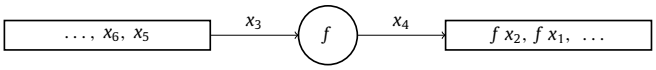
here, farm n is a *task farm* skeleton that replicates its argument n times, fun captures primitive (sequential) functions, $\circ$ is the normal function composition, and $\parallel$ is a parallel pipeline, i.e. the parallel composition of two skeletons. The structure of each parallel implementation can be lifted into an appropriate type signature, so that

$$\texttt{imgMerge}_1 \;:\; \mathsf{List}(\texttt{Img} \times \texttt{Img}) \xrightarrow{\mathsf{FARM}_n(\mathsf{FUN}\ (\texttt{merge}\circ\texttt{mark}))} \mathsf{List}\ \texttt{Img}$$

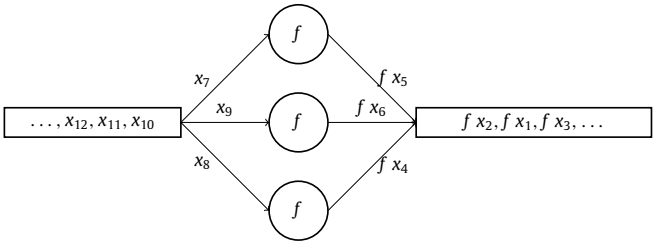
By combining well-known properties of algorithmic skeletons [13] and *hylomorphisms* [7], we can define a *convertibility* relation that relates each of these types for *imgMerge* to any of the other types. Moreover, this same relation also allows the underlying program to be automatically rewritten so that it matches the new type. We can thus automatically select any valid parallel implementation of *imgMerge* simply by changing the type, and without changing its definition. Finally, if we have a cost model for these types, we can now reason about costs, and automatically select the provably best parallel implementation for *imgMerge*, or, indeed, for any appropriately structured parallel program.

2. Algorithmic skeletons

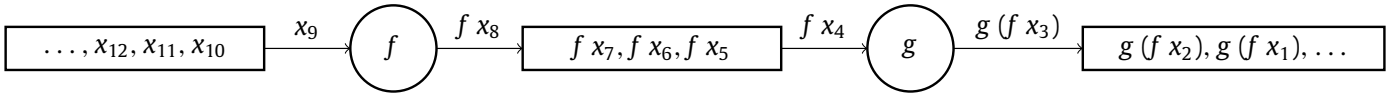
We use *algorithmic skeletons* [2] to represent structured parallel patterns. Skeletons are parameterised templates (*higher-order functions*) that capture the structure of a parallel program. That is, they implement a specific parallel *pattern*, that can be instantiated to produce a specific parallel *algorithm*. In this paper, as in our earlier work [1,14], we use a "pluggable" approach, where all skeletons are *streaming* entities (that is, they operate over multiple inputs and produce multiple results). Each skeleton takes its inputs from an *input queue*, and produces a *result queue*. This approach allows skeletons to be easily nested or linked together into more complex structures, whose sub-components are connected *via* intermediate queues. In this paper, we will consider four common parallel skeletons, *task farms, parallel pipelines, feedbacks*, and *parallel divide-and-conquer*, plus one basic building block, *structured functions*.



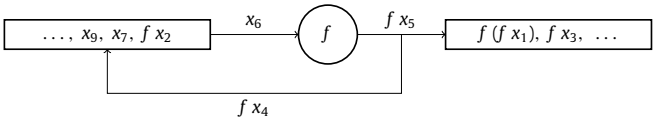
Structured Functions (FUN f) *lift* basic (sequentially executed) functions so that they can be used as building blocks for our skeletons. These may be either named functions; compositions of one or more structured functions, constructed using the *sequential composition* operator ($\circ$); or recursive functions that have been built using common patterns of recursion. Structured functions expose the underlying structure of a computation, enabling possible parallelisations.



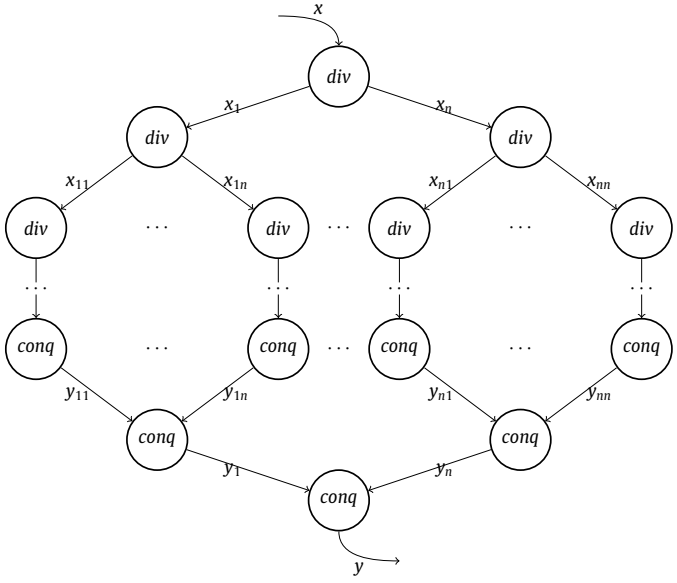
Task farms (farm) apply the same operation to each element of an input stream of tasks, using a fixed number of parallel workers. The input tasks must be independent, and the outputs can be produced in an arbitrary order.



Parallel pipelines ($\|$) compose two other streaming skeletons, in parallel. This can be used to parallelise two or more stages of a computation, e.g. marking and merging in the *image merge* example above.



Feedbacks (fb) capture recursion in a streaming computation, operating over some internally nested skeleton with a given predicate e.g. f. This could be used, for example, to repeatedly transform an image using some parallel skeleton until a certain dynamic condition was met.



Parallel Divide-and-Conquers (dc) are parallel implementations of classical divide-and-conquer algorithms. Parallelism comes from performing each of the recursive calls in the divide-and-conquer in parallel. In our framework, each instance of a divide-and-conquer skeleton divides an input into n sub-inputs using a $n + 1$-ary operation, so that we do not need to assume associativity.

3. A type-level treatment of parallel structure

3.1. Functors

We begin by defining some basic concepts from category theory. A *functor* is a structure-preserving mapping between *categories*. We will only require *endofunctors* on *CPO*, $F : CPO \to CPO$, which will represent type constructors, $F : \texttt{Type} \to \texttt{Type}$. *Bifunctors* are functors that are generalised to multiple arguments. We use them to define polymorphic data types. Our functors are either: (i) standard polynomial functors with constant types; (ii) the left section of a bifunctor; or (iii) a polymorphic type defined as the fixpoint of some bifunctor. Bifunctors are defined using products and sums alone. Functors are defined using a *pointed* notation, with the obvious semantic interpretation. If A and B are type variables, and T is a type, we accept the following definitions:

$$
\begin{array}{lr}
G\,A\,B = T & \text{(bifunctor defined using sums and products)} \\
F\,A = T & \text{(functor defined using sums and products)} \\
F\,A = G\,T\,A & (G \text{ is a bifunctor}) \\
F\,A = \mu(G\,A) & (G \text{ is a bifunctor})
\end{array}
$$

Example 1 (*Lists*)**.** Given the bifunctor $L\ A\ B = 1 + A \times B$, the polymorphic List data type is defined by the fixpoint of $L\ A$: $\mathsf{List}\ A = \mu(L\ A)$. Note that given a base bifunctor $G\ A\ B$, the data type $F\ A = \mu(G\ A)$ is also a functor. The two list constructors are defined in a standard way:

$$
\begin{array}{lcl@{\qquad}lcl}
\texttt{nil} & : & \mathsf{List}\ A & \texttt{cons} & : & A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
\texttt{nil} & = & in_{LA}\,(inj_1\,()) & \texttt{cons}\ x\ l & = & in_{LA}\,(inj_2\,(x, l))
\end{array}
$$

3.2. Hylomorphisms

Hylomorphisms are a well known, very general, recursion pattern [7], that can be seen as a generalised divide-and-conquer pattern. Intuitively, $hylo_F\ f\ g$ is a *regular* recursive algorithm (i.e. with no nested or mutual recursion), where g describes how the algorithm divides the input problem into sub-problems, stored in a structure F, and f describes how these results are combined.

$$
\begin{array}{lcl}
hylo_F & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\
hylo_F\ f\ g & = & f \circ F\ (hylo_F\ f\ g) \circ g
\end{array}
$$

Catamorphisms (folds), *anamorphisms* (unfolds) and *maps* are just special cases of hylomorphisms.

$$
\begin{array}{lcl}
T\ A & = & \mu(F\ A) \\
map_T\ f & = & hylo_{F\,A}\,(in_{F\,B} \circ (F\ f\ id))\ out_{F\,A}, \\
 & & \quad \text{where } A = dom(f) \text{ and } B = codom(f) \\
cata_F\ f & = & hylo_F\ f\ out_F \\
ana_F\ f & = & hylo_F\ in_F\ f.
\end{array}
$$

For any recursive type that is defined as the fixpoint of a functor, μF, the standard $in_F : F\mu F \to \mu F$ and $out_F : \mu F \to F\mu F$ capture the isomorphism between $F\mu F$ and μF. Note that, since $out_F \circ in_F = id$, $hylo_F\ f\ g = cata_F\ f \circ ana_F\ g$.

Example 2 (*Quicksort*)**.** Assuming a type A, and two functions, `leq` and `gt` that filter the elements appropriately, we can implement naïve quicksort as:

$$
\begin{array}{lcl}
\texttt{qsort} & : & \mathsf{List}\,A \to \mathsf{List}\,A \\
\texttt{qsort nil} & = & [] \\
\texttt{qsort}\,(\texttt{cons}\,x\,xs) & = & \texttt{qsort}\,(\texttt{leq}\,x\,xs) \\
 & & +\!+\,\texttt{cons}\,x(\texttt{qsort}\,(\texttt{gt}\,x\,l)).
\end{array}
$$

We make the recursive structure explicit by using a tree. The `split` function unfolds the arguments into this tree, and the `join` function then flattens it.

$$
\begin{array}{lcl}
\texttt{split} & : & \mathsf{List}\,A \to \mathsf{Tree}\,A \\
\texttt{split nil} & = & \texttt{empty} \\
\texttt{split}\,(\texttt{cons}\,x\,l) & = & \texttt{node}\,(\texttt{split}\,(\texttt{leq}\,x\,l))\,x\,(\texttt{split}\,(\texttt{gt}\,x\,l))
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{join} & : & \mathsf{Tree}\,A \to \mathsf{List}\,A \\
\texttt{join empty} & = & \texttt{nil} \\
\texttt{join}\,(\texttt{node}\,l\,x\,r) & = & \texttt{join}\,l +\!+ \texttt{cons}\,x\,(\texttt{join}\,r)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{qsort} & : & \mathsf{List}\,A \to \mathsf{List}\,A \\
\texttt{qsort} & = & \texttt{join} \circ \texttt{split}
\end{array}
$$

We can remove the explicit recursion from these definitions, since `split` is a tree anamorphism, and `join` is a tree catamorphism.

$$
\begin{array}{lcl}
\texttt{split} & : & \mathsf{List}\,A \to T\,A\,(\mathsf{List}\,A) \\
\texttt{split nil} & = & inj_1\,() \\
\texttt{split}\,(\texttt{cons}\,x\,l) & = & inj_2\,(x,\ \texttt{leq}\,x\,l,\ \texttt{gt}\,x\,l)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{join} & : & T\,A\,(\mathsf{List}\,A) \to \mathsf{List}\,A \\
\texttt{join}\,(inj_1\,()) & = & \texttt{nil} \\
\texttt{join}\,(inj_2\,(x, l, r)) & = & l +\!+ \texttt{cons}\,x\,r
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{qsort} & : & \mathsf{List}\,A \to \mathsf{List}\,A \\
\texttt{qsort} & = & cata_{T\,A}\ \texttt{join} \circ ana_{T\,A}\ \texttt{split}
\end{array}
$$

Finally, since we have a composition of a catamorphism and an anamorphism, we can write `qsort` as the equivalent hylomorphism.

$$\texttt{qsort} = hylo_{T\,A}\ \texttt{join}\ \texttt{split}$$

3.3. Type system and semantics

The syntax of expressions, E, is shown below. We distinguish between two levels of expressions: *Structured Expressions (S)* describe primitive (sequential) forms, and *Structured Parallel Processes (P)* introduce specific parallel skeletons.

$$
\begin{array}{lcl}
e \in E & ::= & s \mid \mathsf{par}_T\ p \\
s \in S & ::= & f \mid e_1 \circ e_2 \mid \mathsf{hylo}_F\ e_1\ e_2 \\
p \in P & ::= & \mathsf{fun}\ s \mid p_1 \parallel p_2 \mid \mathsf{dc}_{n,F}\ s_1\ s_2 \mid \mathsf{farm}\ n\ p \mid \mathsf{fb}\ p
\end{array}
$$

3.3.1. Denotational semantics

Our denotational semantics is likewise split into two parts: $\mathcal{S}[\![\cdot]\!]$ describes the base semantics, and $[\![\cdot]\!]$ lifts this to a *streaming* form. We use a global environment for atomic function types, ρ, and the corresponding global environment of functions, $\hat{\rho}$:

$$
\rho = \{f : A \to B,\ \ldots\} \qquad \hat{\rho} = \{[\![f]\!] \in [\![A \to B]\!],\ \ldots\}
$$

$$
\begin{array}{lcl}
\mathcal{S}[\![p : T\,A \to T\,B]\!] & : & [\![A \to B]\!] \\
\mathcal{S}[\![\mathsf{fun}\,f]\!] & = & \hat{\rho}(f) \\
\mathcal{S}[\![p_1 \parallel p_2]\!] & = & \mathcal{S}[\![p_2]\!] \circ \mathcal{S}[\![p_1]\!] \\
\mathcal{S}[\![\mathsf{farm}\,n\,p]\!] & = & \mathcal{S}[\![p]\!] \\
\mathcal{S}[\![\mathsf{fb}\,p]\!] & = & iter\ \mathcal{S}[\![p]\!] \\
\mathcal{S}[\![\mathsf{dc}_{n,T,F}\,f\,g]\!] & = & cata_F\ (\hat{\rho}(f)) \circ ana_F\ (\hat{\rho}(g))
\end{array}
$$

$$
\begin{array}{lcl}
[\![p : T\,A \to T\,B]\!] & : & [\![T\,A \to T\,B]\!] \\
[\![p]\!] & = & map_T\ \mathcal{S}[\![p]\!]
\end{array}
$$

here $\circ$ denotes the usual function composition; *iter* captures the iterative structure of a feedback skeleton (the usual *trampoline* function [1]); and *cata* and *ana* are the usual cata/anamorphisms. Instead of assuming a dynamic termination condition, we require the worker operation, p, of a feedback loop, $\mathsf{fb}\ p$ to return a sum type $A + B$. The feedback skeleton will apply p to a stream of inputs of type A, merging any results of type A with the other inputs, and returning any results of type B as its outputs. Since the *iter*, *cata*, *ana*

$$\frac{\rho(f) = A \to B}{\vdash f \ : \ A \xrightarrow{\mathrm{A}} B} \qquad \frac{\vdash e_1 \ : \ B \xrightarrow{\sigma_1} C \quad \vdash e_2 \ : \ A \xrightarrow{\sigma_2} B}{\vdash e_1 \circ e_2 \ : \ A \xrightarrow{\sigma_1 \circ \sigma_2} C}$$

$$\frac{\vdash e_1 \ : \ F\ B \xrightarrow{\sigma_1} B \quad \vdash e_2 \ : \ A \xrightarrow{\sigma_2} F\ A \quad G \ = \ \mathsf{base}\ F}{\vdash \mathtt{hylo}_F\ e_1\ e_2 \ : \ A \xrightarrow{\mathrm{HYLO}_G\ \sigma_1\ \sigma_2} B} \qquad \frac{\vdash p \ : \ T\ A \xrightarrow{\sigma} T\ B \quad F \ = \ \mathsf{base}\ T}{\vdash \mathsf{par}_T\ p \ : \ T\ A \xrightarrow{\mathrm{PAR}_F\ \sigma} T\ B}$$

$$\frac{\vdash \ s \ : \ A \xrightarrow{\sigma} B}{\vdash \ \mathsf{fun}\ s \ : \ T\ A \xrightarrow{\mathrm{FUN}\ \sigma} TB}$$

$$\frac{\vdash s_1 \ : \ F\ B \xrightarrow{\sigma_1} B \quad \vdash s_2 \ : \ A \xrightarrow{\sigma_2} F\ A \quad G \ = \ \mathsf{base}\ F}{\vdash \mathsf{dc}_{n,F}\ \ s_1\ s_2 \ : \ T\ A \xrightarrow{\mathrm{DC}_{n,G}\ \sigma_1\ \sigma_2} T\ B}$$

$$\frac{n \ : \ \mathbb{N} \quad \vdash p \ : \ T\ A \xrightarrow{\sigma} T\ B}{\vdash \mathsf{farm}\ n\ p \ : \ T\ A \xrightarrow{\mathrm{FARM}_n\ \sigma} T\ B}$$

$$\frac{\vdash p_1 \ : \ T\ A \xrightarrow{\sigma_1} T\ B \quad \vdash p_2 \ : \ T\ B \xrightarrow{\sigma_2} T\ C}{\vdash p_1 \parallel p_2 \ : \ T\ A \xrightarrow{\sigma_1 \parallel \sigma_2} T\ C}$$

$$\frac{\vdash p \ : \ T\ A \xrightarrow{\sigma} T\ (A + B)}{\vdash \mathsf{fb}\ \ p \ : \ T\ A \xrightarrow{\mathrm{FB}\ \sigma} T\ B}$$

Fig. 2. Structure-annotated type system.

and *map* operations are all instances of *hylomorphisms*, it follows that there is a single, underpinning theoretical formulation for all our skeleton forms, and it also follows that our structured programs may be easily transformed to introduce/remove parallelism in a provably sound way. We will achieve this using a type system.

3.3.2. The structure-annotated type system

We annotate top-level program types with an abstraction of the *parallel structure* of the program, $\sigma \in \Sigma$. Fig. 2 shows our structure-annotated type system. Intuitively, Σ is a "pruned" version of E that retains information about *how* the computation is performed, while removing as many details as possible about *what* is being computed. We once again split its definition into two levels, Σ_{s} for sequential structures, and Σ_{p} for parallel ones.

$$\begin{array}{lcl} \sigma \in \Sigma & ::= & \sigma_{\mathsf{s}} \mid \mathrm{PAR}_F\ \sigma_{\mathsf{p}} \\ \sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::= & \mathrm{A} \mid \sigma \circ \sigma \mid \mathrm{HYLO}_F\ \sigma\ \sigma \\ \sigma_{\mathsf{p}} \in \Sigma_{\mathsf{p}} & ::= & \mathrm{FUN}\ \sigma_{\mathsf{s}} \quad \mid \quad \mathrm{DC}_{n,F}\ \sigma_{\mathsf{s}}\ \sigma_{\mathsf{s}} \\ & \mid & \sigma_{\mathsf{p}} \parallel \sigma_{\mathsf{p}} \quad \mid \quad \mathrm{FARM}_n\ \sigma_{\mathsf{p}} \quad \mid \quad \mathrm{FB}\ \sigma_{\mathsf{p}} \end{array}$$

3.3.3. Convertibility

Our type system needs to include a non-structural rule that captures the *convertibility relation*, $\equiv$, for Σ.

$$\frac{\vdash e \ : \ A \xrightarrow{\sigma_1} B \qquad \sigma_1 \equiv \sigma_2}{\vdash e \ : \ A \xrightarrow{\sigma_2} B}$$

This allows us to transform between structurally equivalent forms. For example, a *sequential composition* can be transformed into a *parallel pipeline* or *vice-versa*, or a *task farm* can be introduced wherever we are using any other skeleton. $\equiv$ is defined in terms of the relations $\equiv_{\mathsf{s}}$ (for sequential operations) and $\equiv_{\mathsf{p}}$ (for parallel skeletons), plus a lifting rule that links the two, PAR-EQUIV.

$$\frac{\sigma_1 \equiv_{\mathsf{s}} \sigma_2}{\sigma_1 \equiv \sigma_2} \qquad \frac{\sigma_1 \equiv_{\mathsf{p}} \sigma_2}{\mathrm{PAR}_F\ \sigma_1 \equiv \mathrm{PAR}_F\ \sigma_2}$$

$$\mathrm{PAR}_F\ (\mathrm{FUN}\ \sigma) \equiv \mathrm{MAP}_F\ \sigma \qquad (\text{PAR-EQUIV})$$

In [1], we have defined a number of equivalences, using well-known properties of skeletons (e.g. [13,15]) plus fundamental hylomorphism laws [7]. For example, a parallel pipeline structure ($\parallel$) is functionally equivalent to a function composition; a task farm FARM can be introduced for any structure; and divide-and-conquer DC and feedback FB skeletons can both be derived from hylomorphisms.

$$\begin{array}{rcll} \mathrm{FUN}\ \sigma_1 \parallel \mathrm{FUN}\ \sigma_2 & \equiv_{\mathsf{p}} & \mathrm{FUN}\ (\sigma_2 \circ \sigma_1) & (\text{PIPE-EQUIV}) \\ \mathrm{DC}_{n,F}\ \sigma_1\ \sigma_2 & \equiv_{\mathsf{p}} & \mathrm{FUN}\ (\mathrm{HYLO}_F\ \sigma_1\ \sigma_2) & (\text{DC-EQUIV}) \\ \mathrm{FARM}_n\ \sigma & \equiv_{\mathsf{p}} & \sigma & (\text{FARM-EQUIV}) \\ \mathrm{FB}(\mathrm{FUN}\ \sigma) & \equiv_{\mathsf{p}} & \mathrm{FUN}\ (\mathrm{ITER}\ \sigma) & (\text{FB-EQUIV}) \end{array}$$

These equivalences, plus *reflexivity, symmetry and transitivity*, define an equational theory that allows conversion between different parallel forms, as well as conversion between structured expressions and parallel forms.

3.4. Automatic rewriting using cost information

We can use the convertibility relation to define a *rewriting system* that allows us to automatically rewrite functionally equivalent structured parallel forms. We interpret structured expressions and parallel processes as members of families of functionally equivalent expressions E_s, indexed by a representative structured expression s. For all well-typed $s \ : \ A \overset{\sigma}{\mapsto} B$, σ is an index of the family defined by s, i.e. $\phi_s(\sigma) \in E_s$. The function $\phi_s \ : \ \Sigma \ \to \ E_s$ is a partial function whose result is defined for any structure σ that is an index of the family E_s. The type-checking algorithm needs to decide whether rewriting s to the desired structure would result in a member of the family E_s. We achieve this by defining a *confluent* rewriting system on structures, $\Sigma \rightsquigarrow \Sigma$ [1]. Since, for a given s, the family E_s is finite, we can use cost information to find optimal structures. For small enough parallel structures, exhaustive search is feasible. For larger structures, we can equip a family E_s with a strong ordering $\sigma_1, \sigma_2 \in \mathrm{dom}(\phi_s)$, $\sigma_1 \ \leq \ \sigma_2$ iff $\mathrm{cost}(\sigma_1) \ \leq \ \mathrm{cost}(\sigma_2)$, and use this ordering to optimise the search process, at the cost of not necessarily finding an optimal structure.[1] In order to exploit this technique, we must provide a cost model that is strongly connected to our execution semantics. This is the focus of the remainder of this paper.

4. An operational semantics for queues

In order to obtain a sound cost model, we need to first give an operational semantics for each of our skeletons, and derive then cost information from this semantics. We will build our skeleton semantics (Section 5) on a small-step trace-based operational semantics for the queues that are used to link these skeletons. Each step in the queue semantics will describe a state transition within a simple parallel process abstraction. We first give a number of basic definitions.

State.

A *state* comprises a tuple of three main structures, $\mathcal{W} \times \mathcal{Q} \times \mathcal{S}$:

- an environment of worker definitions, $\mathcal{W}$, which is a mapping from worker identifiers to *worker loop* definitions, i.e. to the code that is run by each worker of the parallel process;
- an environment of *worker states*, $\mathcal{S}$, which represents the instruction that is currently being executed by the worker; and,
- a *queue environment*, $\mathcal{Q}$, which represents the buffers that link the workers.

[1] We conjecture that a family E_s that is equipped with an ordering relation such that $\sigma_1 \ \leq \ \sigma_2$ *iff* $\sigma_2 \ \rightsquigarrow \ \sigma_1$ forms a lattice, and that we can use this information, together with cost information, to optimise the search for an optimal structure within a family.

We will assume that the worker environment, $\mathcal{W}$, is fixed. The rules in our operational semantics therefore have the form:

$$\boxed{(\mathcal{Q}, \mathcal{S}) \xrightarrow{\alpha} (\mathcal{Q}', \mathcal{S}')}$$

They are given in terms of a labelled state transition system. The labels are the actions $\alpha ::= \mathrm{g}^w q \mid \mathrm{e}^w(f, x) \mid \mathrm{p}^w(x, q)$, which respectively *get* an input from a queue, *evaluate* a function f on an input x, or *put* a result on a queue. Actions are performed on specific queues q, and are tagged with the worker, w, that performs the action.

Queue environments.

A queue environment is a mapping from queue identifiers to sequences of values.

$$\mathcal{Q} = \begin{bmatrix} q_0 & \mapsto \langle x_0,\ x_1, \ldots \rangle \\ \ldots & \\ q_n & \mapsto \langle \ldots \rangle \end{bmatrix}$$

Queue operations.

We assume the usual basic, thread-safe, *enqueue* and *dequeue* operations.

$$\begin{array}{lcl} \texttt{enqueue}\,(\mathcal{Q}[q \mapsto vs], x, q) & \to & \mathcal{Q}[q \mapsto \langle x \mid vs \rangle] \\ \texttt{dequeue}\,(\mathcal{Q}[q \mapsto \langle vs \mid x \rangle], q) & \to & \mathcal{Q}[q \mapsto vs],\ x \end{array}$$

We overload the notation for *enqueue*/*dequeue* operations to also work on sums and products. Enqueuing a product type value into a "product queue" results in a pairwise *enqueue* on each sub-queue. Enqueuing a sum type value into a "sum queue" yields a single *enqueue* operation, on the corresponding queue. We use Q to refer to these *queue structures*, and q to refer to queue identifiers.

$$Q \quad ::= \quad q \mid q_1 \times \cdots \times q_n \mid q_1 + \cdots + q_n.$$

We interpret the *enqueue* operation on products of queues as a sequence of simple *enqueue* operations:

$$\begin{array}{l} \texttt{enqueue}\,(\mathcal{Q}, q_1 \times \cdots \times q_n, (x_1, \ldots, x_n)) \\ \quad = \texttt{enqueue}\left(\texttt{enqueue}\left(\ldots \texttt{enqueue}\,(\mathcal{Q}, q_1, x_1) \ldots\right), q_n, x_n\right) \end{array}$$

An *enqueue* on a sum of queues is an *enqueue* on the corresponding queue. For example, if $0 \leq i \leq n$:

$$\texttt{enqueue}\left(\mathcal{Q}, q_1 + \cdots + q_n, \texttt{inj}_1\, x\right) = \texttt{enqueue}\,(\mathcal{Q}, q_i, x)$$

Dequeue operations work similarly. We interpret the *dequeue* operation on products of queues as a sequence of simple *dequeue* operations:

$$\begin{array}{l} \texttt{dequeue}\,(\mathcal{Q}, q_1 \times \cdots \times q_n) = (\texttt{dequeue}\,(\mathcal{Q}, q_1), \ldots, \\ \quad \texttt{dequeue}\,(\mathcal{Q}, q_1)) \end{array}$$

A *dequeue* operation on a sum of queues dequeues an element from the first non-empty queue. For example, if $0 \leq i \leq n$, and for all j, $0 \leq j < i$, $\mathcal{Q}[q_j \mapsto \langle\rangle]$, and $\mathcal{Q}[q_i \mapsto \langle vs|x\rangle]$, then:

$$\texttt{dequeue}\,(\mathcal{Q}, q_1 + \cdots + q_n) = \texttt{inj}_i\,(\texttt{dequeue}\,(\mathcal{Q}, q_i))$$

This is a purely arbitrary choice. Any alternative ordering (e.g. round-robin) is equally acceptable. In the operational semantics, each action will represent an *enqueue* or *dequeue* operation on a single queue. So dequeuing from a tuple of n queues will result in n *dequeue* actions.

$$\frac{\texttt{dequeue}\,(\mathcal{Q}, q_{n+1}) \to (\mathcal{Q}', v)}{(\mathcal{Q}, (v_1, \ldots, v_n, \texttt{dequeue}\,(q_{n+1}), \ldots)) \xrightarrow{\mathrm{g}^w q_{n+1}} (\mathcal{Q}', (v_1, \ldots, v_n, v, \ldots))}$$

$$\frac{\texttt{dequeue}\,(\mathcal{Q}, q_{n+1}) \to (\mathcal{Q}', v)}{(\mathcal{Q}, (v_1, \ldots, v_n, \texttt{dequeue}\,(q_{n+1}))) \xrightarrow{\mathrm{g}^w q_{n+1}} (\mathcal{Q}', \texttt{eval}\,(v_1, \ldots, v_n, v))}$$

$$\frac{\texttt{enqueue}\,(\mathcal{Q}, q_1, x_1) \to \mathcal{Q}'}{(\mathcal{Q}, \texttt{enqueue}\,(q_1, x_1); \ldots) \xrightarrow{\mathrm{p}^w q_1} (\mathcal{Q}', \ldots)}$$

$$\frac{\texttt{enqueue}\,(\mathcal{Q}, q, x) \to \mathcal{Q}' \qquad \mathcal{W}[w \mapsto \texttt{worker}\,(Q_i, f, Q_o)]}{(\mathcal{Q}, \texttt{enqueue}\,(q, x)) \xrightarrow{\mathrm{p}^w q} (\mathcal{Q}', \texttt{dequeue}\,(Q_i, x))}$$

$$\frac{[\![f]\!](x) \ = \ y \qquad \mathcal{W}[w \mapsto \texttt{worker}\,(Q_i, f, Q_o)]}{(\mathcal{Q}, \texttt{eval}\,(x)) \xrightarrow{\mathrm{e}^w(x)} (\mathcal{Q}, \texttt{enqueue}\,(Q_o, y))}$$

Fig. 3. Transition rules for queue operations.

Workers.

In each iteration, a worker first performs a sequence of *dequeues*, then performs its local computation, f, and finally performs a sequence of *enqueue* operations. We represent this as:

$$\mathcal{W} = \left[\begin{array}{c} \ldots \\ w \mapsto \texttt{worker}\,(Q_i,\ f,\ Q_o) \\ \ldots \end{array}\right]$$

where Q_i and Q_o are the input and output queue structures. The example below shows the state of a worker that is in the process of dequeuing from a tuple of queue identifiers, and that has already *dequeue* d n elements from the first n queues:

$$\mathcal{S} = \left[\begin{array}{c} \ldots \\ w \mapsto (x_1, \ldots, x_n, \texttt{dequeue}\,(q_{n+1}), \ldots, \texttt{dequeue}\,(q_m)) \\ \ldots \end{array}\right]$$

Generally, a worker state can be either a sequence (or sum) of *dequeue*/*enqueue* operations, or an *eval* operation. Let v be a value:

$$\begin{array}{lcl} \texttt{st} & ::= & \texttt{inSt} \mid \texttt{outSt} \mid \texttt{eval}\,(x) \\ \texttt{inSt} & ::= & v \mid \texttt{dequeue}\,(q) \mid (\texttt{inSt}, \ldots, \texttt{inSt}) \\ \texttt{outSt} & ::= & v \mid \texttt{enqueue}\,(q, x) \mid \texttt{outSt}; \ldots; \texttt{outSt} \end{array}$$

The transition rules for our operational semantics are given in Fig. 3. Note that we overload *enqueue*/*dequeue* operations to also deal with sums and products. We also assume that *enqueues* on simple queues are trivial. It is now straightforward to define the operational semantics on a sequence of workers:

$$\frac{(\mathcal{Q}, \texttt{st}) \xrightarrow{\alpha^w} (\mathcal{Q}', \texttt{st}')}{(\mathcal{Q}, \mathcal{S}[w \mapsto \texttt{st}]) \xrightarrow{\alpha^w} (\mathcal{Q}', \mathcal{S}[w \mapsto \texttt{st}'])}.$$

It is easy to see that the only rules that affect the output are those involving communication (*enqueue*/*dequeue*).

Definition 4.1 (*Ready and Idle State*)**.** A worker $w \mapsto \texttt{worker}\ (Q_i, f, Q_o)$ is in a *ready* state if it is at the beginning of its worker loop. A worker is *idle* if it is ready and there are no inputs in its input queue structure Q_i.

Definition 4.2 (*Initial and Final State*)**.** A parallel process is *initial* if all its workers are *ready* and all the queues except the input queue are empty. A parallel process is *final* if all its workers are *idle*.

5. A queue-based operational semantics of algorithmic skeletons

We can now build the parallel structures that we require.

Parallel composition of processes.

We define the parallel composition of processes as the union of their queues and workers.

$$(\mathcal{W}_1, \mathcal{Q}_1, \mathcal{S}_1) \parallel (\mathcal{W}_2, \mathcal{Q}_2, \mathcal{S}_2) = (\mathcal{W}_1 \uplus \mathcal{W}_2, \mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{S}_1 \uplus \mathcal{S}_2)$$

Worker.

We lift sequential functions into workers with input and output queue structures. The new worker is *ready*.

$$\begin{aligned}&\mathtt{qfun}(f)(Q_0, Q_1) = \\ &\quad \mathcal{W} = [w \mapsto \mathtt{worker}\,(Q_0, f, Q_1)]\,,\ \mathcal{Q} = Q_0 \cup Q_1,\\ &\quad \mathcal{S} = [w \mapsto \mathtt{dequeue}\,(Q_0)]\end{aligned}$$

Task farm.

A task farm replicates a structure n times:

$$\mathtt{qfarm}(n, \mathcal{P})(Q_0, Q_1) = \overbrace{\mathcal{P}(Q_0, Q_1) \parallel \ldots \parallel \mathcal{P}(Q_0, Q_1)}^{n \text{ times}}$$

Parallel pipeline.

A parallel pipeline is the parallel composition of two structures, linked by an intermediate queue. Let q be a fresh queue identifier:

$$\mathtt{qpipe}(\mathcal{P}_1, \mathcal{P}_2)(Q_0, Q_1) = \mathcal{P}_1(Q_0, q) \parallel \mathcal{P}_2(q, Q_1).$$

Feedback loop.

A feedback loop is created by simply inspecting the output of a structure, and placing it back in the input queue depending on the result:

$$\mathtt{qfeedback}(\mathcal{P})(Q_0, Q_1) = \mathcal{P}(Q_0, Q_0 + Q_1).$$

Parallel hylomorphism.

One of the novelties of this paper lies in describing how to parallelise an arbitrary *hylomorphism* using a divide-and-conquer skeleton. We will first describe a series of simple semantics-preserving transformations for any hylomorphism. The idea is that if the *anamorphism* part of a hylomorphism needs to split an input value into at most n sub-values, then we will create $n + 1$ queues, the first of which will send the corresponding input to the "combine" worker, and the remaining n of which will send the subdivided inputs to the subsequent divide stages. If an input cannot be divided any further, then a synchronisation token, 1, will be sent. Given a functor F described as a combination of sums, products and constants, $F\ A\ =\ T$, where A is nested at most n times within a product in T, we define the functor D_F as $D_F\ B\ =\ (1 + T[1/A]) \times \overbrace{(B \times \cdots \times B)}^{n \text{ times}}$. If we can convert any hylomorphism to this new structure, then we can use its regular structure to create a regular "divide-and-conquer graph" with the following communication structure:

1. The *divide* worker will communicate a value of type $1 + T[1/A]$ to the corresponding *combine* worker, and values of type B to the subsequent *divide* workers.
   - A value of type $1+T[1/A]$ contains the "non-recursive" part of T, plus a unit to indicate that the input could not be divided any further.
   - A *combine* worker can use a value of this type to decide how to recombine the n values of type B that has been received from the previous stages of the divide-and-conquer skeleton.
2. A *divide* worker takes an element of type $1 + A$. If it is 1, then it transmits $\mathtt{inj}_1\ ()$ to all its output queues. This indicates to the subsequent workers that there is no more work to be done. If it receives a value of type A, it divides it, splits the recursive and non-recursive parts of T, and sends the corresponding elements to the output queues.
3. A *combine* worker that receives an element of type 1 from the corresponding *divide* worker will simply discard all values received from the previous level, and send 1 to the *combine* worker of the next level. If it receives an element of type $T[1/A]$, it will need to recombine the corresponding value of type $F\ A$ from the inputs, and apply the *combine* function to it.

We need a way to change a hylomorphism from an F structure to D_F. Note that there are two functions

$$\begin{aligned}d\ &:\ 1 + F\ A \to D_F\ (1 + A)\\ c\ &:\ D_F\ (1 + A) \to 1 + F\ A.\end{aligned}$$

The d function separates the occurrences of values of type A in some structure $F\ A$ into the corresponding $T[1/A]$ and n-tuple of $1 + A$, and the function c recomposes a structure $F\ A$ from the structure of D_F. If d receives $\mathtt{inj}_1\ ()$, then it just returns a $n + 1$ tuple of $\mathtt{inj}_1\ ()$. The c function returns $\mathtt{inj}_1\ ()$ if the first component of the tuple $D_F\ (1 + A)$ is $\mathtt{inj}_1\ ()$. Note that the types F and D_F are not isomorphic, since the function c is *partial*. However, the following properties do hold:

$$\begin{aligned}c \circ d\ &=\ \mathtt{id}\\ d \circ (\underline{1} + F f)\ &=\ D_F\ (\underline{1} + f) \circ d.\end{aligned}$$

Using these properties, we can show that the following condition holds for any functor F:

$$\begin{aligned}&\underline{1} + \mathtt{hylo}_F\ g\ h\\ &=\\ &\underline{1} + g\ \circ\ F\ (\mathtt{hylo}_F\ g\ h) \circ\ h\\ &=\\ &(\underline{1} + g)\ \circ\ (\underline{1} + F\ (\mathtt{hylo}_F\ g\ h)) \circ\ (\underline{1} + h)\\ &=\\ &((\underline{1} + g) \circ c \circ d)\ \circ\ (\underline{1} + F\ (\mathtt{hylo}_F\ g\ h)) \circ\ (\underline{1} + h)\\ &=\\ &((\underline{1} + g) \circ c) \circ\ D_F\ (\underline{1} + \mathtt{hylo}_F\ g\ h)\ \circ\ (d \circ (\underline{1} + h))\\ &=\\ &\mathtt{hylo}_{D_F}\ ((\underline{1} + g) \circ c)\ (d \circ (\underline{1} + h))\end{aligned}$$

Using this, we can convert to a hylomorphism of a different structure. For all $\mathtt{hylo}_F\ g\ h\ :\ A \to B$, given any $x\ :\ B$, then

$$\begin{aligned}&\underline{x} \triangledown \mathtt{id} \circ (\mathtt{hylo}_{D_F}\ ((\underline{1} + g) \circ c)\ (d \circ (\underline{1} + h))) \circ\ \mathtt{inj}_2\\ &=\\ &\underline{x} \triangledown \mathtt{id} \circ (\underline{1} + \mathtt{hylo}_F\ g\ h) \circ\ \mathtt{inj}_2\\ &=\\ &(\underline{x} \circ \underline{1}) \triangledown (\mathtt{id} \circ \mathtt{hylo}_F\ g\ h) \circ\ \mathtt{inj}_2\\ &=\\ &\mathtt{id} \circ \mathtt{hylo}_F\ g\ h\\ &=\\ &\mathtt{hylo}_F\ g\ h\end{aligned}$$

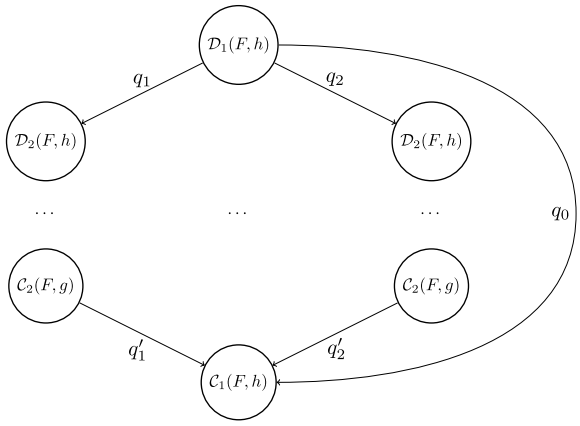


Fig. 4. Divide-and-conquer skeleton: the q_i are the queues; circles represent the workers.

This shows that the first level of a divide-and-conquer must wrap the input in $\mathtt{inj}_2$, and the last level of a divide-and-conquer must unwrap the result using $\underline{x}\triangledown\mathtt{id}$, for any arbitrary $x : B$. We define these as:

$$\mathcal{D}_1(F,h) = d \circ \mathtt{inj}_2 \circ h \qquad \mathcal{C}_2(F,g) = (\underline{1}+g)\circ c$$
$$\mathcal{D}_2(F,h) = d \circ (\underline{1}+h) \qquad \mathcal{C}_1(F,g) = (\underline{x}\triangledown\mathtt{id})\circ(\underline{1}+g)\circ c$$

Although the structure D_F may appear to be complicated, it neatly fits our queue-based model, in that we can create queues that send/receive values of types $(1+T[1/A])$ and $1+A$, and can create workers that split/combine values of these types, as shown in Fig. 4. Values of type 1 are simply used to synchronise the different *divide* and *combine* workers, so that the different levels of a divide-and-conquer skeleton can operate on different inputs in parallel. We define this as:

$$\mathtt{qdc}(n,F,g,h)(Q_0,Q_1) = \mathtt{qdc}'(\mathtt{hylo}_F g\ h,\ n,\ F,\ g,\ h,\ \mathcal{C}_1,\ \mathcal{D}_1)\ (Q_0,Q_1)$$
$$\mathtt{qdc}'(f,0,F,g,h,c,d)(Q_0,Q_1) = \mathtt{qfun}(f)(Q_0,Q_1)$$
$$\begin{aligned}\mathtt{qdc}'(f,n,F,g,h,c,d)(Q_0,Q_1) = {}& \\ & \mathtt{qfun}(d(F,h))(Q_0, q_0\times q_1\times\cdots\times q_n)\\ & \parallel \mathtt{qdc}'(\underline{1}+\mathtt{hylo}_F g\ h, n-1, F,g,h,\mathcal{C}_2,\mathcal{D}_2)(q_1,q_1')\\ & \parallel \ldots\\ & \parallel \mathtt{qdc}'(\underline{1}+\mathtt{hylo}_F g\ h, n-1, F,g,h,\mathcal{C}_2,\mathcal{D}_2)(q_n,q_n')\\ & \parallel \mathtt{qfun}(c(F,g))(q_0\times q_1'\times\cdots\times q_n', Q_1).\end{aligned}$$

Soundness.

It is obvious from these definitions that the soundness of the operational semantics with respect to the denotational semantics can be reduced to correctly connecting the queues of the algorithmic skeletons.

6. Predicting parallel performance

We will now formally derive a set of cost equations from our operational semantics in a systematic way. These cost equations can then be used by our type-system to derive cost-models for the high-level parallel structures. This gives two main benefits: (i) the cost models are sound w.r.t. the operational semantics *by construction*; and (ii) this provides a way to *automatically* derive cost equations for newly defined parallel structures. Defining this automatic process is out of the scope of this paper, but we will discuss later how this can be realised as an algorithm. In this paper, we are mainly interested in predicting realistic *lower bounds* on the execution times of parallel programs. While the more usual *worst-case* predictions might be useful for safety reasons, they would not provide sufficient information to enable us to choose the best parallel implementation. Moreover, when "initialisation" and "finalisation" times are taken into account, the worst-case parallel execution time will generally be very similar to the sequential case, and so provide very little insight into parallel performance. The *amortised* average case timings that we use here are much more useful for predicting the actual parallel performance of the system.

6.1. Costs and sizes

The sequential components of our algorithmic skeletons are given as suitably lifted functions. Before we can define their cost models, we must first explain how we derive cost models for hylomorphisms. Vasconcelos [16] showed how to use sized types [17] for developing an accurate space cost analysis. Sized types have also been used for estimating upper bounds of execution times of parallel functional programs [18]. The success on using sized types for cost analysis motivated our approach. We exploit this previous research on *sized types* [16], as well as previous research on *cost equations* [19]. A *cost equation* for a function $f : A \to B$ provides an estimate of the execution time of f given an input of a given size.

$$\mathtt{cost}_f : \mathtt{size}_A \to \mathtt{time}$$

To obtain the inputs for these cost equations, we use a variant of the usual notion of *sized types* [17]. The only difference is that, since we are interested in amortised costs, we use *average sizes* rather than upper or lower bounds. The notion of *average size* is dependent on each problem, so rather than defining a generic calculation, we require specific definitions for each function and *type constructor*. Suitable definitions include, e.g. the average depth of a tree, the average number of elements in a structure etc.

Type constructors.

These are either constant types C, or recursive types defined as the fixpoint of some base functor μF.

Sizes and size constraints.

We reuse the idea of *stages* [20]. Essentially, sizes are either size variables or sums of sizes, and size constraints are inequalities on sizes. We write $A^i \mid C$ for a sized type A with size i and constraint C.

Sized types.

We require the basic polynomial (bi-) functors to be annotated with size expressions for each alternative (given as a sum-type). For example, the base bifunctor of a binary tree can be annotated as follows:

$$F^{0\vee 1+i+j}\,A\,B = 1 + A\times B^i\times B^j.$$

The size expression $0\vee 1+i+j$ states that functor F either has size 0, i.e. it contains no elements of type A; or it has size $1+i+j$, i.e. it has one element of type A, plus the sizes of the elements of type B, one of size i and the other of size j. Note that there are alternative definitions for the size of a functor F, e.g: $F^{1\vee\max(i,j)}\,A\,B = 1 + A\times B^i\times B^j$. In a more general sense, a sized-type is a type constructor that is annotated with size information, e.g. Int^i, $List^{j+k}\,A$, ... The $\mathtt{in}_F$ and $\mathtt{out}_F$ functions for a sized base functor of a recursive type must have the following type and constraints:

$$\mathtt{in}_F : F^j\mu F \to \mu^i F \mid i = j$$
$$\mathtt{out}_F : \mu^i F \to F^j\mu F \mid i = j.$$

That is, we require that the sizes of the base functor represent the same information as the sizes of the recursive type. Finally, we will use $|A|$ to denote all the nested size information and size constraints in a type A. This is useful for defining cost models that require access to nested size information.

$$
\begin{aligned}
&\mathtt{cost}(\text{FUN}\ \sigma^{n,m}) && = T_{\text{enqueue}}(n) + \mathtt{cost}\, i + T_{\text{dequeue}}(m)\\
&\quad \text{where } |\sigma| = i \to j \mid C\\
&\mathtt{cost}(\text{FARM}\ n\ \sigma^{i,o}) && = \frac{\mathtt{cost}(\sigma^{i\times n, o\times n})}{n}\\
&\mathtt{cost}(\sigma_1^{i,j} \parallel \sigma_2^{k,l}) && = \max\left\{\mathtt{cost}(\sigma_1^{i,j+k}), \mathtt{cost}(\sigma_2^{j+k,l})\right\}\\
&\mathtt{cost}(\text{FB}\ \sigma^{n,m}) && = \mathtt{if}\ (|\sigma| = 0)\\
&&& \quad \mathtt{then}\ \mathtt{cost}(\sigma^{n+m,m})\\
&&& \quad \mathtt{else}\ \mathtt{cost}(\sigma^{n+m,m}) + \mathtt{cost}(\text{FB}\ (\mathtt{resize}(\sigma^{n,m}, |\sigma|)))\\
&\mathtt{cost}(\text{DC}_{n,F}\ \sigma_1\ \sigma_2) && = \max\left\{\begin{array}{l}\mathtt{cost}(\mathcal{D}(F,\sigma_2)),\\ \mathtt{cost}(\text{DC}_{n-1,F}\ \sigma_1\ \sigma_2),\\ \mathtt{cost}(\mathcal{C}(F,\sigma_1))\end{array}\right\}
\end{aligned}
$$

Fig. 5. Cost equations.

Deriving recurrences from hylomorphisms.

We now show how we use sized types to derive recurrences. Essentially, we will define a *cost equation* for each syntactic construct, that takes some cost equation parameters and produces another cost equation. Although the cost of a function is not, in general, compositional, since it may depend on previous computations as well as on other properties of the input data, we will here take a compositional approach under the assumption that only the sizes will affect the execution times. This is a valid way to obtain *amortised* costs. The execution time of each primitive operation is assumed to be a constant that depends on the target architecture. This must be provided as a parameter. The cost of a function composition is the sum of the run-times of each stage, plus some architecture-dependent overhead. The only difficult case involves recursive functions. For recursive functions that are defined as hylomorphisms, we generate recurrence relations, as with Barbosa et al. [21]. In a similar sense to Sands' work [19], our cost equations can be thought of as *functions* that we are integrating into our type-system. For example, assuming that we have the following sized-type for *quicksort*:

$$
\begin{aligned}
&\mathtt{merge} \ :\ T^{0\vee 1+i+j}A\,(\mathsf{List}\,A) \ \to\ \mathsf{List}^{k}\,A \mid k = 0 \ \vee\ k = i+1+j\\
&\mathtt{split} \ :\ \mathsf{List}^{n}\,A \ \to\ T^{0\vee 1+r+s}A\,(\mathsf{List}\,A) \mid n = 0 \ \vee\ n = 1+r+s
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{qs} \ :\ \mathsf{List}^{i}\,A \ \to\ \mathsf{List}^{j}\,A \mid i = j\\
&\mathtt{qs} \ =\ \mathtt{hylo}_{T}\ \mathtt{merge}\ \mathtt{split}
\end{aligned}
$$

Given suitable cost equations for *split* and *merge*, $\mathtt{cost}_{\mathtt{split}}$, $\mathtt{cost}_{\mathtt{merge}}$, then there are two cases. Either $n = 0$, in which case $t = 0$, so we assume some time $\mathtt{cost}_{\mathtt{split}}(0) + \mathtt{cost}_{\mathtt{merge}}(0)$, or $n = 1 + r + s$, in which case:

$$
\begin{aligned}
\mathtt{cost}_{\mathtt{qs}} \ =\ & \mathtt{cost}_{\mathtt{split}}(1+r+s) + \mathtt{cost}_{\mathtt{merge}}(r+1+s)\\
& + \mathtt{cost}_{\mathtt{qs}}(r) + \mathtt{cost}_{\mathtt{qs}}(s)
\end{aligned}
$$

To complete the cost equation, we need now to relate r and s with the input size n. By assuming that these sizes correspond to a balanced computation, we can then automatically calculate an *amortised* cost. Taking more extreme cases into account would, of course, require further programmer input.

$$
\begin{aligned}
\mathtt{cost}_{\mathtt{qs}}(n) \ =\ & \mathtt{cost}_{\mathtt{split}}(1+r+s) + \mathtt{cost}_{\mathtt{merge}}(r+1+s)\\
& + \mathtt{cost}_{\mathtt{qs}}(r) + \mathtt{cost}_{\mathtt{qs}}(s)\\
\text{where } r = (n-1)/2 \text{ and } s = (n-1)/2 &
\end{aligned}
$$

We simplify this internally to generate the desired recurrence relation:

$$
\begin{aligned}
\mathtt{cost}_{\mathtt{qs}}(0) \ =\ & \mathtt{cost}_{\mathtt{split}}(0) + \mathtt{cost}_{\mathtt{merge}}(0)\\
\mathtt{cost}_{\mathtt{qs}}(n) \ =\ & \mathtt{cost}_{\mathtt{split}}(n) + \mathtt{cost}_{\mathtt{merge}}(n) + 2\\
& \times \mathtt{cost}_{\mathtt{qs}}((n-1)/2)
\end{aligned}
$$

6.2. Costing traces of parallel processes

We now describe a systematic way to derive cost equations. We start with a structure C, with some initial empty $\mathcal{P}_i$ and cost c_i. Taking suitably sound simplifications and approximations of $\mathtt{time}(\alpha_1, \ldots)$, we then derive an "amortised cost equation" $\mathtt{cost}_C$ such that for all input l with sized-type $\langle A\rangle^i$ and trace, $C(\mathcal{P}_1, \ldots, \mathcal{P}_n)(q_{in} \mapsto l, q_{out} \mapsto \langle\rangle) \xrightarrow{\alpha_1,\ldots} C(\mathcal{P}_1, \ldots, \mathcal{P}_n)(q_{in} \mapsto \langle\rangle, q_{out} \mapsto l')$, then $i \times \mathtt{cost}_C(c_1, \ldots, c_n)\,|A| \approx \mathtt{time}(\alpha_1, \ldots)$. We differentiate three phases in the execution of a parallel process: an *initialisation phase*, a *steady state*, and a final *flushing phase*. For example, a pipeline of two atomic functions, $w_1 \parallel w_2$, reaches a *steady state* after executing the *initialisation phase* of w_1. At this point, w_2 can run in parallel with w_1:

$$
\left(\begin{bmatrix} q_0 & \mapsto \langle x_1,\ x_2,\ \ldots\rangle,\\ q_1 & \mapsto \langle\rangle,\\ q_2 & \mapsto \langle\rangle \end{bmatrix}, \begin{bmatrix} w_1 & \mapsto \mathtt{worker}\,(q_0, f, q_1)\\ w_2 & \mapsto \mathtt{worker}\,(q_1, g, q_2)\end{bmatrix}\right)
\xrightarrow{\mathrm{g}^{w_1} q_0,\ \mathrm{e}^{w_1} x_1,\ \mathrm{p}^{w_1} q_1}
\left(\begin{bmatrix} q_0 & \mapsto \langle x_2,\ \ldots\ \rangle,\\ q_1 & \mapsto \langle f\ x_1\rangle,\\ q_2 & \mapsto \langle\rangle \end{bmatrix}, \begin{bmatrix} w_1 & \mapsto \mathtt{worker}\,(q_0, f, q_1)\\ w_2 & \mapsto \mathtt{worker}\,(q_1, g, q_2)\end{bmatrix}\right)
$$

We capture these ideas in the definitions below.

Definition 6.1 (*Steady State*)**.** A parallel process $\mathcal{P}$ is in a steady state, $\mathtt{steady}(\mathcal{P})$, if for all $w \in \mathcal{P}$, $\neg\mathtt{idle}(w)$.

Definition 6.2 (*Initialisation Phase*)**.** The initialisation phase for a parallel process $\mathtt{initial}(\mathcal{P}_1)$ is the shortest sequence of $a_1, a_2, \ldots, a_n$, such that if $\mathcal{P}_1 \xrightarrow{a_1,a_2,\ldots,a_n} \mathcal{P}_2$, then $\mathtt{steady}(\mathcal{P})$.

Definition 6.3 (*Flushing Phase*)**.** The flushing phase for a parallel process $\mathcal{P}_1$ is the sequence of $a_1, a_2, \ldots, a_n$, such that for all $i \in [1 \ldots n]$ and for all $w \in \mathcal{P}_1$, $a_i \neq \mathtt{dequeue}(q_{in})^w$, and if $\mathcal{P}_1 \xrightarrow{a_1,a_2,\ldots,a_n} \mathcal{P}_2$, then $\mathtt{final}(\mathcal{P}_2)$.

Total cost and amortised cost:.

Given some initial $\mathcal{P}$ and final $\mathcal{P}'$, where $\mathcal{P} \xrightarrow{\alpha_1,\ldots,\alpha_n} \mathcal{P}'$, the total cost of a parallel computation is $\mathtt{time}(\alpha_1, \ldots, \alpha_n)$. The cost of each action depends on the worker environment. Based on [14], we calculate the queue contention on each queue by simply counting the number of workers in which a queue appears in $\mathcal{W}$. Then, the cost of the g^w and p^w is adjusted according to the corresponding overhead. If we split the trace into $\mathcal{P} \xrightarrow{\mathtt{init}} \mathcal{P}_1 \xrightarrow{\mathtt{steady}} \mathcal{P}_2 \xrightarrow{\mathtt{flush}} \mathcal{P}'$, where $\mathtt{init} = \alpha_1, \ldots, \alpha_i$, $\mathtt{steady} = \alpha_i, \ldots, \alpha_j$, $\mathtt{flush} = \alpha_j, \ldots, \alpha_n$ such that $\mathtt{initial}(\mathcal{P})$, $\mathtt{steady}(\mathcal{P}_1)$, and $\mathtt{final}(\mathcal{P}')$ then the total time is: $\mathtt{time}(\alpha_1, \ldots, \alpha_n) = \mathtt{time}(\mathtt{init}) + \mathtt{time}(\mathtt{steady}) + \mathtt{time}(\mathtt{flush})$. We can systematically derive cost models for $\mathtt{time}(\mathtt{init})$ and $\mathtt{time}(\mathtt{flush})$ using the method shown below.

6.3. Deriving cost equations from the operational semantics

We now show how to systematically derive the cost equations in Fig. 5 from our operational semantics for skeletons. We base our approach on symbolic execution. The following assumptions are used to automatically derive amortised cost equations for our parallel structures.

1. The queues contain enough elements for each *dequeue* to succeed.
2. Each time an element is dequeued, we will return a size that is extracted from its type.

3. Evaluating a function on a size immediately returns the size of the output type of that function, plus the set of constraints that relate the input and output sizes.
4. An *enqueue* operation always succeeds, and has no effect on the queues.
5. Any substructure (e.g. a farm worker) produces a trace that can be safely interleaved with the trace that is produced by other substructures (i.e. without modifying the result of the overall computation), and that has a known cost.

Note that assumption 1 holds only for a `steady` structure, so these cost equations would not be useful for estimating run-times of a computation where `time(init)` and/or `time(flush)` dominate.

Worker cost.

A worker, w, computing $f \;:\; A^i \;\to\; B^j \mid C$ in environment $\mathcal{W}[w \mapsto \texttt{worker}\,(q_i, f, q_o)]$ produces the "symbolic trace": $\mathrm{g}^w q_i,\ \mathrm{e}^w\ |A^i|,\ \mathrm{p}^w q_o$. Assuming there exists some trace $\alpha_1, \ldots, \alpha_n$ that can be safely interleaved with this trace, we want to know the cost of the actions in $\alpha_1, \ldots, \mathrm{g}^w q_i, \ldots, \mathrm{e}^w\, i, \ldots, \mathrm{p}^w q_o, \ldots, \alpha_n$. Here, the cost of $\mathrm{g}^w q_i$ will depend on any other action that is happening in parallel. The only actions that can affect the cost of a queue operation are other queue operations that are acting on the same queue. If n is the number of workers $w_{\text{get}} \in \mathcal{W}$ such that $w_{\text{get}} \neq w_i$ and the number of workers operating on q_o is m, then if we assume that the cost of the *enqueue* and *dequeue* operations is as described by [14], then the cost is:

$$\texttt{time}(\mathrm{g}^w q_i,\ \mathrm{e}^w\ i,\ \mathrm{p}^w q_o) \;=\; T_{\text{enqueue}}(n) + \texttt{cost}_f(i) + T_{\text{dequeue}}(m)$$

We associate this cost with the corresponding high-level structure, so that it can be used by our type-checking algorithm:

$$\texttt{cost}(\textsc{fun}\sigma) \;=\; T_{\text{enqueue}}(n) + \texttt{cost}(\sigma) + T_{\text{dequeue}}(m)$$

The parameters n, m and $\texttt{cost}(\sigma)$ can be instantiated from the context, and added to the structure σ by the type-checking algorithm in a straightforward way. We write $\sigma^{n,m}$ for a structure with n contending workers in the input queue, and with m contending workers in the output queue. Note that the real cost, understood as an equation from an input size to an execution time, would be $\texttt{cost}(\textsc{fun}\sigma^{n,m})(T^i\, j) \;=\; i \times (T_{\text{enqueue}}(n) + \texttt{cost}_\sigma(j) + T_{\text{dequeue}}(m))$, if we assume that T contains i elements of size j. Since we can annotate structures, σ, with sizes, we can abuse our notation for extracting sizes of types, $|\sigma|$, and "overload" the meaning of `cost` to take a structure and yield an amortised cost.

Farm cost.

A farm, $\mathcal{C}(Q_0, Q_1) \parallel \cdots \parallel \mathcal{C}(Q_0, Q_1)$, consists of a number of parallel processes that are joined using our parallel composition operator, and that share input and output queues. Since we symbolically evaluate the *enqueue* and *dequeue* operations, we can take n arbitrary traces, one for each farm worker:

$$\mathcal{C}(Q_0, Q_1) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}', \quad \ldots, \quad \mathcal{C}(Q_0, Q_1) \xrightarrow{\alpha_1^n, \alpha_2^n, \ldots} \mathcal{P}'.$$

Each trace has cost $c_i, \ldots, c_n$. To obtain amortised costs, we assume that each of these values is equal to some average cost c. Because we assume that these actions can be safely interleaved, and because the cost of the workers already considers the queue contention, we simply need to calculate the maximum cost of any worker, assuming that each sub-trace can be performed in parallel:

$$\begin{aligned}\texttt{time}(\alpha_a^i, \alpha_b^j, \ldots) &= \max \begin{Bmatrix} \texttt{time}(\alpha_1^1, \alpha_2^1, \ldots), \\ \texttt{time}(\ldots), \\ \texttt{time}(\alpha_1^n, \alpha_2^n, \ldots) \end{Bmatrix} \\ &= \max\{c_1, \ldots, c_n\} \;=\; c.\end{aligned}$$

Note that if each $\mathcal{P}$ produces k outputs, then $\texttt{qfarm}(n, \mathcal{P})$ produces $k \times n$ outputs. So, in order to obtain an amortised cost, when we associate it with the high-level structure, we need to divide the total cost by n.

$$\texttt{cost}(\textsc{farm}\; n\, \sigma) \;=\; \frac{\texttt{cost}(\sigma)}{n}.$$

Note that, although this cost is obviously correct, the main point is that it was *systematically* derived from the simple queue-based model. It is thus *sound by construction*, and so no longer needs to be a parameter of our type system.

Parallel pipeline.

A pipeline, $\mathcal{C}_1(Q_0, q) \parallel \mathcal{C}_2(q, Q_1)$, consists of two parallel processes that are joined using our parallel composition operator, and connected using an intermediate queue q. Again, we take a similar reasoning process:

$$\mathcal{C}_1(Q_0, q) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}_1 \qquad \mathcal{C}_2(q, Q_1) \xrightarrow{\alpha_1^2, \alpha_2^2, \ldots} \mathcal{P}_2.$$

Assume costs c_1,and c_2 for each process:

$$\texttt{time}(\alpha_a^i, \alpha_b^j, \ldots) \;=\; \max \begin{Bmatrix} \texttt{time}(\alpha_1^1, \alpha_2^1, \ldots), \\ \texttt{time}(\alpha_1^2, \alpha_2^2, \ldots) \end{Bmatrix} \;=\; \max\{c_1, c_2\}.$$

Note that, in order to accurately lift these costs to the high-level structures, we need to consider the size of the output of α_i^1, not the size of the α_j^2 input. We do this by writing $\texttt{resize}(\sigma_2, |\sigma_1|)$, meaning that the input of σ_2 is altered to have the size of the output size in $|\sigma_1|$. We can do this safely since our type-checking algorithm can ensure that sizes meet the necessary constraints. Finally, we associate the cost with the corresponding high-level structure:

$$\texttt{cost}(\sigma_1 \parallel \sigma_2) \;=\; \max\{\texttt{cost}(\sigma_1), \texttt{cost}(\texttt{resize}(\sigma_2, |\sigma_1|))\}$$

Feedback loop.

A feedback loop requires us to take into consideration that an element may be written back to the input queue, $\mathcal{C}_1(Q_0, Q_0 + Q_1)$. The structure C must compute some function f, and we require it to have type:

$$F^{n \vee 0}\, A \;=\; A^n + B, \quad \text{s.t.} \quad f : A^i \to F\, A^j \mid j < i.$$

Since we require j to be strictly smaller than i, we can estimate the number of steps that are required until $i = 0$ to be n, *i.e.* the average number of times an element will need to be put back into the input queue. Basically, a trace $\mathcal{C}_1(Q_0, Q_0 + Q_1) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}_1$ will need to be taken n times to ensure that, on average, at least one element is enqueued into output queue Q_1.

$$\mathcal{C}_1(Q_0, Q_0 + Q_1) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}_1 \xrightarrow{\cdots} \mathcal{P}_n.$$

Since this parameter can be estimated from the sized types, we can again assume that a structure σ is parameterised on it, and can use this in our high-level cost models. Again, we need to use `resize` to generate the appropriate cost equation:

$$\begin{aligned}\texttt{cost}(\textsc{fb}\,\sigma) \;=\; \texttt{if} \quad & (|\sigma| = 0) \\ & \texttt{then}\ \texttt{cost}(\sigma) \\ & \texttt{else}\ \texttt{cost}(\sigma) \;+\; \texttt{cost}(\textsc{fb}\,(\texttt{resize}(\sigma, \sigma)))\end{aligned}$$

Divide-and-conquer.

The divide-and-conquer skeleton requires a little more work, since we first need to transform the structure so that it matches the skeleton. Since this transformation can be done in a fairly

standard way, we initially focus on the cost of the divide-and-conquer skeleton:

$$\begin{aligned}
\mathtt{qdc}'(0,\ F,\ g,\ h)(Q_0, Q_1) &= \mathtt{qfun}(\underline{1} + \mathtt{hylo}_F g\ h)(Q_0, Q_1)\\
\mathtt{qdc}'(n,\ F,\ g,\ h)(Q_0, Q_1) &= \mathtt{qfun}(\mathcal{D}_2(F, h))\\
&\quad (Q_0, q_0 \times q_1 \times \cdots \times q_m)\\
&\quad \| \ \mathtt{qdc}'(n-1, F, g, h)(q_1, q_1')\\
&\quad \| \ \ldots\\
&\quad \| \ \mathtt{qdc}'(n-1, F, g, h)(q_m, q_m')\\
&\quad \| \ \mathtt{qfun}(\mathcal{C}_2(F, g))\\
&\quad (q_0 \times q_1' \times \cdots \times q_m', Q_1)
\end{aligned}$$

Since we define this skeleton inductively for some "depth" n, we start with the base case (depth 0), which is equivalent to the cost of an atomic function:

$$\mathtt{cost}(\mathrm{DC}_{0,F}\ \sigma_1\ \sigma_2) = \mathtt{cost}(\mathrm{FUN}\ (\mathrm{HYLO}_F\ \sigma_1\ \sigma_2))$$

For the recursive case, we assume $2 + m$ traces, one for each "recursive" case, plus the trace of the divide and the trace of the combine parts. Again, we assume that the traces can be safely interleaved, so we can calculate the cost of the total trace as:

$$\begin{aligned}
\mathtt{time}(\alpha_1, \ldots) &= \max \left\{ \begin{array}{l} \mathtt{time}(\alpha^{\mathrm{div}}, \ldots),\\ \mathtt{time}(\alpha^{\mathrm{qdc}}, \ldots),\\ \mathtt{time}(\ldots),\\ \mathtt{time}(\alpha^{\mathrm{qdc}}, \ldots),\\ \mathtt{time}(\alpha^{\mathrm{conq}}, \ldots) \end{array} \right\}\\
&= \max \left\{ \begin{array}{l} \mathtt{time}(\alpha^{\mathrm{div}}, \ldots),\\ \mathtt{time}(\alpha^{\mathrm{qdc}}, \ldots),\\ \mathtt{time}(\alpha^{\mathrm{conq}}, \ldots) \end{array} \right\}
\end{aligned}$$

We can then associate this cost equation with the costs of the high-level structures, by simply substituting the cost of the relevant trace. Note that the elements in the queues decrease in size for each level that we descend into the divide-and-conquer structure. Assuming that our structures are annotated with sizes, we need to update the sizes accordingly in the recursive call to the cost of a divide-and-conquer structure, and in the combine part. Although we can once again use `resize`, we assume that the actual sizes have been correctly updated in the DC structure:

$$\mathtt{cost}(\mathrm{DC}_{n,F}\ \sigma_1\ \sigma_2) = \max \left\{ \begin{array}{l} \mathtt{cost}(\mathcal{D}(F, \sigma_2)),\\ \mathtt{cost}(\mathrm{DC}_{n-1,F}\ \sigma_1\ \sigma_2),\\ \mathtt{cost}(\mathcal{C}(F, \sigma_1)) \end{array} \right\}$$

7. Examples

This section (a) illustrates the cost equations that can be derived from some parallel structures, as they are inferred by the type system; and, (b) compare the costs that are obtained by cost models for some simple structures with actual runtimes, in order to provide evidence for the feasibility of our approach.

7.1. Type-based derivation of cost equations

7.1.1. Image merge

Image merge basically composes two functions: `mark` and `merge`. It can be directly parallelised using different combinations of farms and pipelines.

$$\begin{aligned}
&\mathrm{IM}\ n = \mathrm{PAR}_L\ (m_1 \parallel \mathrm{FARM}\ n\ m_2)\\
&\mathtt{imgMerge} : \mathsf{List}^k(\mathtt{Img} \times^{i+j} \mathtt{Img}) \overset{\mathrm{IM}\ n}{\longmapsto} \mathsf{List}^k(\mathtt{Img}^{\max(i,j)})\\
&\mathtt{imgMerge} = \mathtt{map}_{\mathsf{List}}\ (\mathtt{merge} \circ \mathtt{mark}).
\end{aligned}$$

The type system tries the following substitutions for m_1 and m_2 in IM:

$$\begin{aligned}
\delta &= \{m_1 \sim \mathrm{FUN\ A}, && m_2 \sim \mathrm{FUN\ A}\}\\
\delta_1 &= \{m_1 \sim \mathrm{FARM}\ n_1\ (\mathrm{FUN\ A}), && m_2 \sim \mathrm{FARM}\ n_2\ (\mathrm{FUN\ A})\}\\
\delta_2 &= \{m_1 \sim \mathrm{FUN\ A}, && m_2 \sim \mathrm{FARM}\ n_2\ (\mathrm{FUN\ A})\}\\
\delta_3 &= \{m_1 \sim \mathrm{FARM}\ n_1\ (\mathrm{FUN\ A}), && m_2 \sim \mathrm{FUN\ A}\}\\
\delta_4 &= \{m_1 \sim \mathrm{FUN\ A}, && m_2 \sim \mathrm{FUN\ A}\}
\end{aligned}$$

Note that the substitutions that introduce a farm to m_2 would yield a structure FARM n (FARM n_2) _. In our model, this would not be a problem, since the underlying structure would be equivalent to FARM $(n \times n_2)$ _. We could improve our type-checking implementation by considering the context of a metavariable. This would avoid superfluous rewritings. In our example, we assume that $sz = [d]^{1000}$. This represents the size of 1000 pairs of images of d dimensions. Arithmetic operations on sz are applied to the superscript. The size function of the first stage $|\mathrm{A}_{c_1}|$ is the identity, since we are not modifying the images. The parameters for the number of farm workers are fixed to be those with the least cost, given some maximum number of available cores. For δ_1, we determine that $n_1 = 9$, $n = 3$ and $n_2 = 5$. The values of the costs on those sizes, and the overheads of farms and pipelines are given below. In the example cost equation below, $|\mathrm{A}_{c_1}|(sz)$ denotes the output size of the first stage, obtained using `resize`, and κ is the overhead of the parallel structure. In the following examples, we omit these numbers and just assume a 24-core machine with similar overheads for farms, pipelines, divide-and-conquer and feedbacks.

$$\begin{aligned}
c_1\ [(2048 \times 2048, 2048 \times 2048)]^n &= \times 25.11\ \mathrm{ms}\\
c_2\ [(2048 \times 2048, 2048 \times 2048)]^n &= \times 45.21\ \mathrm{ms}\\
\kappa\ (9) = 29.66\ \mathrm{ms} &\qquad \kappa\ (15) = 60.93\ \mathrm{ms}
\end{aligned}$$

$$\begin{aligned}
&\mathtt{cost}\ (\delta_1 \mathrm{IM}(n))\ sz = \max \left\{ c_1 \left(\frac{sz}{n_1}\right) + \kappa(n_1 + n_2),\ c_2 \left(\frac{|\mathrm{A}_{c_1}|(sz)}{n_2}\right) \right.\\
&\quad \left. + \kappa(n_1 + n_2) \right\} = 3145.69\ \mathrm{ms}\\
&\mathtt{cost}\ (\delta_2 \mathrm{IM}(n))\ sz = 25123.81\ \mathrm{ms}\\
&\mathtt{cost}\ (\delta_3 \mathrm{IM}(n))\ sz = 3189.60\ \mathrm{ms}\\
&\mathtt{cost}\ (\delta_4 \mathrm{IM}(n))\ sz = 25123.81\ \mathrm{ms}
\end{aligned}$$

The structure that results from applying δ_1 is the least cost one, $\delta_1(\mathrm{IM}_2(3))$, with $n_1 = 9$ and $n_2 = 5$.

7.1.2. Quicksort

We will now analyse *quicksort* and show how it can exploit a divide-and-conquer parallel structure. Our cost models show the following estimations, where κ is adjusted to take into account the overheads of a DC structure, again calculated as the cost of the queue contention. In this example, we set the size parameter of our cost model to 1000 lists of 3,000,000 elements:

$$\begin{aligned}
&\mathtt{qsorts} : \mathsf{List}(\mathsf{List}\ A) \mapsto \mathsf{List}(\mathsf{List}\ A)\\
&\mathtt{qsorts} = \mathtt{map}_{\mathsf{List}}\ (\mathtt{hylo}_{F\ A}\ \mathtt{merge}\ \mathtt{div})
\end{aligned}$$

$$\begin{aligned}
&\mathtt{cost}\ (\mathrm{PAR}_L\ (\mathrm{DC}_{n,F}\ \mathrm{A}_{c_1}\ \mathrm{A}_{c_2}))\ sz\\
&= \max\{c_2\ (|\mathrm{A}_{c_2}|^i sz), \ldots\\
&\quad , \mathtt{cost}\ (\mathrm{HYLO}_F\ \mathrm{A}_{c_1}\ \mathrm{A}_{c_2})\ (|\mathrm{A}_{c_2}|^n sz), \ldots\\
&\quad , \max\{c_1\ (|\mathrm{A}_{c_1}|^i |\mathrm{A}_{c_2}|^n sz)\} = 42602.72\ \mathrm{ms}\\
&\mathtt{cost}\ (\mathrm{PAR}_L\ (\mathrm{FARM}_n\ (\mathrm{FUN}\ (\mathrm{ANA}_L\ \mathrm{A}_{c_2})) \parallel (\mathrm{FUN}\ (\mathrm{CATA}_L\ \mathrm{A}_{c_1}))))\ sz\\
&= 27846.13\ \mathrm{ms}\\
&\mathtt{cost}\ (\mathrm{PAR}_L\ (\mathrm{FARM}_n\ (\mathrm{FUN}\ (\mathrm{HYLO}_F\ \mathrm{A}_{c_1}\ \mathrm{A}_{c_2}))))\ sz\\
&= 32179.77\ \mathrm{ms}\\
&\ldots
\end{aligned}$$

Since the most expensive part of the *quicksort* function is the *divide*, and since flattening a tree is linear, the cost of adding a farm to

the *divide* part is less than the cost of using a divide-and-conquer skeleton for this example.

7.1.3. N-body simulation

N-Body simulations are widely used in astrophysics. They comprise a simulation of a dynamic system of particles, usually under the influence of physical forces. The Barnes–Hut simulation recursively divides the n bodies, storing them in a `Octree`, or a 8-ary tree. Each node in the tree represents a region of the space, where the topmost node represents the whole space and the eight children represent the eight octants of the space. The leaves of the tree contain the bodies. We then calculate the cumulative mass and centre of mass of each region of the space. Finally, the algorithm calculates the net force on each particular body by traversing the tree, and updates its velocity and position. This process is repeated for a number of iterations. We will here abstract most of the concrete, well known details of the algorithm, and present its high-level structure, using the following types and functions:

$$\begin{array}{lcl} C & = & \mathbb{Q} \times \mathbb{Q} \\ F\,A\,B & = & A + C \times B^8 \\ G\,A & = & F\,\mathtt{Body} \\ \mathtt{Octree} & = & \mu G \\ \mathtt{insert} & : & \mathtt{Body} \times \mathtt{Octree} \to \mathtt{Octree} \end{array}$$

Since this algorithm also involves iterating for a fixed number of steps, we define iteration as a hylomorphism, as well as its structure.

$$\begin{array}{l} \textsc{loop} \;:\; \Sigma \to \Sigma \\ \textsc{loop}\,\sigma \;=\; \textsc{hylo}_{(+)}\,(\textsc{id} \triangledown \sigma)\,((\textsc{a} + (\textsc{a} \vartriangle (\textsc{a} \circ \textsc{a}))) \circ (\textsc{a} \circ \textsc{a}?)) \end{array}$$

$$\begin{array}{l} \mathtt{loop}_A \;:\; (A \overset{m}{\mapsto} A) \to A \times \mathbb{N} \overset{\textsc{loop}\ m}{\longmapsto} A \\ \mathtt{loop}_A\ s \;= \\ \quad \mathtt{hylo}_{(A+)}\,(\mathsf{id} \triangledown s) \\ \qquad ((\pi_1 + (\pi_1 \vartriangle ((-1) \circ \pi_2))) \circ ((== 0) \circ \pi_2)?) \end{array}$$

This example uses some additional functions: `calcMass` annotates each node with the total mass and centre of mass; `dist` distributes the octree to all the bodies, to allow independent calculations, `calcForce` calculates the force of one body; and `move` updates the velocity and position of the body.

$$\begin{array}{l} \mathtt{calcMass} \;:\; G\,\mathtt{Octree} \to G\,\mathtt{Octree} \\ \mathtt{dist} \;:\; \mathtt{Octree} \times \mathtt{List\ Body} \\ \qquad \to L\,(\mathtt{Octree} \times \mathtt{Body})\,(\mathtt{Octree} \times \mathtt{List\ Body}) \end{array}$$

The algorithm is:

$$\begin{array}{l} \mathtt{nbody} \;:\; \mathtt{List\ Body} \times \mathbb{N} \overset{\textsc{loop}\ \sigma}{\longmapsto} \mathtt{List\ Body} \\ \mathtt{nbody} \;=\; \mathtt{loop}\,(\mathtt{ana}_L\,(L\,(\mathtt{move} \circ \mathtt{calcForce}) \circ \mathtt{dist}) \\ \qquad \circ ((\mathtt{cata}_G\,(\mathtt{in}_G \circ \mathtt{calcMass}) \circ \mathtt{cata}_L\ \mathtt{insert}) \vartriangle \mathsf{id})) \end{array}$$

Note that we do not allow the fixed structure determined by LOOP to be rewritten. Even if we were to allow this, there is a data dependency in its definition that suggests that no parallelism can be extracted from it. However, the loop body does contain sources of parallelism, which our type system is able to exploit. In particular, the structure of the loop body is:

$$\sigma \;=\; \textsc{ana}_L(L\,(\textsc{a} \circ \textsc{a}) \circ \textsc{a}) \circ (\textsc{cata}_G(\textsc{in} \circ \textsc{a}) \circ \textsc{cata}_L\ \textsc{a}) \vartriangle \textsc{id}$$

The normalised structure reveals more opportunities for parallelisation:

$$\sigma \;=\; \textsc{map}_L\textsc{a} \circ \textsc{map}_L\ \textsc{a} \circ \textsc{ana}_L\ \textsc{a} \circ (\textsc{cata}_G(\textsc{in} \circ \textsc{a}) \circ \textsc{cata}_L\ \textsc{a}) \vartriangle \textsc{id}$$

After normalisation, this structure is equivalent to:

$$\sigma \;=\; \textsc{par}_L\,(\textsc{fun}\,(\textsc{a} \circ \textsc{a})) \circ \_$$

The structure makes it clear that there are many possibilities for parallelism using farms and pipelines. As before, parallelism can be introduced semi-automatically using our cost models. For example, setting the input size to 20,000 bodies:

$$\begin{array}{l} \sigma \;= \textsc{par}_L\,(\textsc{farm}\ n\ \_ \parallel \_) \circ \_ \\ \sigma' = \textsc{par}_L\,(\mathtt{min\ cost}\,(\_ \parallel \_)) \circ \_ \end{array}$$

$$\begin{array}{l} \mathtt{cost}\,(\textsc{fun}\ \textsc{a}_{c_1} \parallel \textsc{fun}\ \textsc{a}_{c_2})\,sz \;=\; 310525.67 \text{ ms} \\ \mathtt{cost}\,(\textsc{farm}_6\,(\textsc{fun}\ \textsc{a}_{c_1}) \parallel (\textsc{fun}\ \textsc{a}_{c_2}))\,sz \;=\; 55755.43 \text{ ms} \\ \mathtt{cost}\,(\textsc{fun}\ \textsc{a}_{c_1} \parallel \textsc{farm}_1\,(\textsc{fun}\ \textsc{a}_{c_2}))\,sz \;=\; 310525.67 \text{ ms} \\ \mathtt{cost}\,(\textsc{farm}_{20}(\textsc{fun}\ \textsc{a}_{c_1}) \parallel \textsc{farm}_4(\textsc{fun}\ \textsc{a}_{c_2}))\,sz \;=\; 15730.46 \text{ ms} \end{array}$$

7.1.4. Iterative convolution

Image convolution is also widely used in image processing applications. We assume the type `Img` of images, the type `Kern` of kernels, the functor $F\,A\,B \;=\; A + B \times B \times B \times B$, and the *split*, *combine*, *kern* and *finished* functions. The `split` function splits an image into 4 sub-images with overlapping borders, as required for the kernel. The `combine` function concatenates the sub-images in the corresponding positions. The `kern` function applies a kernel to an image. Finally, the `finished` function tests whether an image has the desired properties, in which case the computation terminates. We can represent image convolution on a list of input images as follows:

$$\begin{array}{l} \mathtt{conv} \;:\; \mathtt{Kern} \to (\mathtt{List\ Img} \overset{\sigma}{\mapsto} \mathtt{List\ Img}) \\ \mathtt{conv}\ k \;= \\ \quad \mathtt{map}_{\mathtt{List}}\,(\mathtt{iter}_{\mathtt{Img}}\,(\mathtt{finished?} \circ \mathtt{hylo}_F\,(\mathtt{combine} \circ F\,(\mathtt{kern}\ k)) \\ \qquad\qquad (\mathtt{split}\ k))) \end{array}$$

The structure of `conv` is equivalent to a feedback loop, which exposes many opportunities for parallelism. Again, we assume a suitable cost model. Our estimates are given for 1000 images, of size 2048×2048.

$$\begin{array}{l} \sigma \;=\; \textsc{par}_L\,(\textsc{fb}\,(\textsc{dc}_{n,L,F}\,(\textsc{a} \circ F\ \textsc{a})\ \textsc{a} \parallel \_)) \\ \quad = \textsc{par}_L\,(\textsc{fb}\,(\ \textsc{farm}\ n\ \_ \parallel \_ \parallel \_)) \\ \quad = \mathtt{min\ cost}\,(\textsc{par}_L\,(\textsc{fb}\,(\_ \parallel \_))) \\ \quad = \;\ldots \end{array}$$

$$\begin{array}{l} \mathtt{cost}\,(\textsc{par}_L\,(\textsc{fb}\,(\textsc{a}_{c_1} \parallel \textsc{a}_{c_2})))\,sz \;= \\ \qquad \displaystyle\sum_{1 \le i, |\textsc{a}_{c_1} \parallel \textsc{a}_{c_2}|^i sz > 0} \mathtt{cost}\,(\textsc{a}_{c_1} \parallel \textsc{a}_{c_2})\,(|\textsc{a}_{c_1} \parallel \textsc{a}_{c_2}|^i sz) \\ \quad = 20923.02 \text{ ms} \\ \mathtt{cost}\,(\textsc{par}_L\,(\textsc{fb}\,(\textsc{farm}_4\,(\textsc{fun}\ \textsc{a}_{c_1}) \parallel (\textsc{fun}\ \textsc{a}_{c_2}))))\,sz \\ \quad = 6649.55 \text{ ms} \\ \mathtt{cost}\,(\textsc{par}_L\,(\textsc{fb}\,(\textsc{fun}\ \textsc{a}_{c_1} \parallel \textsc{farm}_1\,(\textsc{fun}\ \textsc{a}_{c_2}))))\,sz \\ \quad = 20923.02 \text{ ms} \\ \mathtt{cost}\,(\textsc{par}_L\,(\textsc{fb}\,(\textsc{farm}_{14}\,(\textsc{fun}\ \textsc{a}_{c_1}) \parallel \textsc{farm}_4\,(\textsc{fun}\ \textsc{a}_{c_2}))))\,sz \\ \quad = \;2694.30 \text{ ms} \\ \ldots \end{array}$$

Collectively, these four examples have demonstrated the use of our techniques for all the parallel structures that we consider in this paper. In the next section, we provide evidence to validate the cost models that we have derived from our operational semantics against actual parallel executions.

7.2. Actual vs predicted speedups

In order to validate our results, we have compared predicted *versus* actual speedups for a number of example parallel programs. Our results not only validate our cost equations, but also confirm our previous experimental results [14]. We have taken the results of type-checking a number of examples, and implemented the corresponding structures in C, following our model, using the *pthreads*-based queue implementation that we described in [14]. In

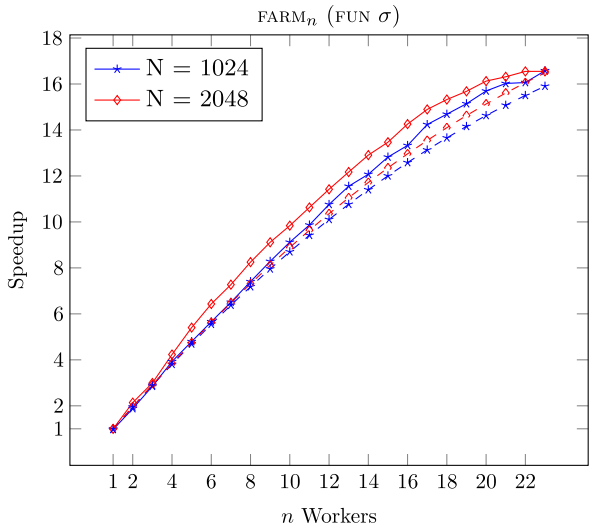


Fig. 6. Speedup (solid lines) vs prediction (dashed lines). Matrix Multiplication of matrices of sizes $N \times N$ (*titanic*).

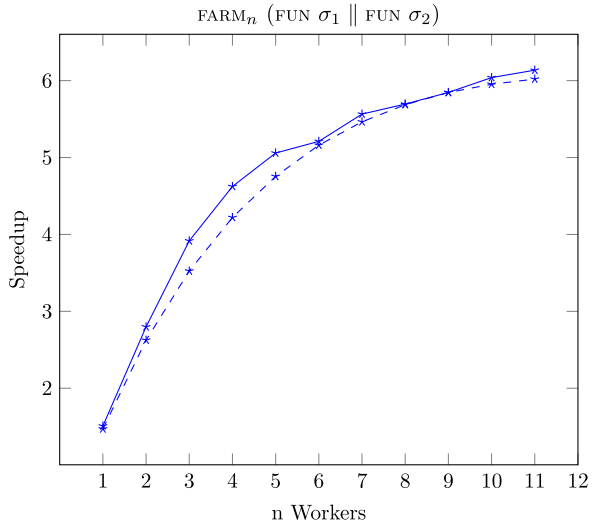


Fig. 7. Speedup (solid lines) vs prediction (dashed lines). Image Merge, 500 input tasks (*titanic*).

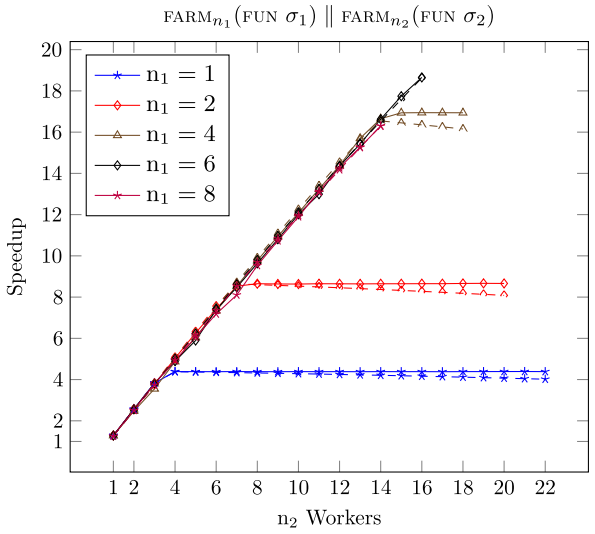


Fig. 8. Speedup (solid lines) vs prediction (dashed lines). Image Convolution, 500 input tasks (*titanic*).

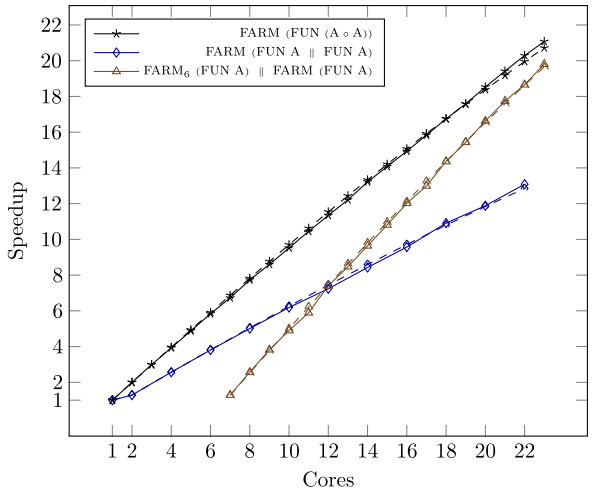


Fig. 9. Speedup (solid lines) vs predicted (dashed lines). Different parallel structures for image convolution, 500 Images 1024 * 1024: *titanic*.

Section 6.3, we showed that most of the cost equations can be simplified to the expected ones, apart from the cost of a FUN σ structure. Basically, the idea is that the overhead of the parallel structures can be safely pushed down into its workers. We show here how that overhead cost affects the actual runtimes, comparing this with the predicted speedups for some of our parallel structures.

We use two different real multicore machines: *titanic*, a 800 MHz 24-core, AMD Opteron 6176, running Centos Linux 2.6.18; and *lovelace*, a 1.4 GHz 64-core, AMD Opteron 6376, running GNU/Linux 2.6.32. All the speedups shown here have been calculated as the mean of ten runs. Fig. 6 shows the real *vs* predicted speedups for task farms, using a simple *matrix multiplication* example; Fig. 7 shows the corresponding speedups for the *image merge* example, parallelised as a farm of a pipeline; and Fig. 8 shows the corresponding speedups for the *image convolution* example. The latter example was originally described using a similar structured expression to that for *image merge*, i.e. as the composition of two functions, read and process. However, the cost models predict that a different parallel structure is optimal: a pipeline of two farms. Finally, Figs. 9 and 10 compare several different possible parallel structures over farms and pipelines for the *image convolution* example. These examples collectively verify the experimental results that we previously showed in [14], and show that the cost models defined in this paper are able to correctly capture queue contentions. We are thus able to predict a safe, tight upper bound on execution times.

8. Related work

There have been a few previous treatments of parallelism using types, but these deal only with sizes and productivity. One line of work uses *sized types* [17] to incorporate a notion of the sizes of streaming data into a type system. This has been extended to cover a small number of skeletons in the Eden dialect of Haskell [22]. More recently, López et al. [23] have presented a type-based methodology, inspired by session types, for verifying that parallel MPI programs follow some given protocol. This approach provides a scalable methodology to statically guarantee that a program satisfies a number of interesting safety properties, such as the absence of deadlocks. Both approaches show that types are indeed useful

to prove a number of interesting properties of parallel programs, e.g. termination and productivity. In contrast, our work focuses on the different, but equally important, properties of semantic equivalence and cost.

The expressive power of hylomorphisms for parallel programming was first explored by Fischer and Gorlatch [24], who showed that a programming language based on catamorphisms and anamorphisms is *Turing-universal*. The idea of using hylomorphisms for parallel programming also appears in Morihata's work [25]. Morihata explores a theory for developing parallelisation theorems based on the *third homomorphism theorem* and *shortcut fusion*, and generalises it to hylomorphisms. In contrast, our work directly exploits the properties of hylomorphisms, in order to choose a suitable parallel skeleton implementation for hylomorphisms. The two lines of work are therefore orthogonal, and we can potentially benefit from Morihata's results in our future work.

Deriving parallel implementations from small, simple specifications has been widely studied. The third homomorphism theorem, list homomorphisms, and the Bird–Meertens Formalism are amongst the many techniques that have been explored [3,10,26–33]. The third homomorphism theorem states that if a function can be written both as a left fold and a right fold, then it can also be evaluated in a divide-and-conquer manner [34]. This theorem has been widely used for parallelism [25,35–40]. The majority of this work enables suitable automation and derivation of efficient parallel implementations. Our work differs in that we allow part of the parallel structure to be chosen in an automated way. This adds flexibility, enabling a parallel implementation to be changed quickly and easily by changing only a single type annotation. One possible extension of our work is to include some automatic transformations derived from the third homomorphism theorem. By parameterising our type system over some cost function on parallel structures, we smoothly integrate the introduction of parallelism with the ability to reason about the run-time behaviour of the parallel program. Skillicorn and Cai [41] have previously shown the utility of such an integration of a cost calculus with derivational software development, illustrating the approach for the Bird–Meertens theory of lists. In this paper, we take this approach one step further by using a more general equational theory based on hylomorphisms. Moreover, our type-based approach introduces new benefits, by providing a mechanism for specifying new parallel structures whose denotational semantics can be described as a composition of hylomorphisms.

Specific algorithmic skeletons are frequently associated with timing cost models [42]. However, these are almost exclusively obtained through measurement or approximation rather than being systematically derived from an operational semantics, as here. Much of the work on developing cost models for parallel execution has focused on data parallelism. The Parallel Random-Access Machine (PRAM) execution model [43] acts as a theory of complexity for parallel algorithms on idealised shared-memory SIMD machines. In the basic PRAM execution model, basic computations and shared-memory accesses are both assumed to take unit time. Unfortunately, PRAM costs underestimate actual machine execution costs, but in an unpredictable way [41]. The Bulk Synchronous Parallel (BSP) model [44] extends the PRAM model in a more realistic way, introducing a synchronising communication step after each set of computation steps. Lisper [45] has investigated the use of a Bulk Synchronous Parallel skeleton for determining worst-case execution times, but only informally, and not in the context of real processor models. Skillicorn and Cai have likewise developed a high-level cost calculus for data parallel computations [41], based on the *shape* of data structures, and known properties of primitive parallel operations, but have not based this on a strong

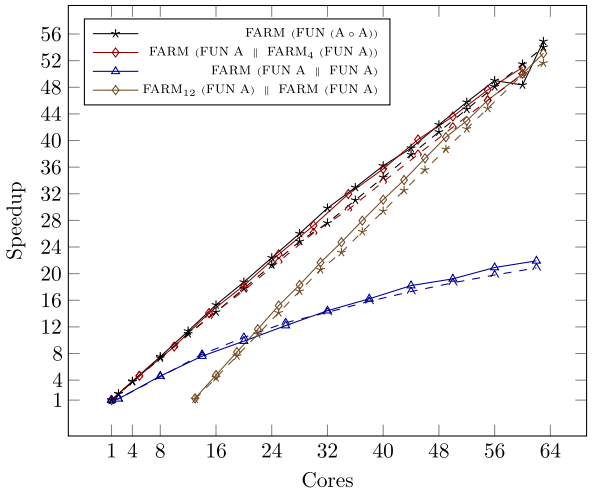


Fig. 10. Speedup (solid lines) vs predicted (dashed lines). Different parallel structures for image convolution, 500 Images 1024 * 1024: *lovelace*.

machine-level semantics. Blelloch and Greiner have demonstrated provable time and space bounds for nested data parallel computations in NESL [46]. None of these models has the generality of the one given here. By linking the model given in this paper with the machine-level operational semantics for queueing that we have previously produced [14], we are now able to give a complete operational model for parallel algorithmic skeletons that accounts for all levels from the memory model upwards, including dealing with real memory effects on *x86* architectures.

In a structured parallelism setting, Janjic et al. [47] define a high-level DSL, the Refactoring Pattern Language (RPL), that aims to represent the parallel structure of a program, and capture its execution time. This DSL is a powerful tool, since it allows suitable parallelisations to be found for a given program, and then to apply them to a real C++ program. There are a number of differences between the approach that is described here and RPL. First, our type-based approach does not need to realise parallelisations as refactoring rules: parallel structures are tied to programs in a systematic way by each syntactic construct. The advantage of our approach is that we can use type information to automatically generate parallel code at compile-time. The corresponding disadvantage is lack of flexibility: the RPL approach can be combined with a number of refactorings that can take into account the user input in a more interactive way. The second important difference is that we use hylomorphisms as an unifying construct. This enables us to use the rich theory of hylomorphisms for parallelism. Moreover, we base our approach on a decision procedure that is derived from an equational theory that is both sound and complete w.r.t. the rules of the underlying equational theory, and we use a standard type unification algorithm to instantiate parallel structures from sequential code. Finally, our parallel structures and cost models are not built-in, but are derived in a systematic way from an underlying cost model and operational semantics.

Finally, Steuwer et al. [12] generate high-performance OpenCL code from a high-level specification by applying a simple set of rewrite rules, and using Monte-Carlo search to traverse the corresponding search space to find an implementation. Our approach provides a way to narrow down this search space, while using cost models to automate the rest of the search. It is, in a sense, more general, since we allow our parallel structures to be easily extended. However, we could benefit from exploiting Steuwer et al.'s work in GPU-specific rewriting rules and skeletons.

9. Conclusions

This paper has introduced new formally-based cost models for common algorithmic skeletons. Unlike previous approaches, these cost models are derived from a formal operational semantics for these skeletons, and consider both queue contention and scheduling. Using our approach, we can be sure that our cost models are sound w.r.t. the operational semantics *by construction* and we can *automatically* derive cost equations for newly defined parallel structures. A key aspect of our approach is the use of *hylomorphisms*, combinations of *catamorphisms* and *anamorphisms* (or *fold*/*unfold* operations). We have used hylomorphisms to capture some common patterns of parallelism: task farms, pipelines, divide-and-conquers and feedbacks. As shown in our examples, this single construct is surprisingly powerful, providing a system of canonical representations that is easy to understand and that is also easy to transform.

We have shown how our cost models can be used to drive type-level decisions about parallelisation, supporting a type-level reasoning system that we recently introduced [1]. Collectively, this allows the construction of provably optimal parallel solutions, that need to take into account only basic, easily obtained, information about the costs of executing sequential operations. The use of a type-level approach avoids the usual separation of analysis from program, allowing structures and costs to be directly and accurately associated with the program. In particular, all transformations are performed internally by the type checker and we ensure the preservation of the underlying functional behaviour *simply by construction*. Moreover, our concrete results show that our cost models are good predictors for actual parallel performance.

A number of obvious extensions can be made to this work. Firstly, there are a few forms of parallel pattern that we have not yet considered: *map (reduce)* and *fold* are clearly instances of hylomorphisms, but *stencil* and *bulk synchronous parallel* patterns, for example, may require deeper thought. Secondly, we could investigate more complex and detailed cost models that take into account even more details of a parallel system. Finally, we believe that hylomorphisms and type structures can form the basis for a good, new parallel programming methodology. We intend to explore this by building our ideas into a new parallel library for Haskell that will enable the automatic introduction of parallelism into normal Haskell programs, under type control.

Acknowledgements

This work has been partially supported by the EU Horizon 2020 grant "RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach" (ICT-644235), by COST Action IC1202 (TACLe), supported by COST (European Cooperation on Science and Technology), and by EPSRC grant EP/M027317/1 "C^3: Scalable & Verified Shared Memory via Consistency-directed Cache Coherence".

References

[1] D. Castro, K. Hammond, S. Sarkar, Farms, pipes, streams and reforestation: Reasoning about structured parallel processes using types and hylomorphisms, in: Proc. ICFP 2016: Intl. Conf. on Functional Programming, 2016, pp. 4–17.
[2] M.I. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, Pitman, London, 1989.
[3] D.B. Skillicorn, The Bird-Meertens Formalism as a Parallel Model, Springer, 1993.
[4] M. Aldinucci, M. Coppola, M. Danelutto, Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff, in: Proc. International Workshop on Constructive Methods for Parallel Programming, Technical Report MIP-9805, University of Passau, Passau, 1998.
[5] M. Aldinucci, Automatic program transformation: The Meta tool for skeleton-based languages, in: Constructive Methods for Parallel Programming, Advances in Computation: Theory and Practice, 2002, pp. 59–78.
[6] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, A. Elliott, Cost-directed refactoring for parallel erlang programs, Int. J. Parallel Program. 42 (4) (2013) 1–19.
[7] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: Proc. ICFP'91: ACM Conf. on Functional Programming, 1991, pp. 124–144.
[8] P. Wadler, Deforestation: Transforming programs to eliminate trees, Theoret. Comput. Sci. 73 (2) (1990) 231–248.
[9] Y. Onoue, Z. Hu, M. Takeichi, H. Iwasaki, A calculational fusion system HYLO, in: Proc. Intl. Workshop on Alg. Lang. and Calculi, 1997, pp. 76–106.
[10] Z. Hu, H. Iwasaki, M. Takechi, Formal derivation of efficient parallel programs by construction of list homomorphisms, ACM Trans. Program. Lang. Syst. 19 (3) (1997) 444–461.
[11] F.A. Rabhi, S. Gorlatch, Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2003.
[12] M. Steuwer, C. Fensch, S. Lindley, C. Dubach, Generating performance portable code using rewrite rules, in: Proc ICFP 2015: 20th ACM Conf. on Functional Prog. Lang. and Comp. Arch., 2015, pp. 205–217.
[13] M. Aldinucci, M. Danelutto, Stream parallel skeleton optimization, in: Proc. PDCS'99: IASTED Intl. Conf. on Parallel and Distributed Computing and Systems, 1999, pp. 955–962.
[14] K. Hammond, C. Brown, S. Sarkar, Timing properties and correctness for structured parallel programs on x86-64 multicores, in: Proc. FOPARA 2015: Foundational and Practical Aspects of Resource Analysis, Revised Selected Papers, in: LNCS, vol. 9964, Springer, 2016, pp. 101–125.
[15] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, A language-independent parallel refactoring framework, in: Proceedings of the Fifth Workshop on Refactoring Tools, ACM, 2012, pp. 54–58.
[16] P.B. Vasconcelos, Space Cost Analysis using Sized Types (Ph.D. thesis), University of St Andrews, 2008.
[17] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: Proc. POPL'96: 23rd ACM Symposium on Principles of Programming Languages, 1996, pp. 410–423.
[18] H.-W. Loidl, K. Hammond, A sized time system for a parallel functional language, in: Proc. Glasgow Workshop on Functional Programming, Ullapool, Scotland, 1996.
[19] D. Sands, A Naïve time analysis and its theory of cost equivalence, J. Logic Comput. 5 (4) (1995) 495–541.
[20] G. Barthe, B. Grégoire, C. Riba, Type-based termination with sized products, in: International Workshop on Computer Science Logic, Springer, 2008, pp. 493–507.
[21] M. Barbosa, A. Cunha, J.S. Pinto, Recursion patterns and time-analysis, ACM SIGPLAN Not. 40 (5) (2005) 45–54.
[22] R. Peña, C. Segura, Sized types for typing Eden skeletons, in: Proc. IFL'01: Intl. Symp. on Impl. of Functional Langs., 2001, pp. 1–17.
[23] H.A. López, E.R.B. Marques, F. Martins, N. Ng, C. Santos, V.T. Vasconcelos, N. Yoshida, Protocol-based verification of message-passing parallel programs, in: Proc. OOPSLA 2015: Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, 2015, pp. 280–298.
[24] J. Fischer, S. Gorlatch, Turing universality of recursive patterns for parallel programming, Parallel Process. Lett. 12 (02) (2002) 229–246.
[25] A. Morihata, A short cut to parallelization theorems, in: Proc. ICFP 2013: 18th ACM Conf. on Functional Programming, 2013, pp. 245–256.
[26] S. Gorlatch, C. Lengauer, Parallelization of divide-and-conquer in the Bird-Meertens formalism, Form. Asp. Comput. 7 (6) (1995) 663–682.
[27] Z. Hu, M. Takeichi, W.-N. Chin, Parallelization in calculational forms, in: Proc. POPL'98: 25th ACM Symposium on Principles of Programming Languages, 1998, pp. 316–328.
[28] G. Keller, M. Chakravarty, Flattening trees, in: Proc. Euro-Par'98: European Conference on Parallelism, 1998, pp. 709–719.
[29] K. Matsuzaki, Z. Hu, M. Takeichi, Towards automatic parallelization of tree reductions in dynamic programming, in: Proc. SPAA 2006: Symp. on Parallelism in Algorithms and Architecture, 2006, pp. 39–48.
[30] J. Misra, Powerlist: A structure for parallel recursion, ACM Trans. Program. Lang. Syst. 16 (6) (1994) 1737–1767.
[31] A. Morihata, K. Matsuzaki, Automatic parallelization of recursive functions using quantifier elimination, in: Proc. FLOPS'10: Functional and Logic Programming, 2010, pp. 321–336.
[32] J.H. Reif, Synthesis of Parallel Algorithms, Morgan Kaufmann, 1993.
[33] D.B. Skillicorn, Models for practical parallel computation, Int. J. Parallel Program. 20 (2) (1991) 133–158.
[34] J. Gibbons, The third homomorphism theorem, J. Funct. Programming 6 (4) (1996) 657–665.
[35] Y.-Y. Chi, S.-C. Mu, Constructing list homomorphisms from proofs, in: Proc. APLAS'11: Asian Symposium on Programming Languages & Systems, 2011, pp. 74–88.
[36] A. Geser, S. Gorlatch, Parallelizing functional programs by generalization, J. Funct. Programming 9 (06) (1999) 649–673.
[37] J. Gibbons, Computing downwards accumulations on trees quickly, Theoret. Comput. Sci. 169 (1) (1996) 67–80.

[38] S. Gorlatch, Extracting and implementing list homomorphisms in parallel program development, Sci. Comput. Program. 33 (1) (1999) 1–27.
[39] Y. Liu, Z. Hu, K. Matsuzaki, Towards systematic parallel programming over mapreduce, in: Proc. Euro-Par 2011: European Conference on Parallelism, 2011, pp. 39–50.
[40] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, M. Takeichi, Automatic inversion generates divide-and-conquer parallel programs, in: Proc. PLDI'07: ACM Conf. on Prog. Lang. Design and Impl., 2007, pp. 146–155.
[41] D.B. Skillicorn, W. Cai, A cost calculus for parallel functional programming, J. Parallel Distrib. Comput. 28 (1) (1995) 65–83.
[42] M.M. Hamdan, A Survey of Cost Models for Algorithmic Skeletons, Technical Report, Heriot-Watt University, Dept. of Computers and Electrical Engineering, 1999.
[43] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proc. STOC'78: 10th Symp. on Theory of Computing, 1978, pp. 114–118.
[44] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[45] A. Gustavsson, J. Gustafsson, B. Lisper, Toward static timing analysis of parallel software, in: OASIcs-OpenAccess Series in Informatics, vol. 23, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.
[46] G.E. Blelloch, NESL: A Nested Data-Parallel Language (Version 3.1), Technical Report, DTIC Document, 1995.
[47] V. Janjic, C. Brown, K. Mackenzie, K. Hammond, M. Danelutto, M. Aldinucci, J.D. Garcia, RPL: A domain-specific language for designing and implementing parallel C++ applications, in: Proc. PDP 2016: Euromicro International Conf. on Parallel, Distributed, and Network-Based Processing, IEEE, 2016, pp. 288–295.

Prof. Kevin Hammond (Principal Investigator) leads the Programming Languages group in the School of Computer Science. He has over 30 years of research experience, having worked extensively in the field of advanced programming language design and implementation, with a focus on understanding and reasoning about extra-functional properties. His work concentrates on declarative language designs, including that of the standard non-strict functional language Haskell [31], where he served on the international design committee, worked on the dominant compiler, GHC [25], and developed the GUM parallel Haskell implementation of GHC with colleagues at Glasgow and elsewhere [9]. In 2011, GHC was awarded the ACM SigPlan prize for significant programming language software. He has published over 100 books, book chapters, journal papers and other refereed publications focusing on functional programming, domain-specific programming languages, type systems, real-time systems, cost issues, adaptive runtime environments, lightweight concurrency, high-level programming language design, parallel programming and performance monitoring/visualisation. He is the recently-appointed Director of SICSA, the pan-Scottish pooling agreement for Informatics and Computer Science, which groups the 14 Scottish universities. He is a Senior Member of the ACM, a founder member of IFIP WG 2.11 (Generative Programming), and an honorary Professor at Heriot–Watt University. He sits on the board of the Scottish government funded Data Lab Innovation Centre. He was awarded a personal award for his services to the parallel programming community at the HiPEAC 2016 conference.