Farms, Pipes, Streams and Reforestation: Reasoning about Structured Parallel Processes using Types and Hylomorphisms

David Castro, Kevin Hammond, Susmit Sarkar
School of Computer Science, University of St Andrews, St Andrews, Scotland
{dc84, kh8, ss265}@st-andrews.ac.uk

Abstract

The increasing importance of parallelism has motivated the creation of better abstractions for writing parallel software, including structured parallelism using nested algorithmic skeletons. Such approaches provide high-level abstractions that avoid common problems, such as race conditions, and often allow strong cost models to be defined. However, choosing a *combination* of algorithmic skeletons that yields good parallel speedups for a program on some specific parallel architecture remains a difficult task. In order to achieve this, it is necessary to simultaneously reason both about the costs of different parallel structures and about the semantic equivalences between them. This paper presents a new type-based mechanism that enables strong static reasoning about these properties. We exploit well-known properties of a very general recursion pattern, *hylomorphisms*, and give a denotational semantics for structured parallel processes in terms of these hylomorphisms. Using our approach, it is possible to determine formally whether it is possible to introduce a desired parallel structure into a program without altering its functional behaviour, and also to choose a version of that parallel structure that minimises some given cost model.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages; D.1.3 [*Programming Techniques*]: Parallel Programming; D.3.1 [*Formal Definitions and Theory*]: Semantics

Keywords Parallelism, type-systems, hylomorphisms, term rewriting systems.

1. Introduction

Providing suitable abstractions to allow reasoning about parallelism is crucial to allow safe exploitation of increasingly parallel hardware. To date, however, there has been only a limited amount of work on this issue: the static analysis community has been concerned primarily with provable safety, whereas the parallelism community has been concerned primarily with practically demonstrable performance. In order to meet both demands, it is necessary to simultaneously consider *both* the functional *and* the extra-functional properties of a parallel program. This paper presents a new approach, a type-based mechanism that enables us to reason about the safe introduction of parallelism, while also providing a good abstraction to reason about cost. This mechanism exploits strong program structure in the form of structured parallel processes [3], combined with properties of *hylomorphisms* [22].

1.1 Motivating Example

We introduce our approach using a simple example, *image merge*, which merges pairs of images taken from an input stream. We start with a simple structure, where the functionality of the program is split into two functions: mark, which marks the pixels that are to be merged; and merge, whichreplaces these pixels. A straightforward implementation would use the usual map construct on lists:

$$\begin{aligned}&\mathsf{imgMerge} \;:\; \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \;\rightarrow\; \mathsf{List\ Img}\\ &\mathsf{imgMerge} \;=\; \mathsf{map}\ (\mathsf{merge} \circ \mathsf{mark})\end{aligned}$$

Given two well-known algorithmic skeletons, *farm* for parallel replication and *pipeline* for parallel composition, even this simple example presents several alternative parallelisations, including:

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

where $\mathsf{fun}(f)$ encapsulates sequential functionality and $p_1 \parallel p_2$ represents a 2-stage parallel pipeline. All these implementations are semantically equivalent to imgMerge: they differ only in *how* the computation is performed, and thus in its performance. In this paper, we will describe a type system that annotates top-level types with the (parallel) structure, and will use this to *automatically* choose a parallelisation. For example, if we define $\textsc{im}(n, m) = \textsc{farm}\ n\ (\textsc{fun a}) \parallel \textsc{farm}\ m\ (\textsc{fun a})$, we could write:

$$\begin{aligned}&\mathsf{imgMerge} \;:\; \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \xmapsto{\textsc{im}\ (n,m)} \mathsf{List\ Img}\\ &\mathsf{imgMerge} \;=\; \mathsf{map}\ (\mathsf{merge} \circ \mathsf{mark})\end{aligned}$$

The type system will then automatically select $\mathsf{imgMerge}_2$. *Rather than manually changing the definition of* imgMerge *to a parallel one, we can simply change the type annotation* $\textsc{im}\ (n, m)$ *and automatically introduce the corresponding parallel implementation.* The soundness of the type system provides strong static guarantees that the resulting program is functionally equivalent to the original form. Moreover, we can even use the type system to *infer* part of this parallel structure. For example, if we define $\textsc{im}_1(n, m) = \_ \parallel \textsc{farm}\ m\ \_$ and $\textsc{im}_2(n, m) = \mathsf{min\ cost}\ (\_ \parallel \textsc{farm}\ m\ \_)$, then the parts of $\textsc{im}_1$ that are denoted by _ will be filled with the simplest possible (parallel) structure and the parts of $\textsc{im}_2$ that are denoted by _ will be filled with the least cost (parallel) structure. This provides a powerful, type-level, mechanism for understanding, reasoning about, and automating the introduction of parallelism.

1.2 Contributions

The paper makes the following main novel contributions:

- We define a denotational semantics for a set of well-known algorithmic skeletons, that cover a good range of parallel programs, in terms of *hylomorphisms* (Section 3).
- We identify semantic equivalences of parallel processes built using nested algorithmic skeletons as part of a *type system* that allows us to rationally choose a suitable parallel structure for a program (Section 4).
- We introduce a decision procedure for such semantic equivalences, based on reintroducing intermediate data structures, "reforestation", rather than the more common approach of eliminating them for efficiency reasons, "deforestation". This enables the type system to introduce parallelism in a semi-automated and sound way (Section 5).

2. Type Preliminaries

Our denotational semantics is phrased in a standard categorical language [10], with a category representing a *model of computation*, the objects of that category representing *types*, the morphisms representing *programs*, and *endofunctors* from the category to itself representing type constructors. In line with common practice [34], we will work in the category of *pointed complete partial orders* (*CPO*). Our type language is entirely standard:

$$A, B ::= \tau \mid 1 \mid A + B \mid A \times B \mid A \to B \mid F\ A \mid \mu F$$

We assume no knowledge about the definition of an *atomic type* (τ), but require it to have an interpretation as an object in the category *CPO*, $[\![\tau]\!] \in CPO$. For the other types, we assume a standard domain-theoretic semantics, where types are interpreted as pointed *CPO*s. The type 1 is interpreted as the unit of *CPO*; the type $A + B$ is interpreted as the separated sum of $[\![A]\!]$ and $[\![B]\!]$; the type $A \times B$ is interpreted as the Cartesian product of $[\![A]\!]$ and $[\![B]\!]$; the type $A \to B$ is interpreted as the set of continuous functions from $[\![A]\!]$ to $[\![B]\!]$; the type $F\ A$ is interpreted as the corresponding endofunctor applied to $[\![A]\!]$; and the type μF is interpreted as the fixpoint of functor $[\![F]\!]$.

2.1 Functors

A *functor* is a structure-preserving mapping between categories. We only need *endofunctors* on CPO, $F : CPO \to CPO$, to represent type constructors, $F : \mathsf{Type} \to \mathsf{Type}$. *Bifunctors* are functors that are generalised to multiple arguments, and we use them to define polymorphic data types. The functors in our language are either standard polynomial functors with constant types, the left section of a bifunctor, or 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:

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

Example 1 (Lists)**.** *Given the bifunctor*

$$L\ A\ B = 1 + A \times B,$$

the polymorphic List *data type is defined by the fixpoint of $L\ A$:*

$$\mathsf{List}\ A = \mu(L\ A).$$

As we will discuss in Section 3, it is well known that given a base bifunctor $F\ A\ B$, the data type $F\ A = \mu(G\ A)$ is also a functor.

$$\begin{array}{l} (\cdot \circ \cdot) : (B \to C) \to (A \to B) \to A \to C \\ f \circ g = \lambda x.f(g\ x) \end{array}$$

$$\begin{array}{l} (\cdot \triangledown \cdot) : (A \to C) \to (B \to C) \to (A + B) \to C \\ (f \triangledown g) = \lambda x.case\ x\ (\lambda y.f\ y)\ (\lambda y.g\ y) \end{array}$$

$$\begin{array}{l} (\cdot + \cdot) : (A \to C) \to (B \to D) \to (A + B) \to (C + D) \\ (f + g) = (inj_1 \circ f) \triangledown (inj_2 \circ g) \end{array}$$

$$\begin{array}{l} (\cdot \vartriangle \cdot) : (A \to B) \to (A \to C) \to A \to (B \times C) \\ (f \vartriangle g) = \lambda x.(f\ x, g\ x) \end{array}$$

$$\begin{array}{l} (\cdot \times \cdot) : (A \to C) \to (B \to D) \to (A \times B) \to (C \times D) \\ (f \times g) = (f \circ \pi_1) \vartriangle (g \circ \pi_2) \end{array}$$

Figure 1: Primitive Combinators.

The two list constructors are defined as expected:

$$\begin{array}{lcl} \mathsf{nil} & : & \mathsf{List}\ A \\ \mathsf{nil} & = & in_{L\,A}\ (inj_1\ ()) \end{array}$$

$$\begin{array}{lcl} \mathsf{cons} & : & A \to \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{cons}\ x\ l & = & in_{L\,A}\ (inj_2\ (x, l)) \end{array}$$

We define the usual notation for lists

$$[x_1, x_2, \ldots, x_n] = \mathsf{cons}\ x_1\ (\mathsf{cons}\ x_2\ (\mathsf{cons}\ \ldots\ (\mathsf{cons}\ x_n\ \mathsf{nil}))).$$

Example 2 (Trees)**.** *The polymorphic binary tree type can be defined in an analogous way:*

$$\begin{array}{lcl} T\ A\ B & = & 1 + A \times B \times B \text{ where} \\ \mathsf{Tree}\ A & = & \mu(T\ A). \end{array}$$

The two tree constructors are defined below:

$$\begin{array}{lcl} \mathsf{empty} & : & \mathsf{Tree}\ A \\ \mathsf{empty} & = & in_{T\,A}\ (inj_1\ ()) \end{array}$$

$$\begin{array}{lcl} \mathsf{node} & : & \mathsf{Tree}\ A \to A \to \mathsf{Tree}\ A \to \mathsf{Tree}\ A \\ \mathsf{node}\ t_1\ x\ t_2 & = & in_{T\,A}\ (inj_2\ (x, t_1, t_2)) \end{array}$$

Finally, we use a syntactic notation $\mathsf{base}\ F$ in our typing rules, $\mathsf{base}\ F$ is either the bifunctor G that is used in the definition of F, or else F itself.

$$\mathsf{base}\ F = \begin{cases} G, & \text{if } F\ A = G\ C\ A \text{ or } F\ A = \mu(G\ A) \\ F, & \text{otherwise.} \end{cases}$$

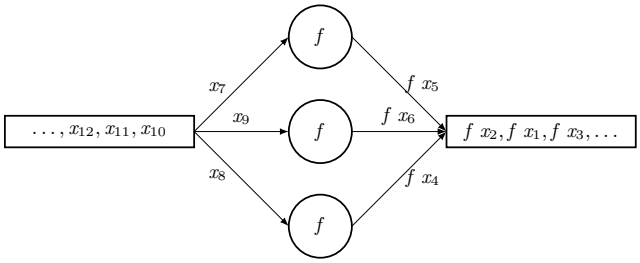
2.2 Primitive Combinators

Our semantics is defined in terms of the primitive combinators in Figure 1. For simplicity, we use a flattened form of sum and product types, e.g. we write $A_1 + A_2 + \cdots + A_n$ rather than $A_1 + (A_2 + (\ldots + A_n))$. Instead of the usual inl/inr and π_1/π_2, we use the $inj_i, 0 < i \leq n$, introduction forms for sum types, $A_1 + A_2 + \cdots + A_n$, and the projection eliminators $\pi_i, 0 < i \leq n$ for product types, $A_1 \times A_2 \times \cdots \times A_n$. The operator $\circ$ denotes function composition, $\triangledown$ is the usual coproduct morphism (*join*), $+$ denotes the map on coproducts, $\vartriangle$ and $\times$ are the respective morphisms and maps on products. Finally, for a recursive type defined as the fixpoint of a functor, μF, the usual $in_F : F\mu F \to \mu F$ and $out_F : \mu F \to F\mu F$ capture the isomorphism between $F\mu F$ and μF.

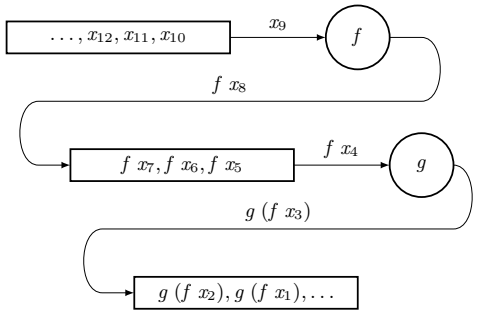
3. Structured Parallel Programs

Structured parallel approaches, such as *algorithmic skeletons* [3], offer many advantages in terms of built-in safety and parallelism-by-construction. For example, they can eliminate *by design* common but hard-to-debug problems including *deadlock* and *race conditions*. Such problems are prevalent in typical low-level *concurrency based* designs for parallel systems, e.g. *pthreads*, OpenMP, etc. Algorithmic skeletons also provide good structural cost models [14, 20]. In this paper, we will use four basic parallel skeletons, each of which operates over a stream of input values, producing a stream of results: task farms, pipelines, feedbacks, and divide-and-conquer.

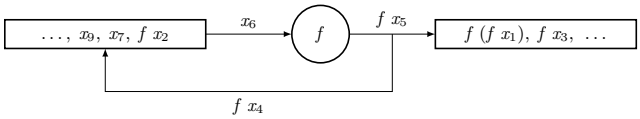
Task farms. The task farm skeleton, farm, applies the same operation to each element of an input stream of tasks, using a fixed number of parallel workers. The input tasks must be independent, and the outputs can be produced in an arbitrary order[1]. For example, a task farm could be used to apply some filter to an input stream of images, in order to parallelise the filter operation.



Pipeline. The pipeline skeleton, ||, composes two streaming computations, in parallel. It can be used to parallelise two or more stages of a computation, e.g. filtering and edge detection.

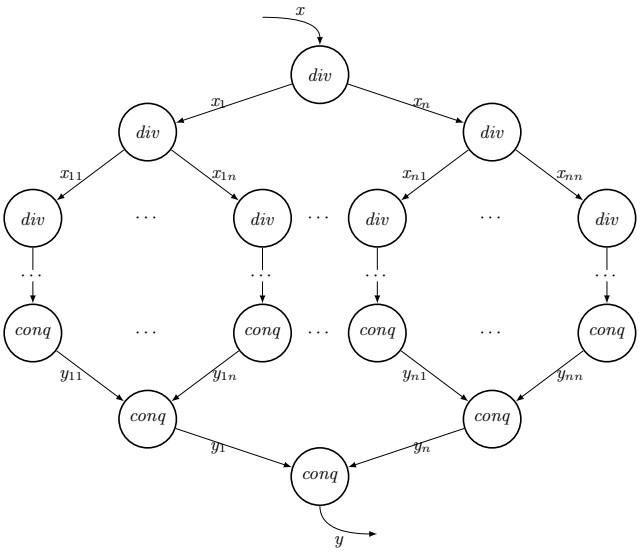


Feedback. The feedback skeleton, fb, captures recursion in a streaming computation. A feedback could be used, for example, to repeatedly transform an image until some dynamic condition was met.



Parallel Divide and Conquer. A divide-and-conquer skeleton, dc, is a parallel implementation of a classical divide-and-conquer algorithm. The parallelism arises from performing each of the recursive calls in parallel.

[1] Although this does complicate the semantics, it improves parallelism.



3.1 Structured Parallel Processes

We define a language P of structured parallel processes, built by composing skeletons over atomic operations.

$$p \in P \ ::= \ \mathsf{fun}_T\ f \mid p_1 \parallel p_2 \mid \mathsf{dc}_{n,T,F}\ f\ g \mid \mathsf{farm}\ n\ p \mid \mathsf{fb}\ p$$

The $\mathsf{fun}_T\ f$ construct *lifts* an atomic function to a streaming operation on a collection T. The arguments of the dc skeleton are: the number of *levels* of the divide-and-conquer, n; the collection T on which the dc skeleton works; and the functor F that describes the divide-and-conquer call tree.

Denotational Semantics. The denotational semantics is 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, $\hat{\rho}$:

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

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

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

An atomic function, f, is applied to all the elements of a collection of data. A parallel pipeline, $p_1 \ \parallel \ p_2$, is the composition of two parallel processes, p_1 and p_2. A task farm, $\mathsf{farm}\ n\ p$, replicates a parallel process, p, so has the same denotational semantics as p. A feedback skeleton, $\mathsf{fb}\ p$, applies the computation p iteratively, i.e. *trampolined*, to the elements in the input collection. Its semantics is given in terms of the function $iter$.

$$\begin{array}{l}
iter \ : \ (A \to A + B) \to A \to B \\
iter\ f \ = \ \mathsf{Y}\ (\lambda\ g.(g \triangledown id) \circ f)
\end{array}$$

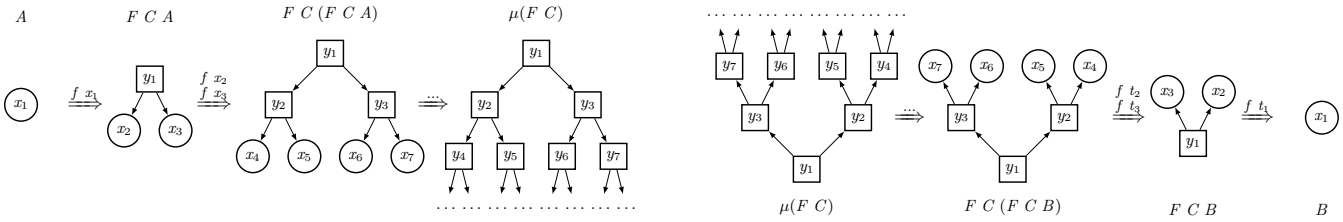


Figure 2: Binary Tree Anamorphism (left) and Catamorphism (right).

Finally, a dc is equivalent to *folding*, using f, the tree-like structure that results from *unfolding* the input using g. It is defined to be the composition of a *catamorphism* with an *anamorphism*.

$$\begin{array}{lcl} cata_F & : & (F\ A \to A) \to \mu F \to A \\ cata_F\ f & = & f \circ F\ (cata_F\ f) \circ out_F \end{array}$$

$$\begin{array}{lcl} ana_F & : & (A \to F\ A) \to A \to \mu F \\ ana_F\ g & = & in_F \circ F\ (ana_F\ g) \circ g \end{array}$$

Example 3 (List catamorphism)**.** *Let L be the base bifunctor of a polymorphic list. We define the function f to be:*

$$\begin{array}{lcl} f & : & L\ \mathbb{N}\ \mathbb{N} \to \mathbb{N} \\ f\ (inj_1\ ()) & = & 0 \\ f\ (inj_2\ (x, n)) & = & \mathsf{add}\ x\ n \end{array}$$

Given an input list $[x_1, x_2, \ldots, x_n]$, the catamorphism $cata_{L\,\mathbb{N}}\ f$ applied to this input list returns the sum of the x_i:

$$cata_{L\,\mathbb{N}}\ f\ [x_1, x_2, \ldots, x_n] = \mathsf{add}\ x_1\ (\mathsf{add}\ x_2\ (\cdots\ (\mathsf{add}\ x_n\ 0))).$$

Example 4 (List anamorphism)**.** *We define a function g that returns $()$ if the input n is zero, and $(n, n-1)$ otherwise.*

$$\begin{array}{lcl} g & : & \mathbb{N} \to L\ \mathbb{N}\ \mathbb{N} \\ g\ n & = & \textit{if } n = 0 \textit{ then } inj_1\ () \textit{ else } inj_2\ (n, n-1) \end{array}$$

The anamorphism $ana_{L\,\mathbb{N}}\ g$ applied to n returns a list of numbers descending from n to 1: $ana_{L\,\mathbb{N}}\ g\ n = [n, n-1, \ldots, 2, 1]$.

Figure 2 shows how catamorphisms and anamorphisms work on binary trees. In the anamorphism, we start with an input value, and apply the operation f recursively until the entire data structure is *unfolded*. In the catamorphism, the operation f is applied recursively until the entire structure is *folded* into a single value. The operation map_T can be defined as a special case of a catamorphism or anamorphism. Given a bifunctor G, a type $T\ A = \mu(G\ A)$ is a polymorphic data type that is also a functor [10]. For all $f : A \to B$, the function $map_T\ f$ is the morphism $T\ f$, defined as:

$$\begin{array}{lcl} map_T\ f & = & cata_{G\,A}(in_{G\,B} \circ G\ f\ id) \\ & = & ana_{G\,B}(G\ f\ id \circ out_{G\,A}) \end{array}$$

For uniformity, we will represent all map_T as catamorphisms.

Example 5 (map_{List})**.** *Given a function $f : A \to B$, $map_{\mathsf{List}}\ f$ applies f to all the elements of the input list:*

$$map_{\mathsf{List}}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

3.2 Hylomorphisms

Hylomorphisms are a well known, and very general, recursion pattern [22]. A *hylomorphism* can be thought of as the generalisation of a divide-and-conquer. Intuitively, $hylo_F\ f\ g$ is a recursive algorithm whose recursive call tree can be represented by μF, where g describes how the algorithm divides the input problem into subproblems, and f describes how the results are combined.

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

Since $out_F \circ in_F = id$, we can easily show that $hylo_F\ f\ g = cata_F\ f \circ ana_F\ g$. Catamorphisms, anamorphisms and map are just special cases of hylomorphisms.

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

Example 6 (Quicksort)**.** *Assuming a type A, and two functions,* leq, gt $: A \to \mathsf{List}\ A \to \mathsf{List}\ A$*, that filter the elements appropriately, we can implement naïve quicksort as:*

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

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.*

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

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

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

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

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

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

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

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

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

The only construct that has not yet been considered is feedback. Although the fixpoint combinator Y can be easily defined as a hylomorphism, we take a different approach. Observe that we can unfold the definition of $iter$ as follows:

$$iter\ f = \mathsf{Y}\ (\lambda\ g.(g \triangledown id) \circ f) = (iter\ f \triangledown id) \circ f$$

$$\frac{\rho(f) \;=\; A \to B}{\vdash f \;:\; A \to B} \qquad \frac{\vdash e_2 \;:\; B \to C \quad \vdash e_1 \;:\; A \to B}{\vdash e_2 \circ e_1 \;:\; A \to C} \qquad \frac{\vdash e_1 \;:\; F\ B \to B \quad \vdash e_2 \;:\; A \to F\ A}{\vdash \mathsf{hylo}_F\ e_1\ e_2 \;:\; A \to B} \qquad \frac{\vdash p \;:\; T\ A \to T\ B}{\vdash \mathsf{par}_T\ p \;:\; T\ A \to T\ B}$$

Figure 3: Simple types for Structured Expressions, E.

$$\frac{\vdash s \;:\; A \to B}{\vdash \mathsf{fun}\ s \;:\; T\ A \to T\ B} \qquad \frac{n \;:\; \mathbb{N} \quad \vdash p \;:\; T\ A \to T\ B}{\vdash \mathsf{farm}\ n\ p \;:\; T\ A \to T\ B} \qquad \frac{\vdash p \;:\; T\ A \to T\ (A+B)}{\vdash \mathsf{fb}\ \ p \;:\; T\ A \to T\ B}$$

$$\frac{\vdash s_1 \;:\; F\ B \to B \quad \vdash s_2 \;:\; A \to F\ A}{\vdash \mathsf{dc}_{n,F}\ \ s_1\ s_2 \;:\; T\ A \to T\ B} \qquad \frac{\vdash p_1 \;:\; T\ A \to T\ B \quad \vdash p_2 \;:\; T\ B \to T\ C}{\vdash p_1 \parallel p_2 \;:\; T\ A \to T\ C}$$

Figure 4: Simple types for Structured Parallel Processes, P.

Note that if $f, g \;:\; A+B \to C$, the function $f \triangledown g \;:\; A+B \to C$ can be written as the composition of $id \triangledown id \;:\; C+C \to C$ and $f \;+\; g \;:\; A+B \to C+C$. We use this to rewrite *iter* as follows:

$$iter\ f \;=\; (iter\ f \triangledown id) \circ f \;=\; (id \triangledown id) \circ (iter\ f + id) \circ f$$

If $f : A \to A+B$, we define the functor $(+B)$, with the morphism $(+\ B)\ f \;=\; f + id$, which trivially preserves identities and composition. Since $iter\ f = (id \triangledown id) \circ (+B)\ (iter\ f) \circ f$, then:

$$iter\ f \;=\; hylo_{(+B)}\ (id \triangledown id)\ f$$

3.3 Structured Expressions

We have now seen that the denotational semantics of all our parallel constructs can be given in terms of hylomorphisms. This semantic correspondence is not unexpected since it has been used to describe the formal foundations of data-parallel algorithmic skeletons [28]. We take this correspondence one step further by using hylomorphisms as a unifying structure, and by then exploiting the reasoning power provided by the fundamental laws of hylomorphisms. In order to define our type-based approach, we will first define a new language, E, that combines two levels, *Structured Expressions* (S), that enable us to describe a program as a composition of hylomorphisms; and *Structured Parallel Processes* (P), that build on S using nested algorithmic skeletons. A program in E is then either a structured expression $s \in S$ or a parallel program $\mathsf{par}_T\ p$, where $p \in P$. Our revised syntax is shown below. Note that since a $p \in P$ can only appear under a par_T construct, we no longer need to annotate each fun and dc with the collection T of tasks.

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

The denotational semantics of P only changes in the rules that mention e, and by providing a semantics for par_T:

$$\begin{array}{lcl} [\![\mathsf{par}_T\ p]\!] & = & map_T\ \mathcal{S}[\![p]\!] \\ \ldots & & \\ \mathcal{S}[\![\mathsf{fun}\ e]\!] & = & [\![e]\!] \\ \mathcal{S}[\![\mathsf{dc}_{n,F}\ e_1\ e_2]\!] & = & hylo_F\ [\![e_2]\!]\ [\![e_1]\!] \\ \ldots & & \end{array}$$

The corresponding typing rules are entirely standard (Figures 3–4). Finally, it is convenient to define the "parallelism erasure of S", $\overline{S}$. Intuitively, $\overline{S}$ contains no nested parallelism: for all $s \in S$, $s \in \overline{S}$ if and only if s contains no occurrences of the par_T construct. This is equivalent to defining s in the erasure of S, $\overline{S}$, if it is just a composition of atomic functions and hylomorphisms:

$$s \in \overline{S} \;::=\; f \;\mid\; s_1 \circ s_2 \;\mid\; \mathsf{hylo}_F\ s_1\ s_2$$

The structure-annotated type system given in Section 4 below describes how to introduce parallelism to an $s \in \overline{S}$ in a sound way.

3.3.1 Soundness and Completeness.

It is straightforward to show that the type system from Figs. 3– 4 is both sound and complete *wrt* our denotational semantics. Our soundness property is: $\forall e \in E;\ A, B \in \mathsf{Type},\ \vdash e : A \to B \implies ([\![e]\!] \in [\![A \to B]\!])$. The proof is by structural induction over the terms in E, using the definitions of $\vdash e : T$ from Figs. 3–4 and $[\![.]\!]$ above. The corresponding completeness property is: $\forall e \in E;\ A, B \in \mathsf{Type},\ ([\![e]\!] \in [\![A \to B]\!]) \implies \vdash e : A \to B$. The proof is also by structural induction over the terms in E, using the definitions of $\vdash e : A \to B$ from Figs. 3– 4 and $[\![.]\!]$ above.

4. A Type System for Introducing Parallelism

In this section, we present a rigorous way to introduce parallelism without affecting a program's functional behaviour. We annotate top-level program types with an abstraction of the *structure* of the program, $\sigma \in \Sigma$. We define the associated type system together with mechanisms for reasoning about these programs using this structure. 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.

Definition 4.1 (Families of equivalent programs)**.** *We say that an $e \in E$ is in the family of programs that are functionally equivalent to $s \in \overline{S}$, $e \in E_s$,* if and only if *$e\ \mathcal{E}\ s$, for the relation $\mathcal{E}$ that is defined later in this section.*

Let $=_{\text{ext}}$ denote extensional equality: $f \;=_{\text{ext}}\; g \Leftrightarrow \forall x, f\ x \;=\; g\ x$. Since this is not decidable, we use instead a decidable relation $\mathcal{E}$ which implies extensional equality. Each E_s is a family of programs indexed by their structure, i.e. for each family E_s, there is a function $\phi_s : \Sigma \to E_s$ that returns an $e \in E_s$ with the desired structure. Note that not all structures $\sigma \in \Sigma$ are indices of a family E_s, so ϕ_s is a partial function. Given a structure $\sigma \in \Sigma$ and a $s \in \overline{S}$, we use a superscript, s^σ, as notation for $\phi_s(\sigma)$. We define the structure Σ and the relation $\mathcal{E}$ later in this section, and the function ϕ_s in Section 5.

Definition 4.2 (Structure-annotated arrows)**.** *Given a structure $\sigma \in \Sigma$, and an $s \in \overline{S}$ with type $A \to B$, we say that s has type $A \overset{\sigma}{\mapsto} B$, if s is equivalent to a parallel program with structure σ.*

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

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

Figure 5: Structure-Annotated Type System for E.

$$\frac{\vdash \ s \ : \ A \xrightarrow{\sigma} B}{\vdash \ \mathsf{fun}\ s \ : \ T\ A \xrightarrow{\text{FUN}\ \sigma} T\ B} \qquad \frac{\vdash s_1 \ : \ F\ B \xrightarrow{\sigma_1} B \quad \vdash s_2 \ : \ A \xrightarrow{\sigma_2} F\ A \quad G \ = \ \mathsf{base}\ F}{\vdash \mathsf{dc}_{n,F}\ \ s_1\ s_2 \ : \ T\ A \xrightarrow{\text{DC}_{n,G}\ \sigma_1\ \sigma_2} T\ B}$$

$$\frac{n \ : \ \mathbb{N} \quad \vdash p \ : \ T\ A \xrightarrow{\sigma} T\ B}{\vdash \mathsf{farm}\ n\ p \ : \ T\ A \xrightarrow{\text{FARM}_n\ \sigma} T\ B} \qquad \frac{\vdash p_1 \ : \ T\ A \xrightarrow{\sigma_1} T\ B \quad \vdash p_2 \ : \ T\ B \xrightarrow{\sigma_2} T\ C}{\vdash p_1 \parallel p_2 \ : \ T\ A \xrightarrow{\sigma_1 \parallel \sigma_2} T\ C} \qquad \frac{\vdash p \ : \ T\ A \xrightarrow{\sigma} T\ (A + B)}{\vdash \mathsf{fb}\ \ p \ : \ T\ A \xrightarrow{\text{FB}\ \sigma} T\ B}$$

Figure 6: Structure-Annotated Type System for P.

By typechecking $s \ : \ A \overset{\sigma}{\longmapsto} B$, the type system guarantees that there is an equivalent program with structure σ, $s^\sigma \in E_s$. That is, the structured expression s typechecks *if and only if* σ is an index of the family E_s. The definition of ϕ_s is actually an algorithm for deriving a parallel program from a sequential program and a type-level structure, i.e. ϕ_s provides a mechanism for selecting a parallel program that is equivalent to s and has structure σ, for well-typed programs. We state this formally in the form of our main soundness and completeness properties later in this section.

4.1 The Structure-Annotated Type System

The program structure abstraction, $\sigma \in \Sigma$, is defined below.

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

Figs. 5– 6 define the annotated type system that extends our simple type system from Figs. 3– 4, and that associates expressions $e \in E$ with structures $\sigma \in \Sigma$. As before, the global environment, ρ, maps primitive functions to their types. The simple annotated arrow, $e \ : \ A \xrightarrow{\sigma} B$, states that e has *exactly* the structure σ. In order to define $A \overset{\sigma}{\longmapsto} B$, we need to extend the type system further with a *convertibility relation*.

4.1.1 Convertibility.

We extend our type system with a non-structural rule that captures the *convertibility relation*, $\equiv$, for Σ.

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

$\equiv$ is defined in terms of the relations $\equiv_\mathsf{s} \in \ \Sigma_\mathsf{s} \times \Sigma_\mathsf{s}$ and $\equiv_\mathsf{p} \in \Sigma_\mathsf{p} \times \Sigma_\mathsf{p}$, plus a rule that links the Σ_s and Σ_p levels, PAR-EQUIV.

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

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

The structures MAP and ITER are defined in Section 5, and represent the structures of the corresponding hylomorphisms. We define a number of equivalences, starting with $\equiv_\mathsf{p}$. A parallel pipeline structure ($\|$) 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 can be derived from hylomorphisms.

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

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, as required by our type system. In these equivalences, we implicitly assume the necessary well-formedness constraints: any structure under a FUN or DC must be in Σ_s, and the structure under FARM must be in Σ_p. Note that, thanks to the transitivity of $\equiv$, we can use these simple equivalences to derive interesting properties of our parallel structures. For example, the associativity of parallel pipelines does not need to be defined explicitly, since it can be derived from the associativity of composition. We defer the definition of $\equiv_\mathsf{s}$ to Section 4.2.

Definition 4.3 (Convertibility in E)**.** *For all convertibility rules in Σ, there is an equivalent rule in E. We define the equivalence relation $\mathcal{E} \in E \times E$ to be the relation $\equiv$ lifted to E.*

An example that illustrates this is that the PIPE-EQUIV rule corresponds to the rule $(\mathsf{fun}\ s_1 \parallel \mathsf{fun}\ s_2) \quad \mathcal{E}_\mathsf{p} \quad (\mathsf{fun}\ (s_2 \circ s_1))$.

Lemma 1. *Semantic equivalence.* $\forall e_1, e_2 \in E, \quad e_1 \ \mathcal{E} \ e_2 \ \Rightarrow \ [\![e_1]\!] =_{ext} [\![e_2]\!]$

Proof Sketch. Straightforward by induction on the structure of the equivalence relation $\mathcal{E}$, using the denotational semantics of P, and the laws of hylomorphisms (Section 5). □

As a consequence of Lemma 1, the $\mathcal{E}$ relation can be used to define the families E_s. Any extension to $\equiv$ and $\mathcal{E}$ may expose more opportunities for parallelisation in E_s. There remains only the definition of the function ϕ_s. We defer this to Section 5, together with the decision procedure for $\equiv$ and $\mathcal{E}$.

4.1.2 Soundness and Completeness.

Since the annotated type system is a simple extension of that from Figs 3–4 and since the convertibility rule only applies to structures, it is trivial to show that the new type system is both sound and complete *wrt* the original system, once structure is removed, since $\forall e \in E, \sigma \in \Sigma, \ \vdash e \ : \ A \overset{\sigma}{\longmapsto} B \implies \vdash e \ : \ A \to B$. We therefore omit these proofs. Our main soundness and completeness

$$
\begin{array}{lcll}
hylo_F\ in_F\ out_F\ =\ id_{\mu F} & & & \text{HYLO-REFLEX}\\
hylo_F\ (f \circ \eta)\ g\ =\ hylo_G\ f\ (\eta \circ g) & \Leftarrow & \eta\ :\ F \to G & \text{HYLO-SHIFT}\\
(hylo_F\ f\ h_1)\ \circ\ (hylo_F\ h_2\ g) & \Leftarrow & h_1 \circ h_2 = id & \text{HYLO-COMPOSE}\\
f_1 \circ (hylo_F\ g_1\ g_2) \circ f_2\ =\ hylo_F\ g_1'\ g_2' & \Leftarrow & f_1 \text{ strict } \wedge\ f_1 \circ g_1 = g_1' \circ F\ f_1\ \wedge\ g_2 \circ f_2 = F\ f_2 \circ g_2' & \text{HYLO-FUSION}\\
hylo_F\ f\ g \text{ strict} & \Leftarrow & f, g \text{ strict} & \text{HYLO-STRICT}
\end{array}
$$

Figure 7: Hylomorphism Laws

theorems for convertibility ensure that the type system derives only functionally equivalent parallel structures from structured expressions. The proofs of these properties build on a number of details that are introduced in Section 5.

Theorem 1. *Soundness of Conversion.*

$$\forall s \in \overline{S}, \sigma \in \Sigma,\ \vdash s : A \overset{\sigma}{\mapsto} B\ \Rightarrow\ s^\sigma \in E_s$$

Proof Sketch. Since s^σ is a synonym for $\phi_s(\sigma)$, s^σ is in E_s if ϕ_s is defined for σ. This property follows directly from the definition of $\phi_s(\sigma)$ (Def. 5.1) and from Thm 3 in Section 5. □

A straightforward consequence of the soundness of the conversion and of Lemma 1 is that if a structured expression typechecks with type $A \overset{\sigma}{\mapsto} B$, then there always exists a functionally equivalent e whose structure is σ.

Corollary 1. $\forall s \in \overline{S}, \sigma \in \Sigma,\ \vdash s : A \overset{\sigma}{\mapsto} B\ \Rightarrow\ \exists e \in E$ *such that* $e\ :\ A \xrightarrow{\sigma} B$ *and* $[\![e]\!] =_{ext} [\![s]\!]$.

Theorem 2. *Completeness of Conversion.*

$$\forall s \in \overline{S}; \sigma, \sigma' \in \Sigma;\ \ s\ :\ A \xrightarrow{\sigma'} B \wedge s^\sigma \in E_s\ \Rightarrow \vdash s : A \overset{\sigma}{\mapsto} B$$

Proof Sketch. This follows directly from the definition of $\phi_s(\sigma)$ (Def. 5.1) and Thm 3 in Sec. 5. □

4.2 Functional Equivalence

The proofs of soundness and completeness rely on a decision procedure for $\equiv$, as well as on the definition of the ϕ_s function for the families E_s. The definition of $\equiv$ requires a definition of $\equiv_{\mathsf{s}} \in \Sigma_{\mathsf{s}} \times \Sigma_{\mathsf{s}}$. Our $\equiv_{\mathsf{s}}$ adapts the well known hylomorphism laws (Fig. 7) [5, 10, 22], using restricted instances of those laws. These restrictions serve two purposes: i) we avoid checking *strictness* conditions by simply ensuring that all the functions we use are strict; ii) because the equivalences that we can capture are very limited if we assume no knowledge of *atomic functions*, due to the side conditions on the rules, we expose extra structure in our programs.

1. We explicitly represent the in_F, out_F and id functions, with the obvious denotational semantics.
2. We explicitly represent the section of a bifunctor F applied to a structured expression s, as $F\ s$ rather than $F\ s\ \mathsf{id}$. This, plus the strictness assumption, enables us to apply some limited forms of HYLO-SHIFT and HYLO-FUSION.
3. We explicitly represent $\triangle$ and ∇ (Fig. 1). Although we do not define equivalences for these combinators, we use them to define the ITER structure later.

$$
\begin{array}{lcl}
s \in S & ::= & f \mid \langle prim \rangle \mid e_1\ \circ\ e_2 \mid \mathsf{hylo}_F\ e_1\ e_2\\
prim & ::= & \mathsf{in}_F \mid \mathsf{out}_F \mid \mathsf{id} \mid e_1\ \langle op \rangle\ e_2 \mid F\ e\\
op & ::= & \nabla \mid\ \triangle\\
\sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::= & \ldots \mid \text{IN} \mid \text{OUT} \mid \text{ID} \mid \sigma_1\ \langle op \rangle\ \sigma_2 \mid F\ \sigma
\end{array}
$$

With these structures, we can define the special cases of HYLO_F:

$$
\begin{array}{lcl}
\text{MAP}_F, \text{CATA}_F, \text{ANA}_F & : & \Sigma \to \Sigma\\
\text{MAP}_F\ \sigma & = & \text{HYLO}_F\ (\text{IN} \circ F\ \sigma)\ \text{OUT}\\
\text{CATA}_F\ \sigma & = & \text{HYLO}_F\ \sigma\ \text{OUT}\\
\text{ANA}_F\ \sigma & = & \text{HYLO}_F\ \text{IN}\ \sigma\\
\text{ITER}\ \sigma & = & \text{HYLO}_{(+)}\ (\text{ID} \nabla \text{ID})\ \sigma
\end{array}
$$

The typing rules are then easily extended.

$$\frac{}{\vdash \mathsf{id} : A \xrightarrow{\text{ID}} A} \qquad \frac{}{\vdash \mathsf{in}_F : F(\mu F) \xrightarrow{\text{IN}} \mu F}$$

$$\frac{}{\vdash \mathsf{out}_F : \mu F \xrightarrow{\text{OUT}} F(\mu F)}$$

$$\frac{\vdash e\ :\ A \xrightarrow{\sigma} B}{\vdash F\ e\ :\ F\ A\ C \xrightarrow{F\ \sigma} F\ B\ C}$$

$$\frac{\vdash e_1 : A \xrightarrow{\sigma_1} B \qquad \vdash e_2 : A \xrightarrow{\sigma_2} C}{\vdash e_1 \triangle e_2 : A \xrightarrow{\sigma_1 \triangle \sigma_2} B \times C}$$

$$\frac{\vdash e_1 : A \xrightarrow{\sigma_1} C \vdash e_2 : \qquad \vdash e_2 : B \xrightarrow{\sigma_2} C}{\vdash e_1 \nabla e_2 : A + B \xrightarrow{\sigma_1 \nabla \sigma_2} C}$$

The *convertibility relation* is also extended to include some equivalences that are derived from the hylomorphism laws:

$$
\begin{array}{lcll}
\text{ID} \circ \sigma & \equiv_{\mathsf{s}} & \sigma & \text{(ID-LEFT)}\\
\sigma \circ \text{ID} & \equiv_{\mathsf{s}} & \sigma & \text{(ID-RIGHT)}\\
\text{OUT} \circ \text{IN} & \equiv_{\mathsf{s}} & \text{ID} & \text{(OUT-IN-ID)}\\
\text{IN} \circ \text{OUT} & \equiv_{\mathsf{s}} & \text{ID} & \text{(IN-OUT-ID)}\\
\text{HYLO}_F\ \text{IN OUT} & \equiv_{\mathsf{s}} & \text{ID} & \text{(HYLO-ID)}\\
F\ (\sigma_1 \circ \sigma_2) & \equiv_{\mathsf{s}} & F\ \sigma_1 \circ F\ \sigma_2 & \text{(F-COMP)}\\
\text{HYLO}_F\ \sigma_1\ \sigma_2 & \equiv_{\mathsf{s}} & \text{CATA}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 & \text{(HYLO-COMP)}\\
\text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2) & \equiv_{\mathsf{s}} & \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2 & \text{(CATA-COMP)}\\
\text{ANA}_F\ (F\ \sigma_1 \circ \sigma_2) & \equiv_{\mathsf{s}} & \text{MAP}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 & \text{(ANA-COMP)}\\
\text{ANA}_F\ (F\ \sigma_1 \circ \text{OUT}) & \equiv_{\mathsf{s}} & \text{MAP}_F\ \sigma_1 & \text{(ANA-MAP)}
\end{array}
$$

We extend $\mathcal{E}$ in the expected way with the lifted $\equiv_{\mathsf{s}}$, $\mathcal{E}_{\mathsf{s}}$. The rule HYLO-COMP is derived from the HYLO-COMPOSE law. The rules CATA-COMP and ANA-COMP are derived from HYLO-FUSION. It is easy to see how any strictness condition holds in those rules. Finally, the rule ANA-MAP is derived from the HYLO-SHIFT law. It is used only to give a uniform representation of the MAP_F structure.

5. Determining Functional Equivalence

Recall that for all $s \in \overline{S}$, there is a Σ-indexed family E_s. For all well-typed structured expression $s\ :\ A \overset{\sigma}{\mapsto} B$, σ is an index of the family defined by s, i.e. $s^\sigma \in E_s$. 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. Given an $s\ :\ A \xrightarrow{\sigma'} B$, both the typechecking algorithm and the function ϕ_s need to decide whether $\sigma \equiv \sigma'$. This problem has been extensively studied for bicartesian closed categories [7, 13, 35], and it is beyond the scope of this paper

$$
\begin{array}{rcll}
\text{ID} \circ \sigma & \rightsquigarrow_{\mathsf{s}} & \sigma & \text{(ID-CANCEL-L)} \\
\sigma \circ \text{ID} & \rightsquigarrow_{\mathsf{s}} & \sigma & \text{(ID-CANCEL-R)} \\
\text{OUT} \circ \text{IN} & \rightsquigarrow_{\mathsf{s}} & \text{ID} & \text{(OUT-IN-CANCEL)} \\
F\ (\sigma_1 \circ \sigma_2) & \rightsquigarrow_{\mathsf{s}} & F\ \sigma_1 \circ F\ \sigma_2 & \text{(F-SPLIT)}
\end{array}
\qquad
\begin{array}{rcll}
\text{IN} \circ \text{OUT} & \rightsquigarrow_{\mathsf{s}} & \text{ID} & \text{(IN-OUT-CANCEL)} \\
\text{HYLO}_F\ \text{IN OUT} & \rightsquigarrow_{\mathsf{s}} & \text{ID} & \text{(HYLO-CANCEL)} \\
F\ \text{ID} & \rightsquigarrow_{\mathsf{s}} & \text{ID} & \text{(F-ID-CANCEL)} \\
\text{ANA}_F\ (F\ \sigma_1 \circ \text{OUT}) & \rightsquigarrow_{\mathsf{s}} & \text{MAP}_F\ \sigma_1 & \text{(ANA-MAP)}
\end{array}
$$

$$
\begin{array}{rcll}
\text{HYLO}_F\ \sigma_1\ \sigma_2 & \rightsquigarrow_{\mathsf{s}} & \text{CATA}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 & \Leftarrow\ \sigma_1 \neq \text{IN} \wedge \sigma_2 \neq \text{OUT}\ \ \text{(HYLO-SPLIT)} \\
\text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2) & \rightsquigarrow_{\mathsf{s}} & \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2 & \Leftarrow\ \sigma_1 \neq \text{IN}\ \ \text{(CATA-SPLIT)} \\
\text{ANA}_F\ (F\ \sigma_1 \circ \sigma_2) & \rightsquigarrow_{\mathsf{s}} & \text{MAP}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 & \Leftarrow\ \sigma_2 \neq \text{OUT}\ \ \text{(ANA-SPLIT)}
\end{array}
$$

Figure 8: Rewriting system in S

to produce a novel decision procedure for the equality of terms. We consequently use a basic decision procedure, but one that enables interesting parallelisations.

5.1 Reforestation.

We take the standard approach of using term rewriting systems to decide equality in Σ (and hence E). It is well known that if a rewriting system is confluent, then two terms have the same normal form *if and only if* they are equal with respect to the underlying equational theory. If we define a confluent term rewriting system with $\equiv$ as underlying theory, we can use the syntactic equality of normalised forms as our decision procedure. We present the rewriting system in two parts. The first part is derived from orienting the rules in $\equiv$ so that parallelism is erased:

$$
\begin{array}{lcl}
\text{FARM}_n\ \sigma_{\mathsf{p}} & \rightsquigarrow_{\mathsf{p}} & \sigma_{\mathsf{p}} \\
\text{FUN}\ \sigma_1 \parallel \text{FUN}\ \sigma_2 & \rightsquigarrow_{\mathsf{p}} & \text{FUN}\ (\sigma_1 \circ \sigma_2) \\
\text{DC}_{n,F}\ \sigma_1\ \sigma_2 & \rightsquigarrow_{\mathsf{p}} & \text{FUN}\ (\text{HYLO}_F\ \sigma_1\ \sigma_2) \\
\text{FB}\ (\text{FUN}\ \sigma_1) & \rightsquigarrow_{\mathsf{p}} & \text{FUN}\ (\text{ITER}\ \sigma_1) \\
\\
\text{PAR}_T\ (\text{FUN}\ \sigma_{\mathsf{s}}) & \rightsquigarrow_{\mathsf{p}} & \text{MAP}_T\ \sigma_{\mathsf{s}}
\end{array}
$$

The first four rules rewrite terms in Σ_{p} to terms in Σ_{p}, and the last rule rewrites terms in Σ to terms in Σ. We define $\overline{\Sigma_{\mathsf{s}}}$ in an analogous way to $\overline{S}$, and erase as any normalisation procedure for a rewriting system $\rightsquigarrow_{\mathsf{p}}$:

$$
\begin{array}{lcl}
\mathsf{erase} & : & \Sigma \to \overline{\Sigma}_{\mathsf{s}} \\
\mathsf{erase}\ \sigma & = & \sigma',\ \text{s.t.}\ \sigma \rightsquigarrow_{\mathsf{p}}^{*} \sigma' \wedge \nexists \sigma''\ \text{s.t.}\ \sigma' \rightsquigarrow_{\mathsf{p}} \sigma''
\end{array}
$$

Lemma 2. *The rewriting system $\rightsquigarrow_{\mathsf{p}}$ is confluent.*

Proof Sketch. The rewriting system is terminating, since the number of redexes is precisely the number of parallel structures (including PAR_T), which is reduced following each rewriting step. It is also easy to show that any critical pairs arising from these rules have the same normal form. For example, a farm of a pipeline reduces to the same expression regardless of which structure is erased first. By Newman's lemma [1] we can conclude that $\rightsquigarrow_{\mathsf{p}}$ is confluent. □

Since $\rightsquigarrow_{\mathsf{p}}$ is confluent, we know that the result of erase is unique. Recall that all the results that are derived from the equational theory $\equiv$ can be lifted to $\mathcal{E}$. This implies that there is an $\mathsf{erase}_E : E \to \overline{S}$ procedure that is equivalent to erase defined with the rewritings lifted to E. The second step is the normalisation of $\sigma \in \overline{\Sigma}_{\mathsf{s}}$. We once again use a confluent rewriting system derived from $\equiv_{\mathsf{s}}$, and define it modulo associativity of the composition $\circ$. The direction of the rewriting is chosen so a "reforestation" rewriting is performed. Hylomorphisms are first split into catamorphisms and anamorphisms, which are then themselves split into compositions of maps, catamorphisms and anamorphisms. We omit some trivial cases, e.g. $F\ \sigma \circ F\ \sigma^{-1} \rightsquigarrow \text{ID}$, and prioritise the rules that deal with ID to simplify the confluence of the rewriting system. We only consider the inverses IN and OUT, F IN and F OUT, etc. For uniformity reasons, ANA-MAP is applied to the anamorphisms that perform a map computation.

Lemma 3. *The term rewriting system $\rightsquigarrow_{\mathsf{s}}$ is confluent.*

Proof Sketch. The rewriting system is terminating, since the preconditions of the rules ensure that no cycles are introduced. The rewriting system is also locally confluent. It is trivial to observe that any critical pair arising from the ID rules have the same normal form. The critical pairs arising from rules MAP-SPLIT and CATA/ANA-SPLIT can be reduced to the same normal form, by applying F-SPLIT and CATA/ANA-SPLIT and/or ANA-MAP in a different order. For example, we can rewrite any $\text{CATA}_F\ (\sigma_1 \circ F\ (\sigma_2 \circ \sigma_3)) \rightsquigarrow_{\mathsf{s}}^{*} \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\ \sigma_3$. Any problems that appear from the critical pairs of the ID rules and the SPLIT rules can be solved by forcing the ID rules to be applied first, and working modulo associativity. As before, Newman's lemma completes the proof. □

Finally, we define the normalisation procedures for $\overline{\Sigma}_{\mathsf{s}}$ and Σ.

$$
\begin{array}{lcl}
\mathsf{norm}_{\mathsf{s}}\ \sigma & = & \sigma',\ \text{s.t.}\ \sigma \rightsquigarrow_{\mathsf{s}}^{*} \sigma' \wedge \nexists \sigma''\ \text{s.t.}\ \sigma' \rightsquigarrow_{\mathsf{s}} \sigma'' \\
\mathsf{norm} & = & \mathsf{norm}_{\mathsf{s}} \circ \mathsf{erase}
\end{array}
$$

We use a subscript, norm_E, to denote this normalisation procedure lifted to E. Given that the underlying equational theory of the term rewriting system is $\mathcal{E}$, we know that: $\forall e_1, e_2 \in E, (\mathsf{norm}_E\ e_1 = \mathsf{norm}_E\ e_2) \Leftrightarrow (e_1 \leftrightsquigarrow^{*} e_2) \Leftrightarrow (e_1\ \mathcal{E}\ e_2)$.

Theorem 3 (norm defines a decision procedure for $\equiv$)**.** *For all $\sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \equiv \sigma_2$ if and only if $\mathsf{norm}\ \sigma_1 = \mathsf{norm}\ \sigma_2$.*

Proof Sketch. From the properties of $\rightsquigarrow_{\mathsf{p}}$, we derive that it is always true that $\sigma_i \equiv \mathsf{erase}\ \sigma_i$. Since $\rightsquigarrow_{\mathsf{s}}$ is confluent, by the properties of term rewriting systems, we know that $\mathsf{erase}\ \sigma_1 \equiv \mathsf{erase}\ \sigma_2$ if and only if $\mathsf{norm}_{\mathsf{s}}(\mathsf{erase}(\sigma_1)) = \mathsf{norm}_{\mathsf{s}}(\mathsf{erase}(\sigma_2))$. We finish the proof by combining these two facts using the transitivity of $\equiv$ with the definition of norm. □

The fact that we can lift the results from Σ to E implies that we can use this rewriting system not only to reason about program equivalences, but also to define an algorithm to *derive* a parallel program from some $s \in \overline{S}$ and a type-level parallel structure. We sketch this algorithm as the definition of ϕ_s.

Definition 5.1 (ϕ_s)**.** *Let $s \in \overline{S}$, $\sigma_1 \in \overline{\Sigma}_{\mathsf{s}}$, such that $\vdash s : A \xrightarrow{\sigma_1} B$, and $\sigma_2 \in \Sigma$. We define $\phi_s(\sigma_2)$ as follows:*

Let $\sigma_i' = \mathsf{norm}\ \sigma_i$. If $\sigma_1' = \sigma_2'$, then:

1. *Reverse the rewriting steps from σ_2 to σ_2': $\sigma_2' \rightsquigarrow^{*} \sigma_2$.*
2. *Obtain the proof of $\sigma_1 \equiv \sigma_2$ by using $\sigma_1 \rightsquigarrow^{*} \sigma_1'$ and (1).*
3. *Obtain the rewriting steps $\sigma_1 \rightsquigarrow^{*} \sigma_2$ from (2).*
4. *Lift the rewriting steps to E, and apply them to s: $s \rightsquigarrow_E^{*} e$.*

$$\text{EQ}\frac{\Delta = \{\{\}\}}{\sigma \sim \sigma \Rightarrow \Delta} \qquad \text{META}_1^r\frac{\Delta = \{\{m \sim \sigma\}\}}{m \sim \sigma \Rightarrow \Delta} \qquad \text{MAP}_1\frac{\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1}{F\ \sigma_1 \sim F\ \sigma_1' \Rightarrow \Delta_1} \qquad \text{MAP}_2^r\frac{\begin{array}{c}\sigma_1 \circ \sigma_2 \sim F\ m_2 \circ F\ m_3 \Rightarrow \Delta_2 \\ \Delta_1 = \{\{m_1 \sim m_2 \circ m_3\}\}\end{array}}{\sigma_1 \circ \sigma_2 \sim F\ m_1 \Rightarrow \Delta_1 \otimes \Delta_2}$$

$$\text{COMP}_1\frac{\begin{array}{c}\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1 \\ \sigma_2 \sim \sigma_2' \Rightarrow \Delta_2\end{array}}{\sigma_1 \circ \sigma_2 \sim \sigma_1' \circ \sigma_2' \Rightarrow \Delta_1 \otimes \Delta_2} \qquad \text{COMP}_2^r\frac{\begin{array}{cc}\sigma_1 \circ \sigma_2 \sim \sigma_1' \Rightarrow \Delta_{11} & \sigma_3 \sim \sigma_2' \Rightarrow \Delta_{12} \\ \sigma_2 \circ \sigma_3 \sim \sigma_2' \Rightarrow \Delta_{22} & \sigma_1 \sim \sigma_1' \Rightarrow \Delta_{21}\end{array}}{\sigma_1 \circ \sigma_2 \circ \sigma_3 \sim \sigma_1' \circ \sigma_2' \Rightarrow \Delta_{11} \otimes \Delta_{12} \cup \Delta_{21} \otimes \Delta_{22}}$$

$$\text{OP}\frac{\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1 \quad \sigma_2 \sim \sigma_2' \Rightarrow \Delta_2}{\sigma_1\langle op\rangle\sigma_2 \sim \sigma_1'\langle op\rangle\sigma_2' \Rightarrow \Delta_1 \otimes \Delta_2} \qquad \text{HYLO}_1\frac{\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1 \quad \sigma_2 \sim \sigma_2' \Rightarrow \Delta_2}{\text{HYLO}_F\ \sigma_1\ \sigma_2 \sim \text{HYLO}_F\ \sigma_1'\ \sigma_2' \Rightarrow \Delta_1 \otimes \Delta_2}$$

$$\text{HYLO}_2^r\frac{\sigma_1 \circ \sigma_2\ \sim\ \text{HYLO}_F\ m_1\ \text{OUT} \circ \text{HYLO}_F\ \text{IN}\ m_2 \Rightarrow \Delta_1 \quad \sigma_1 \circ \sigma_2\ \sim\ \text{HYLO}_F\ m_1\ \text{OUT} \Rightarrow \Delta_2 \quad \sigma_1 \circ \sigma_2\ \sim\ \text{HYLO}_F\ \text{IN}\ m_2 \Rightarrow \Delta_3}{\sigma_1 \circ \sigma_2\ \sim\ \text{HYLO}_F\ m_1\ m_2 \Rightarrow \Delta_1 \cup \{\{m_2 \sim \text{OUT}\}\} \otimes \Delta_2 \cup \{\{m_1 \sim \text{IN}\}\} \otimes \Delta_3}$$

$$\text{HYLO}_3^r\frac{\begin{array}{c}\sigma_1 \circ \sigma_2\ \sim \text{HYLO}_F\ (\text{IN} \circ F\ m_2)\ \text{OUT} \circ \text{HYLO}_F\ \text{IN}\ m_3 \Rightarrow \Delta_1 \\ \sigma_1 \circ \sigma_2\ \sim \text{HYLO}_F\ (\text{IN} \circ F\ m_2)\ \text{OUT} \Rightarrow \Delta_2 \quad \Delta = \{\{m_1 \sim F\ m_2 \circ m_3\}\}\end{array}}{\sigma_1 \circ \sigma_2\ \sim\ \text{HYLO}_F\ \text{IN}\ m_1 \Rightarrow \Delta \otimes \Delta_1 \cup \Delta \otimes \{\{m_3 \sim \text{OUT}\}\} \otimes \Delta_2}$$

$$\text{HYLO}_4^r\frac{\sigma_1 \circ \sigma_2\ \sim \text{HYLO}_F\ m_2\ \text{OUT} \circ \text{HYLO}_F\ (\text{IN} \circ F\ m_3)\ \text{OUT} \Rightarrow \Delta}{\sigma_1 \circ \sigma_2\ \sim\ \text{HYLO}_F\ m_1\ \text{OUT} \Rightarrow \{\{m_1 \sim m_2 \circ F\ m_3\}\} \otimes \Delta}$$

$$\text{HYLO}_5^r\frac{\sigma_1 \circ \sigma_2 \sim \text{HYLO}_F\ m_1\ \text{OUT} \circ \text{HYLO}_F\ \text{IN}\ \sigma_3 \Rightarrow \Delta_1 \quad \sigma_1 \circ \sigma_2 \sim \mathsf{norm}\ (\text{HYLO}_F\ \text{IN}\ \sigma_3) \Rightarrow \Delta_2 \quad \sigma_3 \neq \text{OUT}}{\sigma_1 \circ \sigma_2 \sim \text{HYLO}_F\ m_1\ \sigma_3 \Rightarrow \Delta_1 \cup \{\{m_1 \sim \text{IN}\}\} \otimes \Delta_2}$$

$$\text{HYLO}_6^r\frac{\sigma_1 \circ \sigma_2 \sim \text{HYLO}_F\ \sigma_3\ \text{OUT} \circ \text{HYLO}_F\ \text{IN}\ m_1 \Rightarrow \Delta_1 \quad \sigma_1 \circ \sigma_2 \sim \mathsf{norm}\ (\text{HYLO}_F\ \sigma_3\ \text{OUT}) \Rightarrow \Delta_2 \quad \sigma_3 \neq \text{IN}}{\sigma_1 \circ \sigma_2 \sim \text{HYLO}_F\ \sigma_3\ m_1 \Rightarrow \Delta_1 \cup \{\{m_1 \sim \text{OUT}\}\} \otimes \Delta_2}$$

$$\text{HYLO}_7^r\frac{\sigma_1 \circ \sigma_2 \sim \text{HYLO}_F\ (\text{IN} \circ F\ m_2)\ \text{OUT} \circ \text{HYLO}_F\ (\text{IN} \circ F\ m_3)\ \text{OUT} \Rightarrow \Delta}{\sigma_1 \circ \sigma_2 \sim \text{HYLO}_F\ (\text{IN} \circ F\ m_1)\ \text{OUT} \Rightarrow \{\{m_1 \sim m_2 \circ m_3\}\} \otimes \Delta}$$

Figure 9: Unification Rules.

Since the typechecking algorithm for our type system needs to decide $\sigma_1 \equiv \sigma_2$ (e.g. using Thm. 3), steps *(1)* to *(2)* can be omitted if we know that $\vdash\ s\ :\ A \xrightarrow{\sigma_2} B$ (recall the proof of Thm. 1). Conversely, if there is some $s \vdash A \xrightarrow{\sigma_1} B$, and $s^{\sigma_2} \in E_s$, we know that there is a proof $\sigma_1 \equiv \sigma_2$ (step (2) in Def. 5.1), and therefore $s \vdash A \xrightarrow{\sigma_2} B$ (recall the proof of Thm. 2).

5.2 Structure Unification.

The structure-annotated types that we have presented so far require the specification of a full structure $\sigma \in \Sigma$. However, it is sometimes sufficient, or desirable, to specify only the relevant parts of this structure. We allow this by introducing *structure metavariables* in Σ. Selecting suitable substitutions for these metavariables can be automated in different ways, as we will see later in this section. Given a set of metavariables, $\mathcal{M}$, we extend the syntax of Σ as follows:

$$\begin{array}{llll} m \in \mathcal{M} & & \sigma\ \in \Sigma & ::=\ \ldots \mid m \\ \sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::=\ \ldots \mid m & \sigma_{\mathsf{p}} \in \Sigma_{\mathsf{p}} & ::=\ \ldots \mid m \end{array}$$

The underscore character denotes a fresh metavariable, e.g. given a fresh metavariable m, $\text{FARM}_n\ \_$ is equivalent to $\text{FARM}_n\ m$.

Definition 5.2 (Substitution Environments)**.** *A substitution environment δ is a mapping of metavariables to structures, $\{m_1 \sim \sigma_1, m_2 \sim \sigma_2, \ldots\}$. We use Δ to denote sets of environments δ.*

The two basic operations with substitution environments are the *application* and the *extension*. We apply a substitution environment δ to a structure σ, denoted by $\delta\sigma$, by replacing all metavariables as defined by δ. The extension of δ_1 with δ_2, $\delta_1\delta_2$, is defined in the expected way. If both substitution environments introduce a cycle or conflicting metavariables, the operation fails. Finally, for sets of substitution environments, we define the set of extensions:

$$\Delta_1 \otimes \Delta_2\ =\ \{\delta_1\delta_2 \mid \delta_1 \in \Delta_1 \wedge \delta_2 \in \Delta_2\}$$

Lemma 4. *For all substitutions δ, for all $\sigma \in \Sigma$, $\mathsf{norm}\ \delta\sigma \equiv \delta(\mathsf{norm}\ \sigma)$.*

Proof Sketch. It is obvious that if $\sigma_1 \equiv \sigma_2$, then $\delta\sigma_1 \equiv \delta\sigma_2$, since δ will apply the same substitution in both σ_1 and σ_2. Since $\sigma \equiv \mathsf{norm}\ \sigma$, we conclude using the reflexivity of $\equiv$. □

Note that we can no longer use the relation $\equiv$ in our typechecking rules, since it does not handle metavariables. Instead, we define the relation $\cong$, and define a decision procedure for it.

Definition 5.3 (Equivalence of the Unified Forms)**.** *We say that $\sigma_1 \cong \sigma_2$ if there is at least a substitution δ that makes $\delta\sigma_1 \equiv \delta\sigma_2$.*

$$\sigma_1 \cong \sigma_2 \doteq \exists\delta, \delta\sigma_1 \equiv \delta\sigma_2$$

The type rule for equivalence changes to use the new relation:

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

In order to typecheck a structured expression with a structure containing metavariables, we need to: i) modify the normalisation procedure; and ii) define a *unification algorithm*. The normalisation procedure is modified as follows:

1. Any erase step on a structure with meta-variables always succeeds by simply adding new meta-variables and a substitution environment δ for those metavariables, e.g.
$$m_1 \parallel m_2 \quad \rightsquigarrow \quad \text{FUN}\ (m_2' \circ m_1')$$
$$\delta = \{m_1 \sim \text{FUN}\ m_1',\ m_2 \sim \text{FUN}\ m_2'\}.$$
2. The constraints of the rules used by the norm_S procedure are modified so that they are never satisfied by metavariables, e.g.
$$\text{HYLO}_F\ \sigma_1\ \sigma_2 \ \rightsquigarrow_{\mathsf{s}}\ \text{CATA}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 \qquad \text{(HYLO-SPLIT)}$$
$$\Leftarrow \quad \sigma_1 \neq \text{IN} \ \wedge\ \sigma_2 \neq \text{OUT} \ \wedge\ \sigma_1 \notin \mathcal{M} \ \wedge\ \sigma_2 \notin \mathcal{M}$$

Although condition 2 is not necessary, it simplifies the unification of structures where σ_1 or σ_2 can be unified to IN or OUT.

Unification Rules. Since unifying two structures may lead to different, but valid unifying substitutions, the unification rules yield the set of all possible *unifying substitutions*, Δ. Disambiguating such situations can be done using cost models, or some other procedure. The rules for unification (Fig. 9) define a unification modulo associativity (rule COMP_2). Each rule with superscript r has a symmetric version l. A statement $\sigma_1 \sim \sigma_2 \Rightarrow \Delta$ means that structure σ_1 unifies with structure σ_2, under a non-empty set of substitutions, $\Delta \neq \varnothing$. Whenever two structures do not correspond to the same syntactic structure, the unification rules make any valid assumption about the metavariables that would allow further rewritings to take place.

Theorem 4 (Soundness of the Unification)**.** *For all* $\sigma_1, \sigma_2 \in \Sigma$,
$\sigma_1 \sim \sigma_2 \Rightarrow \Delta \Longrightarrow \Delta \neq \varnothing \wedge \forall \delta \in \Delta,\ \delta\sigma_1 \equiv \delta\sigma_2$

Theorem 5 (Completeness of the Unification)**.** *For all* $\sigma_1, \sigma_2 \in \Sigma$*, and substitution* δ*,*
$\delta\sigma_1 \equiv \delta\sigma_2 \Longrightarrow \exists \Delta$ *s.t.* $\Delta \neq \varnothing \wedge \sigma_1 \sim \sigma_2 \Rightarrow \Delta$

The proofs of those theorems are standard proofs by induction on the derivations of $\sim$ and $\equiv$, and by case analysis on the metavariables.

Corollary 2. $\cong$*:* $\sigma_1 \cong \sigma_2 \ \Leftrightarrow\ \mathsf{norm}\ \sigma_1 \sim \mathsf{norm}\ \sigma_2 \Rightarrow \Delta$*, i.e. the unification algorithm can be used as a decision procedure for* $\cong$*.*

Proof Sketch. For the proof of the $\Rightarrow$ case, we know that there is at least a δ such that $\delta\sigma_1 \equiv \delta\sigma_2$. From the properties of $\equiv$, we know that $\mathsf{norm}\ \delta\sigma_1 = \mathsf{norm}\ \delta\sigma_2$. Using Lemma 4, we derive that $\delta\ (\mathsf{norm}\ \sigma_1) \equiv \delta\ (\mathsf{norm}\ \sigma_2)$. The completeness of the unification allows us to conclude that $\mathsf{norm}\ \sigma_1 \sim \mathsf{norm}\ \sigma_2 \Rightarrow \Delta$.
The proof of $\Leftarrow$ follows from the soundness of the unification algorithm. We know that $\mathsf{norm}\ \sigma_1 \sim \mathsf{norm}\ \sigma_2 \Rightarrow \Delta$ implies that Δ is non-empty, and that for all $\delta \in \Delta$, $\delta\ (\mathsf{norm}\ \sigma_1) \equiv \delta\ (\mathsf{norm}\ \sigma_2)$. We conclude by selecting any δ from Δ, and then using Lemma 4. □

Using metavariables in structures has some implications. Given a $s \vdash A \xrightarrow{\sigma_1} B$, and a structure σ_2 containing one or more metavariables, σ_2 can no longer be used as an index for a family E_s. Since there may be alternative, but valid substitutions for the metavariables, it follows that $s^{\sigma_2} \subseteq E_s$. Given a unification $\sigma_1 \sim \sigma_2 \Rightarrow \Delta$, we need to apply a $\delta \in \Delta$ to σ_2 in order to use it as an index, $s^{\delta\sigma_2} \in E_s$. This implies that there are many ways to use our approach. On one hand, fixing this σ_2 to be a closed structure without any metavariables, or one that unifies with σ_1 with a unique substitution, provides a way to manually parallelise a program. On the other hand, if σ_2 is defined to be a metavariable, then a fully automated method for selecting a parallel structure would be needed. In between, there are a wide range of *semi-automated* possibilities that can be used to reason about the introduction of parallelism to a program. The automated selection mechanism for a $\delta \in \Delta$ can easily be extended with further parallelisation opportunities. Furthermore, it can be parameterised by architecture-specific details, so that compiling a program for different architectures leads to alternative parallelisations. This is, however, beyond the scope of this paper.

Compositionality and Higher-Order Structured-Arrows. We finish this section with a discussion of the compositionality of our approach. Most of this paper deals with the typing rule for structure-annotated arrows. Extending our work for a language with definitions and with a limited-form of higher-order structured arrows can be done using the unification rules from Fig. 9. Applying some $e : A \xrightarrow{\sigma_1'} B$ to a $f : A \xrightarrow{\sigma_1} B \to C \xrightarrow{\sigma_2} D$ typechecks only if $\sigma_1 \sim \sigma_1' \Rightarrow \Delta$, and the type of this would be annotated with a structure resulting from applying any $\delta \in \Delta$, $f\ e : C \xrightarrow{\delta\sigma_2} D$. One advantage of this is to easily allow the specification of new structures that can be later used in our programs. The corresponding erasure rules would then be derived automatically from such a specification. We would need, however, to verify a back-end for such a language in order to ensure that the operational semantics of the new structures are sound with respect to the specification. Although we do not explain this idea in detail in this paper, we will use a small example of a defined structure in Section 6.3.

Although it seems entirely feasible to support full higher-order structured arrows, there is the question of whether this is desirable: what would types such as $A \xrightarrow{\sigma} B \xrightarrow{\sigma'} C$ or $(A \xrightarrow{\sigma} B) \xrightarrow{\sigma'} C$ mean? Intuitively, the first would be a parallel process with structure σ that produces a parallel process with structure σ', and the second would be a parallel process with structure σ' that takes a parallel process with structure σ as input, and produces an output of type C. In contrast to using functions with a type such as $X \xrightarrow{\sigma_1} Y \to A \xrightarrow{\sigma_2} B$, we have so far not seen the benefits of full higher-order functions in our examples, although this is an idea that may be worth exploring in future work.

6. Examples

Our first two examples revisit *image merge* from Section 1, and *quicksort* from Section 3. In both cases, we show how the type system calculates the convertibility of the structured expression to a functionally equivalent parallel process, and how the convertibility proof allows the structured expression to be rewritten to the desired parallel process. The final two examples show how to use types to parallelise different algorithms, described as structured expressions. These examples show how to easily introduce parallel structure to these algorithms using our type system.

6.1 Image Merge

Recall that *image merge* basically composes two functions: mark and merge. It can be directly parallelised using different combinations of farms and pipelines.

$$\text{IM}_1\ (n, m) \ =\ \text{PAR}_L\ (\text{FARM}\ n\ (\text{FUN A}) \parallel \text{FARM}\ m\ (\text{FUN A}))$$

$$\mathsf{imageMerge} : \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \xrightarrow{\text{IM}_1\ (n,m)} \mathsf{List}(\mathsf{Img} \times \mathsf{Img})$$
$$\mathsf{imageMerge} \ =\ \mathsf{map}_{\mathsf{List}}\ (\mathsf{merge} \circ \mathsf{mark})$$

First, we use our annotated typing rules to produce a derivation tree with the structure of the expression (Fig. 10). The key part is the convertibility proof, that $\text{MAP}_L\ (\text{A} \circ \text{A}) \cong \text{IM}_1\ (n, m)$. We use the decision procedure defined in Section 4 to decide the equivalence of both structures. The erase step is applied as follows:

$$\text{IM}_1\ (n, m) \ \rightsquigarrow^*\ \text{MAP}_L\ (\text{A} \circ \text{A})$$

$$\dfrac{\dfrac{}{\mathsf{replace} \;:\; \mathsf{Img} \times \mathsf{Img} \overset{\mathrm{A}}{\longmapsto} \mathsf{Img}} \qquad \dfrac{}{\mathsf{mark} \;:\; \mathsf{Img} \times \mathsf{Img} \overset{\mathrm{A}}{\longmapsto} \mathsf{Img} \times \mathsf{Img}}}{\mathsf{replace} \circ \mathsf{mark} \;:\; \mathsf{Img} \times \mathsf{Img} \overset{\mathrm{A} \circ \mathrm{A}}{\longmapsto} \mathsf{Img} \times \mathsf{Img}}}{\dfrac{\mathsf{map}_{\mathsf{List}}\,(\mathsf{replace} \circ \mathsf{mark}) \;:\; \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \overset{\mathrm{MAP}_L\ (\mathrm{A} \circ \mathrm{A})}{\longmapsto} \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \qquad \mathrm{MAP}_L\ (\mathrm{A} \circ \mathrm{A}) \cong \mathrm{IM}_1\ (n, m)}{\mathsf{map}_{\mathsf{List}}\,(\mathsf{replace} \circ \mathsf{mark}) \;:\; \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \overset{\mathrm{IM}_1\ (n,m)}{\longmapsto} \mathsf{List}(\mathsf{Img} \times \mathsf{Img})}}$$

Figure 10: Typing Derivation Tree for Image Merge

The final step involves applying the decision procedure for equality of $\overline{S}$. Since the expressions are identical, this is a trivial step. We can now apply this equivalence to the original expression:

$$\begin{array}{l}\mathsf{map}_{\mathsf{List}}\,(\mathsf{merge} \circ \mathsf{mark}) \rightsquigarrow^* \\ \quad \mathsf{par}_{\mathsf{List}}\,(\mathsf{farm}\ n\ (\mathsf{fun}\ \mathsf{mark}) \parallel \mathsf{farm}\ m\ (\mathsf{fun}\ \mathsf{merge}))\end{array}$$

We now show an example of unification. We define $\mathrm{IM}_2\ n = \mathrm{PAR}_L\ (\_ \parallel \mathrm{FARM}\ n\ \_)$. First, we instantiate the structure with fresh metavariables m_1 and m_2. Then, we normalise the structure. We start by applying the erase rewriting system:

$$\mathrm{MAP}_L\ (m_2' \circ m_1') \qquad \delta = \{m_1 \sim \mathrm{FUN}\ m_1',\ m_2 \sim \mathrm{FUN}\ m_2'\}$$

We then apply the normalisation in $\overline{\Sigma_s}$, and the unification rules:

$$\mathrm{MAP}_L\ \mathrm{A} \circ \mathrm{MAP}_L\ \mathrm{A} \sim \mathrm{MAP}_L\ m_2' \circ \mathrm{MAP}_L\ m_1' \Rightarrow \{\{m_1' \sim \mathrm{A}, m_2' \sim \mathrm{A}\}\}$$

The final step is to calculate the extension of the environment δ, and the set of environments that are obtained from the unification:

$$\begin{array}{l}\Delta = \{\delta\} \otimes \{\{m_1' \sim \mathrm{A}, m_2' \sim \mathrm{A}\}\} = \\ \quad \{\{m_1 \sim \mathrm{FUN}\ m_1',\ m_2 \sim \mathrm{FUN}\ m_2', m_1' \sim \mathrm{A}, m_2' \sim \mathrm{A}\}\}\end{array}$$

Applying the substitution environment in Δ to the $\mathrm{IM}_2(n)$, we obtain the structure $\mathrm{PAR}_L\ (\mathrm{FUN\ A} \parallel \mathrm{FARM}_n\ (\mathrm{FUN\ A}))$.

Finally, we briefly discuss how to extend the environment further using a procedure min cost. First, min attempts further rewritings to m_1 and m_2. To ensure termination, the process stops whenever the only option is to introduce a task farm to an existing task farm structure.

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

We will show how to select one of those structures using a simple example cost model. In future work, we will consider how to extend and formalise this cost model. A cost model provides size functions $|\sigma|$ over structures, similar to the idea of *sized types* [17]. We assume that all atomic functions are annotated with their cost models, A_c. The cost of a structure is a function that receives a size, sz, and returns an estimation of its run-time in milliseconds.

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. In this example, we assume that a maximum of 24 cores are available. 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 (κ_1 and κ_2) are given below. In the following examples, we omit these numbers and just provide the estimation for a 24-core architecture with similar overheads for farms, pipelines, divide-and-conquer and feedback.

$$\begin{array}{l}c_1\ [(2048 \times 2048, 2048 \times 2048)]^n \;=\; n \times 25.11ms \\ c_2\ [(2048 \times 2048, 2048 \times 2048)]^n \;=\; n \times 45.21ms \\ \kappa_1\ (9) \;=\; 29.66ms \qquad \kappa_1\ (3 \times 5) \;=\; 60.93ms \\ \kappa_2\ (9, 3 \times 5) \;=\; 114.4ms\end{array}$$

$$\begin{array}{l}\mathsf{cost}\ (\delta_1 \mathrm{IM}_2(n))\ sz \\ \;=\; \max\ \{c_1\ (\dfrac{sz}{n_1}) + \kappa_1(n_1),\ c_2\ (\dfrac{|\mathrm{A}_{c_1}|(sz)}{n \times n_2}) + \kappa_1(n \times n_2)\} \\ \qquad +\ \kappa_2(n_1,\ n \times n_2) \;=\; 3145.69ms \\ \mathsf{cost}\ (\delta_2 \mathrm{IM}_2(n))\ sz \;=\; 25123.81ms \\ \mathsf{cost}\ (\delta_3 \mathrm{IM}_2(n))\ sz \;=\; 3189.60ms \\ \mathsf{cost}\ (\delta_4 \mathrm{IM}_2(n))\ sz \;=\; 25123.81ms\end{array}$$

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

6.2 Quicksort

We will now revisit *quicksort* and show how it can exploit a divide-and-conquer parallel structure.

$$\begin{array}{l}\mathsf{qsorts} \;:\; \mathsf{List}(\mathsf{List}\ A) \to \mathsf{List}(\mathsf{List}\ A) \\ \mathsf{qsorts} \;=\; \mathsf{map}_{\mathsf{List}}\,(\mathsf{hylo}_{F\ A}\ \mathsf{merge}\ \mathsf{div})\end{array}$$

In order to introduce a divide-and-conquer parallel structure, the type system needs to decide:

$$\mathrm{MAP}_L(\mathrm{HYLO}_F\ \mathrm{A}\ \mathrm{A}) \cong \mathrm{PAR}_L\ (\mathrm{DC}_{n,F}\ \mathrm{A}\ \mathrm{A})$$

This can be achieved using a simple parallelism erasure. Consider now a slightly more complex structure:

$$\mathrm{MAP}_L\ (\mathrm{HYLO}_F\ \mathrm{A}\ \mathrm{A}) \;\cong\; \mathrm{PAR}_L\ (\mathrm{FARM}_n\ \_ \parallel \_)$$

Let m_1, m_2 be two fresh metavariables. The parallelism erasure of the right hand side returns the following structure and substitution:

$$\begin{array}{c}\mathrm{PAR}_L\ (\mathrm{FARM}\ n\ m_2 \parallel m_1) \;\rightsquigarrow^*\; \mathrm{MAP}_L\ (m_1' \circ m_2') \\ \delta = \{m_1 \;\sim\; \mathrm{FUN}\ m_1', m_2 \sim \mathrm{FUN}\ m_2'\}\end{array}$$

The normalisation procedure continues by normalising the left and right hand sides of the equivalence following a parallelism erasure. The left hand side is normalised by applying HYLO-SPLIT, F-SPLIT and CATA-SPLIT:

$$\mathrm{MAP}_L\ (\mathrm{HYLO}_F\ \mathrm{A}\ \mathrm{A}) \rightsquigarrow^* \mathrm{MAP}_L\ (\mathrm{CATA}_F\ \mathrm{A}) \circ \mathrm{MAP}_L\ (\mathrm{ANA}_F\ \mathrm{A})$$

The right hand side of the equivalence is normalised by applying F-SPLIT and CATA-SPLIT:

$$\mathrm{MAP}_L\ (m_1' \circ m_2') \rightsquigarrow^* \mathrm{MAP}_L\ m_1' \circ \mathrm{MAP}_L\ m_2'$$

The decision procedure finishes by unifying both structures, and extending the substitution δ with all possible unifications.

$$\begin{array}{c}\mathrm{MAP}_L\ (\mathrm{CATA}_F\ \mathrm{A}) \circ \mathrm{MAP}_L\ (\mathrm{ANA}_F\ \mathrm{A}) \sim \mathrm{MAP}_L\ m_1' \circ \mathrm{MAP}_L\ m_2' \\ \Rightarrow \Delta_1 = \{m_1' \sim \mathrm{CATA}_F\ \mathrm{A}, m_2' \sim \mathrm{ANA}_F\ \mathrm{A}\}\end{array}$$

$$\Delta = \{\delta\} \otimes \Delta_1$$

Again, by applying the only substitution in $\delta' \in \Delta$, we select the final structure:

$$\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{ANA}_F\ \text{A}))\ \|\ \text{FUN}\ (\text{CATA}_F\ \text{A}))$$

The full proof of equivalence ($\cong$) allows us to rewrite quicksort to our desired parallel structure:

$$\begin{aligned}&\mathsf{map}_{\mathsf{List}}(\mathsf{hylo}_{F\ A}\mathsf{merge}\ \mathsf{div}) \rightsquigarrow^*\\ &\quad \mathsf{par}_{\mathsf{List}}\ (\mathsf{farm}\ n\ (\mathsf{fun}\ (\mathsf{ana}_{F\ A}\ \mathsf{div}))\ \|\ \mathsf{fun}\ (\mathsf{cata}_{F\ A}\mathsf{merge}))\end{aligned}$$

We can use our cost model again, where κ_3 is the overhead of a divide-and-conquer structure. In this example, we set the size parameter of our cost model to 1000 lists of 3,000,000 elements, and use the following structure:

$$\mathsf{qsorts}\ :\ \mathsf{List}(\mathsf{List}\ A) \xmapsto{\mathsf{min\ cost}\ \_} \mathsf{List}(\mathsf{List}\ A)$$

$$\begin{aligned}&\mathsf{cost}\ (\text{PAR}_L\ (\text{DC}_{n,F}\ \text{A}_{c_1}\ \text{A}_{c_2}))\ sz\\ &=\ \max\{\max_{1\le i\le n}\{c_2\ (|\text{A}_{c_2}|^i sz)\\ &\quad ,\mathsf{cost}\ (\text{HYLO}_F\ \text{A}_{c_1}\ \text{A}_{c_2})\ (|\text{A}_{c_2}|^n sz)\\ &\quad ,\max_{1\le i\le n}\{c_1\ (|\text{A}_{c_1}|^i|\text{A}_{c_2}|^n sz)\}\}\ +\ \kappa_3(n)\ =\ 42602.72ms\\ &\mathsf{cost}\ (\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{ANA}_L\ \text{A}_{c_2}))\ \|\ (\text{FUN}\ (\text{CATA}_L\ \text{A}_{c_1}))))\ sz\\ &=\ 27846.13ms\\ &\mathsf{cost}\ (\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{HYLO}_F\ \text{A}_{c_1}\ \text{A}_{c_2}))))\ sz\\ &=\ 32179.77ms\\ &\ldots\end{aligned}$$

Since the most expensive part of the quicksort is the divide, and flattening a tree is linear, the cost of adding a farm to the divide part is less than using a divide-and-conquer skeleton for this example.

6.3 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 the eight octants of the space. The leaves of the tree contain the bodies. Then, the cumulative mass and center of mass of each region of the space are calculated. 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:

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

Since this algorithm also involves iterating for a fixed number of steps, we define iteration as a hylomorphism. We assume that the combinator $+$ (Fig. 1) is also defined in Σ_{s}. Additionally, we assume a primitive combinator, that tests a predicate on a value, $(\cdot\ ?)\ :\ (A\to\mathsf{Bool})\to A\to A+A$.

$$\begin{aligned}&\text{LOOP}\ :\ \Sigma\to\Sigma\\ &\text{LOOP}\ \sigma\ =\ \text{HYLO}_{(+)}\ (\text{ID}\triangledown\sigma)\ ((\text{A}+(\text{A}\ \triangle\ (\text{A}\circ\text{A})))\circ(\text{A}\circ\text{A}?))\end{aligned}$$

$$\begin{aligned}&\mathsf{loop}_A\ :\ (A\xmapsto{m}A)\to A\times\mathbb{N}\xmapsto{\text{LOOP}\ m}A\\ &\mathsf{loop}_A\ s\ =\\ &\quad \mathsf{hylo}_{(A+)}\ (\mathsf{id}\ \triangledown\ s)\\ &\qquad ((\pi_1+(\pi_1\ \triangle\ ((-1)\circ\pi_2)))\circ((==0)\circ\pi_2)?)\end{aligned}$$

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, so that each can independently calculate its forces and update the velocity and position; calcForce calculates the force of one body; and move updates the velocity and position of the body.

$$\begin{aligned}&\mathsf{calcMass}\ :\ G\ \mathsf{Octree}\to G\ \mathsf{Octree}\\ &\mathsf{dist}\ :\ \mathsf{Octree}\times\mathsf{List\ Body}\\ &\qquad \to L\ (\mathsf{Octree}\times\mathsf{Body})\ (\mathsf{Octree}\times\mathsf{List\ Body})\end{aligned}$$

The algorithm is:

$$\begin{aligned}&\mathsf{nbody}\ :\ \mathsf{List\ Body}\times\mathbb{N}\xmapsto{\text{LOOP}\ \sigma}\mathsf{List\ Body}\\ &\mathsf{nbody}\ =\ \mathsf{loop}\ (\mathsf{ana}_L\ (L\ (\mathsf{move}\circ\mathsf{calcForce})\circ\mathsf{dist})\\ &\qquad \circ((\mathsf{cata}_G\ (\mathsf{in}_G\circ\mathsf{calcMass})\circ\mathsf{cata}_L\ \mathsf{insert})\ \triangle\ \mathsf{id}))\end{aligned}$$

Since the LOOP defines a fixed structure, we do not allow any rewriting that changes this structure. However, note that our type system still enables some interesting rewritings. In particular, the structure of the loop body is:

$$\sigma = \text{ANA}_L(L\ (\text{A}\circ\text{A})\circ\text{A})\circ(\text{CATA}_G(\text{IN}\circ\text{A})\circ\text{CATA}_L\ \text{A})\ \triangle\ \text{ID}$$

The normalised structure reveals more possibilities for introducing parallelism:

$$\sigma = \text{MAP}_L\text{A}\circ\text{MAP}_L\ \text{A}\circ\text{ANA}_L\ \text{A}\circ(\text{CATA}_G(\text{IN}\circ\text{A})\circ\text{CATA}_L\ \text{A})\ \triangle\ \text{ID}$$

After normalisation, this structure is equivalent to:

$$\sigma = \text{PAR}_L\ (\text{FUN}\ (\text{A}\circ\text{A}))\circ\_$$

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 a cost model. For example, setting the input size to 20,000 bodies:

$$\begin{aligned}&\sigma\ =\text{PAR}_L\ (\text{FARM}\ n\ \_\ \|\ \_)\circ\_\\ &\sigma'=\text{PAR}_L\ (\mathsf{min\ cost}\ (\_\ \|\ \_))\circ\_\end{aligned}$$

$$\begin{aligned}&\mathsf{cost}\ (\text{FUN}\ \text{A}_{c_1}\ \|\ \text{FUN}\ \text{A}_{c_2})\ sz\ =\ 310525.67ms\\ &\mathsf{cost}\ (\text{FARM}_6\ (\text{FUN}\ \text{A}_{c_1})\ \|\ (\text{FUN}\ \text{A}_{c_2}))\ sz\ =\ 55755.43ms\\ &\mathsf{cost}\ (\text{FUN}\ \text{A}_{c_1}\ \|\ \text{FARM}_1\ (\text{FUN}\ \text{A}_{c_2}))\ sz\ =\ 310525.67ms\\ &\mathsf{cost}\ (\text{FARM}_{20}(\text{FUN}\ \text{A}_{c_1})\ \|\ \text{FARM}_4(\text{FUN}\ \text{A}_{c_2}))\ sz\ =\ 15730.46ms\end{aligned}$$

6.4 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 following 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:

$$\begin{aligned}&\mathsf{conv}\ :\ \mathsf{Kern}\to(\mathsf{List\ Img}\xmapsto{\sigma}\mathsf{List\ Img})\\ &\mathsf{conv}\ k\ =\\ &\ \mathsf{map}_{\mathsf{List}}\ (\mathsf{iter}_{\mathsf{Img}}\ (\mathsf{finished?}\circ\mathsf{hylo}_F\ (\mathsf{combine}\circ F\ (\mathsf{kern}\ k))\\ &\qquad\qquad (\mathsf{split}\ k)))\end{aligned}$$

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

$$\begin{aligned}\sigma &= \text{PAR}_L\ (\text{FB}\ (\text{DC}_{n,L,F}\ (\text{A} \circ F\ \text{A})\ \text{A} \parallel \_)) \\ &= \text{PAR}_L\ (\text{FB}\ (\ \text{FARM}\ n\ \_ \parallel \_ \parallel \_)) \\ &= \mathsf{min\ cost}\ (\text{PAR}_L\ (\text{FB}\ (\_ \parallel \_))) \\ &= \ldots\end{aligned}$$

$$\begin{aligned}&\mathsf{cost}\ (\text{PAR}_L\ (\text{FB}\ (\text{A}_{c_1} \parallel \text{A}_{c_2}))\ sz\ = \\ &\quad \sum_{1 \le i, |\text{A}_{c_1}\|\text{A}_{c_2}|^i sz > 0} \mathsf{cost}\ (\text{A}_{c_1} \parallel \text{A}_{c_2})\ (|\text{A}_{c_1} \parallel \text{A}_{c_2}|^i sz) \\ &\quad = 20923.02ms \\ &\mathsf{cost}\ (\text{PAR}_L\ (\text{FB}\ (\text{FARM}_4\ (\text{FUN}\ \text{A}_{c_1}) \parallel (\text{FUN}\ \text{A}_{c_2})))\ sz \\ &\quad = 6649.55ms \\ &\mathsf{cost}\ (\text{PAR}_L\ (\text{FB}\ (\text{FUN}\ \text{A}_{c_1} \parallel \text{FARM}_1\ (\text{FUN}\ \text{A}_{c_2})))\ sz \\ &\quad = 20923.02ms \\ &\mathsf{cost}\ (\text{PAR}_L\ (\text{FB}\ (\text{FARM}_{14}\ (\text{FUN}\ \text{A}_{c_1}) \parallel \text{FARM}_4\ (\text{FUN}\ \text{A}_{c_2})))\ sz \\ &\quad = 2694.30ms \\ &\ldots\end{aligned}$$

Collectively, our examples have demonstrated the use of our techniques for all the parallel structures we have considered, showing that we can easily and automatically introduce parallelism according to some required structure, while maintaining the functional equivalence with the original form.

7. Related Work

There have been some previous treatments of parallelism using types, but these deal only with sizes and productivity. One line of work is using *sized types* [17] to internalise a notion of sizes of streaming data into a type system. This has been extended to a small number of skeletons in the Eden language [27]. While types were useful to prove the termination and productivity of Eden skeletons, our own work focuses on the different, but important, properties of semantic equivalence and cost. The expressive power of hylomorphisms for parallel programming was first explored by Fischer and Gorlatch [4], 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 [24]. 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. Both lines of work are therefore orthogonal, and we can potentially benefit from Morihata's results.

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 [12, 15, 16, 18, 21, 23, 25, 29–31]. 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 [9]. This theorem has been widely used for parallelism [2, 6, 8, 11, 19, 24, 26]. 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 a semi-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 runtime behaviour of the parallel program. Skillicorn and Cai [32] 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. 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.

Finally, in a practical setting, Steuwer et al [33] 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 semi-automated approach provides a way to narrow down this search space, while using cost models to automate the rest. Our approach is in a sense more general, since we allow our parallel structures to be easily extended. However, we could benefit from exploiting their work in GPU-specific rewriting rules and skeletons.

8. Conclusions

This paper has introduced a new type-based approach for representing and reasoning about the structure of parallel programs represented as algorithmic skeletons, *the first ever treatment of parallelism at the type level* that combines reasoning about program equivalences and cost. Crucially, our type system is capable of reasoning about both the functional and the non-functional properties of a parallel program. Given a cost model and the underlying equational theory, we have shown how we can take rational choices at the type level between alternative parallel implementations by appropriately instantiating and transforming high-level parallelism abstractions. This avoids the usual static analysis approach that separates analysis from 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*. It also opens the door to further safe type-level program manipulations, for example. A key aspect of our approach is the use of *hylomorphisms*, combinations of *catamorphisms* and *anamorphisms* (or *fold/unfold* operations), as a single, unifying parallel structure. In this paper we have used hylomorphisms to capture many common patterns of parallelism that are found in the literature, including *inter alia* task farms, pipelines, divide-and-conquer and dynamic feedback. As we have shown in our examples, this single construct is surprisingly powerful, providing a system of canonical representations that is easy to understand and to transform.

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, more sophisticated and accurate cost models are both possible and desirable, including ones that consider e.g. the sizes of data structures. The model we have shown here is, however, more precise and realistic than e.g. typical PRAM models that are widely used in parallelism theory. Thirdly and finally, we have only shown a static analysis here. However, type-based approaches can freely admit dynamic analyses as well. We intend to explore this in future in order to obtain even more general and flexible analyses.

Acknowledgements

This work has been partially supported by the EU H2020 grant "RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach" (code 644235), and by EPSRC grant EP/M027317/1 "C^3: Scalable & Verified Shared Memory via Consistency-directed Cache Coherence".

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] Y.-Y. Chi and S.-C. Mu. Constructing List Homomorphisms from Proofs. In *Proc. APLIAS '11: Asian Symposium on Programming Languages & Systems*, pages 74–88. 2011.

[3] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, London, 1989.

[4] J. Fischer and S. Gorlatch. Turing Universality of Recursive Patterns for Parallel Programming. *Parallel Processing Letters*, 12(02):229–246, 2002.

[5] M. M. Fokkinga and E. Meijer. Program Calculation Properties of Continuous Algebras. Technical Report, CWI, 1991.

[6] A. Geser and S. Gorlatch. Parallelizing Functional Programs by Generalization. *Journal of Functional Programming (JFP)*, 9(06): 649–673, 1999.

[7] N. Ghani. $\beta\eta$-Equality for Coproducts. In *Proc. International Conf. on Typed Lambda Calculi and Applications*, pages 171–185. 1995.

[8] J. Gibbons. Computing Downwards Accumulations on Trees Quickly. *Theoretical Computer Science*, 169(1):67–80, 1996.

[9] J. Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming (JFP)*, 6(4):657–665, 1996.

[10] J. Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 151–203. 2002.

[11] S. Gorlatch. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Science of Computer Programming*, 33(1):1 – 27, 1999.

[12] S. Gorlatch and C. Lengauer. Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism. *Formal Aspects of Computing*, 7(6): 663–682, 1995.

[13] T. Hardin. Confluence Results for the Pure Strong Categorical Logic CCL. λ-calculi as Subsystems of CCL. *Theoretical Computer Science*, 65(3):291–342, 1989.

[14] Y. Hayashi and M. Cole. Static Performance Prediction of Skeletal Parallel Programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002.

[15] Z. Hu, H. Iwasaki, and M. Takechi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19 (3):444–461, 1997.

[16] Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in Calculational Forms. In *Proc. POPL '98: 25th ACM Symposium on Principles of Programming Languages*, pages 316–328, 1998.

[17] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL '96: 23rd ACM Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[18] G. Keller and M. Chakravarty. Flattening Trees. In *Proc. Euro-Par '98: European Conference on Parallelism*, pages 709–719. 1998.

[19] Y. Liu, Z. Hu, and K. Matsuzaki. Towards Systematic Parallel Programming over Mapreduce. In *Proc. Euro-Par 2011: European Conference on Parallelism*, pages 39–50. 2011.

[20] O. Lobachev and R. Loogen. Estimating Parallel Performance, a Skeleton-Based Approach. In *Proc. HLPA '10: Intl workshop on High-level Parallel Prog. and Appls.*, pages 25–34, 2010.

[21] K. Matsuzaki, Z. Hu, and M. Takeichi. Towards Automatic Parallelization of Tree Reductions in Dynamic Programming. In *Proc. SPAA 2006: Symposium on Parallelism in Algorithms and Architecture*, pages 39–48, 2006.

[22] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. ICFP '91: ACM Conf. on Functional Programming*, pages 124–144, 1991.

[23] J. Misra. Powerlist: A Structure for Parallel Recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6): 1737–1767, 1994.

[24] A. Morihata. A Short Cut to Parallelization Theorems. In *Proc. ICFP 2013: 18th ACM Conf. on Functional Programming*, pages 245–256, 2013.

[25] A. Morihata and K. Matsuzaki. Automatic Parallelization of Recursive Functions using Quantifier Elimination. In *Proc. FLOPS '10: Functional and Logic Programming*, pages 321–336. 2010.

[26] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *Proc. PLDI '07: ACM Conf. on Programming Language Design and Implementation*, pages 146–155, 2007.

[27] R. Peña and C. Segura. Sized Types for Typing Eden Skeletons. In *Proc. IFL '01: Intl. Symposium on Implementation of Functional Languages*, pages 1–17, 2001.

[28] F. A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[29] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[30] D. B. Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2):133–158, 1991.

[31] D. B. Skillicorn. *The Bird-Meertens Formalism as a Parallel Model*. Springer, 1993.

[32] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[33] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating Performance Portable Code Using Rewrite Rules. In *Proc ICFP 2015: 20th ACM Conf. on Functional Prog. Lang. and Comp. Arch.*, pages 205–217, 2015.

[34] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977. ISBN 0262191474.

[35] H. Yokouchi. Church-Rosser Theorem for a Rewriting System on Categorical Combinators. *Theoretical Computer Science*, 65(3):271–290, 1989.