Under consideration for publication in J. Functional Programming

Structure, Semantics and Speedup: Reasoning about Structured Parallel Programs using Dependent Types

David Castro, Kevin Hammond, Edwin Brady and Susmit Sarkar
School of Computer Science, University of St Andrews, St Andrews, UK.

Abstract

Despite the increasing importance of multi-core/many-core computers, much treatment of parallel programming is still very informal. There is a clear need for a formal, language-level treatment of parallelism that marries high-level abstractions with strong reasoning. This paper explores the use of dependent types to capture the structure, semantics and execution costs of structured parallel programs, so enabling formal reasoning about their behaviour and speedups. We use algorithmic skeletons to structure parallel programs, lifting the program structure and functionality into a *dependent type*. This allows us to reason about the functional equivalence of differently structured parallel programs at the type level. By providing an operational semantics for skeletal programs, we can derive a formal *cost model* for a program. We have proved the soundness of our operational semantics against the denotational semantics, and that of the cost model against the operational semantics, implementing the relations and associated proofs in the *Idris* dependently-typed language. We have verified the accuracy of the formal cost models against real program executions, showing that they have good predictive power for our parallel skeleton implementation, despite lacking some details. We are thus able to reason at the type level about alternative parallel implementations in terms of their expected execution costs, and therefore to produce provably optimal parallel programs with respect to the formal cost model, which have good and predictable actual parallel behaviour.

1 Introduction

This paper studies the use of *dependent types* for reasoning about the performance and functionality of *structured parallel programs.* The main advantages of structured parallel approaches are that: i) they permit "parallelism by construction", avoiding many problems associated with typical concurrency mechanisms, such as unmatched communication pairs (e.g. deadlock, race-conditions, communication failure); ii) they support higher-level (but still, typically, informal) reasoning; and iii) it is possible to associate (again, usually informally derived) cost models with specific parallel structures. A classic example of a structured approach is Google's *map-reduce* pattern, where parallel programs are defined in terms of independent (and so parallelisable) *map* operations whose results are then combined (*reduced*) into simpler forms (also, perhaps, in parallel). In this paper, we will use nested *algorithmic skeletons* (Cole, 1989): *parametric* implementations of common patterns of parallelism that decouple an algorithm and its parallel structure, and that are usually described in terms of higher-order functions.

Although structured approaches guide the programming process, it is still often difficult to choose the *right* parallel structure, i.e. one that provides reasonable (and preferably optimal) speedups over the original, sequential program. While algorithmic skeletons allow us to create reasonably predictable *cost models* (Pelagatti, 1998; Caromel & Leyton, 2007;

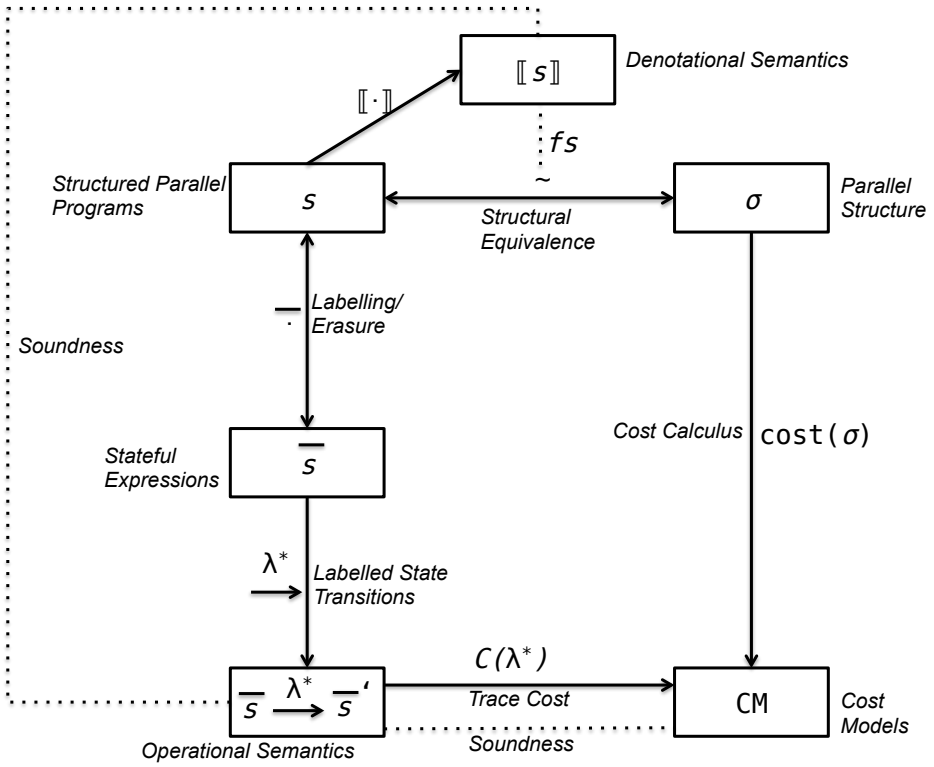


Fig. 1. A Structured Parallel Semantics Approach.

Brown *et al.*, 2013), these cost models are usually derived in a non-rigorous way using e.g. simple profiling techniques, and so lack reasoning and analytical power. What is needed is a source-derived, systematic and formally-motivated approach to determining, *a-priori*, the execution costs and predicted speedups of structured parallel programs. Given suitably strong semantic models, we aim for an automated system that can suggest the best parallelisations for some program, and that can even rewrite it to an equivalent form that offers the best parallel performance. Our key enabling technology is a *dependently-typed* metalanguage that encapsulates a program's semantics, encoding both correctness and cost properties directly into our program types. We use Idris (Brady, 2013a), a full-spectrum dependently-typed programming language that has been previously used to reason about concurrency, state and other effects (Brady & Hammond, 2012; Brady, 2013b).

Figure 1 provides an overview of the approach that we will adopt in this paper. Structured parallel programs ($s \in \mathrm{S}$) can be given a denotational semantics that describes their result as a function over a stream of inputs, $[\![\mathrm{S}]\!]$. They can also be given a trace-based operational semantics, which we derive in two stages, $\bar{s} \in \overline{\mathrm{S}}$ (stateful expressions) and $\xrightarrow{\lambda^*}$ (labelled state transitions). We prove the soundness of the operational semantics with respect to the denotational semantics. We can also derive the structure, σ, of the parallel program from s using the structural equivalence relation, $\backsim_{fs}$, which exposes the functionality of the program, fs, as defined by the denotational semantics for fs, $[\![fs]\!]$. Using our cost calculus, cost, we can produce a *cost model*, *CM* for s. Finally, we prove the soundness of the cost models against the operational semantics, so completing the commuting diagram.

1.1 Example: Introducing Parallelism in Image Convolution

Consider a program that computes the *convolution* of a kernel and a stream of input images. We break this down into two stages: `read` and `process`. The first stage parses the input data and prepares the image for the second stage, which does the actual image convolution.

```
imageConv : Par ICstruct Data Img
imageConv = skel [read, process]

read    : Data -> Img
read    = ...

process : Img  -> Img
process = ...

ICstruct : Struct
ICstruct = ...
```

Here, `imageConv` is a structured (parallel) program, which is defined as a skeleton over the `read` and `process` operations. Its type describes the parallel structure of the convolution as a value of type `Struct`, `ICstruct`, and gives the types of the input and output values. The `skel` function constructs a streaming program from the operations, and ensures that it complies with the type-level description of the parallel structure, `ICStruct`.

One of the key elements of our technique is the use of the type-level `Struct` datatype to abstract over possible parallelisations. Alternative parallelisations will differ only in the type-level annotation, i.e. they will conform to a different instance of `Struct`. By indexing the type of the structured parallel programs on values of the `Struct` type, we provide a way to determine whether any program whose structure is `s1 : Struct` is convertible to a functionally equivalent program whose structure is `s2 : Struct`. Moreover, by enabling cost models to be formally derived from the operational semantics of the structured parallel programs, we provide a way to compare any two instances of `Struct` for any given architecture. Assuming that the cost model is sufficiently strong, this then allows us to derive *provably optimal* parallel programs with respect to that cost model.

For example, given an architectural description of type `Arch`, we can use Idris to determine the costs of different parallel structures `ICstruct1`, `ICstruct2` etc.

```
*ImageConv> cost ICStruct1 uigeadail
2.6497046830091544e-2 : Float

*ImageConv> cost ICStruct2 uigeadail
2.1614735088512402e-2 : Float
```

We can then determine, *at the type level*, which parallel structure has the least cost, so allowing us to reason about the parallel performance of our program through its type.

```
*ImageConv> cost ICStruct2 uigeadail < cost ICStruct1 uigeadail
True : Bool
```

1.2 Novel Contributions

The main novel research contributions of this paper are:

- We derive an *operational semantics* for fundamental parallel skeletons in terms of synchronisation and communication primitives over lock-based queues, and prove the correspondence between our type-level structures and their implementation;
- Based on this operational semantics, we produce a dependently-typed *cost model* for these skeletons, and by instantiating this with the execution costs of actual machine instructions, we encode actual execution costs in the program types; and
- We use type-based reasoning over execution costs to choose the best instantiations of each skeleton, and so to choose the best overall parallel structure and speedup for a given program in terms of a specific cost model.

For the first time, we provide a description of the structure of a parallel program in terms of *dependent types*. We represent parallelism at the type-level, and we show that this can be used to capture a typical set of parallel skeletons. Our cost models and semantics represent the first fully-formal semantics for algorithmic skeletons of which we are aware, in terms of the low-level constructs that are used to implement them[1]. Earlier work on the semantics of skeletons does not capture low-level details about communication and synchronisation of skeletons. Moreover, unlike much earlier work, we are not restricted to highly-regular *data parallelism*, but also consider fundamental *task parallel* constructs. Our cost models accurately[2] predict a tight upper bound on overall execution time and a tight lower bound on speedups for a number of example programs. The full source code of our implementation plus the full proofs described in the paper are available online at `https://bitbucket.org/david_castro/idris_skel`.

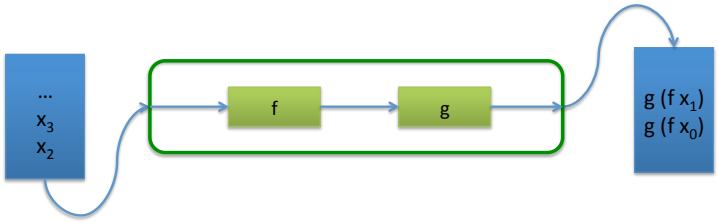
2 Structure

2.1 Algorithmic Skeletons

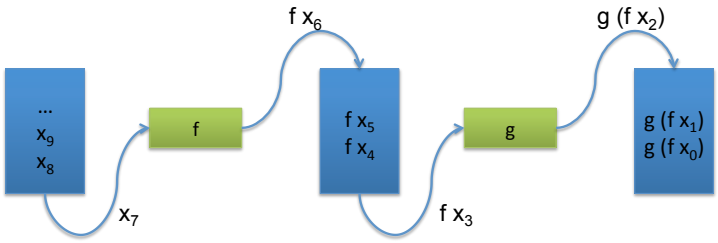
Algorithmic skeletons abstract common patterns of parallelism, providing a clear separation between an algorithm and its parallel structure. Skeletons can be classified as data-parallel, task-parallel or resolution skeletons, depending on their functionality (González Vélez & Leyton, 2010). *Data-parallel skeletons* derive their parallelism from the structure of the data that they operate over. For example a *parallel map* skeleton can apply its function argument in parallel to all elements of a collection type[3]. In the best cases, data-parallelism can give rise to massive amounts of easily detected parallelism. However, data-parallel operations are often quite simple, and it may therefore be necessary to group multiple data items into a single task, for example, in order to obtain acceptable performance. *Task-parallel skeletons* implement less regular patterns of parallelism, often derived from the structure of the program, rather than from the data. For example, a *parallel pipeline* skeleton can be introduced to replace a composition of two or more functions. Finally, *resolution skeletons* describe parallelism for a generic family of problems, such as the *divide & conquer* skeleton. In this paper, we focus on four basic, nestable task-parallel skeletons (shown diagrammatically in Figure 2).

[1] Here, exemplified by lock-based queues and the associated operations.
[2] Though not completely perfectly, as we will see in Section 7.
[3] Provided the data structure has already been fully constructed and/or the elements are independent. In Haskell, for example, this might impose a strictness requirement on the argument list.

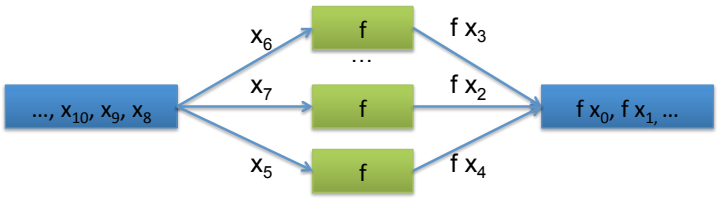
(a) *Func*: Atomic Computations



(b) *Seq*: Sequential composition



(c) *Pipe*: Parallel Pipeline



(d) *Farm*: Parallel Task Farm

Fig. 2. The Four Basic Algorithmic Skeletons used in this paper.

a) *Func* – a wrapper for atomic computations, encapsulated into a streaming skeleton. Given a function $f : A \to B$, we can define the skeleton Func f : *Stream* $A \to$ *Stream* B.

b) *Seq* – the sequential composition of two skeletons. Given two skeletons, s_1 : *Stream* $A \to$ *Stream* B and s_2 : *Stream* $B \to$ *Stream* C, wrapping functions f and g, respectively, the skeleton Seq s_1 s_2 : *Stream* $A \to$ *Stream* C applies $g \circ f$ to all elements of the input stream.

c) *Pipe* – the parallel composition of two skeletons, as a 2-stage pipeline, also computing $g \circ f$, but unlike *Seq*, Pipe s_1 s_2 is defined to execute s_2 on input i *in parallel* with s_1 on input $i+1$.

d) *Farm* – skeleton replication using a task farm. Given the skeleton s : *Stream* $A \to$ *Stream* B, computing f, the skeleton Farm nw s : *Stream* $A \to$ *Stream* B creates nw instances of f, running *in parallel* on separate elements of the shared input stream.

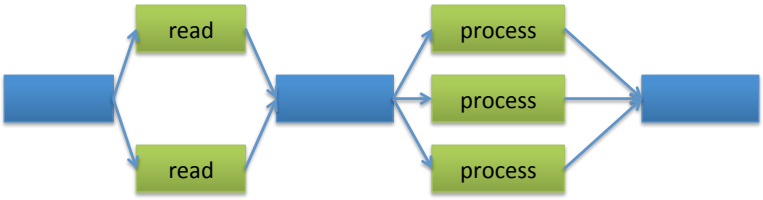


Fig. 3. A two-stage pipeline over two task farms.

We have chosen this set because it is sufficient to express a rich variety of parallel patterns, while still permitting efficient implementations to be produced (Danelutto & Torquati, 2013). Using these skeletons, one possible parallel structure for the image convolution example is as a pipeline of two task farms (Figure 3), where the first stage comprises a farm with two `read` workers, and the second stage comprises a farm with three `process` workers. The structure can be described as:

```
Pipe (Farm 2 (Func read)) (Farm 3 (Func process))
```

Note that, for efficiency, we will allow *Farm* results to be returned in an arbitrary order as they are produced. This will create some complications in the semantics and proofs below, but allows a more general, and more useful, treatment of task farms.

2.2 Types as (Abstract) Interpretations

In a full-spectrum dependently typed language, such as Idris, types may be predicated on *any* value. This allows us to capture, in a formal and machine-checkable way, precise relationships between values. The approach we will take in this paper is to describe the program structure in terms of a dependent type, where the type captures the program's intended *meaning*: whether this is denotational, operational, or both. In a sense, the type gives an *abstract interpretation* of a program. It follows that we can freely manipulate a program's structure, but provided that its type remains the same then we can be confident that it will have exactly the same semantics. Moreover, if part of the type changes as a result of a structural alteration, then we can be confident that the new program is equivalent to the old one under the semantics described by the remainder of the type.

Example: an Expression Language

To illustrate this idea, let us consider a simple expression language with addition, multiplication and numeric literals, implemented as an ordinary algebraic data type:

```
data Expr : Type where
     Plus : Expr -> Expr -> Expr
     Mult : Expr -> Expr -> Expr
     Lit  : Nat
```

We write an evaluator for this expression language as follows:

```
eval : Expr -> Nat
eval (Plus x y) = eval x + eval y
eval (Mult x y) = eval x * eval y
eval (Lit n)    = n
```

Now, suppose we have some expressions in this data type, computed from some input value:

```
exp1, exp2, exp3 : Nat -> Expr
exp1 x = Plus (Plus (Lit x) (Lit x)) Lit x
exp2 x = Plus (Lit x) (Plus (Mult (Lit x) (Lit 2)))
exp3 x = Mult (Lit 3) (Lit x)
```

Which of these expressions are equivalent? That is, which of them will produce the same answer for all possible input values `x`? The answer, as can be seen by writing the expressions in more conventional notation (`x + x + x`, `x + x * 2` and `3 * x`) is *all of them*. One way to show this in Idris would be to write proofs of equalities between the interpretations of each expression by equational reasoning, as follows:

```
exp_equiv : (x : Nat) -> eval (exp1 x) = eval (exp2 x)
exp_equiv = ...
```

By *post hoc* reasoning, therefore, we can show that some expressions are equivalent under some interpretation. In this paper, however, we prefer to take an alternative approach, expressing the interpretation of the expression *in the type itself*. This means that if we require equivalence of expressions under some interpretation then the necessary proof obligations arise naturally in type checking and the equivalence of expressions is shown entirely *by construction*. To achieve this, we represent `Expr` instead as a dependent type that is indexed over its interpretation as a natural number:

```
data Expr : Nat -> Type where
     Plus : Expr x -> Expr y -> Expr (x + y)
     Mult : Expr x -> Expr y -> Expr (x * y)
     Lit  : (x : Nat) -> Expr x
```

The index of an expression gives the corresponding meaning of that expression in Idris. For example, consider the following declaration:

```
three_x : (x : Nat) -> Expr (x + x + x)
```

The type states that the *meaning* of `three_x` is `x + x + x`. All abstract interpretations of `three_x` will comply with this meaning, all implementations which have this type can be considered to be equivalent under the semantics of primitive arithmetic. This is ensured by construction for any type-correct expression. There are several ways to build such an expression. The most straightforward is to add the inputs directly:

```
three_x x = Plus (Plus (Lit x) (Lit x)) (Lit x)
```

Instead of adding the inputs, we may find that `3 * x` is a more efficient implementation:

```
three_x' : (x : Nat) -> Expr (3 * x)
three_x' x = Mult (Lit 3) (Lit x)
```

As it stands, this has a different type to `three_x`, but by using Idris's `rewrite` construct, we can show that `three_x` and `three_x'` are equivalent. The `rewrite` construct takes the following form:

```
rewrite prf in exp
```

Here, `prf` must have a type of the form `a = b`, where expression `a` occurs as a sub-expression in the type of `exp`. Given a proof that `x + x + x = 3 * x`, we can `rewrite` the type of `three_x'` and build an alternative implementation of `three_x` as follows:

```
three_x : (x : Nat) -> Expr (x + x + x)
three_x x = rewrite multThree x in three_x'
   where multThree : (x : Nat) -> x + x + x = 3 * x
         multThree = ...
```

The key idea here, and the essence of our approach, is that we have two different implementations of `three_x`, but we *know* that these are semantically equivalent because they both have the same type. Our strategy is to begin with a direct implementation of an expression, and then to derive an expression with an equivalent denotational semantics (i.e., the same abstract interpretation), but with a different operational semantics (for this paper, that will be an alternative parallelisation). We achieve this by equational reasoning using `rewrite`.

2.3 Capturing Functionality through Types

As described above, we can consider the *meaning* of a program to be defined by its interpretation as a functional value. For skeletal programs, this will be a function from some inputs to some outputs. We can represent this functionality using the `Function` data type, that lifts the functional interpretation directly from the value level to the type level:

```
data Function : Type -> Type -> Type where
  Nil  : Function a a
  (::) : (f : a -> b) -> Function b c -> Function a c
```

An expression of type `Function a b` stores a function of type `a -> b`, built by composing a list of functions. For example,

```
compose : Function b c -> Function a b -> Function a c
compose f g = g :: f :: Nil
```

Exploiting Idris's syntactic sugar for lists, we can write this instead as `compose f g = [g , f]`. This definition of `Function` allows us to reason at the type level about the precise functionality of a program, while maintaining a direct correspondence with the actual program. Unlike model checking approaches, for instance, we do not need to convert our definitions to some simpler form, with the attendant loss of information. The price for this flexibility is, however, that in some cases we must actually evaluate definitions in order to determine concrete type information.

2.4 Types for Algorithmic Skeletons

We can now define a data type `Skel` that is indexed over both its parallel structure (a `Struct`) and its functionality (a `Function a b`), and that describes the skeletal structure of a program:

```
data Skel : Struct -> Function a b -> Type where
     ... -- Skel constructors are defined in Section 4
```

If we have two definitions, `e : Skel p f` and `e' : Skel p' f`, we can therefore see *from the type alone* that although `e` and `e'` have different parallel structures (`p` and `p'`), they have exactly the same functionality (`f`). As a concrete example, consider the function `imageConvFun` that combines a pair of images.

```
imageConvFun : Data -> Image
imageConvFun =  process . read
  where read    : Data  -> Image
        process : Image -> Image


imageConv_1 : Skel (Seq Func Func)          [read, process]
imageConv_2 : Skel (Farm 8 (Seq Func Func)) [read, process]
```

Both `imageConv_1` and `imageConv_2` implement `imageConvFun`. They provide the same functionality, but have different parallel structures: `imageConv_1` composes the two functions sequentially, where `imageConv_2` creates a task farm with 8 workers, each of which composes the two functions sequentially. We will return to this example in Section 4. The remainder of this paper will consider in detail how to describe and reason about such parallel structures using an approach that is informed by a strong semantics basis.

3 Semantics

3.1 Denotational Semantics

Our denotational semantics operates on values in the domain of skeletal expressions, S. defined below. $\mathbb{V}$ is the domain of values (which we will leave partly unspecified), $\mathbb{V} \to \mathbb{V}$ represents functions over those values, and $\mathbb{V}^*$ represents sequences of values (including lists and streams). Σ is the domain of skeleton structures, corresponding to S.

$$
\begin{array}{lcll}
n \in \mathbb{N} & = & \{0,1,\ldots\} & \textit{natural numbers} \\
v,x \in \mathbb{V} & = & \ldots \mid \mathbb{F} \mid \mathbb{L} & \textit{values} \\
f \in \mathbb{F} & = & \mathbb{V} \to \mathbb{V} & \textit{functions} \\
fs \in \mathbb{L} & = & \mathbb{V}^* & \textit{sequences} \\
s \in \mathrm{S} & ::= & \mathsf{func}\, f \mid \mathsf{farm}\, n\, s \mid \mathsf{pipe}\, s\, s \mid \mathsf{seq}\, s\, s & \textit{skeletal expressions} \\
\sigma \in \Sigma & ::= & \mathsf{Func} \mid \mathsf{Farm}\, n\, \sigma \mid \mathsf{Pipe}\, \sigma\, \sigma \mid \mathsf{Seq}\, \sigma\, \sigma & \textit{skeleton structure}
\end{array}
$$

The denotational semantics of a skeletal expression, S is defined by $[\![\mathrm{S}]\!]$. This yields a value of type $\mathbb{V}^* \to \mathbb{V}^*$, that maps a stream of input values to a stream of results:

$$
\begin{array}{l}
[\![\mathrm{S}]\!] \;:\; \mathbb{V}^* \to \mathbb{V}^* \\
\begin{array}{lcl}
[\![\mathsf{func}\, f]\!] & = & \mathsf{map}\, f \\
[\![\mathsf{pipe}\, s_1\, s_2]\!] & = & [\![s_2]\!] \circ [\![s_1]\!] \\
[\![\mathsf{seq}\, s_1\, s_2]\!] & = & [\![s_2]\!] \circ [\![s_1]\!] \\
[\![\mathsf{farm}\, n\, s]\!] & = & \mathsf{perm} \circ [\![s]\!]
\end{array}
\end{array}
$$

Here, $\circ : (\mathbb{V} \to \mathbb{V}) \to (\mathbb{V} \to \mathbb{V}) \to \mathbb{V} \to \mathbb{V}$ denotes the pairwise composition of two functions, $\mathsf{map} : (\mathbb{V} \to \mathbb{V}) \to \mathbb{V}^* \to \mathbb{V}^*$ applies a function to all the elements in a list, and $\mathsf{perm} : \mathbb{V}^* \to \mathbb{V}^*$ arbitrarily permutes a list of elements, so allowing farm results to be returned in an arbitrary order, to allow the option of a more efficient implementation, as described above. We can now define some auxiliary relations that will allow us to state strong properties about $[\![\_]\!]$.

Definition 3.1 (*Structural Equivalence Relation*)
The $\backsim_{fs}$ relation relates values in the domains of S and Σ via the list of functions $fs \in (\mathbb{V} \to \mathbb{V})^*$. $s \backsim_{fs} \sigma$ indicates that s uses the functions fs **in order** in the structure σ.

$$
\frac{}{\mathsf{func}\, f \;\backsim_{\langle f \rangle}\; \mathsf{Func}} \qquad \frac{s \;\backsim_{fs}\; \sigma}{\mathsf{farm}\, n\, s \;\backsim_{fs}\; \mathsf{Farm}\, n\, \sigma} \qquad \frac{s_1 \;\backsim_{fs_1}\; \sigma_1 \qquad s_2 \;\backsim_{fs_2}\; \sigma_2}{\mathsf{pipe}\, s_1\, s_2 \;\backsim_{fs_1 \oplus fs_2}\; \mathsf{Pipe}\, \sigma_1\, \sigma_2}
$$

$$
\frac{s_1 \;\backsim_{fs_1}\; \sigma_1 \qquad s_2 \;\backsim_{fs_2}\; \sigma_2}{\mathsf{seq}\, s_1\, s_2 \;\backsim_{fs_1 \oplus fs_2}\; \mathsf{Seq}\, \sigma_1\, \sigma_2}
$$

Here, $\oplus$ denotes the concatenation of two lists. Given $s \in \mathrm{S}$, $\sigma \in \Sigma$ and $fs \in (\mathbb{V} \to \mathbb{V})^*$, such that $s \backsim_{fs} \sigma$, fs directly determines the denotational semantics of s, $[\![s]\!]$, as shown by Theorem 3.1.

Definition 3.2 (*Reverse Composition*)
The reverse composition of a list of functions, $\bigcirc : (\mathbb{V} \to \mathbb{V})^* \to \mathbb{V} \to \mathbb{V}$ is defined as: $\bigcirc\langle f_1, \ldots, f_n\rangle = f_n \circ \ldots \circ f_1$

3.1.1 Functional Correctness of the Denotational Semantics.

We are now able to state a key soundness result for the denotational semantics. Theorem 3.1 states that the denotational semantics of any $s \in \mathrm{S}$ is some valid permutation of the result of applying the functions that are contained within s. That is, the functionality of s fully determines its results, but in some arbitrary order. This allows the possibility of using one or more *farms*. Given $s \in \mathrm{S}$, we define the *functions of* s, $\mathsf{funcs}(s) = fs$, such that $\exists \sigma \in \Sigma, s \backsim_{fs} \sigma$.

Theorem 3.1
For all $s \in \mathrm{S}$, $[\![s]\!] = \mathsf{perm} \circ \mathsf{map}\ (\bigcirc(\mathsf{funcs}(s))$.

Proof
By induction on the structure of s. The case where $s = \mathsf{func}\ f$ is trivial. In particular, the permutation is the identity function. The seq and pipe cases can be easily proved using fundamental properties of map and perm, *viz.*: $(\mathsf{map}\ f) \circ (\mathsf{map}\ g) = \mathsf{map}\ (f \circ g)$, $\mathsf{map}\ f \circ \mathsf{perm} = \mathsf{perm} \circ \mathsf{map}\ f$ and $\mathsf{perm} \circ \mathsf{perm} = \mathsf{perm}$. The proof of the farm case involves introducing a fresh arbitrary permutation and using the property $\mathsf{perm} \circ \mathsf{perm} = \mathsf{perm}$. □

Corollary 3.1
For all $fs \in (\mathbb{V} \to \mathbb{V})^*$, $s \in \mathrm{S}$ and $\sigma \in \Sigma$, such that $s \backsim_{fs} \sigma$, $[\![s]\!] = \mathsf{map}\ (\bigcirc fs)$ iff s contains no farm subexpressions (transitively).

Corollary 3.1 follows from the proof of Theorem 3.1, where new permutations are introduced only in the farm case.

Definition 3.3 (*Functional Equivalence*)
We say that any two skeletal expressions $s_1, s_2 \in \mathrm{S}$ are *functionally equivalent* iff $[\![s_1]\!] = [\![s_2]\!]$.

Using Theorem 3.1, if $\sigma_1, \sigma_2 \in \Sigma$ and $fs_1, fs_2 \in (\mathbb{V} \to \mathbb{V})^*$ such that $s_1 \backsim_{fs_1} \sigma_1$ and $s_2 \backsim_{fs_2} \sigma_2$, $[\![s_1]\!] = [\![s_2]\!]$ is equivalent to $\mathsf{perm} \circ \mathsf{map}\ (\bigcirc fs_1) = \mathsf{perm} \circ \mathsf{map}\ (\bigcirc fs_2)$. Since perm denotes any arbitrary permutation, we cannot guarantee that both sides of the equality return the same output in the same order unless we use Corollary 3.1 and avoid using task farms. We can, however, weaken this condition by defining *equality modulo permutations*.

Definition 3.4 (*Equality Modulo Permutations*)
For all $xs, ys \in \mathbb{V}^*$ such that $xs = \mathsf{perm}(ys)$, we say that $xs =_{\mathscr{P}} ys$.

It is easy to show that $=_{\mathscr{P}}$ is reflexive, symmetric and transitive. We will abuse this notation slightly and also write $f =_{\mathscr{P}} g$ instead of $\forall xs \in \mathbb{V}^*, f(xs) =_{\mathscr{P}} g(xs)$. This allows us to define a weaker notion of functional equivalence.

Definition 3.5 (*Weak Functional Equivalence*)
Any two skeletal expressions $s_1, s_2 \in \mathrm{S}$ are *weakly functional equivalent* iff $[\![s_1]\!] =_{\mathscr{P}} [\![s_2]\!]$.

This definition allows us to determine the (weak) equivalence of two skeletons.

Theorem 3.2 (*Weak Functional Equivalence of Skeletons*)
For all $s_1, s_2 \in \mathrm{S}$, if $\mathsf{funcs}(s_1) = \mathsf{funcs}(s_2)$, then $[\![s_1]\!] =_{\mathscr{P}} [\![s_2]\!]$.

Proof
Also by using Theorem 3.1, and by annotating each perm function with a mapping function ϕ, the necessary condition for any two $s_1, s_2 \in \mathrm{S}$ is that $\mathsf{perm}_{\phi_1} \circ \mathsf{map}\ (\bigcirc(\mathsf{funcs}(s_1))) = \mathsf{perm}_{\phi_3} \circ (\mathsf{perm}_{\phi_2} \circ \mathsf{map}\ (\bigcirc(\mathsf{funcs}(s_2)))$. Since the ϕ_i are bijections, we can see that the equality is true if $\phi_3 = \phi_1 \circ \phi_2^{-1}$, since $\phi_3 \circ \phi_2 = \phi_1 \circ \phi_2^{-1} \circ \phi_2 = \phi_1$. Since the same perm_{ϕ_1} appears on both sides of the equality, it is sufficient to require that $\mathsf{map}\ (\bigcirc(\mathsf{funcs}(s_1))) = \mathsf{map}\ (\bigcirc(\mathsf{funcs}(s_2)))$. This is true if $\mathsf{funcs}(s_1) = \mathsf{funcs}(s_2)$. □

An important remark about Theorem 3.2 is that if $\mathsf{funcs}(s_1) \neq \mathsf{funcs}(s_2)$, we cannot claim anything about the equivalence of s_1 and s_2, since $\bigcirc fs_1$ might be equal to $\bigcirc fs_2$ even if both lists of functions are not (syntactically) equal. It is, however, reasonable to keep the condition $fs_1 = fs_2$ at a syntactic level, since determining the equivalence of two arbitrary functions is an undecidable problem.

3.2 Operational Semantics

Given some $s \rightharpoonup_{fs} \sigma$, we would like to: (1) use *fs* to reason about the functional behaviour of a skeletal expression; and (2) use σ to reason about its operational behaviour. We have already demonstrated (1) above, using our denotational semantics, but this is not enough for (2), and we must therefore also define an operational semantics. Here, we will use a simple trace semantics to illustrate our approach. Our approach does not, however, depend on this particular semantics, and it would be possible to replace it with a different operational semantics, if desired, e.g. one that also considered more detailed cache behaviours. Our operational semantics is defined as a triple:

$$\langle \mathscr{P}, \Lambda, \rightarrow \rangle$$

where

$\mathscr{P}$	is the set of all valid *program states* (Def. 3.6);
Λ	is a set of *labels* (Def. 3.9);
$\rightarrow$	is the *step* state transition relation (Fig. 4), $(\rightarrow) \in \mathscr{P} \times \Lambda \times \mathscr{P}$.

A valid program execution is one where $p_0 \xrightarrow{\lambda_0} p_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} p_n \xrightarrow{\lambda_n} p$, such that $\forall i : 0 \leq i \leq n,\ p_i \in \mathscr{P} \wedge \lambda_i \in \Lambda$, where p_0 is an *initial* state and $p \in \mathscr{P}$ is a *final* state. We now provide the definitions that are needed to specify this operational semantics, before we show the state transition relation, and state the key termination and soundness theorems. Our first set of definitions relates to the program state. Skeletal computations are defined operationally in terms of streaming processes that transform input queues of arguments into output queues of results in accordance with the denotational semantics. These processes are constructed directly from the structure of a skeletal program, and embed information about evaluation state and input/output queues.

Definition 3.6 (*Stateful Expressions and Program State*)
Let $\omega \in \Omega$, $n \in \mathbb{N}$, $x \in \mathbb{V}$, $q \in \mathbb{V}^*$ and $f \in \mathbb{V} \to \mathbb{V}$. We define the syntax of *stateful expressions*, i.e. expressions with an embedded evaluation state, to be:

$$\begin{array}{lll} st \in ST & ::= & \mathtt{G} \mid \mathtt{E}x \mid \mathtt{P}x \qquad\qquad \textit{state of } \overline{\mathsf{func}} \\ \bar{s} \in \bar{\mathrm{S}} & ::= & \overline{\mathsf{func}}_{\omega}\ st\ f \mid \overline{\mathsf{farm}}\ \bar{s}^* \mid \overline{\mathsf{pipe}}\ q\ \bar{s}\ \bar{s} \mid \overline{\mathsf{seq}}\ q\ \bar{s}\ \bar{s} \qquad \textit{stateful expr.} \end{array}$$

There is a one-to-one correspondence between skeletal expressions and the corresponding stateful expressions. Function skeletons, $\mathsf{func}\ f$, are instantiated to a corresponding process, $\overline{\mathsf{func}}_{\omega}\ st\ f$, which is annotated with a location, ω (defined below) and a state, st. This state may be: $\mathtt{G}$, if the process is idle waiting to read from a queue; $\mathtt{E}x$, if it is ready to evaluate the embedded function using x as argument; or $\mathtt{P}x$, if it is ready to put x in the output queue. The pipe and seq expressions are both instantiated with the state of the intermediate queue that links their sub-computations, q, and farm expressions are instantiated with a sequence of workers, $< \bar{s}_1, \ \ldots, \ \bar{s}_n >$, which must all have the same structure and functionality, s, but which will have different internal states, queues, and locations. A *program state* is then a triple consisting of a *stateful expression* together with a specific input queue and a specific output queue.

$$p \in \mathscr{P} \quad = \quad \langle q_1, \bar{s}, q_2 \rangle \qquad\qquad \textit{program state}$$

Definition 3.7 (*Idle, Initial and Final States*)
We say that a *stateful expression* $\bar{s}$ is *idle*, $\vdash_{\mathsf{idle}} \bar{s}$, if all its intermediate queues are empty and all the atomic function wrappers are in the idle state ($\mathtt{G}$).

$$\frac{}{\vdash_{\mathsf{idle}} \overline{\mathsf{func}}_{\omega}\ \mathtt{G}\ f} \qquad \frac{\vdash_{\mathsf{idle}} \bar{s}_1 \ \vdash_{\mathsf{idle}} \bar{s}_2}{\vdash_{\mathsf{idle}} \overline{\mathsf{seq}}\ \langle\rangle\ \bar{s}_1\ \bar{s}_2} \qquad \frac{\vdash_{\mathsf{idle}} \bar{s}_1 \ \vdash_{\mathsf{idle}} \bar{s}_2}{\vdash_{\mathsf{idle}} \overline{\mathsf{pipe}}\ \langle\rangle\ \bar{s}_1\ \bar{s}_2} \qquad \frac{\vdash_{\mathsf{idle}} \bar{s}_1 \ \ldots \ \vdash_{\mathsf{idle}} \bar{s}_n}{\vdash_{\mathsf{idle}} \overline{\mathsf{farm}}\ \langle \bar{s}_1, \ldots, \bar{s}_n \rangle}$$

A program state, p, is *initial* if $p = \langle q_{\mathrm{in}}, \bar{s}, \langle\rangle\rangle \wedge \vdash_{\mathsf{idle}} \bar{s}$ and a program state, p, is *final* if $p = \langle \langle\rangle, \bar{s}, q_{\mathrm{out}} \rangle \wedge \vdash_{\mathsf{idle}} \bar{s}$.

Locations, Labels and Actions

Our next set of definitions relates to identifying workers and the actions that they perform.

Definition 3.8 (*Locations*)
Locations identify unique workers within a skeletal expression, s.

$$\begin{array}{lll} \Omega & = & \Pi^* \qquad\qquad \text{(locations)} \\ \pi \in \Pi & ::= & \mathsf{pl} \mid \mathsf{pr} \mid \mathsf{sl} \mid \mathsf{sr} \mid \mathsf{w}n \qquad \text{(positions)} \end{array}$$

Starting at the root of s, each position in the sequence uniquely selects one of the sub-workers. The pl and pr positions represent the left and right branches of a pipeline; the sl and sr positions represent the left and right branches of a sequential composition; and the $\mathsf{w}n$ positions represent the nth worker of a farm. When s is a func, the path will be empty, ε. Locations will be used to label atomic worker functions and to map them to physical processor *cores* in some concrete architecture.

Definition 3.9 (*Labels*)
The set of labels Λ is a set of actions, A, annotated with the worker location that performs it, Ω:

$$\begin{array}{lll} \Lambda & = & \mathrm{A}^{\Omega} \qquad\qquad \text{(labels)} \\ \alpha \in \mathrm{A} & ::= & \mathsf{g}q \mid \mathsf{e}(f,x) \mid \mathsf{p}(x,q) \quad \text{(actions)} \end{array}$$

$\mathtt{g}q$ represents the action of dequeuing (*getting*) an element of q; $\mathtt{e}(f,x)$ is the action of evaluating function f with x as argument; and $\mathtt{p}(x,q)$ is the action of enqueuing (*putting*) element x in q.

Definition 3.10 (*Labelling and Erasure*)
Let $\omega \in \Omega$ be the location of a subexpression $s \in \mathrm{S}$, we define the translation scheme $\mathscr{L}$ as a function parameterised with ω, that translates s to an idle $\bar{s}$ with the atomic functions of $\bar{s}$ labelled using ω as prefix for their locations. We also define its inverse $\mathscr{E}$, *state erasure*, that removes the embedded state and the labels from the atomic function wrappers.

$$
\begin{array}{lcl}
\mathscr{L}_\omega[\![\_]\!] & : & \mathrm{S} \to \overline{\mathrm{S}} \\
\mathscr{L}_\omega[\![\mathsf{func}\, f]\!] & = & \overline{\mathsf{func}_\omega}\ \mathtt{G}\, f \\
\mathscr{L}_\omega[\![\mathsf{seq}\ s_1\ s_2]\!] & = & \overline{\mathsf{seq}}\ \langle\rangle\ (\mathscr{L}_{\omega.\mathsf{sl}}[\![s_1]\!])\ (\mathscr{L}_{\omega.\mathsf{sr}}[\![s_2]\!]) \\
\mathscr{L}_\omega[\![\mathsf{pipe}\ s_1\ s_2]\!] & = & \overline{\mathsf{pipe}}\ \langle\rangle\ (\mathscr{L}_{\omega.\mathsf{pl}}[\![s_1]\!])\ (\mathscr{L}_{\omega.\mathsf{pr}}[\![s_2]\!]) \\
\mathscr{L}_\omega[\![\mathsf{farm}\ n\ s]\!] & = & \overline{\mathsf{farm}}\langle \mathscr{L}_{\omega.\mathsf{w}1}[\![s]\!], \ldots, \mathscr{L}_{\omega.\mathsf{w}n}[\![s]\!]\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{E}[\![\_]\!] & : & \overline{\mathrm{S}} \to \mathrm{S} \\
\mathscr{E}[\![\overline{\mathsf{func}_\omega\ st\, f}]\!] & = & \mathsf{func}\, f \\
\mathscr{E}[\![\overline{\mathsf{seq}}\ q\ \bar{s}_1\ \bar{s}_2]\!] & = & \mathsf{seq}\ (\mathscr{E}[\![\bar{s}_1]\!])\ (\mathscr{E}[\![\bar{s}_2]\!]) \\
\mathscr{E}[\![\overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2]\!] & = & \mathsf{pipe}\ (\mathscr{E}[\![\bar{s}_1]\!])\ (\mathscr{E}[\![\bar{s}_2]\!]) \\
\mathscr{E}[\![\overline{\mathsf{farm}}\langle \bar{s}_1, \ldots, \bar{s}_n\rangle]\!] & = & \mathsf{farm}\ n\ s \\
 & & \quad \textit{where}\ \forall i \in [1\ldots n], \mathscr{E}[\![\bar{s}_i]\!] = s
\end{array}
$$

When there is no ambiguity, we will write $\mathscr{L}[\![s]\!]$ for $\mathscr{L}_\varepsilon[\![s]\!]$, and s for $\mathscr{E}[\![\bar{s}]\!]$.

Equivalence of Idle States

Lemma 3.1
For all $\bar{s} \in \overline{\mathrm{S}}$, $\vdash_{\mathsf{idle}} \bar{s}$ iff $\bar{s} = \mathscr{L}(s)$.

This lemma states that all idle $\bar{s}$ are syntactically equal to $\mathscr{L}(s)$. That is, if a $\overline{s_1}$ has the same structure and functionality as some other $\overline{s_2}$, and both are idle, then both are the same. Moreover, if an $\bar{s} = \mathscr{L}(\mathscr{E}(s))$ then s is idle. The lemma can be easily proved by induction on the structure of s.

State Transition Relation

Figure 4 describes the state transition relation of the operational semantics. A rule of the form $\langle q_{\mathrm{in}}, \bar{s}, q_{\mathrm{out}}\rangle \xrightarrow{\alpha^c} \langle q'_{\mathrm{in}}, \bar{s}', q'_{\mathrm{out}}\rangle$ states that an action α executing on core c changes the state of q_{in} to q'_{in}, $\bar{s}$ to $\bar{s}'$ and q_{out} to q'_{out} in a single atomic rewriting step. Since a step rewrites only the state of one atomic function wrapper at a time, at most one queue at a time will be rewritten. The GET, PUT and EVAL rules describe the operations of getting an element from a queue, putting an element into a queue and evaluating a function applied to an argument, respectively. Note that we can choose between $\textsc{Pipe}_1$ and $\textsc{Pipe}_2$ non-deterministically, as well as between the different $\textsc{Farm}_i$. The condition on the $\textsc{Seq}_2$ rule ensures that the sub-expressions in a seq are never run in parallel. We will see in Sec. 5 how to use these sequences of labelled actions to reason about the costs of different traces.

$$\textsc{Get}\ \frac{|q_{\text{in}}| > 0 \qquad (x, q'_{\text{in}}) = \mathsf{dequeue}(q_{\text{in}})}{\langle q_{\text{in}}, \overline{\mathsf{func}}_\omega\ \mathtt{G}\,f, q_{\text{out}}\rangle \xrightarrow{\mathsf{g}^\omega q_{\text{in}}} \langle q'_{\text{in}}, \overline{\mathsf{func}}_\omega\ \mathtt{E}x\,f, q_{\text{out}}\rangle}$$

$$\textsc{Eval}\ \frac{f(x) \Downarrow y}{\langle q_{\text{in}}, \overline{\mathsf{func}}_\omega\ \mathtt{E}x\,f, q_{\text{out}}\rangle \xrightarrow{\mathsf{e}^\omega(f,x)} \langle q_{\text{in}}, \overline{\mathsf{func}}_\omega\ \mathtt{P}y\,f, q_{\text{out}}\rangle}$$

$$\textsc{Put}\ \frac{q'_{\text{out}} = \mathsf{enqueue}(y, q_{\text{out}})}{\langle q_{\text{in}}, \overline{\mathsf{func}}_\omega\ \mathtt{P}y\,f, q_{\text{out}}\rangle \xrightarrow{\mathsf{p}^\omega(y, q_{\text{out}})} \langle q_{\text{in}}, \overline{\mathsf{func}}_\omega\ \mathtt{G}\,f, q'_{\text{out}}\rangle}$$

$$\textsc{Pipe}_1\ \frac{\langle q_{\text{in}}, \bar{s}_1, q\rangle \xrightarrow{\lambda} \langle q'_{\text{in}}, \bar{s}'_1, q'\rangle}{\langle q_{\text{in}}, \overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle q'_{\text{in}}, \overline{\mathsf{pipe}}\ q'\ \bar{s}'_1\ \bar{s}_2, q_{\text{out}}\rangle}$$

$$\textsc{Pipe}_2\ \frac{\langle q, \bar{s}_2, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle q', \bar{s}'_2, q'_{\text{out}}\rangle}{\langle q_{\text{in}}, \overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle q_{\text{in}}, \overline{\mathsf{pipe}}\ q'\ \bar{s}_1\ \bar{s}'_2, q'_{\text{out}}\rangle}$$

$$\textsc{Seq}_1\ \frac{\langle q_{\text{in}}, \bar{s}_1, q\rangle \xrightarrow{\lambda} \langle q'_{\text{in}}, \bar{s}'_1, q'\rangle}{\langle q_{\text{in}}, \overline{\mathsf{seq}}\ q\ \bar{s}_1\ \bar{s}_2, q_2\rangle \xrightarrow{\lambda} \langle q'_{\text{in}}, \overline{\mathsf{seq}}\ q'\ \bar{s}'_1\ \bar{s}_2, q_2\rangle}$$

$$\textsc{Seq}_2\ \frac{\vdash_{\mathsf{idle}} \bar{s}_1 \qquad \langle q, \bar{s}_2, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle q', \bar{s}'_2, q'_{\text{out}}\rangle}{\langle\langle\rangle, \overline{\mathsf{seq}}\ q\ \bar{s}_1\ \bar{s}_2, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle\langle\rangle, \overline{\mathsf{seq}}\ q'\ \bar{s}_1\ \bar{s}'_2, q'_{\text{out}}\rangle}$$

$$\textsc{Farm}_i\ \frac{\langle q_{\text{in}}, \bar{s}_i, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle q'_{\text{in}}, \bar{s}'_i, q'_{\text{out}}\rangle}{\langle q_{\text{in}}, \overline{\mathsf{farm}}\langle \ldots \bar{s}_i \ldots\rangle, q_{\text{out}}\rangle \xrightarrow{\lambda} \langle q'_{\text{in}}, \overline{\mathsf{farm}}\langle \ldots \bar{s}'_i \ldots\rangle, q'_{\text{out}}\rangle}$$

Fig. 4. State Transition Relation.

Termination and Soundness Results

We are now in a position to state the main termination and soundness results.

Theorem 3.3 (*Termination*)
For all $p \in \mathscr{P}$ with a finite number of elements inside atomic function wrappers and queues, there exists no infinite trace $\lambda_1, \lambda_2, \ldots \in \Lambda^*$.

Proof
By induction on the structure of the operational semantics, we can prove that no rule places an element in a previous queue, and that with a finite number of elements it is impossible to ignore an atomic function worker indefinitely. □

In order to prove that the output queue of a skeleton in an idle state contains a valid output with respect to the denotational semantics of the skeletons, we must show that each step of the operational semantics preserves a permutation of the input elements. Since the intermediate and output queues will contain elements that have been operated on, we cannot relate them directly to the original inputs. Our solution is to define a semantic function $\mathscr{O}[\![p]\!]$ that maps the remaining unapplied functions to the elements that are still in the input queue. The idea is to use this definition to reason about the *expected output* of a program in any particular state of the execution.

Definition 3.11 (*Expected Output*)
The function $\mathscr{O}_f[\![\bar{s}]\!]$ denotes the *expected output* of a stateful skeletal subexpression, where f is the function computed by the subsequent subexpressions.

$$
\begin{array}{lcl}
\mathscr{O}_f[\![\_]\!] & : & \overline{\mathrm{S}} \to \mathbb{V}^* \\
\mathscr{O}_f[\![\overline{\mathsf{func}_\omega}\ \mathtt{G}\ g]\!] & = & \langle\rangle \\
\mathscr{O}_f[\![\overline{\mathsf{func}_\omega}\ \mathtt{E}x\ g]\!] & = & \langle (f \circ g)(x)\rangle \\
\mathscr{O}_f[\![\overline{\mathsf{func}_\omega}\ \mathtt{P}x\ g]\!] & = & \langle f(x)\rangle \\
\mathscr{O}_f[\![\overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2]\!] & = & \mathscr{O}_f[\![\bar{s}_2]\!] \oplus \mathsf{map}\ (f \circ f_2)\ q \oplus \mathscr{O}_{f\circ f_2}[\![\bar{s}_1]\!] \\
\mathscr{O}_f[\![\overline{\mathsf{seq}}\ q\ \bar{s}_1\ \bar{s}_2]\!] & = & \mathscr{O}_f[\![\bar{s}_2]\!] \oplus \mathsf{map}\ (f \circ f_2)\ q \oplus \mathscr{O}_{f\circ f_2}[\![\bar{s}_1]\!] \\
 & & \quad \text{where } f_i = \bigcirc fs_i \text{ and } s_i \rightharpoonup_{fs_i} \sigma_i \\
\mathscr{O}_f[\![\overline{\mathsf{farm}}\ \langle \bar{s}_1, \ldots, \bar{s}_n\rangle]\!] & = & \mathscr{O}_f[\![\bar{s}_1]\!] \oplus \ldots \oplus \mathscr{O}_f[\![\bar{s}_n]\!]
\end{array}
$$

We extend this to define the expected output for programs $p \in \mathscr{P}$. Let $\bar{s} \in \overline{\mathrm{S}}$ such that $s \rightharpoonup_{fs} \Sigma$ and $g = \bigcirc fs$, then $\mathscr{O}_f[\![\langle q_1, \bar{s}, q_2\rangle]\!] = \mathsf{map} f\ q_2 \oplus \mathscr{O}_f[\![\bar{s}]\!] \oplus \mathsf{map}\ (f \circ g)\ q_1$. We will write $\mathscr{O}[\![p]\!]$ instead of $\mathscr{O}_{\mathsf{id}}[\![p]\!]$, since $\mathsf{id} \circ f = f \circ \mathsf{id} = f$.

Lemma 3.2
For all $p, p' \in \mathscr{P}$, if $p \xrightarrow{\lambda} p'$ then $\mathscr{O}[\![p']\!] = \mathsf{perm}(\mathscr{O}[\![p]\!])$.

Proof
We prove this lemma by induction on the structure of the step relation. We need only the associativity of $\oplus$, properties of the map function and two properties of the permutations: if $xs = \mathsf{perm}(ys)$ implies that $cs \oplus xs = \mathsf{perm}(cs \oplus ys)$ and $xs \oplus ys = \mathsf{perm}(ys \oplus xs)$. □

Finally, the main soundness result for the operational semantics relates a valid sequence of transitions from some initial state to some final state to the corresponding denotational semantics value.

Theorem 3.4 (*Soundness*)
For all $s \in \mathrm{S}$, $xs, ys \in \mathbb{V}^*$, and traces $\lambda_1, \ldots, \lambda_n$ such that $\langle xs, \mathscr{L}[\![s]\!], \langle\rangle\rangle \xrightarrow{\lambda_1,\ldots,\lambda_n} \langle\langle\rangle, \mathscr{L}[\![s]\!], ys\rangle$, $ys = [\![s]\!](xs)$.

Proof
We can apply Lemma 3.2 to the different steps of the trace to conclude that $\mathscr{O}[\![\langle\langle\rangle, \mathscr{L}[\![s]\!], ys\rangle]\!] = \mathsf{perm}(\mathscr{O}[\![\langle xs, \mathscr{L}[\![s]\!], \langle\rangle\rangle]\!])$. Then unfolding the definition of $\mathscr{O}[\![]\!]$, and proving that if $\vdash_{\mathsf{idle}} \bar{s}$, then $\mathscr{O}[\![\bar{s}]\!] = \langle\rangle$, we prove that $\mathscr{O}[\![\langle\langle\rangle, \mathscr{L}[\![s]\!], ys\rangle]\!] = ys$ and $\mathsf{perm}(\mathscr{O}[\![\langle xs, \mathscr{L}[\![s]\!], \langle\rangle\rangle]\!]) = \mathsf{perm}(\mathsf{map}\ f\ xs) = (\mathsf{perm} \circ \mathsf{map}\ f)(xs)$, where $f = \bigcirc fs$ and $s \rightharpoonup_{fs} \sigma$. We can finish the proof using Theorem 3.1 to conclude that $(\mathsf{perm} \circ \mathsf{map} f)(xs) = [\![s]\!](xs)$. □

4 Representation and Proof in IDRIS

This section describes how to represent algorithmic skeletons, program states and other key entities, including our key proofs and lemmas, in IDRIS. Skeletal expressions are represented using two key datatypes: `Function` (Section 2.4), a list of functions that determines the functional behaviour; and `SkelTy` (Fig. 5a), the structure of the skeletal expressions. The `Skel` datatype (Fig. 5b) then combines values of type `SkelTy` and type `Function` to give a structural representation of a skeleton instance, together with a representation of its functionality, including timing information for Funcs. Finally, the `Par` datatype (Fig. 5c) is a convenience type, defined as a dependent pair of a `Function` and a `SkelTy`, that lifts the functional behaviour from the skeleton instance. We can now, for

```
data SkelTy : Type where
  Func : {default   0 ti:Timing}         -> SkelTy
  Seq  :                SkelTy -> SkelTy -> SkelTy
  Pipe :                SkelTy -> SkelTy -> SkelTy
  Farm : {default (S Z) w:Nat} -> SkelTy -> SkelTy
```

(a) Structure.

```
using ( f1 : Function a b, f2 : Function b c)
 data Skel : SkelTy -> Function a b -> Type where
   func : {ti : Timing} -> (f : a -> b) -> Skel (Func   {ti}) [f]
   seq  :      Skel p1 f1 -> Skel p2 f2 -> Skel (Seq   p1 p2) (f1 ++ f2)
   pipe :      Skel p1 f1 -> Skel p2 f2 -> Skel (Pipe  p1 p2) (f1 ++ f2)
   farm :      {n : Nat}  -> Skel p1 f1 -> Skel (Farm {w=S n} p1) f1
```

(b) Skeletal Expressions.

```
Par : SkelTy -> Type -> Type -> Type
Par p a b = (f : Function a b ** Skel p f)
```

(c) *Par* Datatype.

Fig. 5. IDRIS data types for representing skeletons.

example, describe `imageMerge_1` and `imageMerge_2` as values of type `Par`, whose types capture both the parallel structure of the program and its functionality. `imageMerge_1` simply composes the worker functions `markPixels` and `replacePixels` sequentially, while `imageMerge_1` introduces a parallel task farm with 8 workers.

```
imageMerge_1 : Par (Seq Func Func) (Image,Image) Image
imageMerge_1 = skel [markPixels, replacePixels]

imageMerge_2 : Par (Farm {w=8} (Seq Func Func)) (Image, Image) Image
imageMerge_2 = skel [markPixels, replacePixels]
```

In both cases, we use the auxiliary function `skel`, which will be defined in Section 6, to automatically generate the value corresponding to the given type. As we will see in the rest of this section, different parallel versions of the `imageMerge` will vary only in the type-level structure. Farms can be freely nested inside other skeleton instances, including other task farms, e.g. as shown below.

```
ImFarms : SkelTy
ImFarms = Farm (Seq (Farm Func) Func)

imageMergeFarms : Par ImFarms (Image,Image) Image
imageMergeFarms = skel [markPixels, replacePixels]
```

Here, `ImFarms` abstracts the skeleton structure. We have deliberately used the *default* number of workers (one) for both farms. This improves abstraction, and will allow the number of workers to be automatically instantiated, as described later. These are not the only possible structures. For example, an alternative parallelisation of `imageMerge` is possible using a parallel pipeline (Fig. 2c) to execute `replacePixels` and `markPixels` in parallel within the outer task farm.

```
ImFarmPipe : SkelTy
ImFarmPipe = Farm (Pipe Func Func)

imageMerge_3 : Par ImFarmPipe (Image,Image) Image
imageMerge_3 = skel [markPixels, replacePixels]
```

```
data FSt : Type -> Type -> Type where
  R : FSt a b
  X : a -> FSt a b
  W : b -> FSt a b

using ( f1 : Function a b, f2 : Function b c )
 mutual
  data AnnS :  SkelTy -> Function a b -> Type where
    AFunc : Timing -> (f : a -> b) -> FSt a b
            -> AnnS Func            [f]
    AFarm : (n : Nat) -> Workers (S n) s f1
            -> AnnS (Farm {w=S n} s) f1
    ASeq  : AnnS s1 f1 -> Queue n b -> AnnS s2 f2
            -> AnnS (Seq  s1 s2) (f1 ++ f2)
    APipe : AnnS s1 f1 -> Queue n b -> AnnS s2 f2
            -> AnnS (Pipe s1 s2) (f1 ++ f2)

  data Workers : Nat -> SkelTy -> Function a b -> Type where
    Nil  : Workers Z s f1
    (::) : AnnS s f1 -> Workers w s f1 -> Workers (S w) s f1

 data State : SkelTy -> Function a b -> Type where
   St : Queue n a -> AnnS s f1 -> Queue m b -> State s f1
```

Fig. 6. Program States.

4.1 Representing the Operational Semantics in IDRIS

We have used IDRIS to both yield an implementation of our operational semantics and to automate the associated soundness proofs. Our IDRIS implementation of the operational semantics of Fig. 4 requires the definition of i) queues, ii) program states, iii) queue operations, and iv) the step transition relation from Section 3.

Queues

We have chosen to represent queues as sequences of elements parameterised by the number of elements in the sequence, providing two operations `enqueue` and `dequeue`:

```
enqueue : a -> Queue n a -> Queue (n+1) a
dequeue : Queue (S n) a -> (a, Queue n a)
```

Program States

The `AnnS` datatype is the type of the labelled stateful skeletal expressions $\overline{\mathrm{S}}$ that we defined in Section 3. As we would expect, constructors of `AnnS` closely resemble those of `Skel`, but also including intermediate queues, the state of the `Func` atomic function wrappers and labels. The reason for using a mutually recursive datatype is to define the `AFarm` constructor. Using our own definition instead of e.g. a vector of skeletons simplifies the task of the IDRIS totality checker. The `State` datatype wraps a stateful skeleton inside a *state*, i.e. an input queue with `n` elements and an output queue that contains `m` elements. We define two auxiliary functions, `skelToAnnS` that converts a value of type `Skel` to one of type `AnnS`, and its inverse `annSToSkel`. These functions correspond to the labelling ($\mathscr{L}$) and erasure ($\mathscr{E}$) functions defined in Section 3.

```
skelToAnnS : Skel s f1 -> AnnS s f1
annSToSkel : AnnS s f1 -> Skel s f1
```

Encoding the Operational Semantics Rules

We represent the *step* state transition relation in IDRIS using the `Step` datatype, whose constructors each correspond to one of the rules in Fig. 4.

```
data Step : State k f1 -> State k f1 -> Type where
 E_Write : Step (St q1  (AFunc ti f (W x)) qs2)
                (St q1  (AFunc ti f R    ) (enqueue x qs2))
...
 E_Pipe2 : Step (St q2  s2  q3) (St q2' s2' q3') ->
           Step (St q1  (APipe s1  q2  s2 ) q3)
                (St q1  (APipe s1  q2' s2') q3')
...
 R_Seq2  : Idle s1 -> Step (St q2  s2  q3) (St q2' s2' q3') ->
                      Step (St qZ (ASeq s1 q2  s2 ) q3 )
                           (St qZ (ASeq s1 q2' s2') q3')
...
```

4.2 Properties of the Operational Semantics

The permutation relation $=_{\mathscr{P}}$ is defined in IDRIS as `IsPerm`. This is a datatype indexed by vectors that are equal modulo permutations.

```
data IsPerm : Vect n a -> Vect m a -> Type where
    PNil   :                                IsPerm []       []
    PCons  :                IsPerm xs ys -> IsPerm (x:xs)   (x:ys)

    PSwap  :                                IsPerm (x:y:xs) (y:x:xs)
    PTrans : IsPerm xs ys -> IsPerm ys zs -> IsPerm xs      zs
```

Using this, we can prove the properties that we need for our soundness proof:

```
sym   : IsPerm xs ys -> IsPerm ys xs
refl  : IsPerm xs xs
comm  : IsPerm (xs ++ ys) (ys ++ xs)
const : IsPerm xs ys -> IsPerm (cs ++ xs) (cs ++ ys)
```

We are now ready to prove the main theorem:

```
soundness :
  (s : SkelTy) -> (fs : Function a b)
  -> (sk : Skel s fs) -> (xs : Vect n a) -> (ys : Vect n b)
  -> (trc : Trace (St (toQueue xs) (idleSkel sk) emptyQueue  )
                  (St emptyQueue   (idleSkel sk) (toQueue ys))
  -> IsPerm ys (map (compose fs) xs)
```

The termination proof simply involves showing that for a given program state, the length of all traces is finite. We achieve this using the `Nat` type.

5 Cost Models

In this section, we define a *cost calculus* for Σ that allows us to reason about the run-time behaviour of any $s \in \mathrm{S}$ such that $s \rightharpoonup_{fs} \sigma$, by applying it to this structure $\sigma \in \Sigma$. As we have seen in Section 4, skeletal expressions are represented using a dependent type indexed by its structure σ, which allows us to apply this cost calculus at the type level. We start by defining the *cost model* of the different parallel skeletons. In this paper, we will only describe a simple cost calculus for a parallel skeleton that has the following assumptions:

1. All workers of a skeletal expression can be mapped to different cores.
2. All the tasks have the same granularity (i.e. take the same time to execute).
3. All the input tasks are independent (the only dependencies are the ones imposed by the structure and the trace).

Our queue operations use a simple concurrent lock-based queue implementation, whose costs can be derived formally (Sarkar *et al.*, 2014) from lower (assembly) level semantics. Our technique can easily be adapted for different implementations or cost models, including e.g. lock-free ones. The parameters of our cost models depend on the parallel architecture we are using, $\mathscr{A}$, i.e.

$$\mathscr{C} : \mathscr{A} \to \mathscr{T}$$

For our cost models, we just need the number of cores plus profiling information about the costs of read (t_{read}), write (t_{write}) and exchange (t_{xchg}) operations. Intuitively, the queue *get* operation needs to acquire the lock, read the queue head, and release the lock, while both the acquire and release must write to the lock location. Similarly, the queue *put* operation needs to acquire the lock, and write to the queue head, and release the lock. The lock acquire is also subject to contention and scales linearly. Assuming the parameters as before, with n being the number of threads, our cost model (derived from (Sarkar *et al.*, 2014)) is:

$$\begin{array}{lcl} t_{\mathsf{get}}, t_{\mathsf{put}} & : & \mathbb{N} \to \mathscr{C} \\ t_{\mathsf{get}}(n)(a) & = & n * t_{\mathsf{xchg}}(a) + t_{\mathsf{read}}(a) + 2 * t_{\mathsf{write}}(a) \\ t_{\mathsf{put}}(n)(a) & = & n * t_{\mathsf{xchg}}(a) + 2 * t_{\mathsf{write}}(a) \end{array}$$

In Section 3, we showed that a trace according to the operational semantics is a sequence of actions ($\mathsf{g}^{\omega} q$, $\mathsf{e}^{\omega}(f,x)$ and $\mathsf{p}^{\omega}(x,q)$) that are determined by the transitive closure of the step state transition relation. Since we need to reason about the actions that happen in parallel, we need to map locations to actual cores in an architecture. Given a set of cores C, we map all locations in a trace to the corresponding core, $\Omega \to \mathrm{C}$. This way, whenever we have an action α^{ω}, we will convert it into the corresponding α^{c}, given a $c \in \mathrm{C}$. That alone is not enough to reason about the run-time behaviour, since some interleavings describe unrealistic scenarios. For example, suppose we have two disjoint sets of cores C_1 and C_2 and a pipeline in a state $\langle q_1, \overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2, q_2 \rangle$ such that all $\mathsf{pl}\ldots$ are mapped to cores in C_1, and $\mathsf{pr}\ldots$ to C_2. Let $c1_i \in \mathsf{C}_1$ and $c2_j \in \mathsf{C}_2$. From the operational semantics alone, a trace reaching a final state with the shape:

$$\alpha^{c1_{i1}} \alpha^{c1_{i2}} \ldots, \alpha^{c1_{in}} \alpha^{c2_{j1}} \alpha^{c2_{j2}} \ldots \alpha^{c2_{jm}}$$

is a valid trace, although it does not describe any realistic scenario under our assumptions, since a worker in $\bar{s}_2$ will be able to execute a $\mathsf{g}^{c2_j} q$ in parallel immediately after any action $\mathsf{p}^{c1_i}(x,q)$. In order to avoid such situations, we need to add a validity condition to the traces.

Definition 5.1 (*Valid Trace Condition*)
We define a trace for $\langle q_1, \bar{s}, q_n \rangle \in P$ to be valid, if for all subexpressions $\bar{s'}$ of $\bar{s}$:

1. $\bar{s'} = \overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2$, then after any $\mathtt{p}q$ there is either another $\mathtt{p}q$ or a $\mathtt{g}q$; and
2. $\bar{s'} = \overline{\mathsf{farm}} s\ \bar{s}^n$, then all possible actions α^{c_i} from the workers in the same branch (if any) of a $\overline{\mathsf{seq}}$ subexpression occur with the same frequency in the trace (i.e. no worker is ignored and we step in all of them at the same pace).

It is safe to ask for the second condition only under assumptions 2 and 3 above, that state that all the input tasks are independent and of the same granularity. Using this definition, we can calculate the cost of a valid trace. Provided that we parameterise the queues with the contending workers, and the atomic functions with their profiling information, we know how to calculate the cost of the individual actions from a trace. We now need to define the *dependencies* of an action to calculate the cost of a trace.

Definition 5.2 (*Core Dependency*)
The dependency of a core, $\mathsf{dep}(c, \alpha_1^{c_1} \ldots) = \alpha_1^{c_1} \ldots \alpha_i^{c_i}$, is the minimum trace that precedes all the actions in core c. For the valid traces, it is the trace until an element is placed in the queue that is read by the first action on core c.

We calculate the cost of a trace as the maximum of the cost of the actions per core, plus the cost of the dependencies of each core.

$$\begin{aligned}\mathscr{C}(\alpha_1^{c_i} \ldots \alpha_n^{c_j}) = \mathsf{MAX} \quad & (\mathscr{C}(\mathsf{dep}(c_1, \alpha_1^{c_i} \ldots \alpha_n^{c_j}) + \mathscr{C}(\alpha_{k1}^{c_1}) + \mathscr{C}(\alpha_{k1+1}^{c_1}) + \ldots), \\ & \mathscr{C}(\mathsf{dep}(c_2, \alpha_1^{c_i} \ldots \alpha_n^{c_j}) + \mathscr{C}(\alpha_{k2}^{c_2}) + \mathscr{C}(\alpha_{k2+1}^{c_2}) + \ldots), \ldots)\end{aligned}$$

Although this definition is accurate, it can only be applied to full traces. In the more general situation where we want to estimate the cost of a skeleton with no information about the number of inputs, we instead assume that we know the cost of the different atomic functions for the inputs. We then define the cost of a program in a steady state, where a program is a in steady state when all input, output and intermediate queues in its skeleton have enough elements for all the workers. The cost of a program in a steady state is defined as the cost of producing an output divided by the number of outputs produced.

$$\begin{aligned}
\mathsf{cost}(\langle \langle x \rangle \oplus q_1, \overline{\mathsf{func}}_\omega\ \mathtt{G}\, f, q_2 \rangle) &= \mathscr{C}(\mathsf{g}^{\mu(\omega)} \langle x \rangle \oplus q_1) + \mathscr{C}(\mathsf{e}^{\mu(\omega)}(f, x)) + \mathscr{C}(\mathsf{p}^{\mu(\omega)}(f(x), q_2) \\
\mathsf{cost}(\langle q_1, \overline{\mathsf{pipe}}\ q\ \bar{s}_1\ \bar{s}_2, q_2 \rangle) &= \mathsf{MAX}(\mathscr{C}(\alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}}), \mathscr{C}(\alpha_{21}^{c_{2i}} \ldots \alpha_{2m}^{c_{2j}})), \\
&\quad \text{where } \alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}} \text{ is a trace that produces an output in } q, \\
&\quad \text{and } \alpha_{21}^{c_{2i}} \ldots \alpha_{2m}^{c_{2j}} \text{ is a trace that produces an output in } q_2 \\
\mathsf{cost}(\langle q_1, \overline{\mathsf{seq}}\ q\ \bar{s}_1\ \bar{s}_2, q_2 \rangle) &= (\mathscr{C}(\alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}}) + \mathscr{C}(\alpha_{21}^{c_{2i}} \ldots \alpha_{2m}^{c_{2j}}))/nt, \\
&\quad \text{where } \alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}} \text{ is a trace that produces all } nt \text{ outputs in } q, \\
&\quad \text{and } \alpha_{21}^{c_{2i}} \ldots, \alpha_{2m}^{c_{2j}} \text{ is a trace that produces all } nt \text{ outputs in } q_2, \\
&\quad \text{and } nt \text{ is the number of inputs in } q_1 \\
\mathsf{cost}(\langle q_1, \overline{\mathsf{farm}} s \langle \bar{s}_1 \ldots \bar{s}_w \rangle, q_2 \rangle) &= \mathsf{MAX}(\mathscr{C}(\alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}}), \ldots, \mathscr{C}(\alpha_{w1}^{c_{wi}} \ldots \alpha_{wm}^{c_{wj}}))/w, \\
&\quad \text{where } \alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}} \text{ is a trace for } \bar{s}_1 \text{ that produces an output in } q_2, \\
&\quad \text{and } \alpha_{w1}^{c_{wi}} \ldots \alpha_{wm}^{c_{wj}} \text{ is a trace for } \bar{s}_w \text{ that produces an output in } q_2
\end{aligned}$$

We can simplify this by considering valid traces only. Since a seq skeleton needs its first stage to complete the work before we can apply the rule $\textsc{Seq}_2$, we need to make the whole program complete the work, and then divide by the total number of elements. But given our definition of cost, we can approximate this by the cost of producing an output for each

of the stages.

$$\begin{array}{rcl}\mathsf{cost}(\langle q_1, \overline{\mathsf{seq}}\ q\ \bar{s}_1\ \bar{s}_2, q_2\rangle) & = & \mathscr{C}(\alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}}) + \mathscr{C}(\alpha_{21}^{c_{2i}} \ldots \alpha_{2m}^{c_{2j}}), \\ & & \text{where } \alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}} \text{ is a trace that produces ONE output in } q\ , \\ & & \text{and } \alpha_{21}^{c_{2i}} \ldots, \alpha_{2m}^{c_{2j}} \text{ is a trace that produces ONE output in } q_2\end{array}$$

Note, however, that this is now only an approximation of the cost, since there is no valid interleaving of $\alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}}$ and $\alpha_{21}^{c_{2i}} \ldots, \alpha_{2m}^{c_{2j}}$, because the first queue must be left empty and the first skeleton must be idle before a step is done in the second stage of the composition. The second simplification that we can do is that given that we assume tasks of the same granularity, we can assume that all costs in the MAX expression of the task farm are the same. This assumption allows us to approximate the result of the MAX with the cost of any trace:

$$\begin{array}{rcl}\mathsf{cost}(\langle q_1, \overline{\mathsf{farm}} s \langle \bar{s}_1 \ldots \bar{s}_w \rangle, q_2\rangle) & = & \mathscr{C}(\alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}})/w, \\ \text{where } \alpha_{11}^{c_{1i}} \ldots \alpha_{1n}^{c_{1j}} \text{ is a trace for } \bar{s}_1 \text{ that produces an output in } q_2 & & \end{array}$$

We would like, however, to reason about these costs in terms of the structure. Since we are already using the structure to define the traces for a program in a steady state, and then calculate the cost of these traces, we can skip a step and reason directly about costs using the structure. We define a cost calculus for Σ with this idea, parameterising the structure with profiling information for all the atomic functions:

$$\sigma ::= \mathsf{Func}_t \mid \mathsf{Farm}\ n\ \sigma \mid \mathsf{Pipe}\ \sigma\ \sigma \mid \mathsf{Seq}\ \sigma\ \sigma \qquad \textit{parameterised structure}$$

We can then define the cost in terms of the structure as follows:

$$\begin{array}{lcl}\mathsf{cost}_{c_1,c_2}(\mathsf{Func}_t)(a) & = & t_{\mathsf{get}}(c_1)(a) + t + t_{\mathsf{put}}(c_2)(a) \\ \mathsf{cost}_{c_1,c_2}(\mathsf{Pipe}\ \sigma_1\ \sigma_2)(a) & = & \mathsf{MAX}(\mathsf{cost}_{c_1,ct}(\sigma_1)(a), \mathsf{cost}_{ct,c_2}(\sigma_2)(a)) \\ & & \text{where } ct = \mathsf{contending}(\sigma_1, \sigma_2) \\ \mathsf{cost}_{c_1,c_2}(\mathsf{Seq}\ \sigma_1\ \sigma_2)(a) & = & \mathsf{cost}_{c_1,\mathsf{right}(\sigma_1)}(\sigma_1)(a) + \mathsf{cost}_{\mathsf{left}(\sigma_2),c_2}(\sigma_2)(a) \\ \mathsf{cost}_{c_1,c_2}(\mathsf{Farm}\ n\ \sigma)(a) & = & \mathsf{cost}_{c_1 * n, c_2 * n}(\sigma)(a)/n\end{array}$$

Theorem 5.1
Let steady : $\mathrm{S} \to P$ be a function that produces a program state in steady state from a skeletal expression. Let $s \in \mathrm{S}$, $\sigma \in \Sigma$ and $fs \in (\mathbb{V} \to \mathbb{V})^*$ such that $s \rightharpoonup_{fs} \sigma$ and σ is parameterised by the profiling information of s. Then $\mathsf{cost}(\mathsf{steady}(s)) = \mathsf{cost}(\sigma)$.

Proof
By induction on the structure of s. We see that for the base case, func, in both expressions the result is the time of a *get*, the time of computing the embedded function and the time of a *put* operation. The pipe case can be solved by applying the induction hypotheses. The farm case can be proved using the assumption that all tasks are of the same granularity, and the condition that all the workers produce a step with the same frequency, i.e. all the traces of the MAX expression are of the same length and consist of sequences of the same operations done in different cores. For the seq case, we need to prove that the simplification shown above actually produces the same result than the function cost, in order to be able to apply the induction hypothesis. In the above definition of the cost of the traces of skeletons in steady state, we observe that if *nt* is sufficiently large, then both results are equal. We can define the function steady to specify a sufficiently large number of inputs so that this equality holds. □

6 Rewriting Skeletal Expressions

Any tree of algorithmic skeletons can be rewritten into an alternative parallel structure, provided that it preserves its denotational semantics, i.e. it is possible to use any two $s_1, s_2 \in$ S interchangeably, provided that $[\![s_1]\!] = [\![s_2]\!]$. Theorem 3.2 guarantees that whenever the *fs* are the same, the functional behaviour will be the same (modulo permutations). However, it would be more useful to define an equivalence relation, *structure convertibility*[4], $\equiv_{\mathsf{Cnv}} \in \Sigma \times \Sigma$, that defines families of (weakly) functionally equivalent skeletal expressions.

Definition 6.1 (*Convertibility Relation*)

$$\textsc{Refl}\ \frac{}{\sigma \equiv_{\mathsf{Cnv}} \sigma} \qquad \textsc{Sym}\ \frac{\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2}{\sigma_2 \equiv_{\mathsf{Cnv}} \sigma_1} \qquad \textsc{Trans}\ \frac{\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2 \quad \sigma_2 \equiv_{\mathsf{Cnv}} \sigma_3}{\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_3}$$

$$\textsc{SeqAssoc}\ \frac{}{\mathsf{Seq}(\mathsf{Seq}\ \sigma_1\ \sigma_2)\ \sigma_3 \equiv_{\mathsf{Cnv}} \mathsf{Seq}\ \sigma_1\ (\mathsf{Seq}\ \sigma_2\ \sigma_3)}$$

$$\textsc{PipeIntro}\ \frac{}{\mathsf{Seq}\ \sigma_1\ \sigma_2 \equiv_{\mathsf{Cnv}} \mathsf{Pipe}\ \sigma_1\ \sigma_2} \qquad \textsc{RSeq}\ \frac{\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_3 \quad \sigma_2 \equiv_{\mathsf{Cnv}} \sigma_4}{\mathsf{Seq}\ \sigma_1\ \sigma_2 \equiv_{\mathsf{Cnv}} \mathsf{Seq}\ \sigma_3\ \sigma_4}$$

$$\textsc{FarmIntro}\ \frac{n_1 \in \mathbb{N}}{\sigma \equiv_{\mathsf{Cnv}} \mathsf{Farm}\ n_1\ \sigma}$$

We define a normalisation function $\mathsf{norm} : \Sigma \to \Sigma$ such that for all $\sigma \in \Sigma$, $\mathsf{norm}(\sigma)$ contains no parallel structure and all the occurrences of Seq are right-associative. This function can be defined as the composition of two other functions $\mathsf{norm} = \mathsf{flatten} \circ \mathsf{rmpar}$, where:

$$\begin{array}{lcl}
\mathsf{rmpar} & : & \Sigma \to \Sigma \\
\mathsf{rmpar}(\mathsf{Func}) & = & \mathsf{Func} \\
\mathsf{rmpar}(\mathsf{Seq}\ \sigma_1\ \sigma_2) & = & \mathsf{Seq}\ (\mathsf{rmpar}(\sigma_1))\ (\mathsf{rmpar}(\sigma_2)) \\
\mathsf{rmpar}(\mathsf{Pipe}\ \sigma_1\ \sigma_2) & = & \mathsf{Seq}\ (\mathsf{rmpar}(\sigma_1))\ (\mathsf{rmpar}(\sigma_2)) \\
\mathsf{rmpar}(\mathsf{Farm}\ n\ \sigma) & = & \mathsf{rmpar}(\sigma_1) \\
 & & \\
\mathsf{flatten} & : & \Sigma \to \Sigma \\
\mathsf{flatten}(\mathsf{Seq}\ (\mathsf{Seq}\ \sigma_1\ \sigma_2)\ \sigma_3) & = & \mathsf{flatten}(\mathsf{Seq}\ \sigma_1\ (\mathsf{Seq}\ \sigma_2\ \sigma_3)) \\
\mathsf{flatten}(\mathsf{Seq}\ \sigma_1\ \sigma_2) & = & \mathsf{Seq}\ \sigma_1\ (\mathsf{flatten}(\sigma_2)) \\
\mathsf{flatten}(\sigma_1) & = & \sigma_1
\end{array}$$

Lemma 6.1
For all $\sigma \in \Sigma$, $\sigma \equiv_{\mathsf{Cnv}} \mathsf{norm}(\sigma)$.

Proof
We prove that for all $\sigma \in \Sigma$, $\sigma \equiv_{\mathsf{Cnv}} \mathsf{rmpar}(\sigma)$, by applying rules Sym, FarmIntro and PipeIntro to remove the pipelines and farms. Then, we prove that $\sigma \equiv_{\mathsf{Cnv}} \mathsf{flatten}(\sigma)$ by induction on the number of nested Seq nodes in the leftmost branch. We conclude that $\sigma \equiv_{\mathsf{Cnv}} \mathsf{flatten}(\mathsf{rmpar}(\sigma))$ by using the transitivity rule Trans. □

[4] This is derived from a set of rules that are well-known in the parallel programming literature as "structural equivalences" (Aldinucci *et al.*, 1998; Aldinucci, 2002; Brown *et al.*, 2013). We prefer the term "convertibility" here, since the relation determines whether we can *convert* a skeletal expression that has one structure into a functionally equivalent one that has a different structure.

Lemma 6.2
For all $\sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2 \Longleftrightarrow \mathsf{norm}(\sigma_1) = \mathsf{norm}(\sigma_2)$.

Proof

Case $\Rightarrow$. By induction on the $\equiv_{\mathsf{Cnv}}$ relation, we can apply the rules of the $\equiv_{\mathsf{Cnv}}$ relation to σ_1 and σ_2 to change their structure towards a σ such that $\sigma = \mathsf{norm}(\sigma)$. Then, it becomes obvious that this $\sigma = \mathsf{norm}(\sigma) = \mathsf{norm}(\sigma_1) = \mathsf{norm}(\sigma_2)$.
Case $\Leftarrow$. Since by Lemma 6.1, $\sigma \equiv_{\mathsf{Cnv}} \mathsf{norm}(\sigma)$, and if $\sigma_1 = \sigma_2$ then $\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2$ by rule REFL, using the transitivity and symmetry of $\equiv_{\mathsf{Cnv}}$ we can prove that $\sigma_1 \equiv_{\mathsf{Cnv}} \mathsf{norm}(\sigma_1) \equiv_{\mathsf{Cnv}} \mathsf{norm}(\sigma_2) \equiv_{\mathsf{Cnv}} \sigma_2$. □

Theorem 6.1
For all $s_1 \in \mathrm{S}$ and $\sigma_1, \sigma_2 \in \Sigma$ such that $s_1 \rightharpoonup_{fs} \sigma_1$, $\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2 \Longleftrightarrow \exists s_2 \in \mathrm{S}$ such that $s_2 \rightharpoonup_{fs} \sigma_2$.

Proof

$\Leftarrow$. In this case, we have to prove that whenever we have a $s_2 \rightharpoonup_{fs} \sigma_2$, then $\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2$. By normalising σ_1 and σ_2, we can see that if both s_i, σ_i are related under the same $\rightharpoonup_{fs}$, then $\mathsf{norm}(\sigma_1) = \mathsf{norm}(\sigma_2)$. It is easy to prove this by induction on the structure of the $\rightharpoonup_{fs}$ relation. Then, since $\mathsf{norm}(\sigma_1) = \mathsf{norm}(\sigma_2)$, by using Lemma 6.2, we conclude $\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2$.
$\Rightarrow$. By induction on the structure of the $\equiv_{\mathsf{Cnv}}$ relation, we show how to rewrite the s_1 into an equivalent s_2 with the same functionality but with structure σ_2. That is, we first prove that if $\sigma_1 \equiv_{\mathsf{Cnv}} \sigma_2$, then there exists a *rewriting* from a s_1, such that $s_1 \rightharpoonup_{fs} \sigma_1$, to a s_2, such that $s_2 \rightharpoonup_{fs} \sigma_2$. This rewriting is obtained by lifting the $\equiv_{\mathsf{Cnv}}$ relation to $\mathrm{S} \times \mathrm{S}$. Then, proving that a s_2 exists is trivially achieved by applying this rewriting. □

6.1 Equivalence and Convertibility in IDRIS

The IDRIS datatype `Cnv` captures the convertibility of structures, so providing a machine-checkable proof of the $\equiv_{\mathsf{Cnv}}$ property:

```
data Cnv : SkelTy -> SkelTy -> Type where
   ...
```

We define a normalisation function as described above. In our implementation, `norm` will return a normalised skeleton and a proof of convertibility:

```
norm    : (s1 : SkelTy) -> (s2 : SkelTy ** Cnv s1 s2)
normStr : SkelTy -> SkelTy
```

Using Lemma 6.2, we can define the equivalence of two structures as the equality of their normalised structures. The function `equivCnv` is a proof of the $\Rightarrow$ case of Lemma 6.2. This is convenient when automating proofs of equivalence of any two structures, as we will observe in the definition of the `cast` function.

```
Equiv : SkelTy -> SkelTy -> Type
Equiv s1 s2 = normStr s1 = normStr s2

equivCnv : {s1 : SkelTy} -> {s2 : SkelTy} -> Equiv s1 s2 -> Cnv s1 s2
```

The function `rwSkel` lifts proofs of functional equivalence to rewritings, and it is used in the proof of Theorem 6.1. The function `rwSkelR` returns a rewriting that interprets the $\equiv_{\mathsf{Cnv}}$ relation from left to right, and a `rwSkelL` from right to left.

```
rwSkel : Cnv s1 s2 -> Skel s1 fs -> Skel s2 fs
rwSkel = ...
```

Using these definitions, we can implement a function that automatically rewrites a skeletal expression into a functionally equivalent one. The type-checker ensures that the result is indeed a weakly functionally equivalent expression. We provide this rewriting as a type cast. The `auto` keyword states that whenever there is no value of type `Equiv s1 s2` in the context, the type checker will automatically use the decidable equality relations for skeletons to prove it.

```
cast : {fs : Function a b}
       -> {s1 : SkelTy} -> {s2 : SkelTy}
       -> {auto prf : Equiv s1 s2}
       -> Skel s1 fs -> Skel s2 fs
cast {prf} s = rwSkel (equivCnv prf) s
```

Using a function `nfStruct` that returns the normal form structure of the family of functionally equivalent structures, a predicate on lists of functions `NotEmpty`, and a function `nfSkel` defined in a similar way to `nfStruct` but for values of type `Skel`, we can define the function `skel`:

```
skel : {s1 : SkelTy} -> (fs : Function a b)
       -> {auto pr1 : NotEmpty fs}
       -> {auto pr2 : Equiv (nfStruct fs) s1}
       -> Par s1 a b
```

6.2 Automatic Rewriting of Skeleton Instances

We can use the cost models from Section 5 to generate a skeleton instance that minimises the execution cost, given a suitable model of the target architecture. The algorithm for finding the best structure involves three steps:

1. Generate all possible parallel structures for a skeleton, plus relevant convertibility proofs up to certain depth;
2. Determine the best instantiation of each skeleton;
3. Choose the skeleton instantiation with the least cost.

Step 1: Generate Parallel Structures. We begin by normalising the source skeleton. We then generate all possible combinations of the skeleton that can be obtained by applying associativity and pipe- and farm-introduction rules, but without introducing directly nested farms. This condition ensures termination.

```
mkSkels : (s1 : SkelTy) -> List (s2 : SkelTy ** Cnv s1 s2)
```

Step 2: Determine Best Instantiation. For each possible rewriting, we determine the best possible instantiation by calculating the minimum number of worker threads that are necessary for the skeleton. We use a recursive tiling algorithm, starting at the outermost farm instances, and recursively instantiating each sub-skeleton. In each case, we record only the instantiation that minimises the cost using the lowest number of cores.

```
bestInst : Arch -> (s1 : SkelTy) -> (Timing, Vect (countFarms s1) Nat)
```

Step 3: Select Best Skeleton. The third and final step simply selects the skeleton instance with the least cost, as determined in step 2.

```
bestSkelTy : Arch -> (s1 : SkelTy) -> (s2 : SkelTy ** Cnv s1 s2)
bestOf : Arch -> SkelTy -> SkelTy
```

Using the type-level structure to reason about cost and functional equivalence is very powerful since it allows type-level specifications such as:

```
Auto : Arch -> Type -> Type
Auto arch a b =
    (fs : Function a b ** Skel (bestOf arch (seqStruct fs)) fs)

bestSkel : (fs : Function a b) -> Auto arch a b
bestSkel fs = (fs ** cast (seqSkel fs))

skelProg : Auto titanic Img Img
skelProg = bestSkel [markPixels, replacePixels]
```

7 Speedup

This section shows examples of structured programs in IDRIS, and gives the results of using our techniques on three different benchmarks. We first describe the parallel structure of each benchmark, and show how this is captured in IDRIS. We then show how our types predict the optimal choice of skeleton for our examples given the cost models above, and compare the predictions given by our cost models against actual parallel execution times on two different real multicores: *titanic*, a 800MHz 24-core, AMD Opteron 6176, running Centos Linux 2.6.18-274.e15; using gcc 4.4.6; and *lovelace*, a 1.4GHz 64-core, AMD Opteron 6376, running GNU/Linux 2.6.32-279.22.1.e16. All speedups shown here were calculated as the mean of ten executions. Note that the output of the IDRIS interpreter has been edited slightly to improve readability.

7.1 Matrix Multiplication

We implement matrix multiplication using a task farm, whose worker function, `multRow`, receives a `Matrix` and a `Row` of another matrix, and multiplies the row by the matrix to produce one row of the output. The farm iterates over the rows of the second matrix as its input stream.

```
tmult   : Timing
multRow : Matrix -> Row -> Row
MM : SkelTy
MM = autoInst titanic (Farm (Func {ti=tmult}))

matmult : Matrix -> Par MM Row Row
matmult m = skel [mm]
          where mm : Row -> Row
                mm = multRow m
```

The output of the function `skel` is, as expected, a task farm with the atomic function `mm` as worker.

```
*Results> matmult m
MkSigma [mm] (farm (func mm)) : (f ** Skel (Farm Func) f)
```

The function `autoInst` automatically instantiates the farm using the result of `bestInst` (Sec. 6). IDRIS can then automatically determine the best type, or we can manually apply the cost models to our types. Based on the output given by the cost models, we can decide what the type annotation of `matmult` should be.

```
*Results> cost (Farm {w=20} (Func {ti=tmult})) titanic
1.9597235e-3 : Float

*Results> bestInst titanic (Farm (Func {ti=tmult}))
```

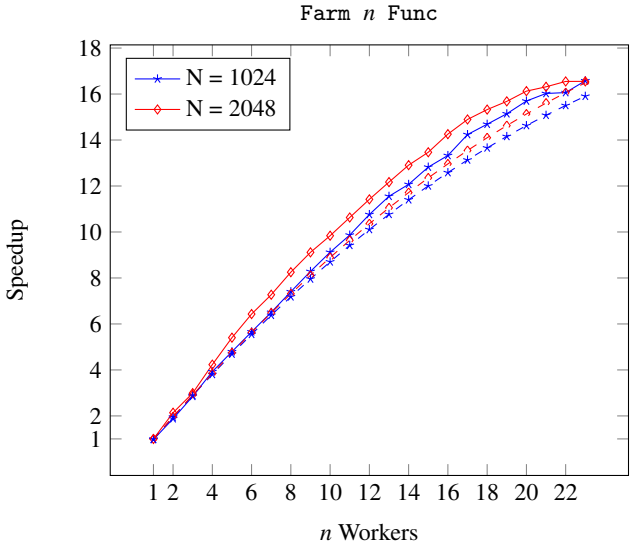


Fig. 7. Speedup (solid lines) vs prediction (dashed lines). Matrix Multiplication of matrices of sizes N×N (*titanic*).

```
(1.8080821739130434e-3, [23]) : (Float, Vect 1 Nat)

*Results> convert (Farm {w=23} (Func {ti=tmult})) matmult
([multRow] ** farm (func multRow)) : (f ** Skel (Farm Func) f)
```

Figure 7 compares the speedups predicted by our cost models with the actual speedups for a farm with a varying number of workers for two different sizes of matrix ($N = 1024$ and $N = 2048$). Predictions are shown as dashed lines, and the actual speedup is shown by the solid lines. It is clear that the cost models give good predictions of the actual speedups for all cases, and do not over-predict the speedup. Moreover, our type system determines that for this example we need to instantiate the `Farm` with 23 workers. We obtain speedups of up to 15.90 on 24 cores.

7.2 Image Merge

Our second example processes a stream of pairs of images and returns a stream of merged images, using `replacePixels . markPixels`. The parallel structure, `IM`, is a farm of a 2-stage pipeline.

```
tmark, treplace : Timing
markPixels    : (Img, Img) -> (Img, Img)
replacePixels : (Img, Img) -> Img
IM : SkelTy
IM = autoInst titanic (Farm (Pipe (Func {ti=tmark})
                                  (Func {ti=treplace})))

imageMerge : Par IM (Img,Img) Img
imageMerge = skel [markPixels, replacePixels]

*Results> imageMerge
MkSigma
  [markPixels, replacePixels]
```

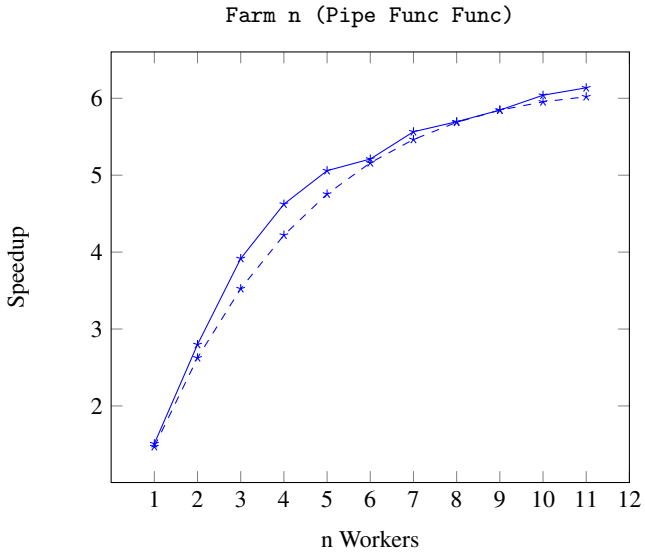


Fig. 8. Speedup (solid lines) vs prediction (dashed lines). Image Merge, 500 input tasks (*titanic*).

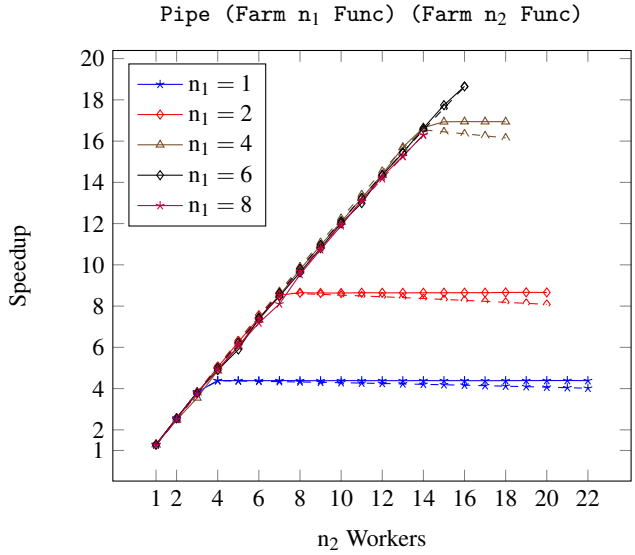


Fig. 9. Speedup (solid lines) vs prediction (dashed lines). Image Convolution, 500 input tasks (*titanic*).

```
(farm
  (pipe
    (func markPixels)
    (func replacePixels))) : (f ** Skel (Farm (Pipe Func Func)) f)
```

The corresponding predicted and actual speedups are shown in Figure 8. Although the absolute speedup is not as good as for the matrix multiplication (6.14 on 23 cores), once again our models are a good lower-bound predictor of the actual speedup. We can again determine the best instantiation for the farm workers:

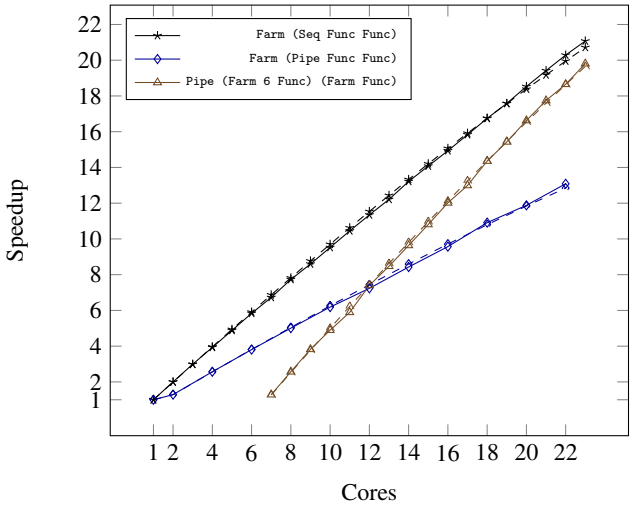


Fig. 10. Speedup (solid lines) vs predicted (dashed lines). Different Parallel Structures for Image Convolution, 500 Images 1024 * 1024: *titanic*

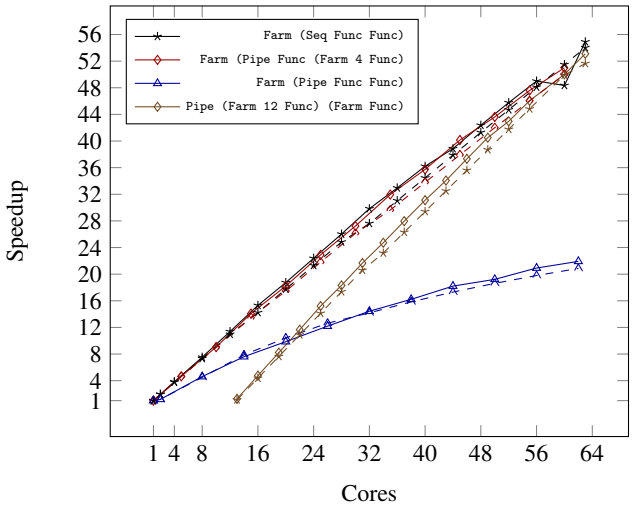


Fig. 11. Speedup (solid lines) vs predicted (dashed lines). Different Parallel Structures for Image Convolution, 500 Images 1024 * 1024: *lovelace*.

```
*Results> bestInst titanic IM
(1.1532345454545454e-3, [11]) : (Float, Vect 1 Nat)
```

7.3 Image convolution

Image convolution is widely used in image processing applications. We express a convolution algorithm as the composition of two functions, `read` and `process`. The `read` function receives the data corresponding to an image, parses it, and prepares it for the next step, `process`, that returns the actual convolution of the image. In this case, the best structure is not a pipeline. We provide two alternative structures: a) the best instantiation for the parameters of a two-stage pipeline, where both stages are task farms (`imageConv1`);

and b) the best structure with respect to the cost models for the `titanic` and `lovelace` architectures (`imageConv2` and `imageConv3`).

```
tread, tproc : Arch -> Timing
IC, BIC : SkelTy
IC  = Pipe
        (Farm (Func {ti=tread titanic}))
        (Farm (Func {ti=tproc titanic}))
BIC = bestInst titanic IC

imageConv1 : Par BIC Data Img
imageConv1 = skel [readI, processI]

imageConv2 : Auto titanic Data Img
imageConv2 = bestSkel [readI, processI] [tread, tproc]

imageConv3 : Auto lovelace Data Img
imageConv3 = bestSkel [readI, processI] [tread, tproc]
```

Figure 9 compares predicted *vs* actual speedups for `IC` with specific numbers of workers in the first stage of the pipeline against a varying number of workers in the second stage on *titanic*. Once again, the cost models are an excellent predictor of the actual speedup, providing tight lower bounds on speedup. In the best case, we achieve a speedup of 19.81 on 23 cores. Figures 10 and 11 compare predicted *vs* actual speedups of the best predicted structures on `titanic` and `lovelace` respectively. The actual speedup closely corresponds to that predicted in most cases, though the prediction is not always a lower bound in this case. In the best case, we achieve a speedup of 44.29 on 63 cores for *lovelace* and 21.08 on 23 cores for *titanic*. As with the previous examples, we can use the cost models directly to obtain cost estimations for our problem and to compare different parallel structures and instantiations. In this case, we can create a function that returns a list of structures sorted by their predicted cost.

```
*Results> compareStructs titanic IM

[(1.5157752173913045e-2, (Farm (Seq Func Func) ** [23])),
 (1.6327986470588234e-2, (Pipe (Farm Func) (Farm Func) ** [6, 17])),
 (1.794705333333333e-2,  (Farm (Pipe Func (Farm Func)) ** [5, 3])),
 (2.425585272727273e-2,  (Farm (Pipe Func Func) ** [11])),
 (7.862535999999999e-2,  (Pipe Func (Farm Func) ** [4])),
 (0.26282868000000004,   (Pipe Func Func ** [])),
 (0.2665590982608696,    (Seq  (Farm Func) Func ** [23])),
 (0.26694033913043486,   (Seq (Farm Func) (Farm Func) ** [23, 23])),
 (0.33985976000000007,   (Seq Func Func ** [])),
 (0.34862830000000006,   (Seq Func (Farm Func) ** [23]))]
      : List (Float, (s ** Vect (countFarms s) Nat))
```

All these structures are convertible, under the convertibility relation (Section 6).

Consider the structures `Farm (Pipe Func (Farm Func))` and `Pipe (Farm Func) (Farm Func)`. IDRIS can determine the convertibility proof that can be used internally to rewrite from one to the other.

```
*Results> cnv (Farm (Pipe Func (Farm Func)))
              (Pipe (Farm Func) (Farm Func))
Trans
  (Trans (Rewrite (Sym FarmIE))
         (Trans (Rewrite (Sym PipeIE))
                (RwComp SERefl (Rewrite (Sym FarmIE)))))
  (Trans (Trans (RwComp SERefl (Rewrite FarmIE))
                (RwComp (Rewrite FarmIE) SERefl))
         (Rewrite PipeIE)) : Cnv (Farm (Pipe Func (Farm Func)))
                                 (Pipe (Farm Func) (Farm Func))
```

Convertibility between both structures can be obtained since the normalised form of both structures is the same. The first part of the proof removes farms and pipelines in order to rewrite the structure into the normalised skeleton. The second part of the proof applies symmetry to the proof of convertibility between the target structure and the normalised skeleton, so that the pipeline and the farms in both stages are introduced.

8 Related Work

8.1 Skeleton Semantics

There are a few formal descriptions of skeleton semantics. For example, Aldinucci and Danelutto propose an operational semantics schema that can be used to describe both functional and parallel behaviour of skeletal programs in a uniform way (Aldinucci & Danelutto, 2004; Aldinucci & Danelutto, 2007). Their semantics enables several interesting analyses of *Lithium* (Danelutto & Teti, 2002) programs, including comparing the performance and resource usage of functionally equivalent programs and determining the maximum parallelism achievable with infinite or finite resources. Falcou and Sérot show how generative and metaprogramming techniques can be applied to the implementation of a skeleton-based parallel programming library, *Quaff* (Falcou & Sérot, 2008). This implementation is derived directly from a set of explicit production rules, in a semantics-oriented style, and is therefore formally sound. None of the previous work attempts to integrate that formal reasoning about skeleton semantics in the program as part of the type system, however, as we have done. We believe that this forms a very powerful tool, since it allows the integration of any semantics-derived analysis. Moreover, previous work on semantics does not consider the low level components that are used to communicate and synchronise the skeleton, as we have done here.

8.2 Cost Models

There has been extensive work into performance prediction of parallel programs. The PRAM model (Fortune & Wyllie, 1978) is widely used as a standard theoretical model of a parallel machine, and it can be used to derive a standard measure of the parallel complexity of a program. Several other models of computations provide reasonably accurate cost models, such as the bulk-synchronous parallel model (Valiant, 1990) and for the LogP model (Culler *et al.*, 1993). Gustavsson *et al.* (2012) describe an algorithm for static timing analysis of parallel software implemented using shared-memory concurrency, using abstract interpretation.

Lobachev and Loogen (2010; 2013) describe the cost of parallel skeletons in terms of the sequential parts and the parallel penalty. They use many methods to obtain this parallel penalty, such as profiling and machine learning techniques. Benoit *et al.* (2004) evaluate the performance of skeleton-based parallel programs by modelling the parallel skeletons in PEPA (Performance Evaluation Process Algebra). Hayashi and Cole (2002) describe the first completely static system that takes into account both computation and communication for predicting the run time of skeletal parallel programs. Much of the previous work on performance prediction provides reasonable accuracy in different contexts. Although the work we describe in this paper provides comparable accuracy for some problems, the main difference lies not in the predictive power. Our cost models are formally derived from the operational semantics of the parallel programs and can be used as type-level annotations integrated with a rewriting system, thus providing more reasoning power (e.g. allowing to specify constraints on the cost of a parallel program, or using the cost models to drive the rewriting system).

8.3 Types and Parallelism

Types have been used to determine some other parallelism properties. For example, Peña and Segura use sized types to reason about the termination and productivity of Eden skeletons. (Peña & Segura, 2005) (Peña & Segura, 2001). However, they do not provide cost models or provide type-level mechanisms to reason about the equivalence of skeletal expressions as we have done. Types have also been used to ensure deterministic evaluation of concurrent/parallel programs. For example, Kawaguchi et al (Kawaguchi *et al.*, 2012) use a type-and-effect system based on refinement types to guarantee that a fine-grained, low-level, shared-memory multithreading program is deterministic and *LVars* support deterministic-by-construction parallel programming (Kuper & Newton, 2013). Bocchino et al (Bocchino Jr *et al.*, 2009; Bocchino Jr *et al.*, 2011) present a language with a type and effect system that supports nondeterministic computations with a deterministic by default guarantee. Finally, Dodds et al. (Dodds *et al.*, 2011) use concurrent abstract predicates, based on separation logic, for the formal specification and verification of barrier constructs that are used to introduce deterministic parallelism.

Finally, dependent types have previously been used in some more limited parallel settings. For example, Lippmeier et al (Lippmeier *et al.*, 2012) describe an extensible parallel Repa-style array fusion using indexed types to specify the internal representation of each array, and Thiemann and Chakravarty (Thiemann & Chakravarty, 2013) use fully-fledged dependent types to create an Agda frontend to Accelerate, a Haskell-embedded language for data parallel programming aimed at GPUs. This front-end takes advantage of dependent types to ensure certain static guarantees. However, their type system does not allow reasoning about the equivalence or cost of parallel programs, and they only consider data-parallelism, where our approach is fully general.

8.4 Session Types

Session types describe the communication structure of a concurrent system passing messages across communication channels. They support static guarantees that a message-passing program follows a particular communication protocol. Recently, the relationship between session types and linear logic has been shown (Takeuchi *et al.*, 1994; Caires & Pfenning, 2010; Wadler, 2012). Behavioral types based on process algebras have been introduced with the aim of characterising the interface of a process not just as a specification of the static type of exchanged messages, but also as specification of its

dynamic behaviour. Caires and Seco introduce the concept of Behavioral Separation as a general principle by dealing with interference in higher-order imperative concurrent programs and present a type-based approach that develops the concept in a concurrent ML-like language (Caires & Seco, 2013). While session types do not deal directly with parallelism, but with concurrency and communication, the properties that they capture can be used for reasoning about parallel computing (Ng *et al.*, 2014). However, it is not clear how to use session types for performance prediction, since details about the computation structure are needed. We aim to provide the corresponding type-level description of the computation structure of a program.

9 Conclusions

This paper has described a new dependently-typed approach to understanding and reasoning about structured parallel programs, based on strong semantic models of parallelism. The type-level description that we have used allows a clear separation of the functionality of the program from its parallel structure: in particular, the parallel structure (and so performance) of a program can be altered simply by changing its type, with the type system ensuring and verifying that its functionality is unaltered. Moreover, by embedding a formal cost model into the type system, we are able to reason about the performance and potential speedup of parallel programs *at the type level*. This allows type-level decisions to be taken about possible parallelisations, including automatically determining both the best parallel structure, and the best values for key performance-affecting parameters. This could be exploited, for example, as part of an automatic rewriting system to transform sequential programs into *provably optimal* parallelisations with respect to the specified operational semantics.

9.1 Further work

In order to achieve the result described in this paper, we have made a number of assumptions and simplifications. The first is in restricting to a very simple, but still usable, set of basic skeletons. Danelutto et al. (Danelutto & Torquati, 2013) have shown that many other skeletons can, in fact, be transformed into combinations of pipelines and farms. However, this may affect timing properties, and some important parallel structures (e.g. ones involving feedback) cannot be captured in this way. In future, we intend to study these more complex skeleton forms, with the aim of extending the applicability of the work reported here.

Secondly, while providing a contribution that we believe is valuable in its own right, the operational semantics that we have given here is primarily intended to describe a sample, formally verifiable, parallel implementation that allows us to reason about the correctness of the underlying program functionality. There are many ways in which it could be improved. In particular, for simplicity, we have used an unbounded queue structure. Restricting to a bounded queue would be an interesting future extension, but would require us to deal with some complex issues, including *back-pressure* on streams[5].

Thirdly, since we are interested mainly in improved speedup, we have only considered upper bounds on costs. This is potentially useful also for a real-time (parallel) setting, where we need to ensure that deadlines are met. However, small changes to the operational semantics and to the corresponding types should also allow us to determine lower-bound

[5] This is especially problematic when combined with skeletons that include feedback.

costs. This would allow us not only to reason about the best speedup, but also to provide guarantees about the minimum speedup that will be achieved by any parallelisation, giving limits on best and worst speedup for any given choice.

Finally, the system of skeletons that we have used appears to have interesting algebraic properties. In this paper, we have captured these rules in the form of a simple convertibility equivalence between alternative parallel forms, but there may be a deeper and more interesting algebraic formulation that gives greater reasoning power.

References

Aldinucci, Marco. (2002). Automatic program transformation: The Meta tool for skeleton-based languages. *Constructive methods for parallel programming, advances in computation: Theory and practice*, 59–78.

Aldinucci, Marco, & Danelutto, Marco. (2004). An operational semantics for skeletons. *Advances in parallel computing*, **13**, 63–70.

Aldinucci, Marco, & Danelutto, Marco. (2007). Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer languages, systems & structures*, **33**(3), 179–192.

Aldinucci, Marco, Coppola, Massimo, & Danelutto, Marco. (1998). Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. *Proc. of international workshop on constructive methods for parallel programming. Technical report MIP-9805. University of Passau. Passau*.

Benoit, Anne, Cole, Murray, Gilmore, Stephen, & Hillston, Jane. (2004). Evaluating the performance of skeleton-based high level parallel programs. *Pages 289–296 of: Computational science-ICCS 2004*. Springer.

Bocchino Jr, Robert L, Adve, Vikram S, Dig, Danny, Adve, Sarita V, Heumann, Stephen, Komuravelli, Rakesh, Overbey, Jeffrey, Simmons, Patrick, Sung, Hyojin, & Vakilian, Mohsen. (2009). A type and effect system for deterministic parallel Java. *Pages 97–116 of: Proc. of the 24th conf. on object oriented programming systems languages and applications.* ACM.

Bocchino Jr, Robert L, Heumann, Stephen, Honarmand, Nima, Adve, Sarita V, Adve, Vikram S, Welc, Adam, & Shpeisman, Tatiana. (2011). Safe nondeterminism in a deterministic-by-default parallel language. *Pages 535–548 of: Proc. of the 38th symp. on principles of programming languages*. ACM.

Brady, Edwin. (2013a). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, **23**(9), 552–593.

Brady, Edwin. (2013b). Programming and reasoning with algebraic effects and dependent types. *Pages 133–144 of: Proc. of the 18th international conference on functional programming*. ACM.

Brady, Edwin, & Hammond, Kevin. (2012). Resource-safe systems programming with embedded domain specific languages. *Pages 242–257 of: Proc. of practical aspects of declarative languages*. Springer Berlin Heidelberg.

Brown, Christopher, Danelutto, Marco, Hammond, Kevin, Kilpatrick, Peter, & Elliott, Archibald. (2013). Cost-directed refactoring for parallel Erlang programs. *International journal of parallel programming*, **42**(4), 1–19.

Caires, Luís, & Pfenning, Frank. (2010). Session types as intuitionistic linear propositions. *Pages 222–236 of: Concur 2010-concurrency theory*. Springer.

Caires, Luís, & Seco, Joao C. (2013). The type discipline of behavioral separation. *Pages 275–286 of: Proc. of the 40th symp. on principles of programming languages*, vol. 48. ACM.

Caromel, Denis, & Leyton, Mario. (2007). Fine tuning algorithmic skeletons. *Pages 72–81 of: Euro-par 2007*. Springer.

Cole, Murray I. (1989). *Algorithmic skeletons: structured management of parallel computation*. Pitman London.

Culler, David, Karp, Richard, Patterson, David, Sahay, Abhijit, Schauser, Klaus Erik, Santos, Eunice, Subramonian, Ramesh, & Von Eicken, Thorsten. (1993). *Logp: Towards a realistic model of parallel computation*. Vol. 28. ACM.

Danelutto, M., & Torquati, M. (2013). A RISC building block set for structured parallel programming. *Pages 46–50 of: 21st euromicro international conference on parallel, distributed, and network-based processing (PDP '13)*.

Danelutto, Marco, & Teti, Paolo. (2002). Lithium: A structured parallel programming environment in Java. *Pages 844–853 of: Computational science—ICCS 2002*. Springer.

Dodds, Mike, Jagannathan, Suresh, & Parkinson, Matthew J. (2011). Modular reasoning for deterministic parallelism. *Pages 259–270 of: Proc. popl'11*. ACM.

Falcou, Joel, & Sérot, Jocelyn. (2008). Formal semantics applied to the implementation of a skeleton-based parallel programming library. *Pages 243–252 of: Proc. of parallel computing: Architectures, algorithms and applications*, vol. 38.

Fortune, Steven, & Wyllie, James. (1978). Parallelism in random access machines. *Pages 114–118 of: Proc. of the 10th symp. on theory of computing*. STOC '78. New York, NY, USA: ACM.

González Vélez, Horacio, & Leyton, Mario. (2010). A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and experience*, **40**(12), 1135–1160.

Gustavsson, Andreas, Gustafsson, Jan, & Lisper, Björn. (2012). Toward static timing analysis of parallel software. *Oasics-openaccess series in informatics*, vol. 23. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Hayashi, Yasushi, & Cole, Murray. (2002). Static performance prediction of skeletal parallel programs. *Parallel algorithms and applications*, **17**(1), 59–84.

Kawaguchi, Ming, Rondon, Patrick, Bakst, Alexander, & Jhala, Ranjit. (2012). Deterministic parallelism via Liquid Effects. *Pages 45–54 of: Proc. of the 33rd conf. on programming language design and implementation.* ACM.

Kuper, Lindsey, & Newton, Ryan R. (2013). LVars: lattice-based data structures for deterministic parallelism. *Pages 71–84 of: Proc. of the 2nd workshop on functional high-performance computing*. ACM.

Lippmeier, Ben, Chakravarty, Manuel, Keller, Gabriele, & Peyton Jones, Simon. (2012). Guiding parallel array fusion with indexed types. *Pages 25–36 of: Proc. of the 2012 haskell symposium*. ACM.

Lobachev, Oleg, & Loogen, Rita. (2010). Estimating parallel performance, a skeleton-based approach. *Pages 25–34 of: Proceedings of the fourth international workshop on high-level parallel programming and applications*. ACM.

Lobachev, Oleg, Guthe, Michael, & Loogen, Rita. (2013). Estimating parallel performance. *Journal of parallel and distributed computing*, **73**(6), 876–887.

Ng, Nicholas, Yoshida, Nobuko, & Luk, Wayne. (2014). Scalable session programming for heterogeneous high-performance systems. *Pages 82–98 of: Software engineering and formal methods*. Springer.

Peña, Ricardo, & Segura, Clara. (2001). Sized types for typing Eden skeletons. *Pages 1–17 of: Proc. of implementation of functional languages*. Springer.

Peña, Ricardo, & Segura, Clara Marıa. (2005). Reasoning about skeletons in Eden. *Parallel computing: Current & future issues of high-end computing*.

Pelagatti, Susanna. (1998). *Structured development of parallel programs*. Vol. 102. Taylor & Francis Abington.

Sarkar, Susmit, Hammond, Kevin, & Brown, Chris. (2014). *Timing properties and correctness for structured parallel programs on x86-64 multicores*. Under submission.

Takeuchi, Kaku, Honda, Kohei, & Kubo, Makoto. (1994). An interaction-based language and its typing system. *Pages 398–413 of: Parallel architectures and languages europe*. Springer.

Thiemann, Peter, & Chakravarty, Manuel MT. (2013). Agda meets Accelerate. *Pages 174–189 of: Implementation and application of functional languages*. Springer.

Valiant, Leslie G. (1990). A bridging model for parallel computation. *Communications of the acm*, **33**(8), 103–111.

Wadler, Philip. (2012). Propositions as sessions. *Pages 273–286 of: Proc. of the 17th international conference on functional programming*, vol. 47. ACM.