Compiling First-Order Functions to Session-Typed Parallel Code

David Castro-Perez
Imperial College London
London, UK
d.castro-perez@imperial.ac.uk

Nobuko Yoshida
Imperial College London
London, UK
n.yoshida@imperial.ac.uk

Abstract

Building correct and efficient message-passing parallel programs still poses many challenges. The incorrect use of message-passing constructs can introduce deadlocks, and a bad task decomposition will not achieve good speedups. Current approaches focus either on correctness or efficiency, but limited work has been done on ensuring both. In this paper, we propose a new parallel programming framework, PAlg, which is a first-order language with *participant annotations* that ensures *deadlock-freedom by construction*. PAlg programs are coupled with an abstraction of their communication structure, a *global type* from the theory of *multiparty session types* (MPST). This global type serves as an output for the programmer to assess the efficiency of their achieved parallelisation. PAlg is implemented as an EDSL in Haskell, from which we: 1. compile to low-level message-passing C code; 2. compile to sequential C code, or interpret as sequential Haskell functions; and, 3. infer the communication protocol followed by the compiled message-passing program. We use the properties of global types to perform message reordering optimisations to the compiled C code. We prove the ***extensional equivalence*** of the compiled code, as well as ***protocol compliance***. We achieve linear speedups on a shared-memory 12-core machine, and a speedup of 16 on a 2-node, 24-core NUMA.

CCS Concepts • **Computing methodologies → Concurrent programming languages**; **Parallel programming languages**; • **Software and its engineering → Parallel programming languages**; **Source code generation**.

Keywords multiparty session types, parallelism, arrows

ACM Reference Format:
David Castro-Perez and Nobuko Yoshida. 2020. Compiling First-Order Functions to Session-Typed Parallel Code. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377555.3377889

1 Introduction

Structured parallel programming is a technique for parallel programming that requires the use of high-level parallel constructs, rather than low-level send/receive operations [51; 61]. A popular approach to structured parallelism is the use of *algorithmic skeletons* [20; 35], i.e. higher-order functions that implement common patterns of parallelism. Programming in terms of high-level constructs rather than low-level send/receive operations is a successful way to avoid common concurrency bugs *by construction* [37]. One limitation of structured parallelism is that it restricts programmers to use a set of fixed, predefined parallel constructs. This is problematic if a function does not match one of the available parallel constructs, or if a program needs to be ported to an architecture where some of the skeletons have not been implemented. Unlike previous structured parallelism approaches, we do not require the existence of an underlying library or implementation of common patterns of parallelism.

In this paper, we propose a structured parallel programming framework whose front-end language is a first-order language based on the *algebra of programming* [2; 3]. The algebra of programming is a mathematical framework that codifies the basic laws of algorithmics, and it has been successfully applied to e.g. *program calculation* techniques [4], *datatype-generic programming* [34], and *parallel computing* [65]. Our framework produces message-passing parallel code from program specifications written in the front-end language. The programmer controls how the program is parallelised by *annotating* the code with *participant identifiers*. To make sure that the achieved parallelisation is satisfactory, we produce as an output a formal description of the *communication protocol* achieved by a particular parallelisation. This formal description is a *global type*, introduced by Honda et al. [41] in the theory of *Multiparty Session Types* (MPST). We prove that the parallelisation, and any optimisation performed to the low-level code respects the inferred protocol. The properties of global types justify the message reordering

---

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7120-9/20/02...$15.00
https://doi.org/10.1145/3377555.3377889

done by our back-end. In particular, we permute send and receive operations whenever sending does not depend on the values received. This is called *asynchronous optimisation* [56], and removes unnecessary synchronisation, while remaining communication-safe.

1.1 Overview

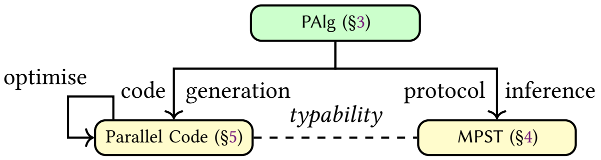
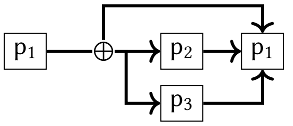


Figure 1. Overview

Our framework has three layers: (1) Parallel Algebraic Language (PAlg), a point-free first-order language with *participant annotations*, which describe which process is in charge of executing which part of the computation; (2) Message Passing Monad (Mp), a monadic language that represents low-level message-passing parallel code, from which we generate parallel C code; and (3) *global types* (from MPST), a formal description of the protocol followed by the output Mp code. Fig. 1 shows how these layers interact. PAlg, highlighted in green, is the input to our framework; and Mp and global types (MPST), highlighted in yellow, are the outputs. We prove that the generated code behaves as prescribed by the global type, and any low-level optimisation performed on the generated code must respect the protocol. As an example, we show below a parallel mergesort. mergesort.

```
msort :: (CVal a, CAlg f) => Int -> f [a] [a]
msort n = fix n $ \ms x -> vlet (vsize x) $ \sz ->
  if sz <= 1 then x
  else vlet (sz / 2) $ \sz2 ->
      vlet (par ms $ vtake sz2 x) $ \xl ->
      vlet (par ms $ vdrop sz2 x) $ \xr ->
      app merge $ pair (sz, pair (xl, xr))
```

The return type of `msort`, `f [a] [a]`, is the type of first-order programs that take lists of *values* `[a]`, and return `[a]`. Constraint `CAlg` restricts the kind of operations that are allowed in the function definition. The integer parameter to function `fix` is used for rewriting the input programs, limiting the depth of *recursion unrolling*. `par` is used to *annotate* the functions that we want to run at different processes, and function `app` is used to run functions at the same participant as their inputs. In case this input comes from different participants, first all values are gathered at any of them, and then the function is applied. We can instantiate `f` either as a sequential program, as a parallel program, or as an MPST protocol. We prove that the sequential program, and output parallel programs are *extensionally equal*, and that the output parallel program *complies* with the inferred protocol. For example, interpreting `msort 1` as a parallel program produces C code that is extensionally equal to its sequential interpretation, and behaves as the following protocol:



This is a depth 1 divide-and-conquer, where p_1 divides the task, sends the sub-tasks to p_2 and p_3, and combines the results. If the input is small, p_1 produces the result directly.

Our prototype implementation is a *tagless-final* encoding [9] in Haskell of a point-free language. Constraint `CAlg` is a first-order form of *arrows* [44; 60], with a syntactic sugar layer that allows us to write code closer to (point-wise) idiomatic Haskell. The remainder of the paper focuses on the language underlying `CAlg`.

Why Multiparty Session Types There are both practical and theoretical advantages. On the theoretical side, the theory of multiparty session types ensures *deadlock-freedom* and ***protocol compliance***. The MPST theory guarantees that the code that we generate complies with the inferred protocol (Theorem 5.2), which greatly simplifies the proof of ***extensional*** equivalence (Theorem 5.3), by allowing us to focus on representative traces, instead of all possible interleavings of actions. On the practical side, we perform message reordering optimisation based on the global types [56]. Moreover, an *explicit* representation of the communication protocol is a valuable output for programmers, since it can be used to assess a parallelisation. (Fig. 4).

1.2 Outline and Contributions

§2 defines the Algebraic Functional Language (Alg), a language inspired by the algebra of programming, that we use as a basis for our work; **§3** proposes the Parallel Algebraic Language (PAlg), our front-end language, as an extension of Alg with participant annotations; **§4** introduces a *protocol inference relation* that associates PAlg expressions with MPST protocols, specified as global types. We prove that the inferred protocols are deadlock-free: i.e. every *send* has a matching *receive*. Moreover, we use the global types to justify message reordering optimisations, while preserving communication safety; **§5** develops a translation scheme which *generates* message-passing code from PAlg, that we prove to preserve the extensionality of the input programs; **§6** demonstrates our approach using a number of examples. We will provide as an ***artifact*** our working prototype implementation, and the examples that we used in §6, with instructions on how to replicate our experiments.

2 Algebraic Functional Language

This section describes the Algebraic Functional Language (Alg) and its combinators. In functional programming languages, it is common to provide these combinators as abstractions defined in a base language. For example, one such combinator is the *split* function ($\triangle$), also known as *fanout*, or `(&&&)`, in the *arrow* literature [44] and `Control.Arrow` Haskell package [60]. Programming in terms of these combinators, avoiding explicit mention of variables is known as *point-free* programming. Another approach is to translate code written in a *pointed* style, i.e. with explicit use of variables, to a point-free style [23; 43]. This translation can be fully automated [23; 29]. In our approach, we define common point-free combinators as syntactic constructs of Alg, and require programs to be implemented in this style. Our implementation provides a layer of syntactic sugar for programmers to refer to variables explicitly, as shown in `msort` in §1, but that builds internally a point-free representation.

2.1 Syntax

$$\begin{aligned} F_1, F_2 &::= \mathsf{I} \mid \mathsf{K}a \mid F_1 + F_2 \mid F_1 \times F_2 \\ a, b &::= 1 \mid \mathsf{int} \mid \ldots \mid a \to b \mid a + b \mid a \times b \mid F\ a \mid \mu F \\ e_1, e_2 &::= f \mid v \mid \mathsf{const}\ e \mid \mathsf{id} \mid e_1 \circ e_2 \mid \pi_i \mid e_1 \triangle e_2 \mid \iota_i \mid e_1 \triangledown e_2 \\ &\quad\ \mid F\ e \mid \mathsf{in}_F \mid \mathsf{out}_F \mid \mathsf{rec}_F\ e_1\ e_2 \end{aligned}$$

In our syntax, $f_1, f_2, \ldots,$ capture ***atomic functions***, which are functions of which we only know their types; v_1, v_2 are ***values*** of primitive types (e.g. integer and boolean); $e_1, e_2, \ldots,$ represent ***expressions***; $F_1, F_2, \ldots,$ are ***functors***; and $a, b, \ldots,$ are ***types***. The syntax and semantics are standard [33; 52].

Constant, ***identity functions***, and ***function composition*** are const, id and $\circ$ respectively. ***Products*** are represented using the standard pair notation: if $x : a$ and $y : b$, then $(x, y) : a \times b$. The functions on product types are π_i and $\triangle$, and they represent, respectively, the ***projections***, and the ***split*** operation: $(f \triangle g)(x) = (f\ x, g\ x)$. ***Coproducts*** have two constructors, the ***injections*** ι_i, that build values of type $a + b$. The $\triangledown$ combinator is the ***case*** operation: $(f_1 \triangledown f_2)(\iota_i\ x) = f_i\ x$. Products and coproducts can be generalised to multiple arguments: $a \times b \times c$ is isomorphic to $a \times (b \times c)$, and to $(a \times b) \times c$. We use $\prod_{i \in [1,n]} a_i$ as notation for the product of more than two types; similarly we use $\sum$ for coproducts. The $\prod$ notation binds tighter than any other construct. Whenever $\forall i, j \in I, a_i = a_j = a$, we use the notation $\prod_n a$ as a synonym for $\prod_{i \in [1,n]} a_i$.

Functors are objects that take types into types, and functions to functions, such that identities and compositions are preserved. In this work, we focus on *polynomial functors* [31], which are defined inductively: I is the identity functor, and takes a type a to itself; $\mathsf{K}b$ is the *constant functor*, and takes any type to b; $F_1 \times F_2$ is the *product functor*, and takes a type a to $F_1\ a \times F_2\ a$; $F_1 + F_2$ is the *coproduct functor*, and takes a type to a coproduct type. A term $F\ e$ behaves as *mapping* term e to the I positions in F. For example, if $F = \mathsf{K}a \times \mathsf{I} \times \mathsf{I}$, then applying $F\ e$ to $(x,\ y,\ z)$ yields $(x,\ e\ y,\ e\ z)$.

Recursion is captured by combinators in, out, rec, and type μF. We use standard isorecursive types [31; 46; 52], where μF is *isomorphic* to $F \mu F$, and the isomorphism is given by the combinators in_F (***roll***) and out_F (***unroll***). For any polynomial functor F, μF, and strict functions in_F and out_F are guaranteed to exist. In our implementation, in_F is just a constructor (like inj_i). Recursion is $\mathsf{rec}_F\ e_1\ e_2$, and it is known as a ***hylomorphism*** [52]. A hylomorphism captures a divide-and-conquer algorithm, with a structure described by F, where e_1 is the *conquer* term and e_2 the *divide* term. Using hylomorphisms requires us to work in a semantic interpretation with *algebraic compactness*, i.e. in which carriers of initial F-algebras and terminal F-coalgebras coincide (or are isomorphic). Hylomorphisms and exponentials $\mathsf{ap} : (a \to b) \times a \to b$ allow the definition of a general fixpoint operator [53]. Working with hylomorphisms implies that our input programs may not terminate. We guarantee that, given a *terminating* input program, we will *not* produce a non-terminating parallelisation (Theorem 5.3).

Example 2.1 (MergeSort in Alg)**.** Assume a type Ls of lists of elements of type a. Functor $T = \mathsf{K}\ (\mathsf{Ls}) + \mathsf{I} \times \mathsf{I}$ captures the recursive structure of $\mathbf{ms} : \mathsf{Ls} \to \mathsf{Ls}$. When splitting some $l : \mathsf{Ls}$, we may find one of the two cases described by T: an empty or singleton list, Ls, or a list of size ≥ 2, that can be split in two halves $\mathsf{Ls} \times \mathsf{Ls}$. Assume that a functions $\mathsf{spl} : \mathsf{Ls} \to T\ \mathsf{Ls}$, and a function $\mathsf{mrg} : T\ \mathsf{Ls} \to \mathsf{Ls}$. We define $\mathbf{ms} = \mathsf{rec}_T\ \mathsf{mrg}\ \mathsf{spl}$. By the definition of rec:

$$\begin{aligned} \mathbf{ms} &= \mathsf{rec}_T\ (\mathsf{id} \triangledown \mathsf{mrg})\ \mathsf{spl} = (\mathsf{id} \triangledown \mathsf{mrg}) \circ\ T\ (\mathsf{rec}_T\ \mathsf{mrg}\ \mathsf{spl}) \circ \mathsf{spl} \\ &= (\mathsf{id} \triangledown \mathsf{mrg}) \circ\ (\mathsf{id} + (\mathsf{rec}_T\ \mathsf{mrg}\ \mathsf{spl}) \times (\mathsf{rec}_T\ \mathsf{mrg}\ \mathsf{spl})) \circ \mathsf{spl} \\ &= (\mathsf{id} \triangledown \mathsf{mrg} \circ (\mathbf{ms} \times \mathbf{ms})) \circ \mathsf{spl} \end{aligned}$$

Function **ms** first applies spl. Then, if the list was empty or singleton, it returns the input unmodified. Otherwise, **ms** applies recursively to the first and second halves. Finally, mrg returns a pair of sorted lists.

3 Parallel Algebraic Language

In the previous section we introduced Alg, a point-free functional language. In this section, we extend this language with *participant annotations*. Annotations occur both at the type and expression levels: at the type level, annotations represent *where* the data of the respective type is; at the expression level, it represents *by whom* the computation is performed. This language extension is called PAlg.

The implicit *dataflow* of the Alg (or PAlg) constructs determines which interactions must take place to evaluate an annotated program. To illustrate this, we use the Cooley-Tukey Fast-Fourier Transform algorithm [21]. The Cooley-Tukey algorithm is based on the observation that an FFT of size n, $\mathbf{fft}_n$ can be described as the combination of two FFTs of size $n/2$. We focus its high-level structure:

$$(\mathsf{add}@\mathsf{p}_1 \triangle \mathsf{sub}@\mathsf{p}_2) \circ ((\mathbf{fft}_{n/2}@\mathsf{p}_3 \circ \pi_1) \triangle ((\mathsf{exp} \circ \mathbf{fft}_{n/2})@\mathsf{p}_4 \circ \pi_2))$$

Assume that the input is a pair of vectors that contain the *deinterleaved* input, i.e. elements at even positions on the left, and odd positions on the right. We first compute the **fft** of size $n/2$ to the even and odd elements at p_3 and p_4 respectively. Then, the first half of the output is produced by adding the results pairwise (at p_1), and the second half by subtracting them (at p_2). In order to evaluate this expression, we need to know where is the input data. This is specified by the programmer as an annotated type, which we call ***interface***. Suppose that the interface specifies that the even elements are at p, and the odd elements at p′. The interface that represents this scenario is $(\mathsf{vec}\times\mathsf{vec})@(\mathsf{p}\times\mathsf{p}')$, i.e. an annotated pair of vectors, with the first component at p, and the second component at p′. By keeping track of the locations of the data, we obtain type $(\mathsf{vec}\times\mathsf{vec})@(\mathsf{p}_1\times\mathsf{p}_2)$, which is the *output* (or *codomain*) interface the PAlg expression. We also refer to the annotations (e.g $\mathsf{p}_1\times\mathsf{p}_2$) as interfaces, whenever there is no ambiguity. We write $\mathbf{fft}_n : (\mathsf{vec}\times\mathsf{vec})@(\mathsf{p}\times\mathsf{p}') \longrightarrow (\mathsf{vec}\times\mathsf{vec})@(\mathsf{p}_1\times\mathsf{p}_2)$ to represent the input and output interfaces of $\mathbf{fft}_n$.

Consider now $e_1@\mathsf{p}_1 \triangledown e_2@\mathsf{p}_2$. The output interface of this expression is either p_1 or p_2, depending on whether the input is the result of applying ι_1 or ι_2. We represent such interfaces using *unions*: $e_1@\mathsf{p}_1 \triangledown e_2@\mathsf{p}_2 : (a+b)@\mathsf{p} \longrightarrow c@(\mathsf{p}_1\cup\mathsf{p}_2)$. Since p contains a value of a sum type $a+b$, p is responsible for notifying both p_1 and p_2 which branch needs to be taken in the control flow. Incorrectly notifying the necessary participants will produce incorrect parallelisations that might deadlock. For example, consider the expression $e_0@\mathsf{p}_0 \circ (e_1@\mathsf{p}_1 \triangledown e_2@\mathsf{p}_2)$. Assuming that the input at p, p needs to notify p_0, otherwise p_0 will be stuck. To avoid such cases, and to compute the interfaces of an expression, we define a type system for PAlg.

3.1 Syntax of PAlg

$$I ::= \mathsf{p} \mid \iota_i\ I \mid I\times I \qquad R ::= I \mid R \cup^{\vec{\mathsf{p}}} R \qquad P ::= R \longrightarrow R$$
$$\mathsf{e} ::= e@\mathsf{p} \mid [\mathsf{p}\oplus\vec{\mathsf{p}}] \mid \mathsf{id} \mid \mathsf{e}\circ\mathsf{e} \mid \pi_i \mid \mathsf{e}\vartriangle\mathsf{e} \mid \iota_i \mid \mathsf{e}\triangledown\mathsf{e}$$

The syntax of PAlg is that of Alg, extended with participant annotations (red). Note that certain Alg constructs can only occur under annotations ($e@\mathsf{p}$), e.g: in, out and rec. This implies that recursive functions need to be annotated at a single participant. To parallelise recursive functions, they need first to be rewritten into a suitable form, and then annotate the resulting expression. At the moment, we support automatic recursion unrolling up to a user-specified depth. We provide an overview of the main syntactic constructs of PAlg: ***annotations***, ***interfaces***, and ***annotated functions***.

Annotations are ranged over by R, R',... We define them in two layers, I, or simple annotations that cannot contain choices ($\cup$), and R. This way, we ensure that choices only occur at the topmost level. Simple annotations are: participant ids p, that identify processes; products of interfaces $I_1\times I_2$; and tagged interfaces $\iota_i\ I$, that keep track of the branch of the choice that led to I. A choice $R_1 \cup^{\vec{\mathsf{p}}} R_2$ describes an scenario that is the result of a branch in the control flow, where a value can be found at either R_1 or R_2. Here, $\vec{\mathsf{p}} = \mathsf{p}_1\cdots\mathsf{p}_n$ are the participants whose behaviour depends on the path in the control flow. Finally, arrows P of the form $R_1 \longrightarrow R_2$ represent the input/output annotations of a parallel program.

Interfaces are annotated types. They range over A, B, …, and are of the form $a@R$, which means that values of type a are distributed across R. We require annotated types to be *well-formed*, $\mathsf{WF}(a@R)$, which implies that the structure of a matches that of R. We write $\mathcal{I}$ to represent *one-hole contexts* for interfaces, with $\mathcal{I}[\mathsf{p}]$ representing the interface that results of placing p at the hole in $\mathcal{I}$.

Annotated functions are ranged over by e, e'. The annotations are introduced using $e@\mathsf{p}$, where e is an unannotated Alg expression, and p is a single participant identifier. These annotations need to be set by the programmer, but their introduction can be also automated. Additionally, we introduce the choice point annotations: $[\mathsf{p}\oplus\vec{\mathsf{p}}]$. This annotation specifies that p performs a choice, and notifies $\vec{\mathsf{p}}$. Choice points can be introduced fully automatically by collecting all participants whose behaviour depends on the value of a sum type.

3.2 Interfaces

An interface represents a *state* in a concurrent system: the set of participants, and the types of the values that they contain. We use mappings from participants to values to represent such states: $V := [\mathsf{p} \mapsto v]_{\mathsf{p}\in\mathcal{P}}$. The programmer, additionally to writing an Alg (PAlg) expression, will need to provide an *input interface*, i.e. *where* is the input to the parallel program. Consider, for example, the interface $\mathsf{int}@\mathsf{p}_i$. Given a concurrent system with participants $\mathsf{p}_o\cdots\mathsf{p}_n$, we know that p_i contains a value of type int: $[\cdots\ \mathsf{p}_i \mapsto 42\ \cdots]$. An interface with a product of participants $(a\times b)@(\mathsf{p}_1\times\mathsf{p}_2)$ represents a state in which p_1 contains an element of type a, and p_2 an element of type b, e.g a possible state represented by $(\mathsf{int}\times\mathsf{vec})@(\mathsf{p}_1\times\mathsf{p}_2)$ is: $[\cdots\mathsf{p}_1\mapsto 42\cdots\mathsf{p}_2\mapsto[1,1,2,\ldots]\ \cdots]$. An interface $\iota_i\ I$ represents the same state as interface I, but we statically know that this state was reached after an i-th injection. Then, if a participant requires the value at I, this participant will apply the necessary injections to the received values. Finally, an interface $a@(R_1\cup^{\vec{\mathsf{p}}} R_2)$ means that the state might be either R_1 or R_2, and that all participants $\vec{\mathsf{p}}$ should be notified of the state.

Well-formedness The above examples are of well-formed interfaces: $\mathsf{int}@\mathsf{p}_i$, $(\mathsf{int}\times\mathsf{vec})@(\mathsf{p}_1\times\mathsf{p}_2)$. Well-formedness ensures that interfaces represent valid states. Generally, $a@R$ is well-formed if a matches the structure of R. For example, $\mathsf{int}@(\mathsf{p}_1\times\mathsf{p}_2)$ is *ill-formed*, since a *single* integer cannot be at *two* different participants. An interface $a@(R_1\cup R_2)$ requires that both $a@R_1$ and $a@R_2$ are well-formed. So, $(\mathsf{vec}\times\mathsf{vec})@((\mathsf{p}_1\times\mathsf{p}_2)\cup\mathsf{p}_3)$ is well-formed because we can have $\mathsf{vec}@\mathsf{p}_1$ and $\mathsf{vec}@\mathsf{p}_2$, or $(\mathsf{vec}\times\mathsf{vec})@\mathsf{p}_3$. However, $\mathsf{int}@((\mathsf{p}_1\times\mathsf{p}_2)\cup\mathsf{p}_3)$ is ill-formed, because $\mathsf{int}@(\mathsf{p}_1\times\mathsf{p}_2)$ is ill-formed.

3.3 Typing of Parallel Algebraic Language

We introduce a relation that associates Alg expressions with potential parallelisations PAlg, and their interfaces. This relation can be seen as a type system for both Alg and PAlg. As a type system for PAlg, this relation provides a way to check or infer the output interface of some e. By using this relation as a type system for Alg, we can explore potential parallelisations of some input expression e. Additionally, the type system ensures that all *choice point* annotations contain every participant that depends on each particular choice.

Typing Rules A judgement of the form $\vdash e \Rightarrow \mathsf{e} : A \longrightarrow B$ means that the PAlg expression e is one potential parallelisation of the Alg expression e, with domain interface A and codomain interface B. The intuition of a judgement $\vdash e \Rightarrow \mathsf{e} : a@R_1 \longrightarrow b@R_2$ is that the participants in e collectively apply computation e to the value of type a distributed across R_1, and produce a value of type b distributed across R_2. We sometimes omit e and write $\vdash \mathsf{e} : A \longrightarrow B$. We ensure that given any e and e such that they are typeable against interfaces $a@R_a \longrightarrow b@R_b$, then e must have type $a \longrightarrow b$.

Lemma 3.1. *If* $e \Rightarrow \mathsf{e} : a@R_a \longrightarrow b@R_b$, *then* $e : a \longrightarrow b$.

The typing rules (Fig. 2) must ensure that the participants involved in a choice are notified, and that Alg expressions are correctly expanded. Rule Choice specifies that a choice point may be introduced at any point when a participant contains a value of a sum-type. In such cases p sends the tag of the sum-type value to any other participant whose behaviour depends on it. After the choice point, the interface is $I[\iota_1\ \mathsf{p}] \cup^{\vec{\mathsf{p}}} I[\iota_2\ \mathsf{p}]$, with the constraint that the participants in $\mathcal{I}[\mathsf{p}]$ must be in $\vec{\mathsf{p}}$. Rule Alt specifies that e must be the parallelisation of e, considering both A_1 and A_2 as input interfaces. The output interface is the union of B_1 and B_2. Any participant in e must be notified of the choice $\mathsf{pids}(\mathsf{e}) \subseteq \vec{\mathsf{p}}$, to make sure that they perform the interactions that correspond to the correct A_i. Rule Alg specifies that given any e and participant p, $e@\mathsf{p}$ is a valid parallelisation, with output interface $b@\mathsf{p}$. Finally, rule Ext is crucial for exploring potential parallelisations. It states that if e is the parallelisation of e_2, and e_2 is extensionally equal to e_1, then e is also a parallelisation of e_1. The undecidability of this rule requires that the programmer specifies *rewriting strategies* both for checking and inference.

Rewriting and Annotation Strategies We use rewriting strategies when exploring potential parallelisations of functions. This is inference problem (2) below. Let **?i** be metavariables. The two inference problems that we are interested in are: 1. Solving $\vdash e \Rightarrow \mathsf{e} : A \longrightarrow$ **?0** obtains the output interface for e, with input interface A. 2. Solutions of $\vdash e \Rightarrow$ **?0** $: A \longrightarrow$ **?1** are potential parallelisations of e, and their output interface. Solving (1) is straightforward. Problem (2) requires to decide how to introduce role annotations (rule Alg), how to perform rewritings (rule Ext), and where to introduce choice points (rule Choice). Introducing choice points is straightforward: we introduce them as early as possible, as soon as an input interface contains a sum-type at a participant. For introducing annotations and doing Alg rewritings, the programmer has to specify *annotation* and *rewriting* strategies. At the moment, our tool allows the developer to introduce annotations explicitly, or to select sub-expressions that will be annotated with fresh new participants. The rewriting strategies that our current implementation supports are unrollings of recursive definitions. However, our tool is extensible: the equivalences used in the rewritings are a parameter.

$$\textsc{Alg}\quad \frac{\vdash e : a \longrightarrow b}{\vdash e \Rightarrow e@\mathsf{p} : a@I \longrightarrow b@\mathsf{p}} \qquad \textsc{Ext}\quad \frac{\vdash e_2 \Rightarrow \mathsf{e} : a@I \longrightarrow B \qquad e_1 =_{\mathrm{ext}} e_2}{\vdash e_1 \Rightarrow \mathsf{e} : a@I \longrightarrow B}$$

$$\textsc{Alt}\quad \frac{\vdash e \Rightarrow \mathsf{e} : A_1 \longrightarrow B_1 \quad \vdash e \Rightarrow \mathsf{e} : A_2 \longrightarrow B_2 \quad A_1 \neq A_2 \quad \mathsf{pids}(\mathsf{e}) \subseteq \vec{r}}{\vdash e \Rightarrow \mathsf{e} : A_1 \cup^{\vec{\mathsf{p}}} A_2 \longrightarrow B_1 \cup^{\vec{\mathsf{p}}} B_2}$$

$$\textsc{Choice}\quad \frac{\begin{array}{c}\mathsf{pids}(\mathcal{I}[\mathsf{p}]) \subseteq \vec{\mathsf{p}} \\ \mathsf{WF}(a@\mathcal{I}[\iota_i\ \mathsf{p}]), i \in [1,2] \quad \vdash e \Rightarrow \mathsf{e} : a@(\mathcal{I}[\iota_1\ \mathsf{p}] \cup^{\vec{\mathsf{p}}} \mathcal{I}[\iota_2\ \mathsf{p}]) \longrightarrow B\end{array}}{\vdash e \Rightarrow \mathsf{e} \circ [\mathsf{p} \oplus \vec{\mathsf{p}}] : a@\mathcal{I}[\mathsf{p}] \longrightarrow B}$$

Figure 2. Typing rules of PAlg (selected)

Example 3.2 (Mergesort)**.** Consider the mergesort definition $\mathbf{ms} = \mathsf{rec}_T\ \mathsf{mrg}\ \mathsf{spl}$. Solutions to the inference problem $\vdash \mathbf{ms} \Rightarrow$ **?0** $: \mathsf{Ls}@\mathsf{p}_0 \longrightarrow$ **?1** provide the alternative parallelisations of **ms**. By choosing a rewriting strategy that unrolls **ms** once, and annotates any remaining instances of **ms** at fresh new participants, we produce the following PAlg expression:

$$\begin{array}{l}\vdash (\mathsf{id} \triangledown (\mathsf{mrg} \circ (\mathbf{ms} \times \mathbf{ms}))) \circ \mathsf{spl} \\ \quad \Rightarrow (\mathsf{id} \triangledown (\mathsf{mrg}@\mathsf{p}_1 \circ (\mathbf{ms}@\mathsf{p}_2 \circ \pi_1@\mathsf{p}_1) \vartriangle (\mathbf{ms}@\mathsf{p}_3 \circ \pi_2@\mathsf{p}_1))) \\ \qquad \circ [\mathsf{p}_1 \oplus \mathsf{p}_1\mathsf{p}_2\mathsf{p}_3] \circ \mathsf{spl}@\mathsf{p}_1 \\ \quad : \mathsf{Ls}@\mathsf{p}_0 \longrightarrow \mathsf{Ls}@\mathsf{p}_1 \cup^{\mathsf{p}_1\mathsf{p}_2\mathsf{p}_3} \mathsf{Ls}@\mathsf{p}_1\end{array}$$

4 Multiparty Session Types for PAlg

The dataflow of the PAlg constructs determine the communication protocol of the annotated expression. However, it is hard to manually check what this communication structure is. Recall the mergesort PAlg expression of §3, **ms**, and suppose that we want to produce a parallelisation for a 32-core machine. Then, we might be interested in using a 5-unfolding of **ms**, so that we have **ms** executing concurrently on all of the cores. How do we know, for such cases, that we produced a *sensible* parallelisation? As an example, suppose we use an annotation strategy that produces the following code:

$$\begin{array}{l}(\mathsf{id} \triangledown (\mathsf{mrg}@\mathsf{p}_1 \circ (\mathbf{ms}@\mathsf{p}_2 \circ \pi_1@\mathsf{p}_1) \vartriangle (\mathbf{ms}@\mathsf{p}_2 \circ \pi_2@\mathsf{p}_1))) \\ \quad \circ [\mathsf{p}_1 \oplus \mathsf{p}_1\mathsf{p}_2] \circ \mathsf{spl}@\mathsf{p}_1 : \mathsf{Ls}@\mathsf{p}_0 \longrightarrow \mathsf{Ls}@\mathsf{p}_1 \cup^{\mathsf{p}_1\mathsf{p}_2} \mathsf{Ls}@\mathsf{p}_1\end{array}$$

Notice that this example will run correctly, and produce the expected result. However, the achieved PAlg expression is *not* parallel! If we represent the ***implicit dataflow*** of this expression as ***explicit communication***, the reason becomes apparent. We use *global types* from *multiparty session types* to provide an explicit representation of the communication structure of the program:

$$\begin{array}{l} \mathsf{p}_0 \to \mathsf{p}_1 : \mathsf{Ls}.\ \mathsf{p}_1 \to \mathsf{p}_2\{\iota_1.\ \mathsf{end}; \\ \quad \iota_2.\ \mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{Ls}.\ \mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{Ls}.\mathsf{p}_2 \to \mathsf{p}_1 : \mathsf{Ls} \times \mathsf{Ls}.\ \mathsf{end}\} \end{array}$$

This global type represents the following protocol: 1. participant p_0 sends a list to p_1; 2. p_1 sends to p_2 either ι_1 or ι_2, and if the label is ι_1, the protocol ends; 3. if p_1 sent ι_2, then p_1 sends to p_2 *two* lists, in two different interactions; and 4. p_2 replies with a message to p_1 with a pair of lists. It is clear from this protocol that p_1 and p_2 are dependent on each others' messages, and that p_2 cannot perform any computation in parallel. The larger the expression is, the harder avoiding these wrong annotations will become. By changing the annotation strategy, we produce the following parallel structure, where p_2 and p_3 can operate in parallel:

$$\begin{array}{l} \mathsf{p}_0 \to \mathsf{p}_1 : \mathsf{Ls}.\ \mathsf{p}_1 \to \{\mathsf{p}_2\mathsf{p}_3\}\ \{\iota_1.\ \mathsf{end}; \\ \quad \iota_2.\ \mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{Ls}.\ \mathsf{p}_1 \to \mathsf{p}_3 : \mathsf{Ls}.\mathsf{p}_2 \to \mathsf{p}_1 : \mathsf{Ls}.\ \mathsf{p}_3 \to \mathsf{p}_1 : \mathsf{Ls}.\ \mathsf{end}\} \end{array}$$

This abstraction of the communication protocol of an achieved parallelisation is therefore useful as an output for the programmer. Additionally, these global types are a *contract* that can be enforced on the generated code. We use this for proving that our back-end is correct, but also for applying low-level code optimisations (e.g. message reordering) guided by this global type, ensuring that they do not introduce any run-time error. For example, when we find in a global type $\mathsf{p}_1 \to \mathsf{p}_2.\ \mathsf{p}_2 \to \mathsf{p}_3$, we mark the send/receive actions for p_2 as point of potential optimisation. If the messages exchanged do not depend on each other, we permute them, performing first the send action, so that p_2 is not blocked by a receive action. This is known as *asynchronous optimisation* [56].

4.1 Multiparty Session Types

Our global types are based on the most commonly used in the literature [22]. We start with a set of *participant identifiers*, $\mathsf{p}_1, \mathsf{p}_2, \ldots$, and a set of *labels*, $\iota_1, \iota_2, \ldots$. These are considered as natural numbers: participant identifiers uniquely identify an independent unit of computation, e.g. thread or process ids; and labels are tags that differentiate branches in the data/control flow. The syntax of global (G) and local (L) types in MPST is given as:

$$\begin{array}{rl} G ::= & \mathsf{p}_1 \longrightarrow \mathsf{p}_2 : a.G \mid \mathsf{p}_1 \longrightarrow \{\mathsf{p}_j\}_{j\in[2,n]} : \{\iota_i.G_i\}_{i\in I} \\ & \mid \mu X.G \mid X \mid \mathsf{end} \\ L ::= & \mathsf{p}!\langle a\rangle.L \mid \mathsf{p}?(a).L \mid \mathsf{p}\ \&\ \{\iota_i.L_i\}_{i\in I} \mid \{\mathsf{p}_j\}_{j\in[2,n]} \oplus \{\iota_i.L_i\}_{i\in I} \\ & \mid \mu X.L \mid X \mid \mathsf{end} \end{array}$$

Global type $\mathsf{p}_1 \longrightarrow \mathsf{p}_2 : a.G$ denotes *data* interactions from p_1 to p_2 with value of type a; *Branching* is represented by $\mathsf{p}_1 \longrightarrow \{\mathsf{p}_j\}_{j\in[2,n]} : \{\iota_i.G_i\}_{i\in I}$ with actions ι_i from p_1 to all $\mathsf{p}_j, j \in [2,n]$. end represents a *termination* of the protocol. $\mu X.G$ represents a *recursive* protocol, which is *equivalent* to $[\mu X.\ G/X]G$. We assume recursive types are guarded.

$$\textsc{Choice}\quad \frac{}{\vDash [\mathsf{p} \oplus \vec{\mathsf{p}}] \Leftarrow a[b+c]@I[\mathsf{p}] \sim \mathsf{p} \longrightarrow \{\vec{\mathsf{p}} \setminus \mathsf{p}\}\{\iota_1.\ \mathsf{end};\ \iota_2.\ \mathsf{end}\}}$$

$$\textsc{Alt}\quad \frac{\vDash \mathsf{e} \Leftarrow A_1 \sim G_1 \qquad \vDash \mathsf{e} \Leftarrow A_2 \sim G_2}{\vDash \mathsf{e} \Leftarrow A_1 \cup^{\vec{\mathsf{p}}} A_2 \sim G_1 \cup G_2} \qquad \textsc{Alg}\quad \frac{\vdash e : a \longrightarrow b}{\vDash e@\mathsf{p} \Leftarrow a@I \sim [a@I \rightsquigarrow \mathsf{p}]}$$

$$\begin{array}{c} [a@\mathsf{p}_1 \rightsquigarrow \mathsf{p}_2] = \mathsf{p}_1 \longrightarrow \mathsf{p}_2 : a.\ \mathsf{end},\ \text{if}\ \mathsf{p}_1 \neq \mathsf{p}_2;\ [a@\mathsf{p} \rightsquigarrow \mathsf{p}] = \mathsf{end}; \\ [(a \times b)@(I_a \times I_b) \rightsquigarrow \mathsf{p}] = [a@I_a \rightsquigarrow \mathsf{p}] \fatsemi [b@I_b \rightsquigarrow \mathsf{p}];\ \text{and} \\ [(a_1 + a_2)@(\iota_i\ I) \rightsquigarrow \mathsf{p}] = [a_i@I \rightsquigarrow \mathsf{p}] \end{array}$$

Figure 3. Protocol Relation (selected)

Each participant in G represents a different participant in a parallel process. *Local session types* represent the communication actions performed by each participant, i.e. the *role* of the participant. Since each participant has a unique role, we sometimes refer to them interchangeably. The *send* type $\mathsf{p}!\langle a\rangle.L$ expresses the action of sending of a value of type a to p followed by interactions specified by L. The *receive* type $\mathsf{p}?(a).L$ is the dual, where a value with type a is received from p. The *selection* type represents the transmission to all p_j of label ι_i chosen in the set of labels ($i \in I$) followed by L_i. The *branching* type is its dual. $\mathsf{pids}(G)/\mathsf{pids}(L)$ denote the set of participants that occur in G/L.

Projection We use a standard definition of *projection* that uses the *full merging* operator [24; 27], which allows more well-formed global types than the original projection rules [41]. We write $G \upharpoonright \mathsf{p}$ for the projection of G onto the role of p. We illustrate the projection with an interaction $\mathsf{p}_0 \to \mathsf{p}_1 : a.G$. The projection onto p_0 is $\mathsf{p}_1!\langle a\rangle.(G \upharpoonright \mathsf{p}_0)$, the projection onto p_1 is $\mathsf{p}_0?(a).(G \upharpoonright \mathsf{p}_1)$, and the projection onto any other role p is $G \upharpoonright \mathsf{p}$. Projection on choices is similar, with the difference that whenever the role is not at the receiving or sending ends of the choice, the different branches must be *merged*. Two local types can be merged when they are the same, or they branch on the same role, and their continuations can be merged.

We use a standard definition of *well-formedness* that states that a global type is well formed if ts projection on all its roles is defined. We denote: $\mathsf{WF}(G) = \forall \mathsf{p} \in \mathsf{pids}(G), \exists L, G \upharpoonright \mathsf{p} = L$.

4.2 Protocol Relation

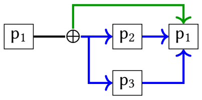
We introduce now the set of rules that associate a PAlg expression and domain interface with their global type (Fig. 3). We extend the syntax of global types with $G_1 \cup^{\vec{\mathsf{p}}} G_2$ to represent the *external choices*, i.e. G_i are the continuations for both branches of a previous choice that affects $\vec{\mathsf{p}}$. We also extend the local types, and projection rules $(G_1 \cup^{\vec{\mathsf{p}}} G_2) = G_1 \upharpoonright \mathsf{p} \cup^{\vec{\mathsf{p}}} G_2 \upharpoonright \mathsf{p}$, and the notion of well-formedness. We say that an external choice is well-formed, $\mathsf{WF}(G_1 \cup^{\vec{\mathrm{p}}} G_2)$, if $\mathsf{WF}(G_1)$, $\mathsf{WF}(G_2)$, and for all $\mathrm{p} \notin \vec{\mathrm{p}}$, $G_1 \restriction \mathrm{p} = G_2 \restriction \mathrm{p}$. We omit the annotation of the participants involved in the choice whenever it is not needed. The relation $\vDash p \Leftarrow A \sim (G, B)$ specifies that the parallel code for p and input interface A will behave as global type G, and output interface B (Fig. 3). The rules are similar to the typing rules of PAlg.

Example 4.1 (Mergesort Protocol)**.** The protocol for Example 3.2 is obtained by solving:

$$\vDash (\mathsf{id} \triangledown (\mathsf{mrg}@\mathrm{p}_1 \circ (\mathbf{ms}@\mathrm{p}_2 \circ \pi_1@\mathrm{p}_1) \vartriangle (\mathbf{ms}@\mathrm{p}_3 \circ \pi_2@\mathrm{p}_1))) \circ [\mathrm{p}_1 \oplus \mathrm{p}_1\mathrm{p}_2\mathrm{p}_3] \circ \mathsf{spl}@\mathrm{p}_1 \Leftarrow \mathsf{Ls}@\mathrm{p}_1 \sim \mathbf{?0}.$$

$$\mathrm{p}_1 \to \{\mathrm{p}_2\mathrm{p}_3\} \left\{ \begin{array}{l} \iota_1.\mathsf{end}; \\ \iota_2.\mathrm{p}_1 \to \mathrm{p}_2 : \mathsf{Ls}.\mathrm{p}_1 \to \mathrm{p}_3 : \mathsf{Ls}.\mathsf{end} \end{array} \right\} \fatsemi \\ (\mathsf{end} \cup (\mathrm{p}_2 \to \mathrm{p}_1 : \mathsf{Ls}.\ \mathrm{p}_3 \to \mathrm{p}_1 : \mathsf{Ls}.\mathsf{end}))$$

$$= \mathrm{p}_1 \to \{\mathrm{p}_2\mathrm{p}_3\} \left\{ \begin{array}{l} \iota_1.\mathsf{end}; \\ \iota_2.\mathrm{p}_1 \to \mathrm{p}_2 : \mathsf{Ls}. \\ \quad \mathrm{p}_1 \to \mathrm{p}_3 : \mathsf{Ls}. \\ \quad \mathrm{p}_2 \to \mathrm{p}_1 : \mathsf{Ls}. \\ \quad \mathrm{p}_3 \to \mathrm{p}_1 : \mathsf{Ls}.\mathsf{end} \end{array} \right\}$$



4.3 Correctness

We guarantee that for e s.t. $\vdash e \Rightarrow \mathrm{e} : A \to B$, with A and B well-formed, there exists a protocol G and that it is well-formed and deadlock-free.

Lemma 4.2. *[Existence of Associated Global Type] For all* $\mathsf{WF}(A)$, *if* $\vdash \mathrm{e} : A \to B$, *then there exists* G *s.t.* $\vDash \mathrm{e} \Leftarrow A \sim G$.

Lemma 4.3. *[Protocol Deadlock-Freedom] For all* $\mathsf{WF}(A)$, *if* $\vdash \mathrm{e} : A \to B$ *and* $\vDash \mathrm{e} \Leftarrow A \sim G$, *then* $\mathsf{WF}(G)$.

Remark. *Since the local type abstracts the behaviour of multiparty typed processes, a well-formed global type ensures the end-point processes (programs) typed by that global type are guaranteed to satisfy the properties (such as safety and deadlock-freedom) of local types [27; 42].*

5 Code Generation

This section addresses the problem of generating low-level parallel code from PAlg expressions. We prove that *the generated code complies with its inferred protocol*, which has several implications: (1) code generation does not introduce any concurrency errors, and the parallel code is therefore deadlock-free; and (2) we can prove that *the generated code is extensionally equal to the input expression* by considering only a representative trace, since any valid interleaving of actions must respect this protocol. The target language of our tool is an indexed monad, the *Message Passing Monad* (Mp). From Mp, we implement our low-level C backend. We implement an untyped version of Mp as a deep embedding in Haskell, and session typing on top of it. This is suitable for code generation: we only generate parallel code if the monadic actions are typeable against the respective local types. Our definition of Mp has significant differences to other embeddings of session types in Haskell, such as the Session monad by Neubauer and Thiemann [57]. First, our Mp monad is deeply embedded in Haskell, and secondly, we use type indices instead of an encoding of session types in terms of type classes. Our approach is better suited for compilation since we manipulate session types, and postpone session typing until code generation.

5.1 Message Passing Monad

Mp comprises four basic operations: send, receive, choice and branching, with a standard (asynchronous) semantics. Additionally, for composing actions that depend on the same choice, we introduce case expressions. Our definition of Mp is based on the *free monad* construction:

$$\begin{array}{ll} v & ::= x \mid (v, v) \mid \iota_i\ v \mid \cdots \mid e\ v \\ m_i & ::= \mathbf{ret}\ v \mid \mathbf{send}\ \mathrm{p}\ v\ m \mid \mathbf{recv}\ \mathrm{p}\ a\ f \mid \mathbf{sel}\ \vec{\mathrm{p}}\ v\ f_1\ f_2 \\ & \quad \mid \mathbf{brn}\ \mathrm{p}\ m_1\ m_2 \mid \mathbf{case}\ f_1\ f_2 \quad f ::= \lambda x.m \end{array}$$

Values v are either primitive values, tagged values $\iota_i\ v$, pairs of values, or the result of applying an Alg expression e to a value. We use standard notation for the monadic unit (**ret**), bind ($\gg\!=$) and Kleisli composition: $f_1 {>\!\!=\!\!>} f_2 = \lambda x.\ f_1\ x {\gg\!=} f_2$. The message-passing constructs are standard, except **sel**, **brn** and **case**, which are used for performing choices, and composing actions that depend on the same choice.

Each monadic computation f or m has a type $m : \mathsf{Mp}\ L\ a$, where a is the return type of m, and L is the type index of Mp, and it represents the local type that corresponds to the behaviour of the term m. There is almost a one to one correspondence between the terms L and the monadic actions m, so we omit the full definition. The types of the constructs that deal with choices use a new type, $\uplus$, that is isomorphic to sum types, but that can only be constructed and eliminated by using the corresponding monadic constructs:

$$\begin{array}{l} \mathbf{sel}\ \vec{\mathrm{p}} : a + b \to (a \to \mathsf{Mp}\ L_1\ c_1) \to (b \to \mathsf{Mp}\ L_2\ c_2) \\ \qquad \to \mathsf{Mp}\ (\vec{\mathrm{p}} \oplus \{\iota_1.L_1; \iota_2.L_2\})\ (c_1 \uplus c_2) \\ \mathbf{brn}\ \mathrm{p} : \mathsf{Mp}\ L_1\ a_1 \to \mathsf{Mp}\ L_2\ a_2 \\ \qquad \to \mathsf{Mp}\ (\mathrm{p}\ \&\ \{\iota_1.L_1; \iota_2.L_2\})\ (a_1 \uplus a_2) \\ \mathbf{case} : (a \to \mathsf{Mp}\ L_1\ c) \to (b \to \mathsf{Mp}\ L_2\ d) \to a \uplus b \\ \qquad \to \mathsf{Mp}\ (L_1 \cup L_2)\ (c \uplus d) \end{array}$$

These constructs ensure that the tag used to build $a \uplus b$ indeed corresponds to the correct branch of the right choice. We use **case** to compose actions that depend on a previous choice. While this treatment of $\uplus$ leads to unnecessary code duplication, our back-end easily optimises cases where we have **case** $f\ f$ to avoid code duplication.

Parallel programs We define the basic constructs of PAlg in a bottom-up way by manipulating *parallel programs*. Parallel programs are mappings from participants to their monadic action: $\mathrm{E} ::= [\mathrm{p}_i \mapsto m_i]_{i \in I}$. If $m_i : \mathsf{Mp}\ L_i\ a_i$ for all $i \in I$, then we write $[\mathrm{p}_i \mapsto m_i]_{i \in I} : \mathsf{Mp}\ [\mathrm{p}_i \mapsto L_i]_{i \in I}\ [\mathrm{p}_i \mapsto a_i]_{i \in I}$. The semantics of both local types and monadic actions is defined in terms of such collections of actions or local types, and shared queues of values W, or queues of types Q, e.g. $\langle \mathsf{E}, W\rangle \rightsquigarrow^{\ell} \langle \mathsf{E}', W'\rangle$ is a transition from E to E', and shared queues W to W' with *observable action* ℓ. We prove a standard safety theorem (Theorem 5.1 below) that guarantees that if a participant does a transition with some observable action, then so does the type index.

Theorem 5.1. *[Soundness] Assume* $\mathsf{E} : \mathsf{Mp}\ C\ A$, $m : \mathsf{Mp}\ L\ a$ *and* $W : Q$. *Suppose* $\langle \mathsf{E}[r \mapsto m], W\rangle \rightsquigarrow^{\ell} \langle \mathsf{E}[r \mapsto m'], W'\rangle$. *Then there exists* $\langle C[r \mapsto L], Q\rangle \to^{\ell} \langle C[r \mapsto L'], Q'\rangle$ *such that* $W' : Q'$ *and* $m' : \mathsf{Mp}\ L'\ a$.

Mp **code generation** The translation scheme for Mp code generation is done recursively on the structure of PAlg expressions. It takes a PAlg expression e, an interface A, and produces a mapping from all participants in e and A to their respective monadic continuations. We write $[\![\mathsf{e}]\!]\,(A)$, and guarantee that $[\![\mathsf{e}]\!]\,(A) : A \to \mathsf{Mp}\ G\ B$, if $\vDash \mathsf{e} \Leftarrow A \sim (G, B)$. This means that if e induces protocol G with interfaces $A \to B$, then the generated code behaves as G, with interfaces A and B. Code generation follows a similar structure to global type inference, and is defined by building PAlg constructs as Mp parallel programs. For example, the translation of $\mathsf{e@p} : a@I \to B$ requires to define the interactions from an interface I that gathers a type a at p: $(\!| a@I \rightsquigarrow \mathsf{p} |\!) : a@I \to \mathsf{Mp}\ [a@I \rightsquigarrow \mathsf{p}]\ (a@\mathsf{p})$. The definition is analogous to that of $[a@I \rightsquigarrow \mathsf{p}]$. The remaining of the translation is straightforward, so we skip the details.

We prove two main correctness results. We guarantee that the generated code behaves as its inferred protocol (Theorem 5.2). We also guarantee that regardless of the annotations and interfaces chosen for e, *the parallel code always produces the same result as the sequential implementation* (Theorem 5.3).

Theorem 5.2. *[Protocol Conformance of the Generated Code] If* $\vDash \mathsf{e} \Leftarrow A \sim G$, *then* $[\![\mathsf{e}]\!]\,(A)$ *complies with protocol* G.

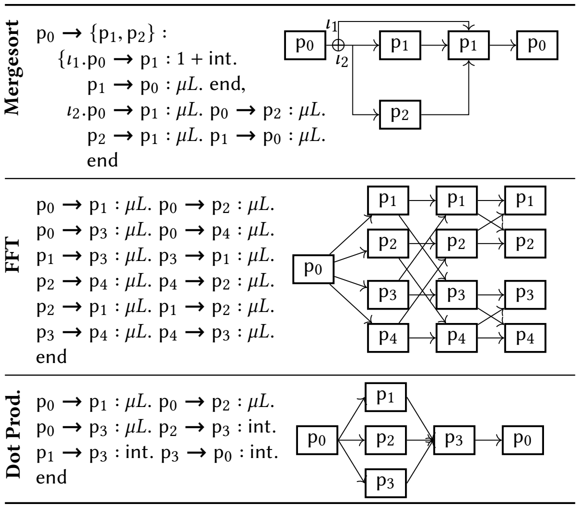
Theorem 5.3. *[Extensionality] Assume* $e \Rightarrow \mathsf{e} : a@\mathsf{p} \to b@R$ *and* $x : a$ *initially at* p. *If* $e\ x = y$, *then the execution of* $[\![\mathsf{e}]\!]\,(\mathsf{p})$ *also produces* y, *distributed across* R.

Example 5.4 (MergeSort Code Generation)**.** We show below the code generation for **ms** (Example 3.2), with p_1 as domain interface:

$$
\begin{aligned}
&\mathsf{p}_1 \mapsto \lambda x.\ \mathbf{sel}\ \{\mathsf{p}_2, \mathsf{p}_3\}\ (\mathsf{spl}\ x)\ (\lambda x.\ \mathbf{ret}\ x)\\
&\qquad\qquad (\lambda x.\ \mathbf{send}\ \mathsf{p}_2\ (\pi_1\ x) \ggg\!= \lambda y.\ \mathbf{send}\ \mathsf{p}_3\ (\pi_2\ x) \ggg\!= \lambda\_.\\
&\qquad\qquad\quad \mathbf{recv}\ \mathsf{p}_2\ \mathsf{Ls} \ggg\!= \lambda x.\ \mathbf{recv}\ \mathsf{p}_3\ \mathsf{Ls} \ggg\!= \lambda y.\ \mathbf{ret}\ (\mathsf{mrg}\ (x, y)))\\
&\mathsf{p}_{2,3} \mapsto \lambda x.\ \mathbf{brn}\ \mathsf{p}_1\ (\mathbf{ret}\ x)\ (\mathbf{recv}\ \mathsf{p}_1\ \mathsf{Ls} \ggg\!= \lambda x. \mathbf{send}\ \mathsf{p}_1\ (\mathbf{ms}\ x))
\end{aligned}
$$

6 Parallel Algorithms and Evaluation

We evaluate our approach using a number of parallel algorithms derived from Alg expressions, and the speedups achieved. The purpose of this is twofold: (i) showing that our approach achieves speedups for an input sequential algorithm, with naïve annotation strategies, and limited optimisations (Fig. 5), and (ii) illustrating the practical value of providing a global type that describes the parallel strategy achieved by a particular annotation strategy (Fig. 4). We run all our experiments on 2 NUMA nodes, 12 cores per node and 62GB of memory, using Intel Xeon CPU E5-2650 v4 @ 2.20GHz chips. We run our experiments first restricting the execution to a single node to avoid NUMA effects, and then on the 2 NUMA nodes.



Mergesort

$$
\begin{aligned}
&\mathsf{p}_0 \to \{\mathsf{p}_1, \mathsf{p}_2\} :\\
&\quad \{\iota_1.\mathsf{p}_0 \to \mathsf{p}_1 : 1 + \mathsf{int}.\\
&\qquad \mathsf{p}_1 \to \mathsf{p}_0 : \mu L.\ \mathsf{end},\\
&\quad\ \iota_2.\mathsf{p}_0 \to \mathsf{p}_1 : \mu L.\ \mathsf{p}_0 \to \mathsf{p}_2 : \mu L.\\
&\qquad \mathsf{p}_2 \to \mathsf{p}_1 : \mu L.\ \mathsf{p}_1 \to \mathsf{p}_0 : \mu L.\\
&\qquad \mathsf{end}
\end{aligned}
$$

FFT

$$
\begin{aligned}
&\mathsf{p}_0 \to \mathsf{p}_1 : \mu L.\ \mathsf{p}_0 \to \mathsf{p}_2 : \mu L.\\
&\mathsf{p}_0 \to \mathsf{p}_3 : \mu L.\ \mathsf{p}_0 \to \mathsf{p}_4 : \mu L.\\
&\mathsf{p}_1 \to \mathsf{p}_3 : \mu L.\ \mathsf{p}_3 \to \mathsf{p}_1 : \mu L.\\
&\mathsf{p}_2 \to \mathsf{p}_4 : \mu L.\ \mathsf{p}_4 \to \mathsf{p}_2 : \mu L.\\
&\mathsf{p}_2 \to \mathsf{p}_1 : \mu L.\ \mathsf{p}_1 \to \mathsf{p}_2 : \mu L.\\
&\mathsf{p}_3 \to \mathsf{p}_4 : \mu L.\ \mathsf{p}_4 \to \mathsf{p}_3 : \mu L.\\
&\mathsf{end}
\end{aligned}
$$

Dot Prod.

$$
\begin{aligned}
&\mathsf{p}_0 \to \mathsf{p}_1 : \mu L.\ \mathsf{p}_0 \to \mathsf{p}_2 : \mu L.\\
&\mathsf{p}_0 \to \mathsf{p}_3 : \mu L.\ \mathsf{p}_2 \to \mathsf{p}_3 : \mathsf{int}.\\
&\mathsf{p}_1 \to \mathsf{p}_3 : \mathsf{int}.\ \mathsf{p}_3 \to \mathsf{p}_0 : \mathsf{int}.\\
&\mathsf{end}
\end{aligned}
$$

Figure 4. Benchmarks: potential parallelisations.

6.1 Benchmarks

Mergesort Mergesort is the usual divide-and-conquer algorithm, using a tree-like parallel reduce.

Cooley-Tukey FFT We use a recursive Cooley-Tukey algorithm. The algorithm starts by splitting the elements of the list into those that are at even and odd positions. Then, it recursively computes the FFT of them, and finally combines the results. To generate a butterfly pattern, we use: products of size n, to store the results of the subsequent interleavings; product associativity to produce a perfect tree; and *asynchronous optimisations*.

Dot Product The dot product algorithm zips the inputs, multiplies them pairwise, and then adds them by folding the result. We use products of size n to derive a scatter-gather.

Additional Algorithms We implemented *scalar prod*, that recursively splits a matrix into sub-matrices, distributes them to different workers, and then multiplies their elements by a scalar, and *quicksort*, with a divide-and-conquer structure.

6.2 Evaluation

We translate Mp monadic actions to C using `pthreads` and shared buffers for communication, and we have a preliminary compilation of the first-order sequential terms to C. We compile the generated C code using gcc version 4.8.5. We take the average of 50 repetitions for each benchmark. Our benchmarks achieve reasonable speedups against the sequential C implementations. Fig. 5 presents the speedups

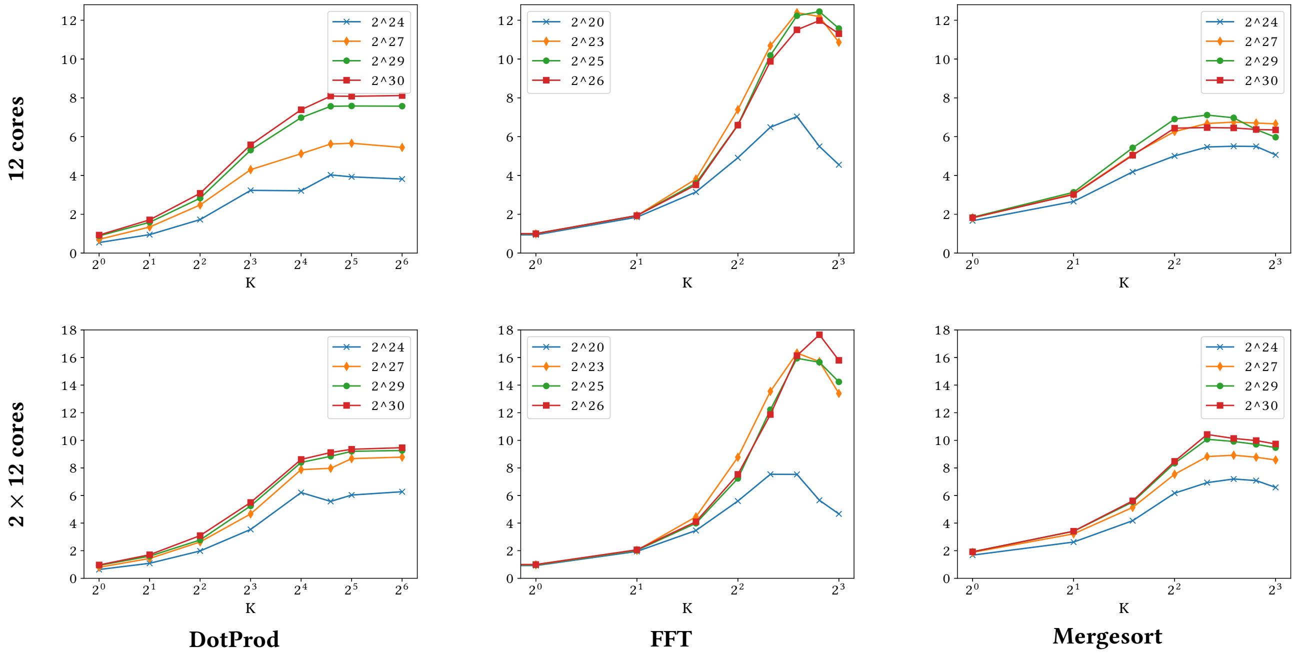


Figure 5. Benchmark speedups, run in 2 NUMA nodes with 12 cores each. The X-axis is the number of workers of the parallel program generated from a set of annotations and recursion unrolling. We show the results for 4 different input sizes.

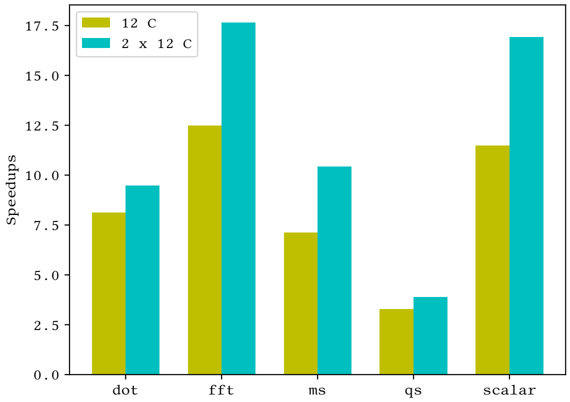


Figure 6. Achieved speedups

against the number of participants for different input sizes, and Fig. 6 present a summary of our speedups for large inputs of size $> 10^9$. We show below an analysis of these results, by plotting the speedups against two factors: 1. the number of participants (threads) produced by a particular annotation and recursion unrolling, named K; and 2. the input size, e.g. number of elements in the input list.

Increasing the number of threads (parameter K), increases the speedups obtained, up to a certain value that depends on the amount of available cores and the input size. For benchmarks that work better with dynamic task creation, our tool does not currently achieve good performance (e.g. quicksort). For FFT, our tool produces the usual butterfly pattern from a straightforward recursive definition, that we can achieve a speedup of 12 when running on a single shared-memory node. The rest of the examples are limited either by *Amdahl's law* (justified by their global types in Fig. 4), or by the overhead of the communication and pthread creation with respect to the cost of the computations, but still achieve speedups of up to 7 and 8 on 12 cores. We can observe that there is a slow down after creating a much larger number of participants than the ones required. This usually depends on how evenly we can distribute the data amongst workers, and whether the amount of workers can be evenly scheduled to different cores. We observe that we can achieve further speedups when running our benchmarks in the 2 NUMA nodes. Overall, we observe that our annotation strategies enable good speedups over the sequential implementation, with relatively little effort. Global types can be used to detect optimisation opportunities that yield efficient parallelisations, such as the Butterfly topology in Fig. 4. Without message-reordering based on the session types, FFT participant p_3 would need to wait for p_1's message before sending its part to p_1, i.e. p_3's local type would be $\mathrm{p}_1?(\mu L).\ \mathrm{p}_1!\langle \mu L \rangle \ldots$. This means that p_3's local computation would only become available to p_1 after it p_1 finishes its own local computation, thus sequentialising the code. Asynchronous permutations [16; 56] allow us to *permute* such actions, and still have communication safety, i.e. $\mathrm{p}_1!\langle \mu L \rangle.\ \mathrm{p}_1?(\mu L) \ldots$. Global types capture the structure of the parallelisation, which can in some cases be used to justify the achieved speedups. For example, we can observe that the mergesort global type contains a part that needs to happen sequentially (p_0 and the last merging point in p_1), and this will prevent us from achieving linear speedups.

7 Related Work

López et al. [49] develop a verification framework for MPI/C inspired by MPST by translating parameterised protocol specifications to protocols in VCC [19]. They focus on verification, not on code or protocol generation. Ng et al. [58; 59] use parameterised MPST [25] to generate an MPI backbone in C that encapsulates the *whole* protocol (i.e., every endpoint), and merges it with user-supplied computation kernels. Several authors (e.g. [10]) generate skeleton API from extensions of Scribble (www.scribble.org). Their approach requires the protocol to be specified beforehand, and it is not extracted from sequential code. Unlike ours, none of the above work formally defines code generation or proves its correctness.

Structured parallelism includes the use of high-level constructs in languages with implicit/data parallelism [5; 12–15; 45; 63], algorithmic skeleton APIs [1; 18; 20; 35; 47], and DSLs/APIs that compile to parallel code [8; 11; 28; 62; 68]. Besides safety, such approaches are often highly optimised. However, most rely on using a fixed, predetermined range of patterns, typically by design with respect to their application domains. By contrast, our work only relies on send/receive operations, which makes it highly portable, and can be easily extended to support further parallel structures by extending the annotation strategies. Optimisations for structured parallel approaches also require to study and define a set of equivalences between patterns [6; 7; 40]. In contrast, our approach does not require the definition of new sets of equivalences, since these are derived from program equivalences. *Lift* is a new language for portable parallel code generation, based on a small set of expressive parallel primitives [39; 66; 67]. Currently, their backend focuses on generating high-performance OpenCL code, while our approach focuses on placing computations on different participants of a concurrent/distributed system. Both approaches could be combined: annotations can be used to generate a high-level message-passing layer that distributes tasks to multiple nodes in a GPU cluster, using the global type to minimise communication costs; then, the code at each participant can be compiled to high-performance GPU code using Lift.

Elliott exploits the idea of giving functional programs multiple interpretations in different categories, and shows examples of applications to multiple domains, including parallelism [29; 30]. Our approach is similar in the sense that we allow the specifications of first-order functional programs to have multiple different interpretations, but we focus on generating parallel code, and provide a finer-grained control over the parallelisations by adding participant annotations. There is a large body of literature in using program equivalences to derive parallel implementations, e.g. [17; 32; 36; 38; 48; 50; 54; 55; 64; 65]. Our framework is orthogonal, in that we focus on tying a low-level C back-end with global types. Our front-end, however, supports some basic form of rewritings, and we plan to extend it in the future with more interesting ones from the literature.

8 Conclusions and Future Work

We have presented a novel approach to protocol inference and code generation. By using this approach, we can reason about *extensionality* of the parallel programs, and alternative mappings of computations to *participants*. We produce the parallel program global type, i.e. its communication protocol, that acts as a contract for the low-level code, can be used to pin-point potential optimisations, or assessing the suitability of a parallelisation. This approach has several benefits: 1. our message-passing code is deadlock-free by construction, since it follows the data-flow of the program, and the optimisations must respect the global type; 2. we prove that our parallelisations are extensionally equivalent to the input function. Additionally, PAlg code could be used for further multiple purposes, such as parallel GPU/FPGA code generation, by combining our approach with other state of the art code generation techniques. We will study this for future work.

Though our approach can already generate representative parallel protocols, our framework is extensible. E.g. we can extend our framework with dynamic participants to handle dynamic task generation [26], and we plan to use this to capture a wider range of communication patterns for parallel computing, such as load-balancing or work-stealing. We plan to study the extension of our back-end to heterogeneous architectures, e.g. GPU clusters, or FPGAs. Our prototype generates code that can achieve speedups against sequential implementations, the optimisations that we support are very basic, and our generated code can be very large. We plan to introduce optimisations that reduce the amount messages exchanged, further message reorderings guided by the global type, and optimisations of the size of the generated code. Finally, we plan to study the instrumentation of global types to estimate statically the speedups of different parallelisations, and optimise communication costs.

Acknowledgements

We thank Shuhao Zhang for his contributions to the C back-end, described in [69]. We thank Francisco Ferreira for the helpful discussions in the early stages of this work. This work was supported in part by EPSRC projects EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, and EP/T006544/1.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. 2011. Accelerating Code on Multicores with FastFlow. In *Proc. of 17th International Euro-Par Conference (Euro-Par 2011) (LNCS)*, Vol. 6853. Springer, 170–181.

[2] John Backus. 1978. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (Aug. 1978), 613–641.

[3] Richard Bird and Oege De Moor. 1996. The algebra of programming. In *NATO ASI DPD*. 167–203.

[4] R. S. Bird. 1989. Algebraic Identities for Program Calculation. *Comput. J.* 32, 2 (01 1989), 122–126.

[5] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[6] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. 2014. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming* 42, 4 (01 Aug 2014), 564–582.

[7] Christopher Brown, Kevin Hammond, Marco Danelutto, and Peter Kilpatrick. 2012. A Language-independent Parallel Refactoring Framework. In *Proc. of the 5th Workshop on Refactoring Tools (WRT '12)*. ACM, New York, NY, USA, 54–58.

[8] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proc. of 2011 Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 89–100.

[9] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (2009), 509–543.

[10] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role Parametric Session Types in Go. In *46th Symp. on Principles of Programming Languages (POPL'19)*, Vol. 3. ACM, 29:1–29:30.

[11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-Specific Approach to Heterogeneous Parallelism. In *Proc. of the 16th Symp. on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, 35–46.

[12] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: a Status Report. In *Proc. of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, (DAMP'07)* (2007). ACM, 10–18.

[13] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *IJHPCA* 21, 3 (2007), 291–312.

[14] Rohit Chandra. 2001. *Parallel Programming in OpenMP*. Morgan Kaufmann.

[15] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proc. of the 20th Conf. on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA05)*. ACM, San Diego, CA, USA, 519–538.

[16] Tzu-chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science* Volume 13, Issue 2 (June 2017).

[17] Yun-Yan Chi and Shin-Cheng Mu. 2011. Constructing List Homomorphisms from Proofs. In *Proc. APLIAS '11: Asian Symposium on Programming Languages & Systems*. 74–88.

[18] Philipp Ciechanowicz and Herbert Kuchen. 2010. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *12th Intl. Conf. on High Performance Computing and Communications, (HPCC'10)*. IEEE, 108–113.

[19] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics, (TPHOLs 2009) (LNCS)*, Vol. 5674. Springer, 23–42.

[20] Murray Cole. 1988. *Algorithmic skeletons : a structured approach to the management of parallel computation*. Ph.D. Dissertation. University of Edinburgh, UK.

[21] James W Cooley and John W Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of computation* 19, 90 (1965), 297–301.

[22] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming (LNCS)*, Vol. 9104. Springer, 146–178.

[23] Alcino Cunha, Jorge Sousa Pinto, and José Proença. 2006. A Framework for Point-Free Program Transformation. In *Proc. of 17th Intl. Workshop on Implementation and Application of Functional Languages (IFL 2005)*. Springer, 1–18.

[24] Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR 2012 – Concurrency Theory*. Springer, 272–286.

[25] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* 8, 4 (2012).

[26] Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic Multirole Session Types *(POPL'11)*. ACM, New York, NY, USA, 435–446.

[27] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming (LNCS)*, Vol. 7966. Springer, 174–186.

[28] Zach DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: a Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011*. ACM, 9:1–9:12.

[29] Conal Elliott. 2017. Compiling to Categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 27 (Aug. 2017), 27 pages.

[30] Conal Elliott. 2017. Generic functional parallel algorithms: Scan and FFT. *Proc. ACM Program. Lang.* 1, ICFP, Article 48 (Sept. 2017), 24 pages.

[31] Maarten M. Fokkinga and Erik Meijer. 1991. *Program Calculation Properties of Continuous Algebras*. Number CS-R91 in Report / Department of Computer Science. CWI.

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

[33] Jeremy Gibbons. 2002. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer, Chapter 5, 151–203.

[34] Jeremy Gibbons. 2007. Datatype-Generic Programming. In *Datatype-Generic Programming*. Springer, 1–71.

[35] Horacio González-Vélez and Mario Leyton. 2010. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw., Pract. Exper.* 40, 12 (2010), 1135–1160.

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

[37] Sergei Gorlatch. 2004. Send-receive Considered Harmful: Myths and Realities of Message Passing. *ACM Trans. Program. Lang. Syst.* 26, 1 (Jan. 2004), 47–56.

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

[39] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proc. of the 2018 Intl. Symp. on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 100–112.

[40] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. 2013. *The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems*. Springer, 218–236.

[41] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proc. of 35th Symp. on Princ. of Prog. Lang. (POPL '08)*. ACM, New York, NY, USA, 273–284.

[42] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67.

[43] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. 1996. Deriving Structural Hylomorphisms from Recursive Definitions. In *Proc. ICFP '96: ACM Int. Conf. on Functional Programming (ICFP '96)*. ACM, New York, NY, USA, 73–82.

[44] John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1 (2000), 67 – 111.

[45] Guy L. Steele Jr. 2005. Parallel Programming and Parallel Abstractions in Fortress. In *14th International Conference on Parallel Architecture and Compilation Techniques (PACT 2005), 17-21 September 2005, St. Louis, MO, USA*. IEEE Computer Society, 157.

[46] Daniel J. Lehmann and Michael B. Smyth. 1981. Algebraic Specification of Data Types: A Synthetic Approach. *Mathematical Systems Theory* 14, 1 (01 Dec 1981), 97–139.

[47] Mario Leyton and José M. Piquer. 2010. Skandium: Multi-core Programming with Algorithmic Skeletons. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*. IEEE Computer Society, 289–296.

[48] Yu Liu, Zhenjiang Hu, and Kiminori Matsuzaki. 2011. Towards Systematic Parallel Programming over Mapreduce. In *Proc. Euro-Par 2011: European Conference on Parallelism*. 39–50.

[49] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-Based Verification of Message-Passing Parallel Programs. In *Proc. of the 2015 Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, 280–298.

[50] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. 2017. Calculating Parallel Programs in Coq Using List Homomorphisms. *International Journal of Parallel Programming* 45, 2 (01 Apr 2017), 300–319.

[51] Michael McCool, Arch Robison, and James Reinders. 2012. *Structured parallel programming: patterns for efficient computation*. Elsevier.

[52] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture*. Springer, 124–144.

[53] Erik Meijer and Graham Hutton. 1995. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, New York, NY, USA, 324–333.

[54] Akimasa Morihata. 2013. A Short Cut to Parallelization Theorems. In *Proc. ICFP 2013: 18th Int. Conf. on Functional Programming*. 245–256.

[55] Akimasa Morihata and Kiminori Matsuzaki. 2010. Automatic Parallelization of Recursive Functions using Quantifier Elimination. In *Proc. FLOPS '10: Functional and Logic Programming*. 321–336.

[56] Dimitris Mostrous and Nobuko Yoshida. 2009. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In *Proc. of the 9th Intl. Conf on Typed Lambda Calculi and Applications (LNCS)*, Vol. 5608. Springer, 203–218.

[57] Matthias Neubauer and Peter Thiemann. 2004. An Implementation of Session Types. In *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*. 56–70.

[58] Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. 2015. Protocols by Default - Safe MPI Code Generation Based on Session Types. In *Proc. of the 24th Intl. Conf. on Compiler Construction (LNCS)*, Vol. 9031. Springer, 212–232.

[59] Nicholas Ng and Nobuko Yoshida. 2015. Pabble: parameterised Scribble. *Service Oriented Computing and Applications* 9, 3-4 (2015), 269–284.

[60] Ross Paterson. 2018. arrows: Arrow classes and transformers. http://hackage.haskell.org/package/arrows-0.4.4.2.

[61] Fethi A. Rabhi and Sergei Gorlatch (Eds.). 2003. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer.

[62] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. of Conf. on Programming Language Design and Implementation, PLDI '13*. ACM, 519–530.

[63] James Reinders. 2007. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly.

[64] Rodrigo C. O. Rocha, Luís Fabrício Wanderley Góes, and Fernando Magno Quintão Pereira. 2016. An Algebraic Framework for Parallelizing Recurrence in Functional Programming. In *Proc. of the 20th Brazilian Symposium on Programming Languages, SBLP 2016 (LNCS)*, Vol. 9889. Springer, 140–155.

[65] David B Skillicorn. 1993. *The Bird-Meertens Formalism as a Parallel Model*. Springer.

[66] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules. In *Proc ICFP 2015: 20th ACM Conf. on Functional Prog. Lang. and Comp. Arch.* 205–217.

[67] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. ACM, 74–85.

[68] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proc. of the 28th Intl. Conf. on Machine Learning (ICML'11)*. Omnipress, 609–616.

[69] Shuhao Zhang. 2019. *Session Arrows: A Session-Type Based Framework For Parallel Code Generation*. Master's thesis. Imperial College London.