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

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
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](image)

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.

```hs
1  msort :: (CVal a, CAlg f) => Int -> f [a] [a]
2  msort n = fix n $ \ms x -> vlet (vsize x) $ \sz ->
3      if sz <= 1 then x
4  else vlet (sz / 2) $ \sz2 ->
5      vlet (par ms $ vtake sz2 x) $ \xl ->
6      vlet (par ms $ vdrop sz2 x) $ \xr ->
7      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:

![Protocol](image)

This is a depth 1 divide-and-conquer, where p₁ divides the task, sends the sub-tasks to p₂ and p₃, and combines the results. If the input is small, p₁ 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 combinatorics. 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 (\(\lambda\)), 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 \texttt{msort} in §1, but that builds internally a point-free representation.

2.1 Syntax

\[
F_1, F_2 ::= I \mid Ka \mid F_1 + F_2 \mid F_1 \times F_2
\]

\[
a, b ::= 1 \mid \text{int} \mid \ldots \mid a \rightarrow b \mid a + b \mid a \times b \mid F a \mid \mu F
\]

\[
e_1, e_2 ::= f \mid \text{const} e \mid \text{id} F e_1 \circ e_2 \mid \pi_1 \mid e_1 \Delta e_2 \mid i_1 \mid e_1 \triangledown e_2
\]

In our syntax, \(F_1, F_2, \ldots\) capture atomic functions, which are functions of which we only know their types; \(e_1, e_2\) are values of primitive types (e.g. integer and boolean); \(e_1, e_2, \ldots\) represent expressions; \(F_1, F_2, \ldots\) are functions; 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 \(\pi_i\) and \(\Delta\), and they represent, respectively, the projections, and the split operation: \((f \Delta g)(x) = (f x, g x)\). Coproducts have two constructors, the injections \(i_1, i_2\), that build values of type \(a + b\). The \(\lor\) combinator is the case operation: \((f_1 \lor f_2)(i_1 x) = f_1 x\). Products and coproducts can be generalised to multiple arguments: \(a \times c\) is isomorphic to \(a \times (b \times c)\), and to \((a \times b) \times c\). We use \(\Pi_{i \in [1,n]} a_i\) as notation for the product of more than two types; similarly we use \(\Sigma\) for coproducts. The \(\Pi\) notation binds tighter than any other constructor. Whenever \(\forall i, j \in I, a_i = a_j = a\), we use the notation \(\Pi_{i \in [1,n]} a\) as a synonym for \(\Pi_{i \in [1,n]} a_i\).

Functions 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; \(Kb\) 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 \(a\) to a coproduct type. A term \(F e\) behaves as mapping term \(e\) to the \(l\) positions in \(F\). For example, if \(F = Ka \times I \times I\), then applying \(F e\) to \((x, y, z)\) yields \((x, e y, e z)\).

Recursion is captured by combinators in, out, rec, and type \(\mu F\). We use standard isorecursive types [31; 46; 52], where \(\mu F\) is isomorphic to \(F \mu F\), and the isomorphism is given by the combinators \(\text{in}_F\) (roll) and \(\text{out}_F\) (unroll). For any polynomial functor \(F, \mu F\), and strict functions \(\text{in}_F\) and \(\text{out}_F\) are guaranteed to exist. In our implementation, \(\text{in}_F\) is just a constructor (like \(\text{inj}_I\)). Recursion is \(\text{rec}_F e_1 e_2\), 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 \(a : (a \rightarrow b) \times a \rightarrow b\) allow the definition of a general fix-point 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 \(T = K (Ls) + I \times I\) captures the recursive structure of \(ms : Ls \rightarrow Ls\). When splitting some \(l : Ls\), we may find one of the two cases described by \(T\): an empty or singleton list, \(Ls\), or a list of size \(\geq 2\), that can be split in two halves \(Ls \times Ls\). Assume that a functions \(spl : Ls \rightarrow T Ls\), and a function \(mrg : T Ls \rightarrow Ls\). We define \(ms = \text{rec}_T mrg\ spl\). By the definition of \(rec\):

\[
ms = \text{rec}_T (id \downarrow mrg) \circ (id \downarrow mrg) \circ T (\text{rec}_T mrg \ spl) \circ spl
\]

Function \(ms\) first applies \(spl\). Then, if the list was empty or singleton, it returns the input unmodified. Otherwise, \(mrg\) 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 \(\text{Alg}\), a point-free functional language. In this section, we extend this language with \textit{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 \(\text{PALg}\).

The implicit \textit{dataflow} of the \(\text{Alg}\) (or \(\text{PALg}\)) 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 \(n\) FFT of size \(n\), \(fft n\), can be described as the combination of two FFTs of size \(n/2\). We focus its high-level structure:

\[
(\text{add} \circ \text{sub} \circ \text{p2}) \circ ((\text{fft} n/2 \circ \text{p3} \circ \pi_1) \Delta ((\exp \circ \text{fft} n/2) \circ \text{p4} \circ \pi_2))
\]
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 $\text{fft}$ of size $n/2$ to the even and odd elements at $p_1$ and $p_2$ 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 $(\text{vec} \times \text{vec})@((p \times p'))$, 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 $(\text{vec} \times \text{vec})@((p_1 \times p_2))$, which is the output (or codomain) interface the $\text{PAAlg}$ expression. We also refer to the annotations (e.g. $p_1 \times p_2$) as interfaces, whenever there is no ambiguity. We write $\text{fft}_{n_1} : ((\text{vec} \times \text{vec})@((p \times p'))) 
rightarrow (\text{vec} \times \text{vec})@((p_1 \times p_2))$ to represent the input and output interfaces of $\text{fft}_{n_1}$.

Consider now $e_1@p_1 \lor e_2@p_2$. The output interface of this expression is either $p_1$ or $p_2$, depending on whether the input is the result of applying $i_1$ or $i_2$. We represent such interfaces using unions: $e_1@p_1 \lor e_2@p_2 : (a + b)@p \nrightarrow c@((p_1 \lor p_2))$. 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 $e_0@p_0 \circ (e_1@p_1 \lor e_2@p_2)$. Assuming that the input at $p$, $p_0$, 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 $\text{PAAlg}$.

### 3.1 Syntax of $\text{PAAlg}$

$I := p \mid i_1 I \mid I \times I \quad R := I \mid R \cup \bar{R} \quad P := R \nrightarrow R \quad e := e@p \mid [p \oplus \bar{p}] \mid id \mid ee \mid \pi_i \mid \Delta e \mid i_1 \mid e \lor e$

The syntax of $\text{PAAlg}$ is that of Alg, extended with participant annotations (red). Note that certain Alg constructs can only occur under annotations ($e@p$), e.g. $\text{in}$, $\text{out}$ and $\text{rec}$. 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 $\text{PAAlg}$: annotations, interfaces, and annotated functions.

**Annotations** are ranged over by $R, R' \ldots$ 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 $i, I$, that keep track of the branch of the choice that led to $I$. A choice $R_1 \cup \bar{R}_2$ 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, $\bar{p} = p_1 \cdots p_n$ are the participants whose behaviour depends on the path in the control flow. Finally, arrows $P$ of the form $R_1 \nrightarrow R_2$ represent the input/output annotations of a parallel program.

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

**Annotated functions** are ranged over by $e, e'$. The annotations are introduced using $e@p$, 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: $[p \oplus \bar{p}]$. This annotation specifies that $p$ performs a choice, and notifies $\bar{p}$. Choice points can be points 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: $V := [p \mapsto v]_{p \in p}$. The programmer, additionally to writing an Alg ($\text{PAAlg}$) expression, will need to provide an input interface, i.e. where is the input to the parallel program. Consider, for example, the interface $\text{int}@p_1$. Given a concurrent system with participants $p_0 \cdots p_n$, we know that $p_1$ contains a value of type $\text{int}$: $[\cdots p_1 \mapsto 42 \cdots]$. An interface with a product of participants $(a \times b)@((p_1 \times p_2))$ 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 $(\text{int} \times \text{vec})@((p_1 \times p_2))$: $[\cdots p_1 \mapsto 42 \cdots p_2 \mapsto [1, 1, 2, \ldots]]$. An interface $i, I$ 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 $a@((R_1 \cup \bar{R}_2))$ means that the state might be either $R_1$ or $R_2$, and that all participants $\bar{p}$ should be notified of the state.

**Well-formedness** The above examples are of well-formed interfaces: $\text{int}@p_1$, $(\text{int} \times \text{vec})@((p_1 \times p_2))$. Well-formedness ensures that interfaces represent valid states. Generally, $a@R$ is well-formed if $a$ matches the structure of $R$. For example, $\text{int}@((p_1 \times p_2))$ is ill-formed, since a single integer cannot be at two different participants. An interface $a@((R_1 \cup \bar{R}_2))$ requires that both $a@R_1$ and $a@R_2$ are well-formed. So, $(\text{vec} \times \text{vec})@((p_1 \times p_2) \cup p_3)$ is well-formed because we can have $\text{vec}@p_1$ and $\text{vec}@p_2$, or $(\text{vec} \times \text{vec})@p_3$. However, $\text{int}@((p_1 \times p_2) \cup p_3)$ is ill-formed, because $\text{int}@((p_1 \times p_2))$ 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 ⊢ e ⇒ e : A → 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 ⊢ e ⇒ e : a@R_a → b@R_b is that the participants in e collectively apply computation e to the value of type a distributed across R_a and produce a value of type b distributed across R_b. We sometimes omit e and write ⊢ e : A → B. We ensure that given any e and e such that they are typeable against interfaces a@R_a → b@R_b, then e must have type a → b.

Lemma 3.1. If e ⇒ e : a@R_a → b@R_b, then e : a → 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{[1], p} ∪ I{[2], p}, with the constraint that the participants in I{[p]} must be in 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 pids(e) ⊆ 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@p is a valid parallelisation, with output interface b@p. Finally, rule Ext is crucial for exploring potential parallelisations. It states that if e ⇒ e : A → ? and e : a@R → B holds, then e : a → B.

Example 3.2 (Mergesort). Consider the mergesort definition ms = rec_T mrg spl. Solutions to the inference problem ⊢ ms ⇒ ?0 : Ls@p_0 → ?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:

\[
\vdash (id ∨ (mrg@p_0 ∘ (ms@p_2 ∘ π_1@p_1) ∩ (ms@p_3 ∘ π_2@p_1))) ∩ [p_1 @ p_1, p_2] ∩ spl@p_1 @ Ls@p_0 \rightarrow Ls@p_1 ∪ p_1@p_2 @ Ls@p_1
\]

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:

\[
(id ∨ (mrg@p_1 ∘ (ms@p_2 ∘ π_1@p_1) ∩ (ms@p_3 ∘ π_2@p_1)))) ∩ [p_1 @ p_1, p_2] ∩ spl@p_1 @ Ls@p_0 \rightarrow Ls@p_1 ∪ p_1@p_2 @ Ls@p_1
\]
Notice that this example will run correctly, and produce the expected result. However, the achieved PAlog 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:

\[
\begin{align*}
p_0 &\rightarrow p_1 : Ls. p_1 \rightarrow p_2 \langle i_1. \end{end} \& \\
i_2. p_1 &\rightarrow p_2 : Ls. p_1 \rightarrow p_2 : Ls. p_2 \rightarrow p_1 : Ls \times Ls. \end{end}
\]

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 \(i_1\) or \(i_2\), and if the label is \(i_1\), the protocol ends; 3. if \(p_1\) sent \(i_2\), 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:

\[
\begin{align*}
p_0 &\rightarrow p_1 : Ls. p_1 \rightarrow \{p_1 p_3\} \langle i_1. \end{end} \& \\
i_2. p_1 &\rightarrow p_2 : Ls. p_1 \rightarrow p_3 : Ls. p_2 \rightarrow p_1 : Ls. p_3 \rightarrow p_1 : Ls. \end{end}
\]

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 \(p_1 \rightarrow p_2. p_2 \rightarrow p_3\), 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, \(p_1, p_2, \ldots\), and a set of labels, \(i_1, i_2, \ldots\). 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:

\[
\begin{align*}
G &::= p_1 \rightarrow p_2 : a.G | p_1 \rightarrow \{p_j\}_{j\in[2,n]} : \{i_l, L_l\}_{l\in L} \\
&\quad | \mu X.G \mid X \end{end} \& \\
L &::= p!(a) L | p?(a) L \mid p \& \{i_l, L_l\}_{l\in L} \cup \{p_j\}_{j\in[2,n]} \cup \{i_l, L_l\}_{l\in L} \\
&\quad | p.X.L \mid X \end{end} \& \\
G &::= p \rightarrow p_1 : a.G \mid p \rightarrow \{p_j\}_{j\in[2,n]} : \{i_l, L_l\}_{l\in L}
\]

Global type \(p_1 \rightarrow p_2 : a.G\) denotes data interactions from \(p_1\) to \(p_2\) with value of type \(a\). Branching is represented by \(p_1 \rightarrow \{p_j\}_{j\in[2,n]} : \{i_l, L_l\}_{l\in L}\) with actions \(i_l\) from \(p_1\) to all

4.2 Protocol Relation

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

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

$$\vdash (id \cup (\text{merge@}p_1 \circ (\text{ms}@p_2 \circ \pi_1@p_1) \triangle (\text{ms}@p_3 \circ \pi_2@p_1)) \circ [p_1 \oplus p_2,p_3] \circ \text{split@}p_1 \equiv Ls@p_1 \sim ?.0) \rightarrow Ls\oplus_1.$$  

$$\begin{array}{ll}
  p_1 \rightarrow \{p_2,p_3\} & f_1 \cdot \text{end};
  f_2 \cdot p_2 \rightarrow \text{Ls};
  f_3 \cdot p_3 \rightarrow \text{Ls};
  (f_1 \cup (f_2 \rightarrow f_1 : f_1 \rightarrow f_1 : f_2 \rightarrow f_1 : \text{Ls.end})).
\end{array}$$

$$= p_1 \rightarrow \{p_2,p_3\}$$

4.3 Correctness

We guarantee that for $e \iff e : A \rightarrow B$, 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 $WF(A)$, if $e : A \rightarrow B$, then there exists $G$ s.t. $G \equiv e \iff A \approx G$.

**Lemma 4.3.** [Protocol Deadlock-Freedom] For all $WF(A)$, if $e : A \rightarrow B$ and $e \equiv A \approx G$, then $WF(G)$.

**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 PAig 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 with 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:

$$v \equiv x \mid (v, v) \mid i_1 v \mid \cdots \mid e v$$

$$m_i \equiv \text{ret } v \mid \text{send } p v m \mid \text{recv } v a f \mid \text{sel } p v f_1 f_2 \mid \text{brn } p m_1 m_2 \mid \text{case } f_1 f_2 f \equiv \lambda x.m$$

Values $v$ are either primitive values, tagged values $i_1 v, p$ of values, or the result of applying an Alg expression $e$ to a value. We use standard notation for the monadic unit (ret), bind ($\triangleright=$) and Kleisli composition: $f_1 \triangleright=f_2 = \lambda x. f_1 x \triangleright f_2$. 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 $m : Mp L a$, 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, $\otimes i$, that is isomorphic to sum types, but that can only be constructed and eliminated by using the corresponding monadic constructions:

$$\begin{array}{ll}
  \text{sel } p : a + b & \rightarrow (a \rightarrow Mp L_1 c_1) \rightarrow (b \rightarrow Mp L_2 c_2) \rightarrow Mp (p @ \{a_1 L_1 ; b_2 L_2\}) (c_1 \otimes c_2) \\
  \text{brn } p : Mp L_1 a_1 & \rightarrow Mp L_2 a_2 \rightarrow Mp (p \& \{a_1 ; a_2\}) (c_1 \otimes c_2) \\
  \text{case } (a \rightarrow Mp L_1 c) & \rightarrow (b \rightarrow Mp L_2 d) \rightarrow a \otimes b \rightarrow Mp (L_1 \cup L_2) (c \otimes d)
\end{array}$$

These constructs ensure that the tag used to build $a \otimes b$ 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 $\otimes$ 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 PAig in a bottom-up way by manipulating _parallel programs_. Parallel programs are mappings from participants to their monadic action: $E := [p_i \mapsto m_{i}]_{i \in I}$. If $m_{i} : Mp L_i a_{i}$ for all $i \in I$, then we write $[p_i \mapsto m_{i}]_{i \in I} : Mp [p_i \mapsto L_i]_{i \in I} [p_i \mapsto a_{i}]_{i \in I}$. 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. $(E, W') \sim^\ell E'$, which is a transition from $E$ to $E'$, and shared queues $W$ to $W'$ with observable action $\ell$. 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 $E : Mp C A, m : Mp L a$ and $W : Q$. Suppose $(E'[r \mapsto m], W') \sim^\ell (E[r \mapsto m'], W')$. Then there exists $(C[r \mapsto I], Q) \sim^\ell (C[r \mapsto I'], Q')$ such that $W' : Q'$ and $m' : Mp L' a$.

**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 $[e](A)$, and guarantee that $[e](A) : A \rightarrow Mp G B$, if $e \equiv A \sim (G, B)$. This means that if $e$ induces protocol $G$ with interfaces $A \rightarrow B$, 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 $e\circ p : a@l \rightarrow B$ requires the definition of the interactions from an interface $I$ that groups a type $a$ at position $p : [a@l \mapsto p] : a@l \rightarrow Mp [a@l \mapsto p] (a@lp)$. The definition is analogous to that of $[a@l \sim p]$. 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] $\text{if } e \equiv A \sim G$, then $[e](A)$ complies with protocol $G$.

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

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

$p_1 \mapsto \lambda x. \text{sel} \{p_2, p_3\} (\text{spl } x) (\lambda x. \text{ret } x) \\
(\lambda x. \text{send } p_2 (\pi_1 x) \mapsto \lambda y. \text{send } p_1 (\pi_2 x) \mapsto \lambda_.) \\
\text{recv } p_2 Ls \mapsto \lambda x. \text{recv } p_3 Ls \mapsto \lambda y. \text{ret } (\text{mg } x, y)) \\
p_{2,3} \mapsto \lambda x. \text{brn } p_1 (\text{ret } x) \text{recv } p_1 Ls \mapsto \lambda x. \text{send } p_1 (\text{ms } x)$

## 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 naive 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.

### 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 asynchronus 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
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 $p_1!(\langle \mu \rangle L)$. This means that $p_3$’s local computation would only become available to $p_1$ after it 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. $p_1!(\langle \mu \rangle L)$. 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 verifica-
tion, 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 end-
point), and merges it with user-supplied computation ker-
nels. 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 con-
structs 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 equiv-
alences, since these are derived from program equivalences.

Lift is a new language for portable parallel code gener-
ation, 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 par-
allelism [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 con-
trol over the parallelisations by adding participant anno-
tations. There is a large body of literature in using pro-
gram equivalences to derive parallel implementations, e.g.
[17; 32; 36; 38; 48; 50; 54; 55; 64; 65]. Our framework is or-
thogonal, 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 C, can be used to
pin-point potential optimisations, or assessing the suitabil-
ity of a parallelisation. This approach has several benefits:
1. our message-passing code is deadlock-free by construc-
tion, 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, PAalg code could be used for further
multiple purposes, such as parallel GPU/FPGA code genera-
tion, 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 paral-
lel 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


