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, PaAlg, which is a first-order language with participant annotations that ensures deadlock-freedom by construction. PaAlg 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. PaAlg 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.

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 [52; 62]. A popular approach to structured parallelism is the use of algorithmic skeletons [20; 36], 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 [38]. 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 [35], and parallel computing [66]. 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. [42] 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 [37], and removes unnecessary synchronisation, while remaining communication-safe.

1.1 Overview

Our framework has three layers: (1) Parallel Algebraic Language (PaAlg), 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

![Figure 1. Overview](image-url)
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.

We prove that the sequential program, and output parallel

Why Multiparty Session Types

Language underlying

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 ($\triangledown$), also known as fanout, or (&&), in the arrow literature [45] and Control.Arrow Haskell package [61]. 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; 44]. 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

\[
F_1, F_2 ::= l \mid K a \mid F_1 + F_2 \mid F_1 \times F_2 \\
a, b ::= \mid \ldots \mid a \rightarrow b \mid a + b \mid a \times b \mid F a \mid \mu F \\
e_1, e_2 ::= l \mid v \mid \mathsf{const} e \mid \mathsf{id} e_1 \circ e_2 \mid \Pi_1 e_1 \triangleleft e_2 \mid \Pi_I e_1 \triangledown e_2 \\
| F e | \mathsf{in}_F | \mathsf{out}_F | \mathsf{rec}_F e_1 e_2
\]
In our syntax, $f_1$, $f_2$, ..., capture atomic functions, which are functions of which we only know their types; $u_1$, $u_2$ are values of primitive types (e.g. integer and boolean); $e_1, e_2, ...$, represent expressions; $F_1$, $F_2$, ..., are functors; and $a, b, ...$, are types. The syntax and semantics are standard [34; 53].

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: $(f \triangle y)(x) = (f, x, g, x)$. Coproducts have two constructors, the injections $i_1, i_2$, that build values of type $a + b$. The $\vee$ combinator is the case operation: $(f_1 \triangledown f_2)(i, x) = f_1 x$. 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 $\Pi$ notation binds tighter than any other construct. Whenever $i, j \in I$, $a_i = a_j = a$, we use the notation $\Pi n a$ as a synonym for $\prod_{i \in [1, n]} a_i$.

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 to a coproduct type. A term $F e$ behaves as mapping term $e$ to the $I$ positions in $F$. For example, if $F = K a \times I$ $I$, 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; 47; 53], where $\mu F$ 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 $F$, $\mu F$, and strict functions $\in F$ and $\out F$ are guaranteed to exist. In our implementation, $\in F$ is just a constructor (like $\in I$), Recursion is $\rec F e_1 e_2$, and it is known as a hylo morphism [53]. A hylo morphism 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 hylo morphisms 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). Hylo morphisms and exponentials $a \mapsto (a \to b) \times a \to b$ allow the definition of a general fixpoint operator [54]. Working with hylo morphisms 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 $L s$ of lists of elements of type $a$. Functor $T = K (L s) \times I \times I$ captures the recursive structure of $\ms : L s \to L s$. When splitting some $l : L s$, we may find one of the two cases described by $T$: an empty or singleton list, $L s$, or a list of size $\ge 2$, that can be split in two halves $L s \times L s$. Assume that a functions $\spl : T L s \to T L s$, and a function $\mrg : T L s \to L s$. We define $\ms = \rec T \mrg \ms$. By the definition of $\rec$:

$$\ms = \rec T (\id \triangledown \mrg) \ms = (\id \triangledown \mrg) \circ T (\rec T \mrg \ms) \circ \ms = (\id \triangledown \mrg) \circ (\id \triangledown (\rec T \mrg \ms) \circ (\rec T \mrg \ms)) \circ \ms = (\id \triangledown \mrg \circ (\ms \times \ms)) \circ \ms$$

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 $P A l g$.

The implicit dataflow of the Alg (or $P A l g$) 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 $n$, $\ff n$, can be described as the combination of two FFTs of size $n/2$. We focus its high-level structure:

$$(\add @ p_1 \triangle \sub @ p_2) \circ ((\ff n / 2 @ p_3 \circ p_1) \circ (\exp @ \ff n / 2 @ p 4 @ p_2))$$

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 $\ff$ 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 $(\vec \times \vec)(p \times p')$, 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 $(\vec \times \vec)(p_1 \times p_2)$, which is the output (or codomain) interface the $P A l g$ expression. We also refer to the annotations (e.g. $p_1 \times p_2$) as interfaces, whenever there is no ambiguity.

We write $\ff n : (\vec \times \vec)(p \times p') \rightarrow (\vec \times \vec)(p_1 \times p_2)$ to represent the input and output interfaces of $\ff n$.

Consider now $e_1 \circ p_1 \triangledown e_2 \circ p_2$. 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: $e_1 \circ p_1 \triangledown e_2 \circ p_2 : (a + b)\circ p \rightarrow c \circ (p_1 \cup p_2)$. 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 $e_0 \oplus p_0 \circ (e_1 \oplus p_1 \lor e_2 \oplus p_2)$. 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 $PAlg$

$$I ::= p \mid i_1 I \mid I \times I \quad R ::= I \mid R \uplus R \quad P ::= R \rightarrow R$$
$$e ::= e \circ p \mid [p \oplus \tilde{p}] \mid id \mid e \odot e \mid \pi_i \mid e \triangle e \mid i_1 \mid e \triangledown e$$

The syntax of $PAlg$ is that of Alg, extended with participant annotations (red). Note that certain $Alg$ constructs can only occur under annotations ($e \circ p$), 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 $i_1 I$, that keep track of the branch of the choice that led to $I$. A choice $R_1 \uplus R_2$ 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, $\tilde{p} = p_1 \ldots p_n$ are the participants whose behaviour depends on the path in the control flow. Finally, arrows $P$ of the form $R_1 \rightarrow R_2$ represent the input/output annotations of a parallel program.

**Interfaces** are annotated types. They range over $A, B, \ldots$, and are of the form $a \circ R$, which means that values of type $a$ are distributed across $R$. We require annotated types to be **well-formed**, $WF(a \circ R)$, 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 $e \circ p$, 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: $[p \oplus \tilde{p}]$. This annotation specifies that $p$ performs a choice, and notifies $\tilde{p}$. 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: $V \equiv [p \mapsto v]_{p \in P}$. 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 int$\text{\texttt{p}}_1$.

Given a concurrent system with participants $p_0 \circ \ldots \circ p_n$, we know that $p_1$ contains a value of type 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 $\text{int} \times \text{vec}((p_1 \times p_2))$ is: $[\cdots p_1 \mapsto 42 \cdots p_2 \mapsto [1, 2, \ldots]]$. An interface $i_1 I$ 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 $a \circ (R_1 \uplus R_2)$ means that the state might be either $R_1$ or $R_2$, and that all participants $\tilde{p}$ should be notified of the state.

#### Well-formedness

The above examples are of well-formed interfaces: $\text{int} \circ \tilde{p}_1$, $(\text{int} \times \text{vec}) \circ (p_1 \times p_2)$. Well-formedness ensures that interfaces represent valid states. Generally, $a \circ R$ is well-formed if $a$ matches the structure of $R$. For example, int$\times(p_1 \times p_2)$ is ill-formed, since a single integer cannot be at two different participants. An interface $a \circ (R_1 \cup R_2)$ requires that both $a \circ R_1$ and $a \circ R_2$ are well-formed. So, $(\text{vec} \times \text{vec}) \circ (p_1 \times p_2 \cup p_3)$ is well-formed because we can have vec$\circ p_1$ and vec$\circ p_2$, or $(\text{vec} \times \text{vec}) \circ p_3$. However, int$\circ (p_1 \times p_2 \cup p_3)$ is ill-formed, because int$\circ (p_1 \times p_2)$ 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 $\vdash e \Rightarrow e : A \rightarrow B$ 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 $\vdash e \Rightarrow e : a \circ R_1 \rightarrow b \circ R_0$ is that the participants in $e$ collectivelly 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 $\vdash e : A \rightarrow B$. We ensure that given any $e$ and $e$ such that they are typeable against interfaces $a \circ R_1 \rightarrow b \circ R_0$, then $e$ must have type $a \rightarrow b$.

**Lemma 3.1.** If $\vdash e \Rightarrow e : a \circ R_a \rightarrow b \circ R_b$, then $e : a \rightarrow b$.

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 \( J[I_p] \cup \Pi[J[I_p]] \), with the constraint that the participants in \( J[p] \) must be in \( \bar{p} \). 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(\( e \)) \( \subseteq \bar{p} \), 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 \( e \) 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 an inference problem (2) below. Let \( ?i \) be metavariables. The two inference problems that we are interested in are: 1. Solving \( \vdash e : A \rightarrow ?0 \) obtains the output interface for \( e \), with input interface \( A \). 2. Solutions of \( \vdash e : ?0 : A \rightarrow ?1 \) are potential parallelisations of \( e \), and their output interface. Solving (1) is straightforward. Problem (2) requires to decide how to introduce role annotations (rule Alt), 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.

Example 3.2 (Mergesort). Consider the mergesort definition \( ms = \text{rec}_\tau \text{mrq} \text{spl} \). Solutions to the inference problem \( \vdash ms \Rightarrow ?0 : \text{ls}@p_0 \rightarrow ?1 \) 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 PA\( \text{Alg} \) expression:

\[
\frac{\vdash e : a \to b}{\vdash e \Rightarrow e@p : a@l \to b@p} \quad \frac{\vdash e_2 \Rightarrow e@l \to B}{\vdash e_1 \Rightarrow e : a@l \to B}
\]

\[
\frac{\vdash e : A_1 \to B_1}{\vdash e \Rightarrow e : A_2 \to B_2 \text{ } A_1 \neq A_2 \text{ } \text{pids}(e) \subseteq \bar{p}}
\]

\[
\frac{\text{pids}(I[p]) \subseteq \bar{p}}{\text{WF}(a@I[I_1(p)], i \in [1, 2]) \Rightarrow \vdash e : a@I[I[p]] \cup \Pi I[I_2(p)] \to B}
\]

Figure 2. Typing rules for PA\( \text{Alg} \) (selected)

4 Multiparty Session Types for PA\( \text{Alg} \)

The dataflow of the PA\( \text{Alg} \) constructs determine the communication protocol of the annotated expression. However, it is hard to manually check what this communication structure is. Recall the mergesort PA\( \text{Alg} \) 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:

\[
(id \vee (\text{ms}@p_1 \circ (\text{ms}@p_2 \circ \text{ms}@p_1) \circ (\text{ms}@p_2 \circ \text{ms}@p_1)))
\]

\[
\circ p_1 \circ p_2 \circ \text{ms}@p_0 \Rightarrow \text{ls}@p_0 \cup \Pi p_1 : \text{ls}@p_0 \cup \Pi p_2 : \text{ls}@p_1,
\]

Notice that this example will run correctly, and produce the expected result. However, the achieved PA\( \text{Alg} \) 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:

\[
p_0 \to p_1 : \text{ls}, p_1 \to p_2 \{t_1, \text{end}\}
\]

\[
t_2, p_1 \to p_2 : \text{ls}, p_1 \to p_2 : \text{ls}, p_2 \to p_1 : \text{ls} \times \text{ls} \text{.end} \}
\]

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 \( t_1 \) or \( t_2 \), and if the label is \( t_1 \), the protocol ends; 3. if \( p_1 \) sent \( t_2 \), 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:

\[
p_0 \to p_1 : \text{ls}, p_1 \to \{p_2, p_3\} \{t_1, \text{end}\}
\]

\[
t_2, p_1 \to p_2 : \text{ls}, p_1 \to p_3 : \text{ls}, p_2 \to p_1 : \text{ls}, p_3 \to p_1 : \text{ls} \text{.end} \}
\]

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 ensuring 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 [57].

### 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:

\[
G \equiv p_1 \rightarrow p_2 : a.G \mid p_1 \rightarrow \{p_j\}_{j \in [2,n]} : \{ G_{i_1} \}_{i_1 \in I} \\
| \mu X.G[ X ] \mid e \end{align}
\]

\[
L \equiv p_1.\langle a \rangle .L \mid p_2.\langle a \rangle .L \mid p \& \{ p \}_{j \in [2,n]} : \{ G_{i_1} \}_{i_1 \in I} \\
| \mu X.G[ X ] \mid e \end{align}
\]

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 \( p_1 \rightarrow \{ p_j \}_{j \in [2,n]} : \{ G_{i_1} \}_{i_1 \in I} \) with actions \( a \) from \( p_1 \) to all \( p_j, j \in [2,n], \) end represents a termination of the protocol. \( L \) represents a recursive protocol, which is equivalent to \( \mu X.G[ X ]G \). We assume recursive types are guarded.

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 \( p_1.\langle a \rangle .L \) expresses the action of sending a value of type \( a \) to \( p \) followed by interactions specified by \( L \). The receive type \( p_2.\langle a \rangle .L \) is the dual, where a value with type \( a \) is received from \( p \). The selection type \( \{ p \}_{j \in [2,n]} : \{ G_{i_1} \}_{i_1 \in I} \) represents the transmission to all \( p_j \) of label \( i_1 \) chosen in the set of labels \( i \in I \) followed by \( L_i \). The branching type is its dual. \( \text{pids}(G) / \text{pids}(L) \) denote the set of participants that occur in \( G \) or \( L \).

### 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 [42]. We write \( G \mid p \) for the projection of \( G \) onto the role of \( p \). We illustrate the projection with an interaction \( p_0 \rightarrow p_1 : a.G \). The projection onto \( p_0 \) is \( p_1.\langle a \rangle .G \mid p_0 \), the projection onto \( p_1 \) is \( p_0.\langle a \rangle .G[ \{ p_1 \} ] \), and the projection onto any other role \( p \) is \( G \mid p \). 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 its projection on all its roles is defined. We denote: \( \text{WF}(G) = \forall p \in \text{pids}(G), \exists L, G \mid p = L \).

### 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 \( G_1 \) \& \( G_2 \) to represent the external choices, i.e. \( G_1 \) are the continuations for both branches of a previous choice that affects \( \bar{p} \). We also extend the local types, and projection rules \( (G_1 \& G_2) = G_1 \mid p \& G_2 \mid p \) and the notion of well-formedness. We say that an external choice is well-formed, \( \text{WF}(G_1 \& G_2) \), if \( \text{WF}(G_1), \text{WF}(G_2) \), 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 \( G \vdash p \iff A \sim (G \cup B) \) 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:

\[
\vdash (\text{id} \cup (\text{merge} @ p_1 \circ (\text{split} @ p_2 \circ \text{split} @ p_1) \& (\text{split} @ p_3 \circ \text{split} @ p_1))) \circ [p_1 @ p_1.p_2.p_3] \circ \text{split} @ p_1 \Leftrightarrow \text{sort} @ p_1 \iff \textbf{70}.
\]

\[
p_1 \rightarrow \{p_2.p_3\} \begin{cases} \text{i1.end}; & \\
\text{i2.p1} \rightarrow p_2 : \text{ls.p1} \rightarrow p_3 : \text{ls.end} & \end{cases}
\]

4.3 Correctness

We guarantee that for \( e \) s.t. \( e \Rightarrow e : A \rightarrow B \), 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 \(WF(A), \text{if } \vdash e : A \rightarrow B, \text{then there exists } G \text{ s.t. } \vdash e \iff A \sim G.\)

Lemma 4.3. [Protocol Deadlock-Freedom] For all \(WF(A), \text{if } \vdash e : A \rightarrow B \text{ and } \vdash e \iff A \sim G, \text{then } WF(G).\)

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; 43].

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 [58]. 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:

\[
\begin{align*}
\nu & ::= x \mid (\nu, \nu) \mid t_1 \nu \mid \cdots \mid e \nu \\
\text{m_1} & ::= \text{ret } \nu \mid \text{send } p \nu m \mid \text{recv } p a f \mid \text{sel } p \varphi_1 \nu f_1 f_2 \\
\text{brn } p \text{ m_1 m_2} & \mid \text{case } f_1 f_2 f & \text{val } := \lambda x . m \\
\end{align*}
\]

Values \(\nu\) are either primitive values, tagged values \(t_i \nu\), pairs of values, or the result of applying an \(ALg\) expression \(e\) to a value. We use standard notation for the monadic unit (\(\text{ret}\)), bind (\(\eta\)) and Kleisli composition: \(f_1 \gg f_2 = \lambda x . f_1 x \gg f_2\).

The message-passing constructs are standard, except \(\text{sel}, \text{brn}\) and \(\text{case}\), which are used for performing choices, and composing actions that depend on the same choice.

Each monadic computation \(f\) or \(m\) has a type \(A : M_p, L, a\), where \(a\) is the return type of \(m\), and \(L\) is the type index of \(M_p\). We use standard notation for the monadic unit (\(\text{ret}\)) and bind (\(\eta\)) to encode the monadic actions.

For composing actions that depend on the same value. We use standard notation for the monadic unit (\(\text{ret}\)) and bind (\(\eta\)) to encode the monadic actions. Values \(\varphi_1\) are either primitive values, tagged values \(t_i \nu\), pairs of values, or the result of applying an \(ALg\) expression \(e\) to a value. We use standard notation for the monadic unit (\(\text{ret}\)), bind (\(\eta\)) and Kleisli composition: \(f_1 \gg f_2 = \lambda x . f_1 x \gg f_2\).

The message-passing constructs are standard, except \(\text{sel}, \text{brn}\) and \(\text{case}\), which are used for performing choices, and composing actions that depend on the same choice.

Each monadic computation \(f\) or \(m\) has a type \(A : M_p, L, a\), where \(a\) is the return type of \(m\), and \(L\) is the type index of \(M_p\). We use standard notation for the monadic unit (\(\text{ret}\)) and bind (\(\eta\)) to encode the monadic actions.

For composing actions that depend on the same value. We use standard notation for the monadic unit (\(\text{ret}\)) and bind (\(\eta\)) to encode the monadic actions. Values \(\varphi_1\) are either primitive values, tagged values \(t_i \nu\), pairs of values, or the result of applying an \(ALg\) expression \(e\) to a value. We use standard notation for the monadic unit (\(\text{ret}\)), bind (\(\eta\)) and Kleisli composition: \(f_1 \gg f_2 = \lambda x . f_1 x \gg f_2\).

The message-passing constructs are standard, except \(\text{sel}, \text{brn}\) and \(\text{case}\), which are used for performing choices, and composing actions that depend on the same choice.
is analogous to that of $a@I \sim p$. 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] If $e = A \sim G$, then $\llbracket e \rrbracket (A)$ 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 $\llbracket e \rrbracket (p)$ 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:

\begin{align*}
    p_1 & \mapsto \lambda x . \text{ sel } \langle p_2, p_1 \rangle (\text{ spl } x) (\lambda x . \text{ ret } x) \\
    (\lambda x . \text{ send } p_2 (\pi x) \Rightarrow \lambda y . \text{ send } p_1 (\pi y) \Rightarrow \lambda_. \\
    \text{recv } p_2 \text{ L}$s$\Rightarrow \lambda x . \text{ recv } p_3 \text{ L}$s$\Rightarrow \lambda y . \text{ ret } (\text{ msg } (x, y)))
    \end{align*}

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 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 $\text{Mp}$ monadic actions to C using $\text{pthread}$ 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 against the number of participants for different input sizes, and Fig. 6 present a summary of our speedups for large inputs of size $> 10^5$. 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

![Figure 4. Benchmarks: potential parallelisations.](image-url)
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

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 $p_1!(µL)$. This means that $p_1$’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; 57] allow us to permute such actions, and still have communication safety, i.e. $p_1!(µL), p_3?(µL) . . .$. 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. [50] 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. [59; 60] 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; 46; 64], algorithmic skeleton APIs [1; 18; 20; 36; 48], and DSLs/APIs that compile to parallel code [8; 11; 28; 63; 69]. 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; 41]. 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 [40; 67; 68]. 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; 37; 39; 49; 51; 55; 56; 65; 66]. 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 [70]. 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/K034431/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, and EP/T006544/1.

References


A Further Definitions

A.1 Algebraic Functional Language

\[ F_1, F_2 \vdash a \mid F_1 + F_2 \mid F_1 \times F_2 \]
\[ a, b \vdash a \mid \text{int} \mid \ldots \mid a \rightarrow b \mid a + b \mid a \times b \mid F \mid a \mid \mu F \]
\[ e_1, e_2 \vdash f \mid v \mid \text{const}_1 \mid \text{id} \mid e_1 \circ e_2 \mid | \pi_1 \mid e_1 \bowtie e_2 \mid e_1 \mid e_2 \]
\[ | F \mid | e \mid \text{in}_F \mid | \text{out}_F \mid | \text{rec}_F \]
\[ f : a \rightarrow b \in \Gamma \]
\[ e : a \]
\[ \vdash f : a \rightarrow b \]
\[ \vdash e : a \rightarrow b \]
\[ \vdash \text{const}_1 : b \rightarrow a \]
\[ \vdash \text{id} : a \rightarrow a \]
\[ \vdash \text{in}_F : F \mu F \rightarrow \mu F \]
\[ \vdash \text{out}_F : \mu F \rightarrow F \mu F \]
\[ \vdash e_1 : b \rightarrow c \]
\[ \vdash e_2 : a \rightarrow b \]
\[ \vdash e_1 \circ e_2 : a \rightarrow c \]
\[ i \in \{1, 2\} \]
\[ \vdash \pi_1 : a_1 \times a_2 \rightarrow a_1 \]
\[ \vdash e_1 \bowtie e_2 : a \rightarrow b \times c \]
\[ \vdash e_1 \bowtie e_2 : a \rightarrow b \times c \]
\[ i \in \{1, 2\} \]
\[ \vdash \pi_1 : a_1 \rightarrow a_1 + a_2 \]
\[ \vdash e_1 : a \rightarrow c \]
\[ \vdash e_2 : b \rightarrow c \]
\[ \vdash e_1 \bowtie e_2 : a \rightarrow b \]
\[ \vdash \text{rec}_F e_1 e_2 : a \rightarrow b \]

Figure 7. Syntax and types of Alg.

A.1.1 Properties of Alg Constructs

Alg constructs are characterised by well-known properties. Sum and product functions, are uniquely determined by their universal properties. Composition and identity must satisfy the associativity and cancellation properties. These basic properties are summarised in Fig. 9. Functors preserve identities, composition, and rec satisfy the hylomorphism laws (Fig. 9b). The laws of hylomorphisms can be performed some common program optimisations. For example, the well-known deforestation transformation can be
Constant, Identity and Composition

\[ \text{const } e = \lambda x. e \quad \text{id } = \lambda x. x \quad e_1 \circ e_2 = \lambda x. e_1 (e_2 (x)) \]

Products

\[ \pi_i = \lambda (x_1, x_2). \quad x_i, \quad i \in \{1, 2\} \quad e_1 \times e_2 = \lambda x. (e_1 (x), e_2 (x)) \]

Coproducts

\[ \iota_i = \lambda x. \text{inj}_i x \quad e_1 \uplus e_2 = \lambda (\text{inj}_i(x)). \quad e_1 x \quad e_1 \mid e_2 = (\iota_1 \circ \iota_1) (\uplus (e_2 \circ \pi_2)) \]

Functors

\[ 1 a = a \quad K a b = a \quad (F_1 \uplus F_2) a = F_1 a \uplus F_2 a \quad \iota \in \{+, \times\} \]

Recursion

\[ \text{in}_F = \lambda x. \text{in}_F x \quad \text{out}_F = \lambda (\text{in}_F x). x \quad \text{rec}_F e_1 e_2 f = f \quad \text{where } f = e_1 \circ F \circ f \circ e_2 \]

Figure 8. Semantics of Alg combiners.

Example A.1 (Rewriting and annotation strategies). We illustrate how rewriting and annotation strategies work by showing the mergesort (ms) example. Consider the merge-sort definition \( ms \rightarrow \text{rec}_T \ms \) spl. Solutions to the inference problem \( \vdash ms \rightarrow ?0 : \text{Lspl} \rightarrow ?1 \) provide the alternative parallelisations of ms. The only two rules that can be applied are Alg or Ext. By rule Alg, we can annotate ms at some \( p_1 \vdash ms \rightarrow ?0 : \text{Lspl} \rightarrow ?1 \). Alternatively, we can use the hylomorphism equation, and apply rule Ext:

\[ ms \rightarrow \text{rec}_T \ms \ms \rightarrow \ms \circ \text{ms} \ms \ms = \text{rec}_T \ms \ms \ms \rightarrow \ms \circ \text{ms} = \ms \circ (\text{id } + \ms \times \ms) \circ \text{spl} \]

We decide which of the rules to apply by querying a collection of rewriting hints, that we call \( \text{rewriting strategy} \). The collection of hints is of the form \( \{ e_1 : \text{rw}_1, \ldots, \} \), and must be specified by the programmer. The rewriting rules are essentially proofs that \( e_i = e_i' \) by applying equations in Figure 9. Once a hint is used, it is removed from the collection of hints. For the mergesort example, if we use the rewriting strategy \( \{ ms \rightarrow \text{unr11 }1 \} \), we will apply rule Ext, unravel the hylomorphism equation once, and continue with an empty strategy [].

With an empty rewriting strategy, the only possibility once we find the atomic function spl is to use rule Alg. To select a participant, we query the annotation strategy. The annotation strategy is a collection of expressions that we require to place at distinct participants. Suppose that our annotation strategy is \( \{ \text{spl} \} \). Then, we would need to select a fresh participant \( p_1 \vdash \text{spl} \rightarrow \text{spl}_{p_1} \rightarrow ((1 + a) \times \text{Lspl})_{p_1} \). If the annotation strategy does not contain spl, then we would select any participant from the input interface, to minimise the amount of messages exchanged: \( \vdash \text{spl} \rightarrow \text{spl}_{p_0} \rightarrow ((1 + a) \times \text{Lspl})_{p_0} \). Suppose that the annotation strategy is \( \{ \text{spl} \} \). Then, after spl we have a sum type at \( p_1 \). This requires us to introduce a choice point:
1485 We also write injections of interfaces that contain choices: a
1486 tion for annotation down the type structure, e.g.
1487 a faces that contain choices. We do the product of the re-
1488 pression follows a divide-and-conquer parallel structure, that may
1489 bypass the structure of
1490 Figure 9. Properties of point-free combinators
1491
1492 (b) Hylomorphism Laws
1493
1494 Definition A.2 (Product and Injection of Choice Interfaces).
1495 We sometimes write $A \times B$ to represent the product of in-
1496terfaces that contain choices. We do the product of the respec-
1497tive interfaces, after performing first the choices in $A$, and
1498then the choices in $B$:
1499
1500 \[
1501 (R_1 \uplus R_2) \times R_3 = (R_1 \times R_3) \uplus (R_2 \times R_3)
1502\]
1503
1504We also write injections of interfaces that contain choices:
1505
1506\[
1507i_l : (R_1 \uplus R_2) \to i_l : R_1 \uplus i_l : R_2
1508\]
1509
1510Definition A.3 (Well-formedness of interfaces: WF($a@R$)).
1511Interface $a@R$ is well formed if $R$ matches the structure of $a$:
1512
1513\[
1514\text{WF}(a@R) \iff \frac{WF(a@\ell) \quad WF((a+x)@\ell(i_l l)) \quad WF((a \times \overline{x})@\ell(i_l l))}{WF(a@R)}
1515\]
1516
1517For well-formed interfaces, we sometimes propagate the anno-
1518tation down the type structure, e.g. $a@R_1 \uplus b@R_2$ is nota-
1519tion for $(a \times b)@\ell(i_1 l R_1 \uplus i_2 l R_2)$. We also define sums of interfaces
1520$a@R_1 \uplus b@R_2$ as notation for $(a \times b)@\ell(i_1 l R_1 \uplus i_2 l R_2)$.
1521
1522\[
1523\frac{WF(a@\ell) \quad WF((a+x)@\ell(i_l l)) \quad WF((a \times \overline{x})@\ell(i_l l))}{WF(a@R)}
1524\]
1525
1526\[
1527\frac{WF(a@\ell) \quad WF((a+x)@\ell(i_l l)) \quad WF((a \times \overline{x})@\ell(i_l l))}{WF(a@R)}
1528\]
Two local types can be merged only if they are the same, or if they branch on the same role.

Two local types can be merged only if they are the same, or if they branch on the same role. For example, p₁’s local type of the global type: μX.p₁ → p₂.l₁ → p₂.l₃ → l₅.p₁.p₂.l₁.l₅.end is μX.p₂ & {l₁.p₂ & {l₃.X, l₄.p₂ & {l₅.p₁ & {l₆.end}}}.

Definition A.5. LTS for Local Type Configurations

(C, Q) ∼ fs (C’, Q’) = [p₁ → L₁]i∈I Q = [p₁ → w]i∈I
(C[p₀ → p₁(a), L], Q[p₀p₁ → w])
(p₁ → t(a), L), Q[p₁ → w ⋅ a]
(C[p₁ → p₁(t, L), Q[p₁p₀ → w])
(C[p₁ → p₁[ι, l₁]i∈I], Q[p₁p₀ → w])
(C[p₁ → p₁[ι, l₁]i∈I], Q[p₁p₀ → w])

Definition A.6 (Interface projection: A | p). The projection of interface A onto role p is the part of interface A that is located at p. We define the projection for a@R inductively on the structure of R:

a@p₀ | p₁ = {a, if p₀ = p₁, 1, otherwise}
(a × b)(R₁ × R₂) | p = (a @ R₁) × (b @ R₂)
(a₁ + a₂)(ι, R) | p = (a₁ @ R) + p
(a₁ ⊕ a₂)(ι, R) | p = a₁ ⊕ a₂
(R₁ ⊕ R₂) | p =
(a @ R₁) ⊕ (a @ R₂) | p if p ∈ p
a’ if a’ = (a @ R₁) + p | p = (a @ R₂) + p

A.3 MPST

Definition A.7 (Label Broadcasting). We define a macro that represents the broadcasting of a label to multiple participants in a choice. We write:

• p → {p₁}j∈[1,n] : {λi,G₁}i∈I for p → p₁:i→p₂i

It is straightforward to show that (p₁ → {p₂}j∈j : {λi,G₁}i∈I) \ p₂ = (p₂)j∈j : {λi,G₁}i∈I, if p₂ is a choice, and (p₁ → {p₂}j∈j : {λi,G₁}i∈I) \ p₂ = p₁ & {λi,G₁}i∈I, if p₂ is p₂ for some j ∈ J.

The relation ≡ p ⇐ A ∼ G associates p and A with the global type G (Fig. 11).

Rules Ind, Inj, Proj, and Case are straightforward. Rule Comp associates two PAlg expressions with the sequencing of their respective global types, G₁ ∼ G₂. The sequencing produces the global type that results of performing first G₁, and then G₂, by taking into account branching and choices:

Example A.8 (Mergesort Protocol). Recall the PAlg expression for ms in Example A.1:

We need to solve (a @ p₁) o (id + (ms @ p₂ +π₁ @ p₁)) o (ms @ p₁ + π₁ @ p₁) o (ms @ p₁ + π₁ @ p₁) 0 (p₁ @ p₂p₃) 0 spl @ p₁ : Ls @ p₀ → Ls @ p₁ ∪ P ; P ; P Ls @ p₁

At this point, the input interface is:

This means that we need to obtain two sub-protocols, for the left and the right branches respectively. The left branch is solved by applying rule Inj₂, while the right branch is solved by rules Case, Inj₂, Split, Comp and Alg:
A.4 Mp

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.

\[ v \equiv x \mid (\tau, v) | i_1 \cdot v | \cdots | e \cdot v \]

\[ m_i \equiv \text{ret } v \mid m_2 \Rightarrow f \mid \text{send } p \cdot v \mid \text{recv } p \cdot a \mid \text{sel } \tilde{p} \cdot v \cdot f_2 \]

\[ \text{brn } p \cdot m_1 \cdot m_2 \mid \text{case } f_1 \cdot f_2 \mid f \equiv \lambda x. m \]

Values \( v \) are either primitive values, tagged values \( i \cdot v \), pairs of values, or the result of applying an Alg expression \( e \) to a value. We use standard notation for the monadic unit (\( \text{ret} \)) and bind (\( \Rightarrow \)). The term \( \lambda x. m \) is a monadic continuation. We write \( \lambda \cdot m \) when the continuation discards the result of the previous monadic action. We use the standard Kleisli composition: \( f_1 \Rightarrow f_2 \equiv \lambda x. f_1 x \Rightarrow f_2 \).

The message-passing constructs are standard, except \( \text{sel} \), \( \text{brn} \) and \( \text{case} \), which are used for performing choices, and composing actions that depend on the same choice. We explain them in detail below. We include select and branching as syntactic constructs to simplify the typeability of parallel code against local types, but their semantics can be defined in terms of standard pattern matching, plus send and receive operations.

Each monadic composition \( f \) or \( m \) has a type \( m : M p L a \), where \( a \) is the return type of \( m \), and \( L \) is the type index of \( M p \), 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 refer the reader to Appendix A (Fig. 16) for the full definition.

**Composing Choices** The types of the constructs that deal with choices use a new type, \( \mathcal{C} \), that is isomorphic to sum types, but that can only be constructed and eliminated by using the following monadic constructs:

\[ \text{sel } \tilde{p} : a + b \to (a \to M p L_1 c_1) \to (b \to M p L_2 c_2) \]

\[ \Rightarrow M p (\tilde{p} \odot \{1, L_1; 1, L_2\}) (c_1 \uplus c_2) \]

\[ \text{brn } p : M p L_1 a_1 \to M p L_2 a_2 \]

\[ \Rightarrow M p (p \odot \{1, L_1; 1, L_2\}) (a_1 \uplus a_2) \]

\[ \text{case } (a : M p L_1 c) \to (b : M p L_2 d) \to a \uplus b \]

\[ \Rightarrow M p (L_1 \cup L_2) (c \uplus d) \]

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. It may seem that this treatment of \( \mathcal{C} \) leads to unnecessary code duplication, e.g. the only possibility to compose a single action \( f \) after a branch is using case: \( \text{brn } p \cdot m_1 \cdot m_2 \Rightarrow \text{case } f \cdot f \).

Our back-end easily optimises those cases to avoid code duplication.
Parallel programs We define the basic constructs of PAg 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 $m_i : Mp \{ a \ union b \} \ L_i$ for all $i \in I$, then we write $[p_i \mapsto m_i]_{i \in I} : Mp \{ [p_i \mapsto a_i]_{i \in I} \}$. The semantics of both local types and monadic actions is defined in terms of such collections of actions or types, and shared queues of values $W$, or queues of types $Q$, e.g. $(E, W') \rightarrow^E (E', W')$ is a transition from $E$ to $E'$, and shared queues $W$ to $W'$ with observable action $E$. 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 $E : Mp \{ a \ union b \} \ L a$ and $W : Q$. Suppose $(E[r \mapsto m], W') \rightarrow^E (E[r \mapsto m'], W')$. Then there exists $(C[r \mapsto L], Q') \rightarrow^C (C[r \mapsto L'], Q')$ such that $W' : Q'$ and $m' : Mp \{ a \}$.

Notations and Operations for Parallel Programs We simplify the notation for $E$, when all $L_i$ are projections of the same global type, and the $a_i$ are projections of the same interface. We define the projection of an interface at a participant, $A \mid p$, to be the part of $A$ that is at $p$ (Appendix A.6). Whenever we have $m_p : Mp \{ G \mid p \} \ (A \mid p)$ for all participants in $p \in G$, we use the notation $[p \mapsto m]_{p \in pids(G)} : Mp \{ G \}$. This means that the collection of all actions $m_p$ behave as prescribed by $G$, and produce their result in interface $A$. Finally, if we have $E = [p \mapsto f] : A \mid p \mapsto Mp \{ G \mid p \} (B \mid p)$, we write $E : A \mapsto Mp \{ G \}$. Parallel programs have a default value for participants that are not in their domain. Unless otherwise specified, this default value is the identity. For example, $E(p) = f$ if $E = E[p \mapsto f]$, and $E(p) = \lambda x. x$ if $p \notin E$. We specify the default value using the underscore character as a key in the mapping from participants to monadic actions: $[ \_ \mapsto f ]$.

Distributed Values and Execution We define the execution of a parallel program on a distributed value below. A distributed value $V : a \oplus R$ is a mapping from participants to the value that they hold in the respective interface: $[p_i \mapsto (\xi_j : (a \oplus R) \ L_i)]_{i \in I} : (a \oplus R)$. Additionally, we require unit to be the default value, so if $p \notin pids(R)$, then $V(p) = ()$.

Definition A.9 (Execution). Given $E = [p_i \mapsto f_i]_{i \in I}$ and $X = [p_i \mapsto x_i]_{i \in I}$, we define $E(X) = [p_i \mapsto f_i, X(p_i)]_{i \in I}$, with $X(p_i) = x_i$ if $i \in J$, or $X(p_i) = ()$ otherwise. Given $Y = [p_k \mapsto y_k]_{k \in K}$, we say that $P(X)$ executes to $Y$, $P(X) \rightsquigarrow * Y$, if there is a trace $P(X), \emptyset \rightarrow^* \{ [p_i \mapsto \text{ret} Y(p_i)]_{i \in I}, \emptyset \}$.

We write $P(X) = Y$, whenever there is a unique $Y$ s.t. for all $Z$, $P(X) \rightsquigarrow * Z$ implies that $Z = Y$.

Composition and Identity Composition is defined as the standard Kleisli composition, extended to parallel programs as follows: $E_1 \circ E_2 = [p \mapsto E_1(p) \circ E_2(p)]_{p \in pids(E_1) \cap pids(E_2)}$. Then, $E_2 \circ E_1 = E_1 \circ E_2$. Identity is simply the empty program with just the default value, id $= []$.

Split and Projection The split operation is the participantwise split, and the $i$-th projection is the environment with the projection $i$ as the default value:

$$E_1 \uplus E_2 = [p \mapsto \lambda x. E_1(p) x : m \lambda y. E_2(p) x : m \lambda z. \text{ret}(y, z)]_{p \in pids(E_1) \cup pids(E_2)}.$$ 

Case and Injection Case expressions will never occur during code generation, since they will be resolved by choices. Injections only tag a branch in the protocol, and so we define them as the identity: $i = []$.

Choices Choices are performed by the participant holding a value of a sum-type, and the tag is notified to the list of participants that depend on them. The definition uses functions $get_f(x)$ and $put_f(y, x)$ to extract the value of a sum-type from the hole of a one-hole context $I$ (§3.1), and to replace the value at the hole respectively.

$$[p_0 \oplus p_0 \oplus \ldots \oplus p_n] =$$

$$[\lambda y. \text{ret}(get_f(x)) [(\lambda y. \text{ret}(put_f(y, x))) (\lambda y. \text{ret}(put_f(y, x)))]$$

$$p_1 \mapsto \lambda x. \text{brn} \ p \ (\text{ret} x) \ (\text{ret} x);$$

$$\ldots$$

$$p_n \mapsto \lambda x. \text{brn} \ p \ (\text{ret} x) \ (\text{ret} x)$$

The presence of type $\uplus$ means that we might require to perform a case expression to inspect the result of a previous choice; we define $E_1 \uplus E_2$ for this:

$$E_1 \uplus E_2 = [p \mapsto \lambda x. \text{case} E_1(p) E_2(p)]_{p \in pids(E_1) \cup pids(E_2) \cup \{ p \}}.$$ 

The definition of $\uplus$ means that the participants involved in a choice will perform a case expression to inspect which branch they need to take, while the rest of the participants will continue as specified by either $E_1$ or $E_2$. Note $(E_1 \uplus E_2)(p)$ will produce $E_2(p)$, if $p \in E_1 \cap E_2$. This will not be an issue during code generation: any participant that is not involved in a choice will have the same continuation in both branches.

A.5 Mp code generation The translation scheme for $M p$ code generation (Fig. 12) is done recursively on the structure of PAg expressions. It takes a PAg expression $e$, an interface $A$, and produces a mapping from all participants in $e$ and $A$ to their respective monadic continuations. We write $[e]_p(A)$, and guarantee that $[e]_p(A) : A \mapsto Mp \{ G \} B$, if $p \models e \equiv A \rightarrow (G, B)$. This means that if $e$ induces protocol $G$ with interfaces $A \rightarrow B$, then the generated code behaves as $G$, with interfaces $A$ and $B$. 
Theorem 5.2. \( a \) the same result as evaluating of and protocol compliance, we prove that if \( \lambda \text{rec}_{TT} p \rightarrow \lambda \text{rec}_{TT} p \), the remaining of the translation is straightforward on top of the previous definitions.

\[ \begin{align*}
(\lambda \text{rec}_{TT} p_1 \rightarrow p_0) & = [p_1 \mapsto \lambda x. \text{send} x \times p_0 & \mapsto \lambda \text{rec}_{TT} p_1 \rightarrow p_0] \\
[(a_1 + a_2)(\lambda i_1 \rightarrow p)] & = [a_1(\lambda i_1 \rightarrow p) \Rightarrow [p \mapsto \lambda x. \text{ret}(x) \times p]] \\
[(a \times b)(\lambda (i_1 \times i_2) \rightarrow p)] & = [a(\lambda i_1 \rightarrow p) \times b(\lambda i_2 \rightarrow p)]
\end{align*} \]

Example A.10 (MergeSort Code Generation). We start with the annotated \( ms \) from Example A.1, and we use \( p_1 \) as master role to avoid the initial communication from \( p_0 \) to \( p_1 \):

\[ \begin{align*}
ms & = \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \\
\circ & = \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \\
\circ & = \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms}
\end{align*} \]

Note that \( \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \) would be equivalent to \( \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \) because the output interface is already of the form \( p_1 \cup p_1 \). Therefore, for simplicity, we use \( \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \circ \text{ms} \), and use it to produce the runnable parallel program. Fig. 14 show the code generation process using a table, where the \( i \)-th column is the current code for participant \( p_i \), and the last column shows the expression and input interface that we are translating next.

From this point, we need to produce the code for the two branches. The left branch is straightforward, and is simply \( \lambda x. \text{ret} x \) for all participants. The result for the right branch is shown in Fig. 15.

Next, we combine both branches using case, and compose it with the previous result. To avoid unnecessarily pattern-matching expressions such as case, we optimise the code using rules of the form:

\[ \begin{align*}
& \text{case } f_1 f_2 \Rightarrow \text{case } f_3 f_4 \\
& = \text{case } f_5 f_6 \Rightarrow \text{case } f_7 f_8
\end{align*} \]

Additionally, we optimise all instances of \( x \) (\( x(i) \)) using the fact that \( 1 \times a \equiv a \times 1 \equiv a \). We show below the code for all \( p_i \), after composing it with \( \text{join}(p_1 \cup p_2) \), and applying these optimisations. We use different colours to highlight the different branches of the protocol.
p₁ → sel {p₂, p₃} (spl v) (λx. ret (mrg (t₁ x))) (λx. send p₂ (t₁ x) ≅ λy. send p₃ (t₂ x) ≅ _).
recv p₂ Ls ≅ λx. recv p₃ Ls ≅ λy. ret (mrg (t₂ (x, y))) ≅ λx. ret (join x)

Figure 13. Step-by-step execution of the parallel code for ms. Input is [p₁ → v], with spl v = t₂ (v₁, v₂).

Figure 14. Example Translation

Figure 15. Right branch for mergesort.

Typing Mp against Local Types  We define a relation between Mp code and the local type that captures their communication behaviour (Fig. 16). We define a judgement of the form Γ ⊢ m : Mp L a, where Mp L a is the type of an Mp expression that conforms to protocol L and returns a value of type a. The types are parameterised by a variable l that represents a local type continuation. The rules in Fig. 16 are straightforward, since they relate in a one-to-one way to the constructs of local types.

Semantics  The operational semantics of Mp terms is standard, and mirrors that of the local type configurations in [27].

The operational semantics is defined as an LTS with transitions of the form [p₁ → mᵢ] ∈ L, W) → [p₁ → mᵢ⁺] ∈ L, W⁺. Here l⁺ := p₀p₁!⟨a⟩ | p₀p₁?⟨a⟩ | p₀p₁ ⊗ i | p₀p₁ & i is the observable action that takes place, and represents, respectively, p₀ sends to p₁ a value of type a, p₁ receives from p₀, p₀ sends label i to p₁, and p₁ receives label i from p₀. We use the special symbol ⊗ to represent that no communication took place. Finally, W is a mapping from ordered pairs of roles to unbounded buffers that contain the data sent between participants.

Definition A.11. LTS for Mp Terms

P, W transitions to P’, W’ with action l. The transition rules are defined in Fig. 17.

Similarly, we define ⟨C, Q⟩ → [P, W] → [P’, W’]. Here, C is a collection of local types, C = {p₁ → Lᵢ} ∈ L, and Q is a mapping from ordered pairs of roles to unbounded buffers that contain types of the data exchanged. We also say that W is compatible with Q, W : Q, if for all pair p₁,p₂, if w₁ ··· wₙ = W(p₁,p₂) then a₁ ··· aₙ = Q(p₁,p₂), and for all i, wᵢ : aᵢ.

Definition A.12 (Get/Set for Types from One Hole Contexts). Whenever a role performs a choice, we have a type
Case \textbf{Choice}, \vdash e \Rightarrow \{ p \oplus p_0 : a \oplus I \mid [p_0] \rightarrow a \oplus I \mid (p_1, p_2) \} \cup \{ \top \mid (t_1, t_2) \}. By rule \textbf{Choice}, \models [p \oplus p] \Rightarrow [p \oplus p_1] \Rightarrow p \rightarrow \{ p \} \mid (t_1, \text{end}; t_2, \text{end})

Case \textbf{Proj}. By rule \textbf{Proj}, \models p_1 \Rightarrow A_1 \times A_2 \Rightarrow \text{end}.

Case \textbf{Comp}. \vdash e_1 \circ e_2 \Rightarrow e_1 \circ e_2 : A \rightarrow C. This implies that \vdash e_1 \Rightarrow e_1 : B \rightarrow C and \vdash e_2 \Rightarrow e_2 : A \rightarrow B. By the IH, \models e_1 \Rightarrow a \oplus I \Rightarrow A \rightarrow G_2, and \models e_2 \Rightarrow b \oplus I \Rightarrow \text{end} \Rightarrow C_1. We proceed by induction on the size of the base. The base case is \textbf{\textit{end}}. In this case, \textbf{\textit{end}} must not be \textbf{\textit{end}} or contain any choices. Therefore, \textbf{\textit{end}} is defined, and equal to \{G_2/\text{end}\}G_1. If \textbf{\textit{end}} = b@\text{\textit{end}}, then \textbf{\textit{end}} = G_2 \cup G_2. There are two cases: (1) \textbf{\textit{end}} = G_1 \cup G_1, or (2) \textbf{\textit{end}} is a choice in \textbf{\textit{end}}, i.e. \textbf{\textit{end}} = [p \oplus p] \Rightarrow (t_1, G_{11}; t_2, G_{12})/\text{end}\}G_1. In the first case, we have \models e_1 \Rightarrow a@\text{\textit{end}} \Rightarrow G_1, and \models e_2 \Rightarrow a@\text{\textit{end}} \Rightarrow G_2, which by the IH implies that \textbf{\textit{end}} is defined. Therefore, \{G_1 \cup G_1\} \Rightarrow \{G_1 \cup G_2\} = \{G_1 \cup G_1\}\mid (t_1, G_{11}; t_2, G_{12}) \Rightarrow \{G_1 \cup G_2\}. In the second case, there must be two sub-expressions of \textbf{\textit{end}}, \textbf{\textit{end}} and \textbf{\textit{end}} s.t. \models e_1 \Rightarrow a@G_1 \Rightarrow G_1, and \models e_2 \Rightarrow a@G_2 \Rightarrow G_2, and which implies that \textbf{\textit{end}} = \{G_1 \cup G_2\} \Rightarrow \{G_2/\text{end}\}G_1, and \textbf{\textit{end}} is also defined.

Case \textbf{Case}. \vdash e_1 \vee e_2 \Rightarrow e_1 \vee e_2 : t_1 \Rightarrow A \rightarrow B. By inversion, \vdash e_1 \Rightarrow e_1 : A \rightarrow B. By the IH, \models e_1 \sim A \Rightarrow G. By \textbf{\textit{Case}}, \models e_1 \vee e_2 \Rightarrow t_1 \Rightarrow A \sim G.

Case \textbf{Split}. \vdash \text{det} e_1 \wedge e_2 \Rightarrow e_1 \wedge e_2 : a@I \Rightarrow B \times C. By inversion, \vdash e_1 \Rightarrow e_1 : a@I \Rightarrow B \Rightarrow \textbf{\textit{end}} \Rightarrow C. By the IH, \models e_1 \Rightarrow a@I \sim G_1 \sim e_2 \Rightarrow a@I \sim G_2. Therefore, \models t_1 \wedge e_2 \Rightarrow a@I \sim \{G_2/\text{end}\}G_1.

\textbf{B.2 Proof of Lemma 4.3}

\textbf{Lemma 4.3.} \{Protocol Deadlock-Freedom\} For all WF(A), if \vdash e : A \rightarrow B and \models e \sim A \Rightarrow G, then WF(G).

\textit{Proof}. By induction on the structure of the derivation \vdash e : a@I \rightarrow B.

Case \textbf{JOIN}, \vdash e \Rightarrow e : A \cup \top \Rightarrow A \Rightarrow B \cup \top \Rightarrow B. By the IH, \models e \sim A \sim G \Rightarrow B \Rightarrow G \Rightarrow G .

Case \textbf{ALT}, \vdash e \Rightarrow e : A \cup \top \Rightarrow A \sim G \Rightarrow G \Rightarrow G.

Case \textbf{ALT}, \vdash e \Rightarrow e : A \cup \top \Rightarrow A \sim G \Rightarrow G \Rightarrow G.

Case \textbf{ALT}, \vdash e \Rightarrow e : A \cup \top \Rightarrow A \sim G \Rightarrow G \Rightarrow G.

Case \textbf{ALT}, \vdash e \Rightarrow e : A \cup \top \Rightarrow A \sim G \Rightarrow G \Rightarrow G.

Case \textbf{ALT}, \vdash e \Rightarrow e : A \cup \top \Rightarrow A \sim G \Rightarrow G \Rightarrow G.
\[ (P, W) \rightsquigarrow^f (P', W') \quad P = \{ p_i \mapsto m_i \}_{i \in I} \quad W = \{ p_i \mapsto w \}_{i \in I} \]

\[ (P[p \mapsto m], W) \rightsquigarrow^f (P[p \mapsto m'], W') \]

\[ (P[p \mapsto m \, \beta e], W) \rightsquigarrow^f (P[p \mapsto m' \, \beta e], W') \]

\[ (P[p_1 \mapsto \text{send } p_2 (v : a)], W) \rightsquigarrow^f (P[p_1 \mapsto \text{F}$1$](a)) \quad (P[p_1 \mapsto \text{ret } ()], W[p_1 p_2 \mapsto v \cdot w]) \]

\[ (P[p_1 \mapsto \text{recv } p_2 \, a], W[p_2 p_1 \mapsto w \cdot v]) \rightsquigarrow^f (P[p_1 \mapsto \text{ret } v], W[p_2 p_1 \mapsto w]) \]

\[ (P[p_0 \mapsto \text{sel } (i : v) \{ f_1 \, f_2 \}], W) \rightsquigarrow^f (P[p_0 \mapsto f_1 \, v \mapsto \lambda x. \text{ret } (br_i \, x)], W) \]

\[ (P[p_0 \mapsto \text{sel } (i : v)\{p_1 \ldots p_n\} \, f_1 \, f_2 \}, W[p_0 p_1 \mapsto w]) \rightsquigarrow^f (P[p_1 \mapsto \text{ret } v], W[p_2 p_1 \mapsto w]) \]

\[ (P[p_1 \mapsto \text{brn } p_2 \, m_1 \, m_2], W[p_2 p_1 \mapsto w \cdot i]) \rightsquigarrow^f (P[p_1 \mapsto \text{ret } v], W[p_2 p_1 \mapsto w]) \]

\[ (P[p_1 \mapsto \text{case } f_1 \, f_2 \,(br_i \, v)], W) \rightsquigarrow^f (P[p_1 \mapsto f_1 \, v], W) \]

**Figure 17.** Rules for the LTS of Mp terms

\[ \text{WF}(G_2). \text{ Since pids}(G_1) \subseteq \text{pids}(p) \cup \text{pids}(A_1) \cup \text{pids}(A_2) \subseteq \text{pids}(r), \text{WF}(G_1 \cup \bar{p} \cdot G_2). \]

**Case In.** Trivial by WF(\text{end}).

**Case Choice.** \( e \mapsto e \circ [p \oplus \bar{p}] : a \circ I[p] \mapsto B_1 \cup \bar{p} B_2 \), where pids(e) \( \subseteq \bar{p} \) and \( J[p] \subseteq \bar{p} \). By the Choice and \text{Alt} typing rules, \( e \mapsto e \circ [p \oplus \bar{p}] \rightarrow a \circ I[p] \rightarrow B_1 \cup \bar{p} B_2 \). By inversion, the protocol rule must be also Choice: \( \vdash [p \oplus \bar{p}] \rightarrow a \circ I[p] \rightarrow B_1 \cup \bar{p} B_2 \). By the Choice protocol rule, \( \vdash e \rightarrow a \circ I[p] \rightarrow B_1 \cup \bar{p} B_2 \). By the IH, \( \text{WF}(G_i) \). Since pids(G_i) \( \subseteq \text{pids}(e) \cup \text{pids}(J[p]) \subseteq \bar{p} \), then for all \( p' \in G_i \), \( p \rightarrow \bar{p}(i, G_i)_{i \in [1,2]} \) \( \rightarrow p' \) must be defined. Therefore, \( \text{WF}(\text{p} \rightarrow \bar{p}(i, G_i)_{i \in [1,2]}) \).

**Case Proc.** Trivial by WF(\text{end}).

**Case Comp.** \( e_1 \circ e_2 \rightarrow e_1 \circ e_2 : A \rightarrow C \), with \( e_1 \rightarrow e_1 : B \rightarrow C \) and \( e_2 \rightarrow e_2 : A \rightarrow B \). By inversion, the only possible protocol rule is also Comp. Therefore, \( e_1 \circ e_2 \rightarrow A \rightarrow G_2 \circ G_1 \), with \( e_1 \rightarrow A \rightarrow G_2 \) and \( e_1 \rightarrow B \rightarrow G_1 \). By the IH, \( \text{WF}(G_1) \) and \( \text{WF}(G_2) \). Also, by the induction on the derivation of \( e \), we know that \( A_1 \cup \bar{p} A_2 \), if \( A_1 \neq A_2 \), then pids(A_1) \( \subseteq \bar{p} \). This implies that if \( G_1 \) is \( G_1 \cup \bar{p} G_2 \), then either the projection onto \( p \) of \( G_1 \) is the same, or \( p \in \bar{p} \). By the Choice rule, \( G \) must be of the form \( G' \rightarrow \bar{p}(i, G_1)_{i \in [1,2]} \), therefore, for all \( p' \in \text{pids}(G_1) \), the projection of \( (G_2 \cup G_1) \rightarrow p' \) must be defined, which implies that \( G_2 \circ G_1 \) is defined.

**Case Case.** \( e_1 \lor e_2 \rightarrow e_1 \lor e_2 : a \circ (i_1 \circ I) \rightarrow B \) and \( e_1 \lor e_2 \rightarrow e_1 \circ (i_1 \circ I) \rightarrow A \). By the Case_{(1)} protocol and typing rules, \( e_1 \rightarrow A \rightarrow G \) and \( e_1 \rightarrow e_1 : A \rightarrow B \). We conclude by the IH that \( \text{WF}(G) \).

**Case Split.** \( e_1 \land e_2 \rightarrow e_1 \land e_2 : (b \rightarrow c) \circ (R_1 \times R_2) \) and \( e_1 \land e_2 \rightarrow A \rightarrow [G_2 \circ \text{end}]G_1 \). By the IH, we know that \( \text{WF}(G_2) \) and \( \text{WF}(G_3) \). By straightforward induction on the structure of \( G_1 \), if \( \text{WF}(G_1) \), then \( \text{WF}([G_2/\text{end}]G_1) \).

**B.3 Proof of Theorem 5.2**

**Theorem 5.2.** [Protocol Conformance of the Generated Code]

If \( e \iff e : A \rightarrow \bar{G} \), then \( \llbracket e \rrbracket (A) \text{ complies with protocol } G \).

**Proof.** By induction on the structure of the derivation \( \vdash e \rightarrow A \rightarrow \bar{G} \).

Case Alt. \( \vdash e \rightarrow A_1 \cup \bar{p} A_2 \rightarrow \bar{G}_1 \cup \bar{p} \bar{G}_2 \), with \( \vdash e \rightarrow A_1 \rightarrow \bar{G}_1 \), \( \vdash e \rightarrow A_2 \rightarrow \bar{G}_2 \). By the IH, \( \llbracket e \rrbracket (A_1) : (A_1 \rightarrow p) \rightarrow \text{M}_p (\bar{G}_1 \cup p) (B_1 \cup p) \). Moreover, we know that if \( p \in \bar{p} \), then \( \llbracket e \rrbracket (A_1)(p) = \llbracket e \rrbracket (A_2)(p) \), and \( G_1 \cup p = G_2 \cup p \). Therefore, by the definition of \( E_1 \circ E_2, \llbracket e \rrbracket (A_1) \cup \bar{p} \llbracket e \rrbracket (A_2) : \text{M}_p (G_1 \cup \bar{p} G_2) (B_1 \cup \bar{p} B_2) \).

Case In. \( \llbracket \text{id} \rrbracket \rightarrow \text{M}_p (p \circ \bar{p} \rightarrow \text{M}_p \cup p) \rightarrow \text{M}_p \text{end} \).

Case In_{(1)}. \( \llbracket i_1 \rrbracket \rightarrow \text{M}_p \text{end} \). By definition, \( \llbracket i_1 \rrbracket (A) = \llbracket e \rrbracket (A) \rightarrow \text{M}_p \text{end} \rightarrow \text{M}_p \text{end} \).

Case Alg. \( \llbracket e \circ p_e \rightarrow e \circ \text{M}_p \rightarrow a \circ \text{M}_p \rightarrow p_e \) : \( e : a \rightarrow b \). We prove by straightforward induction on the structure of \( I \) that \( f = (a \circ \text{M}_p \rightarrow p_e) = (a \circ \text{M}_p \rightarrow p_e) : (a \circ \text{M}_p \rightarrow p_e) : (a \circ \text{M}_p \rightarrow p_e) \).

Case Comp. \( \llbracket e_1 \circ e_2 \rightarrow A \rightarrow G_2 \circ G_1 \), with \( \text{Comp} \).

Case Split. \( e_1 \land e_2 \rightarrow A \rightarrow G_2 \circ G_1 \), with \( \text{Split} \).

Case Case. \( e_1 \lor e_2 \rightarrow A \rightarrow G_2 \circ G_1 \), with \( \text{Case} \).

**Figure 18.** Get/Set for Types from One Hole Contexts

\[ \text{WF}(G_2) \text{ and } \text{WF}(G_3) \text{. By straightforward induction on the structure of } G_1, \text{ if } \text{WF}(G_1), \text{ then } \text{WF}([G_2/\text{end}]G_1). \]

\[ \text{WF}(G_2) \text{ and } \text{WF}(G_3) \text{. By straightforward induction on the structure of } G_1, \text{ if } \text{WF}(G_1), \text{ then } \text{WF}([G_2/\text{end}]G_1). \]
If $p \in [p] \perp \{p\}$, then $\{p\} \rightarrow \{p\}$. By the definition of $[p] \perp \{p\}$, we have $\{p\} \rightarrow \{p\}$. Therefore, $[p] \perp \{p\}$.

**Lemma C.2.** Assume $G, A, B, X : A$, and $f_i : A \rightarrow p_i \rightarrow Mp (G \mid p_i) (B \mid p_i)$ for all $i \in I$. Let $P = [p_i \rightarrow f_i]_{i \in I}$ then there is a unique $Y$ s.t. $P(X) = Y$.

**Proof.** Straightforward consequence of Lemma 5.1, and Theorem 3.1 in [27]. We know that the traces for $G$ can only differ in the order of the actions, and that this order must preserve the dependencies laid out by $G$. Therefore, there the result of any possible execution must respect the data dependencies specified by $G$.

**Lemma C.3.** If $(P, W) \parallel X$ and $(P, W) \rightarrow (P', W')$, then $(P', W') \parallel X$.

**Theorem 5.3.** [Extensionality] Assume $e : a \rightarrow e : a \rightarrow b \rightarrow R$ of a psl expression $e : A \rightarrow B$, each applied to the corresponding value of type $a : A \vdash p$ represents an execution of the parallel algorithm on an input of type $a$, if $a \in R = A$. Recall from Sec. 5 that the transitions are of the form $(P, W) \rightarrow (P', W')$, where $P$ is an environment that contains the code executed by all roles that collaborate to compute the parallel algorithm, and $W$ represents the shared unbounded buffers used by each pair of participants to communicate. We write $w$ for such buffers, where $\emptyset$ is the empty buffer, $\psi \cdot w$ is the buffer $w$ extended with value $\psi$ at the rightmost position, and $w \cdot \psi$ is the buffer $w$ extended with value $\psi$ at the leftmost position.

**Definition C.1** (Type buffers). We write $Q = [p_i : p_j \rightarrow q_{i \in I, j \in I}]$, where $q$ is a buffer of types, that can be either $\emptyset$, $a \cdot q$ or $q \cdot a$. Note that values include singleton types that represent labels: $l : l$. We say that a buffer $w = v_1 \cdots v_n$ contains types $q = a_1 \cdots a_m$, $w : q$ if: $n = m$ and $v_i : a_i$ for all $i \in [1, n]$. We say that $W : Q$ if for all pairs of roles, $p_i : p_j$, $W(p_i, p_j) : Q(p_i, p_j)$.

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

**Proof.** Straightforward induction on $L_i$, and case analysis on $m_i$ and the rules $\rightarrow$ and $\rightarrow$, since there is a one-to-one correspondence between the rules syntactic constructs in $M$ and the local types. For $\rightarrow$ we need to take several $\epsilon$ transitions until communication $\ell$ happens.

We prove the following generalised statement. Let $e : a \rightarrow b$ s.t. $e : a \rightarrow b : a$, and $\overline{i} \vdash \overline{s}$. Then, there is a $\overline{s}$ s.t. $[e]((\overline{s}))(x) = \overline{s}((\overline{s}))$. We define $\overline{s}((\overline{s})) : A$ as follows:

$$\overline{s}(x) = \overline{s}(x)$$

Proof. We proceed by induction on the structure of the derivation $\vdash e : a \rightarrow b$:

- **Case Join.** We have $\vdash e : A \cup B \rightarrow A \cup B$ with $\vdash e : A \rightarrow B$. By definition, $[e] : (A \cup B) \rightarrow [e](A) \cup B$

  $$[e]((A \cup B)) = [e](A) \cup B$$

  Then, by the induction hypothesis, there exists $\overline{i} s.t.$

  $$(\overline{s})(A) = \overline{s}(x)$$

  Then, by the induction hypothesis, there exists $\overline{i} s.t.$

  $$(\overline{s})(A) = \overline{s}(x)$$

  Then, by the induction hypothesis, there exists $\overline{i} s.t.$

  $$(\overline{s})(A) = \overline{s}(x)$$

  Then, by the induction hypothesis, there exists $\overline{i} s.t.$

  $$(\overline{s})(A) = \overline{s}(x)$$

  Then, by the induction hypothesis, there exists $\overline{i} s.t.$

  $$(\overline{s})(A) = \overline{s}(x)$$

- **Case Alt.** We have $\vdash e : A_1 \cup B_2 \rightarrow B_1 \cup B_2$ with $\vdash e : A_1 \rightarrow B_1$, $\vdash e : A_2 \rightarrow B_2$ and $A_1 \neq A_2$. Then, $[e]((A_1 \cup B_2) ((A_1 \cup B_2))) = ([e]((A_1) \cup B_2))$
\[ \begin{align*}
\text{Case } \text{At}. \quad & \text{with } \delta \text{ and } \pi, \text{ Case } \text{and } \text{Case } \text{since } \pi \text{ ret } \delta \text{ by straightforward induction on } I, \text{ there exists a trace } (\left\llbracket p \mapsto (a@I \sim p) (p_i) \right\rrbracket)
\end{align*} \]

We have
\[ (p \mapsto \lambda x. \text{ recv } x) \left( \left\llbracket p \mapsto \lambda x. \text{ ret } x \right\rrbracket \right) \]

By case analysis, if \( \lambda x. \text{recv } x \) = \( \lambda x. \text{ret } x \), then we have:
\[ 1. \ p \mapsto \lambda x. \text{put } x (x) \]
\[ 2. \ \lambda x. \text{recv } x \mapsto \lambda x. \text{ret } x \]

This is clearly \( \left\llbracket p \mapsto \text{recv } x \right\rrbracket \delta \left\llbracket I \mapsto \text{ret } x \right\rrbracket \left\llbracket p \mapsto \text{recv } x \right\rrbracket \)

\[ \square \]

\section{D Generated Code}

We show now the generated code for mergesort, unrolling the recursive function once.

\subsection{D.1 Input Alg expression}

\begin{verbatim}
fftTree :: forall n. SINat n -> Int
    -> Tree n (D [Complex Double])
    => Tree n (D [Complex Double])

fftTree SZ w
    = lift (intlit SZ &&& (lit w &&& id)
    >=> prim "baseFFT")

fftTree (SS w) x
    = withCDict (cdictTree @(D [Complex Double]) x) (w &&& x)
    $ [$
    | Recursive FFT to EVENs and ODDs$]$
    (fftTree x w)
    | Multiply right side by exponential $-$
    >>> id
    | *** mapTree x (lit ps2x &&& id $>$ => mapExp)
    | 0
    >>>$
    |$ zipWith add (swap arguments to force butterfly pattern
    | - &&& zipWith sub$
    | - ]$
    | zipTree x True 1lw w addr
    | &&& zipTree x False 1lw (w + 2^ toInteger x)
    | subc

where

1lw :: Int
1lw = fromInteger (toInteger (SS x) + 1)
ps2x :: Int
ps2x = 2^ toInteger (SS x)

fft :: SINat n -> (D [Complex Double]) => D [Complex Double]

fft n
    = withCDict (cdictTree @(D [Complex Double]) n)
        $]
    $ tsplit n deinterleave => fftTree n 0 >>> tfold
        (append @ 0)$
    fft5 :: D [Complex Double] => D [Complex Double]
\end{verbatim}
fft5 = withSize 5 fft

Listing 1. Fragment of FFT.hs

D.2 Main C Code and Atomic Functions

These need to be implemented by the programmer.

```c
#include "FFT.h"
#include <inttypes.h>
#include <errno.h>
#include <string.h>
#include <sys/time.h>
#include <stdlib.h>
#include <math.h>
#define REPETITIONS 50
#define BENCHMARKSEQ(s, f) { 
  time = 0; 
  time_diff = 0; 
  time_old = 0; 
  var = 0; 
  for(int i=0; i<REPETITIONS; i++){ 
    in = randvec(s, size); 
    start = get_time(); 
    out = f(in); 
    end = get_time(); 
    free_fftvec(in); 
    time_diff = end - start; 
    time_old = time; 
    time += (time_diff - time)/(i+1); 
    var += (time_diff - time) * (time_diff - time_old); 
  } 
  printf("K: %d
"), s); 
  printf("K: %d
"), time); 
  printf("K: %d
"), sqrt(var / (REPETITIONS - 1))); 
}
#define WARMUP(f) { 
  for(int i=0; i<REPETITIONS; i++){ 
    in = randvec(0, size); 
    out = f(in); 
    free_fftvec(in); 
  } 
}
double PI = atan2(1, 1) * 4;
```

```c
vec_cplx_t zip_add(
  pair_pair_int_int_pair_vec_cplx_vec_cplx_t in)
{
  int lvl = in.fst.fst;
  int wid = in.fst.snd;
  vec_cplx_t l = in.snd.fst;
  vec_cplx_t r = in.snd.snd;
  vec_cplx_t lout = stages[lvl][wid];
  for(int i = 0; i < l.size; i++){
    lout.elems[i] = l.elems[i] + r.elems[i];
  }
  return lout;
}
```

```c
vec_cplx_t zip_sub(
  pair_pair_int_int_pair_vec_cplx_vec_cplx_t in)
{
  int lvl = in.fst.fst;
  int wid = in.fst.snd;
  vec_cplx_t l = in.snd.fst;
  vec_cplx_t r = in.snd.snd;
  vec_cplx_t lout = stages[lvl][wid];
  for(int i = 0; i < l.size; i++){
    lout.elems[i] = l.elems[i] - r.elems[i];
  }
  return lout;
}
```

```c
vec_cplx_t cat(pair_vec_cplx_vec_cplx_t in){
in.fst.size *= 2;
return in.fst;
}
```

```c
void _fft(cplx_t buf[], cplx_t out[], int n, int step)
{
  if(step < n) {
    _fft(out, buf, n, step * 2);
    _fft(out + step, buf + step, n, step * 2);
  }
}
```

```c
void show(const char * s, vec_cplx_t in) {
  printf("%s", s);
  for(int i = 0; i < in.size; i++)
    if(!cimag(in.elems[i])
      printf("%g", creal(in.elems[i]));
```
else
    printf("(%g, %g) ", creal(in.elems[i]), cimag(in.elems[i]));
    printf("n");
}

void showstep(intstp, const char * s, vec_cplx_t in) {
    printf("%s", s);
    for (int i = 0; i < in.size; i+=stp)
        if (!cimag(in.elems[i]))
            printf("%g ", creal(in.elems[i]));
        else
            printf("(%g, %g) ", creal(in.elems[i]), cimag(in.elems[i]));
    printf("n");
}

vec_cplx_t baseFFT(pair_int_pair_int_vec_cplx_t in)
{
    int lvl = in.fst;
    int wid = in.snd.fst;
    cplx_t *buf = stages[lvl][wid].elems;
    int n = in.snd.size;
    _fft(buf, in.snd.elems, n, 1);
    return stages[lvl][wid];
}

vec_cplx_t seqfft(vec_cplx_t in)
{
    pair_int_pair_int_vec_cplx_t i = {1, {0, in}};
    return baseFFT(i);
}

vec_cplx_t map_exp(pair_int_pair_int_vec_cplx_t iv)
{
    int i = iv.snd.fst;
    int ps2x = iv.fst;
    vec_cplx_t in = iv.snd.snd;
    int step = i * in.size;
    for(int k = 0; k < in.size; k++)
    {
        in.elems[k] = in.elems[k] * cexp(2 * -I * PI * (k + step) / (ps2x * in.size));
    }
    return in;
}

void free_fftvec()
{
    for(int i = 0; i < num_stages; i++)
    {
        free(stages[i][0].elems);
        free(stages[i]);
    }
    free(stages);
}

pair_vec_cplx_vec_cplx_t deinterleave(
    pair_int_int_t iin)
{
    int wl = iin.fst;
    int wr = iin.snd;
    int mid = stages[0][wl].size/2;
    stages[1][wl].size = mid;
    stages[1][wr].size = mid;
    stages[1][wr].elems = stages[1][wl].elems + mid;
    for(int i = 0; i < stages[0][wl].size; i++)
    {
        stages[1][wl].elems[i/2] = stages[0][wl].elems[i];
        stages[1][wr].elems[i/2] = stages[0][wl].elems[i+1];
    }
    memcpy(stages[0][wl].elems, stages[1][wl].elems, stages[0][wl].size * sizeof(cplx_t));
    stages[0][wr].elems = stages[0][wl].elems + mid;
    stages[0][wr].size = mid;
    for (int i = 2; i < num_stages; i++)
    {
        memcpy(stages[i][wl].elems, stages[i][wl].elems, stages[0][wl].size * sizeof(cplx_t));
        stages[i][wl].size = mid;
        stages[i][wr].elems = stages[i][wl].elems + mid;
        stages[i][wr].size = mid;
    }
    stages[0][wl].size = mid;
    return (pair_vec_cplx_vec_cplx_t) { stages[0][wl], stages[0][wr] };
}

vec_cplx_t randvec(int depth, size_t s)
{
    num_workers = depth <= 1? 1 : 1 << depth - 1;
    num_stages = depth <= 1? 2 : 1 + depth ;
    stages = (vec_cplx_t **)malloc(num_stages * sizeof(vec_cplx_t *));
    for (int i = 0; i < num_stages; i++)
    {
        stages[i][0].elems = (cplx_t *)malloc(num_workers * sizeof(cplx_t));
        stages[i][0].size = s;
    }
}
```c
srand(time(NULL));

for (int i = 0; i < s; i++) {
    double rand_r = (double)rand() / (double)RAND_MAX;
    double rand_i = (double)rand() / (double)RAND_MAX;
    stages[0][0].elems[i] = rand_r + rand_i * I;
}

for (int j = 1; j < num_stages; j++) {
    memcpy(stages[j][0].elems, stages[0][0].elems, s * sizeof(vec_cplx_t));
    stages[j][0].size = s / num_workers;
}

for (int i = 0; i < num_stages; i++) {
    for (int j = 1; j < num_workers; j++) {
        stages[i][j] = stages[i][j-1];
    }
}

return stages[0][0];

void usage(const char *nm){
    printf("Usage: %s <input_size >
", nm);
    exit(-1);
}

int main(int argc, const char *argv[]){
    setbuf(stdout, NULL);
    if (argc <= 1) {
        usage(argv[0]);
    }
    char *endptr = NULL;
    errno = 0;
    size_t size = strtoumax(argv[1], &endptr, 10);
    size = (size_t) 1 << (long)ceil(log2(size));
    if (errno != 0) {
        printf("%s", strerror(errno));
        usage(argv[0]);
    }
    if (endptr != NULL & *endptr != 0) {
        usage(argv[0]);
    }
    vec_cplx_t *in, out;
    out = fft5(in);
    free_fftvec();
}

D.3 Automatically Generated C Code

#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<complex.h>

typedef double _Complex cplx_t;

typedef struct vec_cplx {
    cplx_t * elems; size_t size;
} vec_cplx_t;

typedef struct q_vec_cplx {
    volatile unsigned int q_size;
    int q_head;
    int q_tail;
    pthread_mutex_t q_mutex;
    pthread_cond_t q_full;
    pthread_cond_t q_empty;
    vec_cplx_t q_mem[1];
} q_vec_cplx_t;

void q_vec_cplx_put(q_vec_cplx_t *, vec_cplx_t);
vec_cplx_t q_vec_cplx_get(q_vec_cplx_t *);

typedef enum unit {
    Unit
} unit_t;

typedef struct pair_int_vec_cplx {
    int fst; vec_cplx_t snd;
} pair_int_vec_cplx_t;

typedef struct pair_int_pair_int_vec_cplx {
    int fst; pair_int_vec_cplx_t snd;
} pair_int_pair_int_vec_cplx_t;

vec_cplx_t baseFFT(pair_int_pair_int_vec_cplx_t);
vec_cplx_t fft0(vec_cplx_t);
vec_cplx_t fft1(vec_cplx_t);
vec_cplx_t fft5(vec_cplx_t);
```

Listing 2. Generated FFT.h

```c
#include "FFT.h"

q_vec_cplx_t ch0 = { 0, 0, 0, { } }
q_vec_cplx_t ch2 = { 0, 0, 0, { } }
q_vec_cplx_t ch3 = { 0, 0, 0, { } }

vec_cplx_t fft2_part_0(vec_cplx_t v_s)
{
    pair_int_int_t v_t;
    v_t.fst = 0;
    v_t.snd = 1;
    pair_vec_cplx_vec_cplx_t v_u;
    v_u = deinterleave(v_t);
    vec_cplx_t v_v;
    v_v = v_u.fst;
    q_vec_cplx_put(&ch0, v_v);
    vec_cplx_t v_w;
    v_w = v_u.snd;
    q_vec_cplx_put(&ch2, v_w);
    vec_cplx_t v_x;
    v_x = q_vec_cplx_get(&ch1);
    vec_cplx_t v_y;
    v_y = q_vec_cplx_get(&ch3);
    pair_vec_cplx_vec_cplx_t v_z;
    v_z.fst = v_x;
    v_z.snd = v_y;
    vec_cplx_t v_aa;
    v_aa = cat(v_z);
    return v_aa;
}

q_vec_cplx_t ch4 = { 0, 0, 0, { } }
q_vec_cplx_t ch5 = { 0, 0, 0, { } }
unit_t fft2_part_1()
{
    vec_cplx_t v_ba;
    v_ba = q_vec_cplx_get(&ch0);
    pair_int_pair_int_vec_cplx_t v_ca;
    v_ca.fst = 1;
    pair_int_vec_cplx_t v_da;
    v_da.fst = 0;
    v_da.snd = v_ba;
    v_ca.snd = v_da;
    vec_cplx_t v_ea;
    v_ea = baseFFT(v_ca);
    q_vec_cplx_put(&ch4, v_ea);
    vec_cplx_t v_fa;
    v_fa = q_vec_cplx_get(&ch5);
    pair_pair_int_int_pair_vec_cplx_vec_cplx_t v_ga;
    pair_int_int_t v_ha;
    v_ha.fst = 2;
    v_ha.snd = 0;
    v_ga.fst = v_ha;
    pair_vec_cplx_vec_cplx_t v_ia;
    v_ia.fst = v_ea;
    v_ia.snd = v_fa;
    v_ga.snd = v_ia;
    vec_cplx_t v_ja;
    #endif
```

---

27
CC ’20, February 22–23, 2020, San Diego, CA, USA

David Castro-Perez and Nobuko Yoshida

v_ja = zip_add(v_ga);
q_vec_cplx_put(&ch1, v_ja);
return Unit;
}

unit_t fft2_part_2()
{
  vec_cplx_t v_ka;
  v_ka = q_vec_cplx_get(&ch2);
  pair_int_pair_int_vec_cplx_t v_la;
  v_la.fst = 1;
  pair_int_vec_cplx_t v_ma;
  v_ma.fst = 1;
  v_ma.snd = v_ka;
  v_la.snd = v_ma;
  vec_cplx_t v_na;
  v_na = baseFFT(v_la);
  pair_int_pair_int_vec_cplx_t v_oa;
  v_oa.fst = 2;
  pair_int_vec_cplx_t v_pa;
  v_pa.fst = 0;
  pair_int_vec_cplx_t v_pa;
  v_pa.snd = v_na;
  v_oa.snd = v_pa;
  vec_cplx_t v_ba;
  v_ba = map_exp(v_oa);
  q_vec_cplx_put(&ch5, v_ba);
  vec_cplx_t v_ab;
  v_ab = baseFFT(v_ba);
  pair_pair_int_int_pair_vec_cplx_vec_cplx_t
    v_sa;
  pair_int_int_t v_ta;
  v_ta.fst = 2;
  v_ta.snd = 1;
  v_sa.fst = v_ta;
  pair_vec_cplx_vec_cplx_t v_us;
  v_us.fst = v_ra;
  v_us.snd = v_qa;
  v_ssa.snd = v_us;
  vec_cplx_t v_vb;
  v_vb = zip_sub(v_sa);
  q_vec_cplx_put(&ch3, v_vb);
return Unit;
}

void * fun_thread_1_1(void * arg)
{
  fft2_part_1();
return NULL;
}

void * fun_thread_2(void * arg)
{
  fft2_part_2();
return NULL;

vec_cplx_t fft2(vec_cplx_t v_wa)
{
  vec_cplx_t v_xa;
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, fun_thread_1_1, NULL);
  pthread_create(&thread2, NULL, fun_thread_2, NULL);
  v_xa = fft2_part_0(v_wa);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
return v_xa;
}

Listing 3. Fragment of generated FFT.c

E Artifact Appendix

E.1 Abstract
This artifact provides a prototype implementation of PAlg, embedded in Haskell, along with a number of benchmarks used to test the scalability of our approach. We provide scripts to regenerate the execution time measurements that we used in our paper. This will allow to evaluate our results on any multi-core shared-memory architecture.

We also provide a small tutorial that is meant to guide a programmer, step-by-step, in the implementation of a message-passing parallel algorithm using our library. The tutorial includes a guide on how to visualise the global types that correspond to the achieved parallelisations, as well as any asynchronous optimisations applicable to the generated message-passing code.

E.2 Artifact check-list (meta-information)

- **Algorithm**: Message-passing C code generation from first-order Haskell functions. Global type inference of the communication protocol followed by the parallelisation.
- **Program**: Haskell libraries Language.CAlg, noindent Language.CAlg.CSyn and dependencies, as well as session-arrc, to compile to C Haskell functions built using such libraries.
- **Compilation**: GHC >= 8.6 && < 8.8, and C compiler that supports C11.
- **Transformations**: Compilation to C, and asynchronous optimisation pass.
- **Binary**: Source code and scripts included to generate the binaries from the sources.
- **Data set**: Included original run-time measurements for comparison.
- **Hardware**: We used a 12-core Intel Xeon CPU E5-2650 v4 @ 2.20GHz. We recommend a shared-memory architecture, with uniform access times, to measure the overheads of our approach, not message latencies.
• **Execution**: We include a script to run the benchmarks.
• **Output**: Benchmark execution times.
• **Experiments**: Small, representative benchmarks of common parallel algorithms.
• **How much memory required (approximately)?**: 64GB for using the maximum benchmark input size.
• **How much time is needed to complete experiments (approximately)?**: 5 days on the hardware stated in §E.3.2.
• **Publicly available?**: Yes.
• **Code licenses (if publicly available)?**: BSD-3.

### E.3 Description

#### E.3.1 How delivered

We provide a docker image with the necessary dependencies: https://imperialcollegelondon.box.com/v/cc20-artifact-p43. After downloading, the image can be loaded using:

$ sudo docker load -i cc20-artifact-p43.docker

To run the image, run the command:

$ sudo docker run -ti cc20-artifact-p43

File README.md inside the docker image contains additional instructions. Our benchmarks, source code and scripts are also publicly available on Github, in https://github.com/session-arr/session-arr. Our benchmarks, source code and scripts are also publicly available on Github, in https://github.com/session-arr/session-arr.

#### E.3.2 Hardware dependencies

We used a 12-core Intel Xeon CPU E5-2650 v4 @ 2.20GHz. We recommend using a shared-memory architecture, with uniform access times, to measure the overheads of our approach, not message latencies.

#### E.3.3 Software dependencies

All our dependencies are listed in the Dockerfile in our public repository. We list them below. To compile our tool:

1. GH C >= 8.6 (not tested with GH C >= 8.8)
2. stack Version 1.9.1

To run our experiments:

1. C compiler that supports C11 (tested with GCC >= 4.8 && &< 8.3)
2. glibc (tested with versions >= 2.17 && &< 2.29)
3. numacl

To generate the graphs:

1. python (== 2.7)
2. python-matplotlib (== 2)
3. python-pint (== 0.7)

### E.3.4 Data sets

We include as part of the artifact the raw data that we obtained for our benchmarks. These are included under benchmarks/<bench_name>/data/t_<num_cores>, where <num_cores> is either 12 or 24. There is additionally a file t_48, that uses all 24 cores + hyperthreading. The structure of the files is:

```plaintext
size: <size>
K: seq
  mean: <avg_execution_time>
  stddev: <std_dev>
K: 1
  mean: ...
  stddev: ...
...
```

Keyword size denotes the size of the inputs for the particular benchmark. Keyword mean is the average execution time. Keyword stddev is the standard deviation. We write K: to denote the number of recursion unfoldings used to produce the parallel version.

Examples of global types for each benchmark are under benchmarks/<bench_name>/protocol/<bench_name>_<fun_name>.mpst, where <fun_name> is the function name in <bench_name> hs that corresponds to this protocol.

### E.4 Installation

**Note:** this section can be omitted if using our docker image.

We recommend using Stack (https://docs.haskellstack.org/en/stable/README/#how-to-install). To build our tool:

$ git clone https://github.com/session-arr/session-arr
$ cd session-arr
$ stack build
$ stack install

There is no need to install the tool. However, to install it, run:

$ stack install

This will copy the binary session-ar cc to a local directory, usually ${HOME}/.local/bin.

Manual compilation and installation using GHC is also possible, but we discourage it. Read session-ar cc/package.yaml to find out which haskell packages are required.

### E.5 Experiment workflow

#### E.5.1 Automatic

We included script session-ar cc/benchmark.sh to compile and run all the benchmarks used in the paper. To customise the amount of cores, the number of repetitions per experiment and the maximum input size, run:

```bash
$ CORES=<ncores> REPETITIONS=<nreps> \n  MAXSIZE=<nsize> ./benchmark.sh
```

The defaults are:

1. CORES: number of physical cores on your machine
2. REPETITIONS: 50
3. MAXSIZE: 30

The script requires that MAXSIZE ≥ 15.

**Note:** using MAXSIZE= 30 requires a machine with a large amount of memory. We run our experiments on a machine with 64GB of memory. 

```bash
$ CORES=12 REPETITIONS=100 MAXSIZE=30 ./benchmark.sh
```


E.5.2 Manual

Clone and build the repository:

$ git clone https://github.com/session-arr/session-arr

$ cd session-arr

To run our tool, run session-arrc using stack, with the .hs file as input:

$ stack exec session-arrc -- FFT.hs

The tool should output the list of functions found in module FFT.hs that are going to be compiled to C, and produce two files FFT.c and FFT.h. The interface file contains the type definitions and function signatures of the functions in FFT.c. Finally, compile main.c:

$ gcc FFT.c main.c -o bench -lpthread -lm

To configure the number of repetitions, recompile the benchmark as follows:

$ gcc FFT.c main.c -DREPETITIONS=<num_reps> \ -o bench -lpthread -lm

You may use run.sh to run the benchmark on a range of inputs. The usage is:

$ ./run.sh <num_cores> <max_size>

For example, ./run.sh 2 10 will run the benchmark with sizes $2^9$ and $2^{10}$. The maximum size must be > 9. To generate the graphs, you need measurements using at least 7 different sizes, i.e. size must be > 14.

Running each benchmark manually Pass a valid input size to bench, the output looks as follows (run in a 4-core machine):

$ ./bench $((2**17))
K: seq
mean: 0.039446
stddev: 0.000713
...
K: 4
mean: 0.011952
stddev: 0.000636
...

Save all execution times to files with the format described in §E.3.4, as follows:

$ mkdir data
$ echo "size: <sizeN>" > data/t_<num_cores>
$ ./bench <sizeN> >> data/t_<num_cores>

Ensure that there are measurements with at least $N > 14$ sizes.

Plotting the speedups: Navigate to examples/. The speedups can be plotted using scripts plotall.sh and plot.py, these will re-generate the graphs used in our paper. The usage is:

$ ./plotall.sh BENCHMARK_DIR CORES

This will generate the graphs for FFT run on 4 cores under examples/plots.

E.6 Evaluation and expected result

If you followed the experiment workflow, you should find under examples/plots a series of graphs with the speedups for each benchmark. To visualise them, we recommend copying them to a local directory, by running docker cp from outside the docker container:

$ docker cp /<NM>/home/cc20-artifact/session-arr/examples/plots /<DIR>

Here, <NM> is the container name obtained via docker ps -a, and <DIR> is the destination path.

Outcome When run on similar hardware to the one that we describe in the paper, following our workflow, comparable speedups and scalability to the ones that we reported in the paper should be observed.

Note: for more reliable results, execution should be done outside the docker container. Use the container to generate all C code, then copy it running docker cp from outside the container, as well as the necessary scripts run.sh, and proceed locally. If you decide to run the experiments locally, please check §E.3.3 and ensure that you have all required software.

E.7 Experiment customization

Several aspects can be customised in the benchmark source code, execution scripts and compilation options:

• Annotations to functions in the .hs files should produce different parallelisations.
• Files main.c can be compiled using different numbers of repetitions using -DREPETITIONS=<num_reps>.
• The script benchmark.sh can be run with such number of repetitions, to reduce execution times. The maximum input size for the benchmarks, and the number of cores can also be customised:

$ REPETITIONS=<num_reps> CORES=<cores> \ -o bench -lpthread -lm

$ ./benchmark.sh

E.8 Methodology

Submission, reviewing and badging methodology:

• http://cTuning.org/ae/submission-20190109.html
• http://cTuning.org/ae/reviewing-20190109.html
• https://www.acm.org/publications/policies/artifact-review-badging