Imperial College London

MENG INDIVIDUAL PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Session Arrows: A Session-Types Based Framework For Parallel Code Generation

Author:
Shuhao Zhang

Supervisor:
Prof. Nobuko Yoshida
Dr. David Castro-Perez

Second Marker:
Dr. Iain Phillips

June 18, 2019

Abstract

Parallel code is notorious for its difficulties in writing, verification and maintenance. However, it is of increasing importance, following the end of Moore's law. Modern programmers are expected to utilize the power of multi-core CPUs and face the challenges brought by parallel programs.

This project builds an embedded framework in Haskell to generate parallel code. Combining the power of multiparty session types with parallel computation, it creates a session typed monadic language as the middle layer and use Arrows, a general interface to computation as an abstraction layer on top of the language. With the help of the Arrow interface, we convert the data-flow of the computation to communication and generate parallel code according to the communication pattern between participants involved in the computation. Thanks to the addition of session types, not only the generated code is guaranteed to be deadlock-free, but also we gain a set of local types so that it is possible to reason about the communication structure of the parallel computation.

In order to show that the framework is as expressive as usual programming languages, we write several common parallel computation patterns and three algorithms to benchmark using our framework. They demonstrate that users can express computation similar to traditional sequential code and gain, for free, high-performance parallel code in low-level target languages such as C. Moreover, this framework is not limited to a standalone tool for parallel computation; we show the framework can act as a code generation backend for other data-flow based high-level parallel languages with an example. Benchmarks show the generated code can have up to 12X speedup on certain input sizes on a 32-core machine.

Acknowledgements

I would like to take the opportunity to thank my supervisor, Prof. Nobuko Yoshida for her invaluable advice and insight. Her brilliant course on session types is one of the main reasons why I want to pursue this project. I must thank my second supervisor, Dr. David Castro-Perez, for his incredible dedication, contagious cheerfulness and willingness to help me during this year, and for igniting my interest in functional programming. Every meeting with him is a delight, and I have learned a large amount of knowledge from him. It is hard to imagine to complete this project without their supports.

Contents

Chapter 1

Introduction

1.1 Motivation

Writing parallel software is not a trivial task. Parallel code is hard to write because it is usually done in low-level languages with verbose and non-idiomatic decorations, hard to debug because machines, where the code is written, are usually different from machines where the code is intended to run and hard to maintain and reuse. This is because even though the underlying algorithms are not changed, multiple versions of the parallel code are needed to tackle various platform and evolution of architectures.

There are many on-going pieces of research aimed at helping programmers to write correct parallel programs smoothly. A common approach is to develop a high-level language and compile programs in this language to parallel code. There are many high-level frameworks for parallel programming (e.g. algorithmic skeletons [1], domain-specific languages for parallelism [2] or the famous MapReduce parallel model [3]). An example is to use arrow terms (see in Section 2.1) to describe data-flow implicitly and hence generate parallel code [4].

The workflow of writing parallel code has evolved from writing it directly in the target platform to writing software in a high-level language designed for parallel computation and then compiling to the target platform. In this project, we present a method to improve the backend of parallel code generation by introducing a monadic domain-specific language: SPar to act as a bridge between high-level and target low-level parallel languages.

This specific language needs to be general enough so that it supports multiple high-level parallel programming frameworks. It can be used to generate different parallel code such as Message Passing Interface (MPI) and Cuda. Moreover, we developed a simulator to aid debugging parallel programs.

Our framework couples with multiparty session type (MPST) [5]. It is done by inferring session types for the computation of each participant. We can take advantages of the collection of local types to enable aggressive optimization but ensuring code correctness and meaningful static analysis; e.g. cost modelling for parallel programming according to properties of MPST.

1.2 Contributions

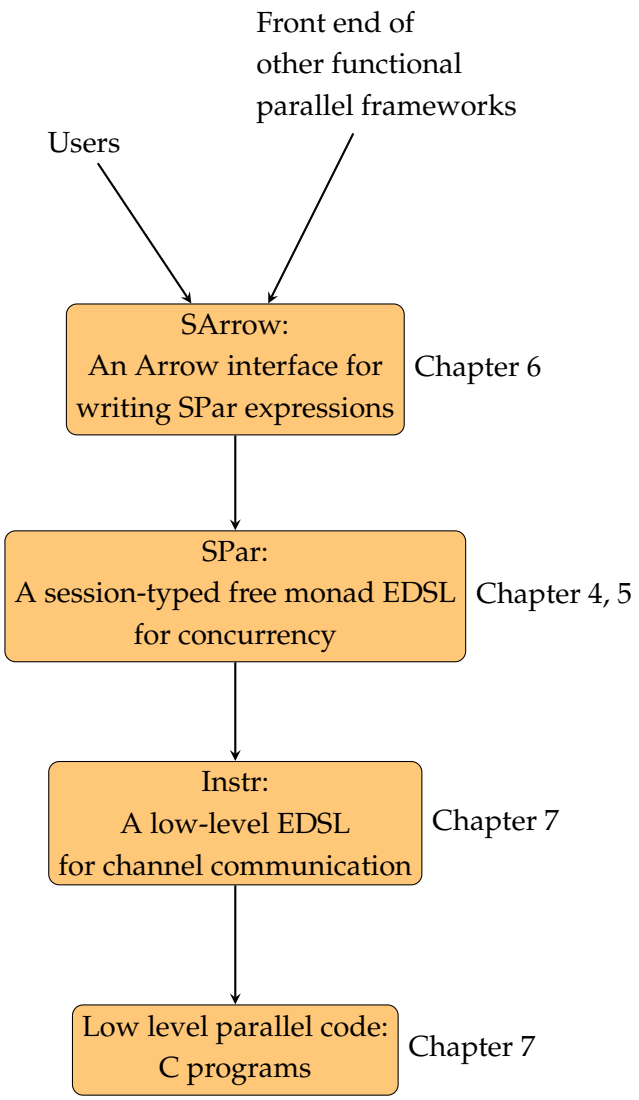


Figure 1.1: Visualization of the workflow

The result of the project is an embedded high-level framework in Haskell that is capable of generating low-level parallel code. The major contributions are:

1. **Session-typed intermediate language.** We create **SPar**: a session typed free monad EDSL for message-passing concurrency. SPar is built using free monad technique, and it contains primitive operations for channel communication as well as our representation of computation. This language can be typed by local types, and hence, we can apply multiple results from the multiparty session types to our framework, especially in terms of safety of generated code and reasoning of communication patterns.

2. **Intuitive user interface.** One innovation of this project is that we apply the Arrow interface for users to express parallel computations. Arrow is a general interface for computations (see in Section 2.1). We call our interface **SArrow**: an Arrow interface

for writing SPar expressions. It is an abstraction layer on top of SPar, which hides communication primitives from users so that users can express parallel algorithms similar to what they would write for sequential programs.

3. **Multiple backends.** We create a backend to generate parallel C code from SPar expressions. The core of the backend is **Instr:** a low-level EDSL for channel communication as well as operations on variables and resources management. It is independent of target languages. This means that we can support multiple target languages with ease without re-implementing multiple backends. In addition to the code generation backend, we implement an interpreter backend in Haskell for experimenting and fast verification.

4. **Evaluations.** Finally, we show the expressive power of the framework by implementing several common computation patterns and three algorithms using our interface. We evaluate the performance of the generated code from the algorithms on high-performance computers.

The Figure 1.1 summaries the workflow of the framework visually. The main principle supporting the framework is that **we convert data-flow into communication and from the communication pattern, we gain parallel code**. The results of expressing computation in the framework are 1) compilation to efficient deadlock-free low-level parallel programs and 2) a set of local types to reason the structure of the parallel computation.

At the end of the project, we have discovered two use case of the framework. The primary application is a stand-alone tool to generate parallel C code from programs written in Haskell, and another is a backend for other data-flow based parallel frameworks.

1.3 Report outline

Chapter 2 gives an overview of the background and related research. We present the syntax and the semantics of SPar in Chapter 4 followed by Chapter 5 introducing the implementation aspect of SPar, like session typing and developing the interpreter. Chapter 6 demonstrates the Arrow interface with examples of parallel patterns formed by the interface and justification of the interface satisfying the arrow laws. The discussion about some implementation specific issues like role allocation is also contained in Chapter 6. In Chapter 7, we show the code generation backend and discuss our solutions to challenges when compiling to C, i.e. the problem of representing polymorphic algebraic data structures in C. Chapter 8 explains our benchmark and shows the performance of the generated code. This chapter can also be regarded as a tutorial on how to use the framework. Finally, we conclude with potential future improvements and remarks on this project. We also include the generated C code in the appendix for the curious readers.

Chapter 2

Background

This section is an overview of techniques that influence the design choices of our framework. First of all, we give an overview of techniques that can be applied in high-level parallel frameworks: arrows (Section 2.1) and recursion schemes (Section 2.2). We then introduce several techniques for message-passing concurrency: multiparty session types (Section 2.3) and monadic languages for concurrency (Section 2.4). In the end, we introduce free monads (Section 2.5), a technique valuable in implementing embedded domain-specific languages (EDSL).

2.1 Arrows

Arrow is a general interface to describe computation. It can ease the process of writing structured code suitable for parallelizing. It also demonstrates a common feature of the frameworks: parallelizability is empowered by underlying implicit but precise data-flow. On the other hand, converting to low-level message-passing code, which requires programmers to define communication using message-passing function and primitives, makes the data-flow explicit.

2.1.1 Definition

Listing 1 shows the Arrow definition in Haskell. Intuitively, an arrow type `y a b`(that is, the application of the parameterized type `y` to the two parameter types `b` and `c`) can be regarded as a computation with input of type `b` and output of type `b`[6]. Visually, arrows are like pipelines (shown in Figure 2.1). In Haskell, an arrow `y` is a type that implements the following interface (type classes in Haskell are roughly interfaces). `arr` converts an arbitrary function into an arrow. `>>>` sequences two arrows (illustrated in Figure 2.1b). Taking two input, `first` apply the arrow to the first input while keeping the second untouched (Figure 2.1a). Conversely, `second` modifies the second input and keeps the first one unchanged. `***` applies two arrows to two input side by side (Figure 2.1d). `&&&` takes one input and applies two separate arrows to the input and its duplications (Figure 2.1c).

The simplest instance of arrow class is the function type (shown in Listing 2). It

is worth noticing that only `arr` and `***` need to be implemented. The rest of functions in the arrow type class can be defined in terms of the two functions. For example, `f &&& g = (f *** g) . arr (\b -> (b, b))` and `first = (*** id)`

```
class Arrow y where
  arr :: (a -> b) -> y a b
  first :: y a b -> y (a, c) (b, c)
  second :: y a b -> y (c, a) (c, b)
  (***) :: y a c -> y b d -> y (a, b) (c, d)
  (&&&) :: y a b -> y a c -> y a (b, c)
```

Listing 1: Arrow class in Haskell

```
instance Arrow (->) where
  arr f = f
  (***) f g ~(x,y) = (f x, g y)
```

Listing 2: $(\rightarrow)$ instance of Arrow class

2.1.2 Example: Calculate the mean

Consider the function to calculate the mean from a list of floating point numbers, we present the implementation using arrows compared with a point-free Haskell definition. Implementation using arrows can be regarded as point-free programming. Point-free programming is programming paradigm where function definitions only involve combinators and function composition without mentioning variables [8].

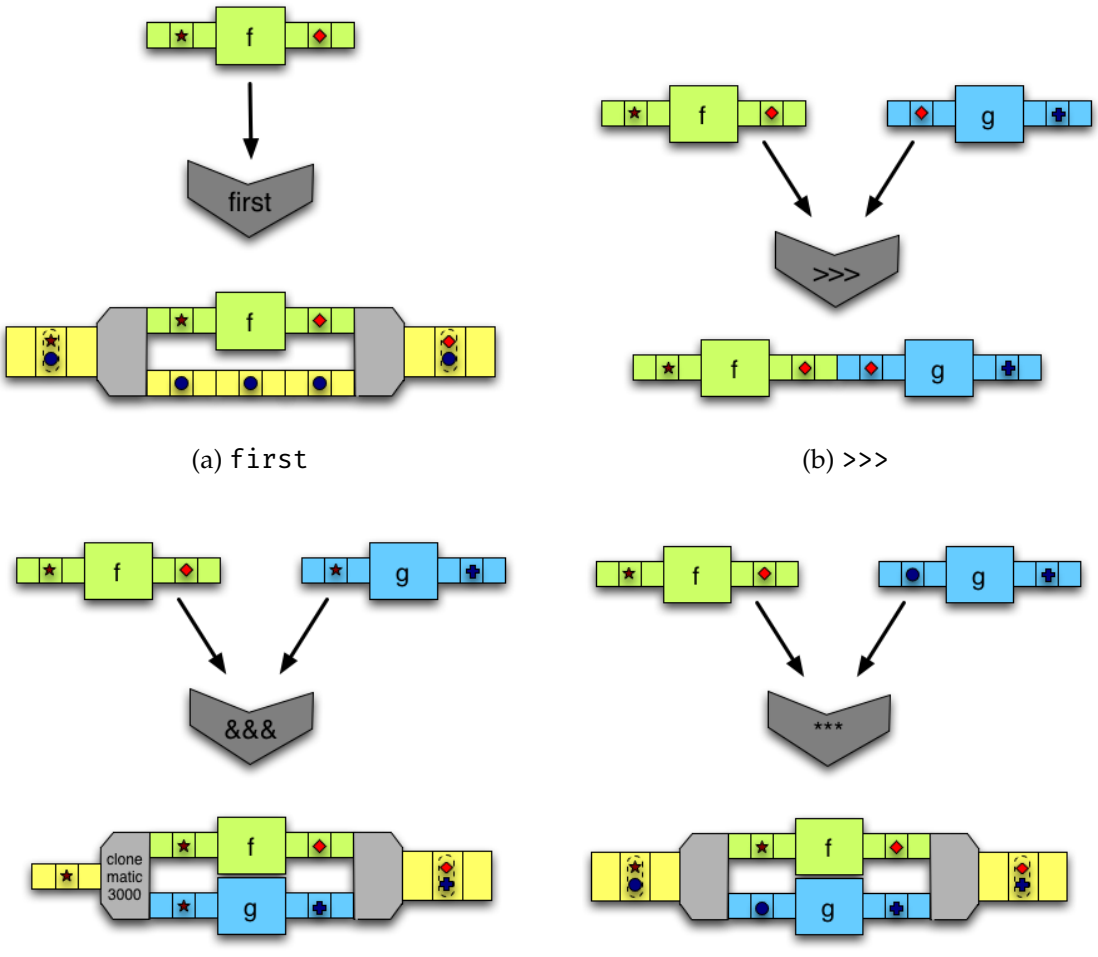
```
mean :: [Float] -> Float
mean xs = sum xs / (fromIntegral . length) xs

mean' :: [Float] -> Float
mean' = (sum &&& (length >>> fromIntegral)) >>> uncurry (/)
```

The arrows implementation can be visualized in Figure 2.2.

```
mean'' :: [Float] -> Float
mean'' = liftM2 (/) sum (fromIntegral . length)
```

The above code snippet is the more traditional approach form of point-free mean function in Haskell. Arrows are not the only way to form point-free programs. We can argue this form of point-free function is more difficult to understand compared to arrows because it involves knowledge of monads (`liftM2`) and does not map to the intuitive data-flow.

The simple example demonstrates that arrows combinators make writing point-free programs easier. Arrows unite the implementation of the algorithm and data-flow in the algorithm.



(a) `first` (b) `>>>`

(c) `&&&` (d) `***`

Figure 2.1: The visual representations of arrow combinators[7]

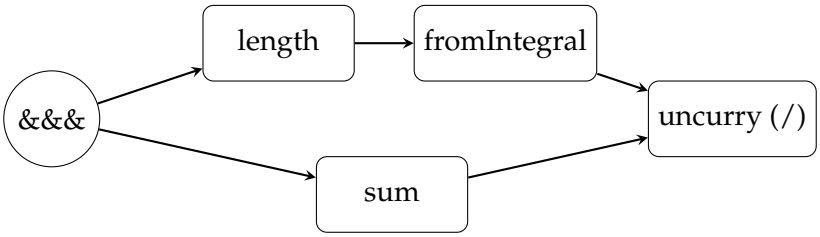
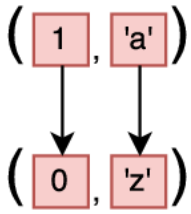


Figure 2.2: Visualization of mean'

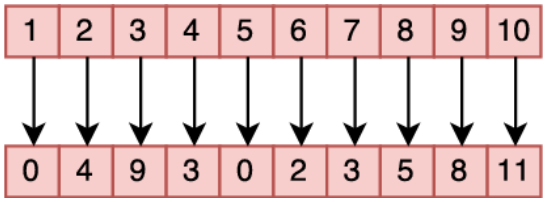
2.1.3 Application in parallel computation

From the previous example, the data-flow of programs written using arrow combinators can be easily visualized (shown in Figure 2.1). It is intuitive to recognize that the clean separation between the flow of data and actual computation will be useful in generating parallel code. Indeed, arrow describes data-flow implicitly, and it is an example of the so-called algebraic pattern. Many works has been done to generate parallel code from algebraic patterns [4, 9, 10]. In particular, details of [10] are introduced in Chapter 3.

We will use some figures to explain the idea behind arrows as a framework for parallel computation. For example, as shown in the Figure 2.3a, `f *** g` means computations of `f` and `g` happen in parallel. Figure 2.3b shows that it can be used to implement parallel map in terms of arrows, taking an arrow computation `arr a b` and returning a list of computation in parallel `arr [a] [b]`.



(a) Visualization of parallel `***` [4]



(b) Visualization of parMap [4]

2.2 Recursion Schemes

Recursion schemes are patterns for expressing recursive functions. In particular, they are high order function abstracting common patterns of recursion.

2.2.1 Definition

We will introduce three typical recursion schemes: catamorphisms representing folds, anamorphisms representing unfolds and hylomorphisms representing divide-and-conquer algorithms (seen in Listing 4). Recursion schemes express recursion with the help of data structures that mirror the control structure of the recursion.

Anamorphisms takes a function `a -> f a` (called the co-algebra) and a value `a` and return the `Fix f`. Using `TreeF` as an example, anamorphisms takes a single value with type a and applies the co-algebra to the value. It continues to apply itself to the branches

```haskell
newtype Fix f = Fix { unfix :: f (Fix f) }

data TreeF a =
    Node a a
  | Leaf int
  | Empty
  deriving Functor

type Tree = Fix TreeF
```

Listing 3: Definition of fix point of data structures

of the `TreeF` recursively and finally expands a single value to a complete tree. Intuitively, anamorphism unfolds a single value to a complicated data structure in a top-down way.

Catamorphisms is the reverse of anamorphisms, folding a data structure to a single value in a bottom-up way. It takes a function `f a -> a` (called the algebra) and data `Fix f` to fold and return a single value `a`. Catamorphisms and anamorphisms describe the process globally while co-algebra and algebra capture what happened locally. The elegant part is while co-algebra and algebra do not involve with any recursion data structure (`TreeF` is not recursive), catamorphisms can consume recursive data structures and anamorphism can build recursion data structures.

Hylomorphisms applies anamorphism followed by catamorphisms. It is the most common pattern to use. We will use an example to illustrate its usefulness. It can be thought of as an abstract divide and conquer algorithm.

```haskell
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg

cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unfix

hylo :: (f b -> b) -> (a -> f a) -> b -> a
hylo g f = f . fmap (hylo f g) . g
```

Listing 4: Recursion schemes in haskell

2.2.2 Example: Merge sort

We can write merge sort recursively. First of all, we split the list in half and then apply the merge sort recursively to both parts and finally we merge two lists into a single list.

To write merge sort in terms of recursion scheme, we need to define the recursive structure to represent the control structure. By the definition of merge sort, this structure must have a case with two branches, a base case representing a singleton list and a

base case representing an empty list hence this structure is the `TreeF` we defined above. Splitting a list is a co-algebra while merging is an algebra. We use the hylomorphisms to combine them and then get the implementation of merge sort (seen in Listing 5).

```
mergeSort :: [Int] -> [Int]
mergeSort = hylo merge split where
  merge Empty      = []
  merge (Leaf c)   = [c]
  merge (Node l r) = usualMerge l r

  split []  = Empty
  split [x] = Leaf x
  split xs  = Node l r where
    (l, r) = splitAt (length xs div 2) xs
```

Listing 5: Merge sort using hylomorphisms

2.3 Multiparty session types

In complicated distributed systems, participants agree on a protocol, specifying the type and direction of data exchanged. Multiparty session types are a branch of behavioral types specifically targeted at describing protocols in distributed systems based on asynchronous communication [5]. They are a type formalism used to model communication-based programming by codifying the structure of the communication. The evolution of computing from the era of data processing to the era of communication witnessed the growth and significance of the theory of session types.

The theory of multiparty session types contains three main elements. Global types (seen in Section 2.3.1), local (session) types and processes. Processes are the concrete descriptions of the behavior of the peers involved in the distributed system [5] using a formal language. The most used and the original language is π-calculus [11]. The coming sections are an intuitive introduction of session types by examples.

2.3.1 Global types and local types

A global type is at the most abstract level, describing a communication protocol from a neutral viewpoint between two or more participants[5]. The syntax of global types is shown in Figure 2.4 and an example of global types is shown in Figure 2.6.

Local types or session types characterise the same communication protocol as the global type, but from the viewpoint of each peer [5]. Each process is typed by a local type. The syntax of local types is shown in Figure 2.5 and an example of local type is shown in Figure 2.7.

The relationship between global types and local types are established by the projection operator (seen in the Section 2.3.1.1), and a type system performs syntactic checks,

ensuring that processes are typed by their corresponding local types. Hence, at the compile time, three important properties follow [5].

- **communication safety**: Mismatches between the types of sent and expected messages, despite the same communication channel is used for exchanging messages of different types, do not exist [5].
- **protocol fidelity**: The interactions that occur are accounted for by the global type and therefore are allowed by the protocol [5].
- **progress**: Every message sent is eventually received, and every process waiting for a message eventually receives one [5].

We will learn that these properties are valuable not only in the distributed system but also in the domain of parallel computing in Section 2.3.2.

$G ::=$	Global types
$p \to q : \langle S \rangle.G$	Value exchange
$p \to q : \langle T \rangle.G$	Channel exchange
$p \to q : \{l_i : G_i\}_{i \in I}$	Bracnhing
$\mu \mathbf{t}.G \mid \mathbf{t} \mid \text{end}$	Recursion/End

Figure 2.4: Global types

$T ::=$	Session types/local types
$!\langle p, S \rangle.T$	Send value
$!\langle p, T \rangle.T$	Send channel
$?(p, T).T$	Channel Receive
$?(p, S).T$	Sorts Receive
$\oplus \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Selection
$\&(p, \{l_i : T_i\}_{i \in I})$	Bracnhing
$\mu \mathbf{t}.T \mid \mathbf{t} \mid \text{end}$	Recursion/End

$S ::=$	Sorts
bool | nat | string	
$\ldots$	

Figure 2.5: Session types/local types

2.3.1.1 Projection between global types and local types

Projection is the formalization of the relationship between global and local types. It is an operation extracting the local type of each peer from the global type [5]. The definition of projection is shown in Figure 2.8.

As an example, a projection of global type in Figure 2.6 is

$$G \upharpoonright 0 = !\langle 1, \text{string} \rangle; ?(1, \text{string}); \&(1, \{\text{accept} : ?(1, \text{string}); !\langle 2, \text{string} \rangle; ?(2, \text{int}), \text{reject} : \text{end}\})$$

1. Customer(0) sends an order number to Agency(1), and the Agency sends back a quote to the customer.
2. If Customer is happy with the price, then Customer selects accept option and notifies Agency.
3. If Customer thinks the price is too high, then Customer terminate the trade by selecting reject.
4. If accept is selected, the Agency notifies both Customer and Agency2(2).
5. Customer sends an address to Agency2 and Agency2 sends back a delivery date.

$$
\begin{aligned}
G = \quad & \\
& 0 \to 1 : \quad \langle \text{string} \rangle . \\
& 1 \to 0 : \quad \langle \text{int} \rangle . \\
& 0 \to 1 : \{ \quad \text{accept} : \\
& \qquad\qquad 1 \to \{0, 2\} : \langle \text{string} \rangle . \\
& \qquad\qquad 0 \to 2 : \langle \text{string} \rangle . \\
& \qquad\qquad 2 \to 0 : \langle \text{int} \rangle . \text{end}, \\
& \qquad\qquad \text{reject} : \text{end} \}
\end{aligned}
$$

Figure 2.6: An example of a protocol described by global types G

$$
\begin{aligned}
S \triangleq \mu t.(\&\{&\text{balance} : ![\text{nat}]; t, \\
&\text{deposit} : ?[\text{nat}]; ![\text{nat}]; t, \\
&\text{exit} : \text{end}\})
\end{aligned}
\qquad\qquad
\begin{aligned}
C \triangleq \oplus\{&\text{balance} : ?[\text{nat}]; \text{end}, \\
&\text{deposit} : ![\text{nat}]; ?[\text{nat}]; \text{end}\}
\end{aligned}
$$

Figure 2.7: Session types of client and server end point of a ATM service

$$
(\mathtt{p} \to \mathtt{p}' : \langle U \rangle . G') \upharpoonright \mathtt{q} = \begin{cases} !\langle \mathtt{p}', U \rangle . (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?(\mathtt{p}, U) . (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}', \\ G' \upharpoonright \mathtt{q} & \text{otherwise}. \end{cases}
$$

$$
(\mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i \in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus \langle \mathtt{p}', \{l_i : T_i\}_{i \in I} \rangle & \text{if } \mathtt{q} = \mathtt{p} \\ \&(\mathtt{p}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}) & \text{if } \mathtt{q} = \mathtt{p}' \\ G_{i_0} \upharpoonright \mathtt{q} & \text{where } i_0 \in I \text{ if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \neq \mathtt{p}' \\ & \text{and } G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q} \text{ for all } i, j \in I. \end{cases}
$$

$$
(\mu \boldsymbol{t} . G) \upharpoonright \mathtt{q} = \begin{cases} \mu \boldsymbol{t} . (G \upharpoonright \mathtt{q}) & \text{if } G \upharpoonright \mathtt{q} \neq \boldsymbol{t}, \\ \mathsf{end} & \text{otherwise}. \end{cases} \qquad \boldsymbol{t} \upharpoonright \mathtt{q} = \boldsymbol{t} \qquad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}.
$$

Figure 2.8: The definition of projection of a global type G onto a participants q[5]

2.3.1.2 Duality of session types

In binary session types where all protocals are pairwise, duality formalises the relationship between the types of opposite endpoints. For a type T, its dual or co type, written $\bar{T}$ is defined inductively as in Figure 2.9.

$$\begin{array}{rclcrclcrcl}
\overline{?[\widetilde{S}];\, T} & = & ![\widetilde{S}];\, \overline{T} & \quad & \overline{\oplus\{l_i : T_i\}_{i\in I}} & = & \&\{l_i : \overline{T_i}\}_{i\in I} & \quad & \overline{?[T];\, T'} & = & ![T];\, \overline{T'} \\
\overline{![\widetilde{S}];\, T} & = & ?[\widetilde{S}];\, \overline{T} & & \overline{\&\{l_i : T_i\}_{i\in I}} & = & \oplus\{l_i : \overline{T_i}\}_{i\in I} & & \overline{![T];\, T'} & = & ?[T];\, \overline{T'} \\
\overline{\texttt{end}} & = & \texttt{end} & & \overline{\mu t.T} & = & \mu t.\overline{T} & & \overline{t} & = & t
\end{array}$$

Figure 2.9: Inductive definition of duality

Duality is essential for checking type compatibility. Compatible types mean that each common channel k is associated with complementary behavior: this ensures that the interactions on k run without errors.

In order to apply duality into multiparty session types in which more than two participants are allowed, the partial projection operation (seen in [5]) from multiparty session type to binary session type was introduced to allow reusing the definition of duality after applying the partial projection.

2.3.2 Applications in parallel computing

Multiparty session types not only have rich applications in distributed systems but also value in the domain of parallel computation.

Existing work[12] has shown how to generate Message Passing Interface (MPI) [13] programs using session types. Users describe the communication topology as a skeleton using a protocol language, which is type checked by session types. After that, an MPI program is generated by merging the skeleton and user-provided kernels for each peer. The parallel code obtained in this way is guaranteed to be deadlock-free and progressing.

2.4 Message passing concurrency

This section introduces some interfaces for message passing concurrency from the primitive case: channel to more advanced one: monad for message passing concurrency.

For simplicity, they are represented in Haskell, but in general, most languages can implement similar interfaces.

2.4.1 Primitives for message-passing concurrency

In Section 2.3, channels are bi-directional and used for communication between two parties. In Haskell, channel primitives are represented in Listing 6. However, just using these primitives cannot guarantee progress or communication safety. For example, a program that has one thread writing channel once combined with another thread reading channel

twice is type-correct but will cause deadlock. Many kinds of research to encode MPST using Haskell's type system are presented in [14] so that an (MPST) type-correct Haskell program assures progress, communication safety and session fidelity.

```haskell
data Chan a
    newChan :: IO (Chan a)
    writeChan :: Chan a -> a -> IO ()
    readChan :: Chan a -> IO a
    dupChan :: Chan a -> IO (Chan a)
```

Listing 6: Channel primitives in Haskell

2.4.2 Concurrency Monads

The work done by [15] constructs a monad to express concurrent computation. The definition is in Listing 7. `Action` is the algebraic datatype representing basic concurrency primitives. `Atom`, the atomic unit of computation, is a computation (wrapped in the `IO` monad) followed by an action. `Fork` is two parallel action. `Stop` is the termination of an action. type `C` is a special case of the continuation monad. The continuation monad is an encapsulation of computations in continuation-passing style (CPS)[1]. SO `C a` is a CPS computation that produces an intermediate result of type `a` within a CPS computation whose final result type is `Action`. With the help of the monad `C`, sequencing and composing actions can use monadic bind.

```haskell
data Action =
    Atom (IO Action)
  | Fork Action Action
  | Stop

newtype C a = C { runC :: (a -> Action) -> Action }

instance Monad C where
    (>>=) :: C a -> (a -> C b) -> C b
    m >>= f  = C $ \k -> runC m (\v -> runC (f v) k)
    return :: a -> C a
    return x = C $ \k -> k x
```

Listing 7: The definition of concurrency monad

The idea is using continuation to represent the "future" so that computation can pause and resume as well as expressing sequential computation. Atom wraps the actual computation and Fork is responsible for spawning threads. In addition, in order to write

[1] In continuation-passing style function result is not returned, but instead is passed to another function, received as a parameter (continuation)[16]

programs in a monadic way easier, some helper functions are defined (shown in Listing 8). `atom` lifts an IO computation to `C`. And `fork` takes a computation in `C` and return a `C` which involves the `Fork` action. Given a `C a`, `action` gives the result of running the CPS computation. We use `\_. Stop` to represent the final continuation (`Stop` action is the last action). An example of programme written in the concurrency monad is shown

```
atom :: IO a -> C a
atom m = C $ \k -> Atom $ do
    r <- m
    return $ k r

fork :: C () -> C ()
fork m = C $ \k -> Fork (runC m (const Stop)) (k ())

action :: C a -> Action
action m = m (\_. Stop)
```

Listing 8: Helper functions

below.

```
example :: C ()
example = do
  atom $ putStrLn "Hello"
  name <- atom getLine
  fork $ atom $ putStrLn "World"
  atom $ putStrLn name
```

We can easily define a round-robin scheduler for programs in this monad. We can regard a list of action as a queue of threads that are running concurrently. `schedule` will pattern match on the head of the list. If it is `Atom` then the scheduler will run the computation (seen `a <- ioa` at ⑦) and pause its remaining computation and put it at the end of the thread queue (seen at ⑧). If it is `Fork` then the scheduler will spawn the thread and put the new thread and the current thread to the bottom of the queue (seen at ⑨). Finally, If it is `Stop` then it means this thread has finished, and the scheduler will resume with the rest of threads in the queue. For example, to run the above example, we call `schedule [action example]`.

```
schedule :: [Action] -> IO ()
schedule [] = return ()
schedule (a:as) = sched as a

sched :: [Action] -> Action -> IO ()
sched as (Atom ioa) = do
    a <- ioa ⑦
    schedule $ as ++ [a] ⑧
```

```haskell
sched as (Fork a1 a2) = schedule $ as ++ [a2, a1] ⑨
sched as Stop = schedule as
```

The concurrency monad can be extended to support many features. For example, work done by [17] modifies the definition of Action as well as implements a work-stealing parallel scheduler (seen in Listing 9) to build a monad for parallel computation.

Besides, extending the concurrency monad to monad for message-passing concurrency can be done by adding channel primitives like newChan, writeChan and readChan into the Action. Since channel primitives are possible to represent in this monad, we naturally think of its prospect in connecting with MPST (will be discussed in Section 4.3.2)

```haskell
newtype IVar a = IVar (IORef (IVarContents a)) ①
data IVarContents a = Full a | Blocked [a -> Action.]

data Action .=
    Fork Action Action
  | Stop
  | forall a . Get (IVar a) (a -> Action) ⑦
  | forall a . Put (IVar a) a Action ⑧
  | forall a . New (IVar a -> Action)
```

Listing 9: Par Monad

① Parent threads and child threads communicate data via IVar

⑦ Get operation blocks when the underlying IVarContents is Blocked

⑧ Put operation updates the underlying IVarContetns to Full with the result a and resume the list of blocking threads by applying a to the continuation.

We draw inspiration from this monad to design our intermediate language.

2.5 Free monad

Free monad [18] is a concept from category theory. Intuitively, a free monad as a programming abstraction is a technique for implementing EDSLs, where a functor represents basic actions of the EDSL and the free monad of this Functor provides a way to sequence and compose actions. Speaking of the advantages, we are particularly interested in its benefits in flexible interpretations which will be illustrated by an example (Section 2.5.2) and discussed further (Section 2.5.3).

2.5.1 Definition

In practice, a free monad in Haskell can be defined as an algebraic data type(ADT) (shown in Listing 10). `Free f` is the monad produced given a functor `f`. `Free` has two

type constructors: `Pure` and `Free`. `Monad (Free f)` is the Haskell implementation of the Monad interface for `Free f`. Many useful helper functions are derived from the simple definition of the free monad (shown in Listing 11). `liftF` lift the functor to its free monad representations. `freeM` maps a natural transformation of functor (`f a -> g a`) to the natural transformation of their free monad versions. Given `m` is a monad, `freeM` is a special case of interpreting `Free m a`: to the `m` monad itself. Finally, `interpret` shows the power of free monad. We can interpret the free monad version of a functor `f` to any monad `m` given a natural transformation from `f` to `m`.

```haskell
data Free f a
  = Pure a
  | Free f (Free f a)

instance Functor f => Monad (Free f) where
  return = pure
  (Pure x) >>= fab = fab x
  (Free fx) >>= fab = Free $ fmap (>>= fab) fx
```

Listing 10: Free monad in Haskell

```haskell
liftF :: Functor f => f a -> Free f a
liftF = Free . fmap Pure

freeM :: (Functor f, Functor g)
    => (f a -> g a)
    -> (Free f a)
    -> (Free g a)
freeM phi (Pure x ) = Pure x
freeM phi (Free fa) = Free $ phi (fmap (freeM phi) fa)

monad :: Monad m => Free m a -> m a
monad (Pure x  ) = pure x
monad (Free mfx) = mfx >>= monad

interpret :: (Functor f, Monad m)
    => (f a -> m a)
    -> (Free f a -> m a)
interpret phi = monad . freeM phi
```

Listing 11: Helper functions based on free monad

2.5.2 Example

Free monad is useful in interpreting an abstract syntax tree (AST). In order to apply the free monad technique to a given AST, we can follow a routine [19].

1. Create an AST, usually represented as an ADT
2. Implement functor for the ADT
3. Create helper constructors to Free ADT for each type constructor in ADT by liftF
4. Write a monadic program using helper constructors. It is essentially a program written in EDSL operations.
5. Build interpreters for Free ADT by interpreting
6. Interpret the program by the interpreter.

We will demonstrate the above procedure by a made-up example. We would like to build a simple EDSL for getting customers' name and greeting customers. First of all, we build a functor `GreetingF` to represent the basic operations: getting the name and greeting. Then we wrap the functor with `Free` constructor so that a program written in our EDSL can be regarded as a Haskell expression with type `Free GreetingF a`.

```haskell
data GreetingF next
  = Getname (String -> next)
  | Greet String next
  deriving Functor

type Greeting = Free GreetingF
```

Then we create helper functions of Greeting using liftF.

```haskell
getName = liftF $ Getname id
greet str = liftF $ Greet str ()
```

Then we can write a simple program using operations provided by Greeting.

```haskell
exampleProgram :: Greeting ()
exampleProgram = do
    a <- getName
    greet a
    b <- getName
    greet b
```

Then we can easily implement an interpreter for the example program

```haskell
goodMorningInterpreter :: Greeting a -> IO a
goodMorningInterpreter = interpret helper
    where
        helper (Getname next) =
          fmap next getLine
        helper (Greet str next) =
          putStrLn ("Good morning " ++ str) >> return next
```

Finally, execute the program.

```
ghci:> goodMorningInterpreter exampleProgram
Tom
Good morning Tom
Mary
Good morning Mary
```

2.5.3 Applications

As illustrated by the example (Section 2.5.2), a free monad decouples the abstract syntax tree of domain-specific language and the interpreter. Interpreters with different purposes can be implemented without changing the syntax.

In the project, we apply free monads to the intermediate language so not only we make the language monadic for free but also benefit from decoupling the interpreter and the syntax to implement different interpreters, e.g., Simulator, code generators to different platforms easily.

Chapter 3

Alg : Algebraic Functional Language

Algebraic Functional Language (Alg) is an example of a high-level language to generate parallel code, proposed in the paper [10]. The work done by [10] also proposes a method to do the code generation. Part of this project is about implementing an alternative code generation backend for this language. In the evaluation section, we will compare the speed of the generated code of our method against the original method. We will give an overview of the language in this section.

3.1 Alg

3.1.1 Syntax

$F_1, F_2 ::=$	
I	Identity functor
Kt	Constant functor
$F_1 + F_2$	Sum functor
$F_1 \times F_2$	Product functor

$t_1, t_2 ::=$	Type
$() \mid \text{int} \mid \ldots$	Primitive types
$a \to b$	Function types
$a + b$	Sum types
$a \times b$	Product types
$F\ t_1$	Functor types
$\mu.F$	Recursive types

$e_1, e_2 ::=$ Expression

$f \mid v \mid \text{const}\ e \mid e_1 \circ e_2 \mid \pi_i \mid e_1 \vartriangle e_2 \mid e_1 \triangledown e_2 \mid l_i \mid F\ e \mid \text{in}_F \mid \text{out}_F \mid \text{rec}_F\ e_1\ e_2$

Figure 3.1: Syntax of Alg language

```
newtype L = K () + K Int * I
type List = Rec L
```

Listing 12: Type of integer list in PAL

The syntax of Alg is shown in Figure 3.1. In terms of the syntax of expressions, f represents atomic functions which are functions of which we only know their types [10].

The presence of atomic functions is important in the code generation, which we will discuss in the later chapter. v is a primitive value like integer 1.F represents functor and a, b are types. Besides primitive types, function types and the recursive type constructor μ, Alg uses four functors to form more types hence representing data structures by composing them. For example, a list of integers is expressed in Listing 12. Alg is a point-free language.

$$\frac{f : a \to b \in \Gamma}{\vdash f : a \to b}$$

$$\frac{\vdash e : a}{\vdash \text{cons } e : b \to a}$$

$$\frac{}{\vdash \text{id} : a \to a}$$

$$\frac{}{\vdash \text{in}_F : F\ \mu F \to \mu F}$$

$$\frac{}{\vdash \text{out}_F : \mu F \to F\ \mu F}$$

$$\frac{\vdash e_1 : b \to c, \quad \vdash e_2 : a \to b}{e_1 \circ e_2 : a \to b}$$

$$\frac{i \in [1, 2]}{\pi_i : a_1 \times a_2 \to a_i}$$

$$\frac{i \in [1, 2]}{l_i : a_i \to a_1 + a_2}$$

$$\frac{\vdash e_1 : a \to b, \quad \vdash e_2 : a \to b}{e_1 \vartriangle e_2 : a \to b \times c}$$

$$\frac{\vdash e_1 : a \to c, \quad \vdash e_2 : b \to c}{e_1 \triangledown e_2 : a + b \to c}$$

$$\frac{\vdash e : a \to b}{F\ e : F\ a \to F\ b}$$

$$\frac{\vdash e_1 : F\ b \to b, \quad \vdash e_2 : a \to F\ a}{\text{rec}_F\ e_1\ e_2 : a \to b}$$

Figure 3.2: Typing rules for Alg

The typing rules for Alg are expressed in Figure 3.2 and the semantics for Al is shown in Figure 3.3.

An important feature of Alg is the lack of usual control flow like if a branch or while loop to build algorithms, instead, it uses the flow of transformation of data structures to replace conventional control flow. For example, $\triangledown$ combinator is the case operation whose types is $a+b \to c$. It can be seen as an analogy of if branch in normal programming languages. $\vartriangle$ combinator represents split operation. More importantly, the combinator rec_F uses the idea of recursion schemes (explained in Section 2.2) to build divide-and-conquer algorithms.

To summarize, algorithms in Alg are represented as a series of transformations of data, making it easy to transform the Alg programs to programs in arrows mechanically since arrows also express the flow of data and their transformations naturally. This property allows us to generate parallel code without burdens.

Constant, Identity and Composition $\quad \mathsf{const}\ e = \lambda x.\ e \quad \mathsf{id} = \lambda x.\ x \quad e_1 \circ e_2 = \lambda x.\ e_1\ (e_2\ x)$

Products

$\pi_i = \lambda(x_1, x_2).\ x_i \ \text{ if } i \in [1,2] \quad e_1 \vartriangle e_2 = \lambda x.\ (e_1\ x,\ e_2\ x) \quad e_1 \times e_2 = (e_1 \circ \pi_1) \vartriangle (e_2 \circ \pi_2)$

Coproducts

$\iota_i = \lambda x.\ inj_i\ x \quad e_1 \triangledown e_2 = \lambda x.\ e_i\ y \ \text{ if } x = inj_i\ y \quad e_1 + e_2 = (\iota_1 \circ e_1) \triangledown (\iota_2 \circ e_2)$

Functors

$\mathsf{I}\ a = a \quad \mathsf{K}a\ b = a \quad (F_1 \times F_2)\ a = F_1\ a \times F_2\ a \quad (F_1 + F_2)\ a = F_1\ a + F_2\ a$

$\mathsf{I}\ e = e \quad \mathsf{K}a\ e = \mathsf{id} \quad (F_1 \times F_2)\ e = F_1\ e \times F_2\ e \quad (F_1 + F_2)\ e = F_1\ e + F_2\ e$

Recursion

$\mathsf{in}_F = \lambda x.\ in_F\ x \quad \mathsf{out}_F = \lambda x.\ y \ \text{ where } \ in_F\ y = x \quad \mathsf{rec}_F\ e_1\ e_2 = f \ \text{ where } f = e_1 \circ F\ f \circ e_2$

Figure 3.3: Semantics of Alg expression[10]

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

Listing 13: Merge sort in Alg

By making the hylomorphism as a built-in combinator rec_F, Alg can express merge sort similarly as the example shown in Section 2.2.2. We use the functor $T = K() + Ka + I \times I$ to be substitute the functor F in rec_F and the type Ls to represent a list of elements whose type is a. We also have two atomic function spl : $Ls \to T\ Ls$ and mrg : $T\ Ls \to Ls$. Finally we express merge sort as $\mathbf{ms} = \text{rec}_T$ mrg spl. It is shown in Listing 13. From the example, we observe that $\mathbf{ms}$ can expanded infinitely. Later, we will exploit this property to generate parallel code.

3.2 ParAlg: Alg + role annotations

Alg programs are point-free programs which have an implicit but precise data-flow that does not rely on an external context. For example, $e_1 \circ e_2$ is the function composition so the output of applying e_2 will be used as the input of e_1. We can interpret this as e_2 sends a message to e_1. Parallel Algebraic Language is the formalization of the above idea. In essence, it is Alg with role annotations, converting implicit data-flow to explicit role communication. In this section, we will present the main results and use examples to build intuitions about its principles. More details and proofs can be found in [10]. We will briefly mention the results that will be useful in our project.

3.2.1 Syntax

The syntax of ParAlg is shown in Figure 3.4. r is the role identifier representing a unit of computation. $R_1 \times R_2$ specifies that the input is split across roles R_1 and R_2 [10]. More explanation of different constructor can be seen in [10]. In short, ParAlg is just Alg with

$$\rho_1, \rho_2 ::= r \mid \oplus_k^r[\rho] \mid \mathsf{id} \mid \rho_1 \circ \rho_2 \mid \pi_i \mid \rho_1 \vartriangle \rho_2 \mid \iota_i \mid \rho_1 \triangledown \rho_2 \qquad R_1, R_2 ::= r \mid \iota_i\ R \mid R_1 \times R_2$$
$$p_1, p_2 ::= e@r \mid \oplus_k^r[p] \mid \mathsf{id} \mid p_1 \circ p_2 \mid \pi_i \mid p_1 \vartriangle p_2 \mid \iota_i \mid p_1 \triangledown p_2 \qquad A, B ::= a@r \mid \iota_i\ A \mid A \times B$$
$$x \in X \implies x\ \in \mathcal{T}_X \qquad T_x \in \mathcal{T}_X \wedge T_y \in \mathcal{T}_X \implies T_x \cup^k T_y \in \mathcal{T}_X$$

Figure 3.4: Syntax of ParAlg

role annotations at certain points.

Notice that rec_F in Alg does not belong to constructs of ParAlg. So we need to unroll recursion a fixed number of times and then apply the role annotations to the unrolled expression. The unroll process is shown in Listing 13 and in this example, the number of unrollings is 1. The unroll-and-annotate operation will parallelize the recursive functions.

3.2.2 Inferring global types

ParAlg turns implicit data-flow into communication, and hence, we should be able to derive its communication protocols from valid ParAlg programs. In [10], global types (explained in Section 2.3) are used to represent communication protocols for ParAlg, which means for any valid ParAlg program, we can infer its corresponding global type. The inference rules and typing for ParAlg can be seen in [10].

$G =$
$r_0 \to r_1$: Rec L
$r_1 \to \{r_2, r_3\}$
$\{l_0 : r_1 \to r_3 : () + \text{int}.$
end;
$l_1 : r_1 \to r_2$: Rec L.
$r_1 \to r_3$: Rec L.
$r_2 \to r_3$: Rec L.
end$\}$;
$r_3 \to r_0$: Rec L.
end

Listing 14: Global types for merge sort

An example of the inferred global type is in Listing 14. It is the inferred global type of rec_T mrg spl with the number of unrollings equal to two. This global type tells us that r_1 receives an input list from r_0, based on the length of the input list, r_1 either sends it to r_3 directly when the list is a singleton or empty or splits the list into halves and sends them to r_2 and r_3 respectively. In the second case, r_2 processes the received list and sends it to

r_3, r_3 processes its received part and waits for another half of list to be received from r_2. After both r_2 and r_3 finish, r_3 will process the combined results. Finally r_3 sends the data back to r_0.

3.2.3 Example: Parallel merge sort

```
par_fun msp
    : (Rec L)@r0 -> (Rec L)@r0
    = id@r0
       . (merge@r3
           . (inj[0]
               ||| (inj[1]
                     . (((merge
                          . (inj[0]
                          ||| (inj[1]
                          . ((ms. proj[0])&&& (ms. proj[1, 2])))))
                          . split)@r2
                           . proj[0]@r1)
                          &&& ((merge
                                . (inj[0]
                                ||| (inj[1]
                                . ((ms. proj[0])&&& (ms. proj[1])))))
                                . split)@r3
                                 . proj[1]@r1))))
           . split@r1)
```

Listing 15: ParAlg for merge sort

The ParAlg expression for two unrolling of the merge sort is shown in Listing 15. We use arrow combinators `&&&` and `|||` to replace $\triangledown$ and $\triangle$ in the actual ParAlg expression since they are equivalent. A similar reason also applies to `inj`, the replacement of l_i and `proj`, the replacement of π_i. Its inferred global type is shown in the last subsection.

3.3 Conclusion

A visualization of the compilation pipeline can be seen in the Figure 3.5. More details about Alg and ParAlg can found in [10].

Alg is a point-free language. Without the use of explicit variables, point-free programs express the underlying data-flow of the computation clearly. Adding role annotations transforms Alg programs to ParAlg programs converting the implicit data-flow to explicit communication. Communication will aid us to generate parallel code using message-passing concurrency. We will introduce our method from the next chapter.

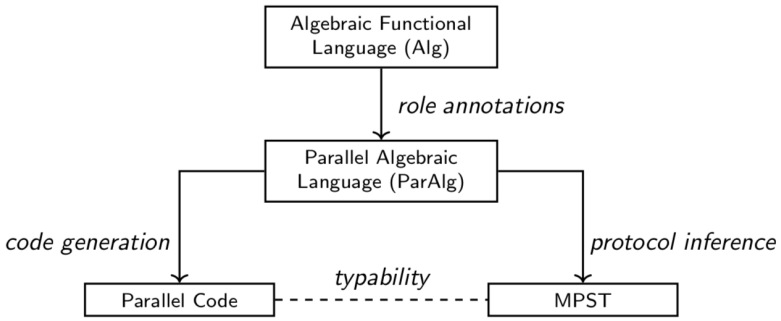


Figure 3.5: Overview of code generating pipeline from Alg[10]

Chapter 4

SPar: A session-typed free monad EDSL for concurrency

To generate parallel code from ParAlg, we first introduce the syntax of our intermediate language, the session-typed free monad EDSL for concurrency hosted in Haskell (SPar). SPar is comprised of two components: Core and Proc. Core is the language expressing sequential computation while Proc is a monadic language with message-passing primitives, communicating Core expressions between different roles. We use a group of Proc interacting with each other to represent parallel computations. In addition, session typing a group of Procs ensures that the computation is deadlock-free [20].

4.1 Computation: The Core EDSL

Core is the elemental computation. The syntax of Core is mostly inspired by Alg [10] and the work done by Svenningsson and Axelsson [21]. For this project, we chose to implement Core syntax as small as possible without sacrificing expressibility.

4.1.1 Syntax

The syntax of Core is shown in Listing 16. Inl and Inr are for the construction of sum types. Pair is responsible for constructing values of product types while Fst and Snd extract value from product type. Supporting sum, product and inductive types (see in the next section) is enough to express any data structure in any computation. In addition to these actions manipulating basic data structures, we have Lit which can be directly interpreted by the Haskell interpreter (see Section 5.2) via unwrapping and Var, a constructor which is useful when what we do not evaluate the Core expression but inspect their static structure. It is used for code generation (see Section 7.3.2) and session typing (see Section 5.1). Id is the identity function, and Const is the constant function. Prim represents user-defined functions takes, and two fields: the name and the Haskell implementation. The first field will be useful in the code generation (see Section 7.3.2) when applying user-defined function calls. The second field will be used by the Haskell interpreter directly when interpreting Prim expressions.

```
data Core a where
    Lit :: a -> Core a
    Var :: Int -> Core a
    Prim  :: String
             -> a
             -> Core a

    Ap :: Core (a -> b) -> Core a -> Core b
    Id :: Core (a -> a)
    Const :: Core a -> Core (b -> a)

    Fst :: Core ((a, b) -> a)
    Snd :: Core ((a, b) -> b)
    Pair :: Core a -> Core b -> Core (a, b)

    Inl :: Core (a -> Either a b)
    Inr :: Core (b -> Either a b)
```

Listing 16: The syntax of Core

4.1.2 Representation of recursive data structures

Core has primitives to operate on sum and product types. Representing recursive types like $\mu\text{list}.() + \text{Int} \times \text{list}$ will be covered in this section. The method is taken from the implementation of the Alg language in [10]. First of all, we extend the core with the following two operations. `In` represents the fold operation on iso-recursive types and `Out` represents the unfold operation on iso-recursive types. `:@:` is a type family which is a function acting on types instead of values. `:@:` converts `f` to sum and product type in Haskell which are `(,)` and `Either`. Consider a recursive type $\mu\alpha.\tau$, the type parameter `t` is equivalent to α, `f :@: t` is equivalent to τ and the typeclass `Data f t` is a recursive datatype equivalent to the fix-point of `f`.

```
data Core a where
    In   :: Data f t => Core (f :@: t -> t)
    Out  :: Data f t => Core (t -> f :@: t)

type family (:@:) (a :: Poly Type) (b :: Type) :: Type where
    'PId :@: x = x
    'PK y :@: _ = y
    'PProd f g :@: x = (f :@: x, g :@: x)
    'PSum f g :@: x = Either (f :@: x) (g :@: x)

class Data (f :: Poly Type) t | t -> f where
    roll :: f :@: t -> t
    unroll :: t -> f :@: t
```

A concrete example is shown below. We know a list has recursive type: $\mu\alpha.() + a \times \alpha$. So the `f` is `('PSum ('PK ()) ('PProd ('PK a) 'PId))` and we use Haskell list type `[a]` to present α (equivalent to `t` in `f :@: t`). `f :@: t` is evaluated to the type `Either () (a, [a])`.

```
instance Data ('PSum ('PK ())  ('PProd ('PK a) 'PId)) [a] where
    roll (Left _) = []
    roll (Right (a, b)) = a : b

    unroll [] = Left ()
    unroll (x:xs) = Right (x,xs)
```

However, most of our examples use Prim for representing recursive data structure hiding the implementation details (see in Section 7.3.2) In, and Out are low-level operations and less used.

4.2 Communication: The Proc EDSL

Proc is a free monad EDSL for message passing. As introduced in the free monad section in the background, the first thing to do it to define the algebra of message-passing concurrency: `ProcF` and `Proc` is defined using free monad constructor and `ProcF`. The definition is shown in the Listing 17. Careful reader might notice that `Proc` and `ProcF` are defined mutually with each others in `Branch`, `Select` and `Broadcast`.

The semantics will be defined in the next subsection in terms of a group of Proc programs because a single Proc program is either sequential or deadlock. The operational semantics is only worth discussing when given a group of Proc programs interacting with each other.

4.3 Concurrent computation: A group of Proc

We have introduced syntax for computation and communication. We also know that a single Proc expression is meaningless since there does not exist another party to interact with; hence, the computation has no progress. Naturally, we use a group of Proc to represent concurrent computations. To be more precise, a collection of Proc with their own role identifiers can be treated as a system of roles executing their own programs concurrently. In most of the cases, in order to make a group of Proc meaningful, we will allocate a start role in the system acting as the original data provider and an end role whose Proc program will receive data from others, process and output the final computation which is wrapped by the Pure constructor at the end of the Proc program.

Readers might find it easy to visualize a group of Proc as a computation graph. The start role is the source node, and the end role is the sink node. A pair of nodes are connected if they communicate data with each other.

```
data ProcF next where
    Send :: Nat -> Core a -> next -> ProcF next

    Recv :: Nat -> (Core a -> next) -> ProcF next

    Select :: Nat
           -> Core (Either a b)
           -> (Core a -> Proc c)
           -> (Core b -> Proc c)
           -> next
           -> ProcF next

    Branch :: Nat
           -> Proc c
           -> Proc c
           -> (Core c -> next)
           -> ProcF next

    Broadcast :: [Nat]
              -> Core (Either a b)
              -> (Core a -> Proc c)
              -> (Core b -> Proc c)
              -> next
              -> ProcF next

type Proc a = Free ProcF (Core a)
```

Listing 17: The algebra for message-passing

4.3.1 Operational semantics

Due to the similarities between Proc and multiparty session calculus introduced in [5], we borrow some syntax and operational semantics rules from their calculus to define the operational semantics of Proc. P, Q denote Proc programs. A message queue is h which contains messages (q, p, v) meaning that the sender q sends the receiver p with value v. $h \cdot m$ is a message queue whose bottom element is message m. h are runtime syntax to model the asynchronous message communication where the order of the messages are retained [5]. $e \downarrow v$ means the evaluation of the Core expression e to the value v. Figure 4.1 shows the small step semantics for Proc. Rule (Init) describes the initialization of a group of Proc programs with an empty message queue at the beginning. Rule (Send) appends the value to the message queue. Its complementary rule: (Recv) will recv the value at the top of the message queue. Rule (Branch) is also the complementary rule of the rule (Sel). Broadcast is defined in terms of Select, and it broadcasts the label to a group of receivers. Even though its semantics can be expressed in terms of Select, the reason why we still treat broadcast as an independent operation in SPar is because 1) this operation is very

$$(P_1, r_1) \mid (P_2, r_2) \mid \ldots \mid (P_n, r_n) \rightarrow \qquad \text{(Init)}$$
$$\quad (P_1, r_1) \mid (P_2, r_2) \mid \ldots \mid (P_n, r_n) \mid \emptyset$$

$$(\text{Free (Send } r_j\ e\ \text{next}), r_i) \mid \ldots \mid h \rightarrow \qquad \text{(Send)}$$
$$\quad (\text{next}, r_i) \mid \ldots \mid h \cdot (r_i, r_j, v) \qquad (e \downarrow v)$$

$$(\text{Free (Recv } r_i\ \text{cont}), r_j) \mid \ldots \mid (r_i, r_j, v) \cdot h \rightarrow \qquad \text{(Recv)}$$
$$\quad (p, r_j) \mid \ldots \mid h \qquad (\text{cont } v \downarrow p)$$

$$(\text{Free (Select } r_j\ e\ \text{cont1 cont2 next}), r_i) \mid \ldots \mid h \rightarrow \qquad \text{(Sel-Left)}$$
$$\quad (\text{cont1 } v \gg \text{next}, r_i) \mid \ldots \mid h \cdot (r_i, r_j, \text{L}) \qquad (e \downarrow v, \text{label}(v) \downarrow \text{L})$$

$$(\text{Free (Select } r_j\ e\ \text{cont1 cont2 next}), r_i) \mid \ldots \mid h \rightarrow \qquad \text{(Sel-Right)}$$
$$\quad (\text{cont2 } v \gg \text{next}, r_i) \mid \ldots \mid h \cdot (r_i, r_j, \text{R}) \qquad (e \downarrow v, \text{label}(v) \downarrow \text{R})$$

$$(\text{Free (Branch } r_i\ \text{next1 next2 cont}), r_j) \mid \ldots \mid (r_i, r_j, \text{L}) \cdot h \rightarrow \qquad \text{(Branch-Left)}$$
$$\quad (\text{next1} \gg\!= \text{cont}, r_j) \mid \ldots \mid h$$

$$(\text{Free (Branch } r_i\ \text{next1 next2 cont}), r_j) \mid \ldots \mid (r_i, r_j, \text{R}) \cdot h \rightarrow \qquad \text{(Branch-Right)}$$
$$\quad (\text{next2} \gg\!= \text{cont}, r_j) \mid \ldots \mid h$$

$$(\text{Free (Broadcast } [r_{k_1}, \ldots, r_{k_n}]\ e\ \text{cont1 cont2 next}), r_i) \mid \ldots \mid h \rightarrow \qquad \text{(Broadcast-1)}$$
$$\quad (\text{Free (Select } r_{k_1}\ v\ \text{c c (Pure ())}), r_i) \mid \ldots \mid h$$
$$\quad \text{where } (e \downarrow v, \text{c} = \text{Free (Broadcast } [r_{k_2}, \ldots, r_{k_n}]\ \text{cont1 cont2 next}))$$

$$(\text{Free (Broadcast } [r_{k_1}]\ e\ \text{cont1 cont2 next}), r_i) \mid \ldots \mid h \rightarrow \qquad \text{(Broadcast-2)}$$
$$\quad (\text{Free (Select } r_{k_1}\ v\ \text{cont1 cont2 next}), r_i) \mid \ldots \mid h \qquad (e \downarrow v)$$

$$(\text{Pure } v, r_i) \mid (P_i, r_i) \mid \ldots \mid (P_j, r_j) \mid h \rightarrow \qquad \text{(Pure)}$$
$$\quad (P_i, r_i) \mid \ldots \mid (P_j, r_j) \mid h$$

Figure 4.1: Small step semantics for Proc

common in communication, including Broadcast as a primitive operation is beneficial for user to write code 2) In the code generation stages, we can generate more efficient code for Broadcast operation instead of generating a series of Select operations. Broadcast operation can be treated as a syntax sugar in the Proc language.

4.3.2 Session types and duality checking

Immediately, we notice that a Proc program can be typed by session types. `Send` operation in ProcF corresponds to the type !⟨p, S⟩.T, `Select` corresponds to $\oplus\langle \text{p}, \{l_i : T_i\}_{i \in I}\rangle$ and so on. One exception is the `broadcast` operation which does not correspond to any type of the session types. We will discuss how to handle `Broadcast` in Section 5.1.

Figure 4.2 shows the session typing rule for a Proc expression. We borrowed some notation from the work done [10]. $\Gamma \vdash e$: Proc L a is the type of a Proc expression that follows protocol L and returns a value of type a. The types are parameterized by

$$\text{Inst}\frac{\Gamma \vdash e : \forall l.\text{Proc } L\ a}{\Gamma \vdash e : \text{Proc } ([\text{end}/l]L)\ a}$$

$$\text{Gen}\frac{\Gamma \vdash e : \forall l.\text{Proc } L\ a, \quad \text{fresh } l}{\Gamma \vdash e : \text{Proc } ([l/\text{end}]L)\ a}$$

$$\text{Ret}\frac{\Gamma \vdash v : a}{\Gamma \vdash \text{Pure } v : \forall l.\text{Proc } l\ a}$$

$$\text{Abs}\frac{\Gamma, x : a \vdash e : \forall l.\text{Proc } L\ b}{\Gamma \vdash \lambda x.e : a \to \forall l.\text{Proc } L\ b}$$

$$\text{Bind}\frac{\Gamma \vdash m : \forall l_1.\text{Proc } L_1\ a, \quad \Gamma \vdash f : a \to \forall l_2.\text{Proc } L_2\ b}{\Gamma \vdash m \gg\!= f : \forall l_2.\text{Proc } [L_2/L_1]L_1\ b}$$

$$\text{Send}\frac{\Gamma \vdash v : a}{\Gamma \vdash \text{Free (Send } r\ v\ \text{(Pure ())}) : \forall l.\text{Proc } (r!\langle a\rangle.l)\ ()}$$

$$\text{Recv}\frac{}{\Gamma \vdash \text{Free (Recv } r\ (\backslash v \to \text{Pure } v)) : \forall l.\text{Proc } (r?(a).l)\ a}$$

$$\text{Select}\frac{\Gamma \vdash v : a + b, \quad \Gamma \vdash f_1 : a \to \forall l_1\text{Proc } L_1\ c, \quad \Gamma \vdash f_2 : b \to \forall l_2.\text{Proc } L_2\ c}{\Gamma \vdash \text{Free (Select } r\ v\ f_1\ f_2\ \text{(Pure ())}) : \forall l.\text{Proc } r \oplus \{L : [l/l_1]L_1, R : [l/l_2]L_2\}\ ()}$$

$$\text{Branch}\frac{\Gamma \vdash \text{next1} : \forall l_1\text{Proc } L_1\ c, \quad \Gamma \vdash \text{next2} : \forall l_2.\text{Proc } L_2\ c}{\begin{array}{l}\Gamma \vdash \text{Free (Branch } r\ \text{next1 next2 } (\backslash x \to \text{Pure } x)) : \\ \forall l.\text{Proc } r\ \&\{L : [l/l_1]L_1, R : [l/l_2]L_2\}\ ()\end{array}}$$

Figure 4.2: Typing rules for Proc expressions

a variable l representing the continuation of a local type L. The typing rules contain all the operations except `Broadcast`. This is because `Broadcast` can be expressed in terms of a series of `Select` as explained in the previous subsection. So its session type $\oplus\{r_j\}_{j\in[1,n]}L : L_1, R : L_2$ can be expanded to $r_1 \oplus \{l : r_2 \oplus \{\ldots \oplus \{r_n \ldots\}\}, \ldots\}_{l\in[L,R]}$.

We have argued that a Proc program can be typed by multiparty session types. To utilize this property, we should check the duality of each pair of Proc in the group. In short, the duality check examines whether any pair of Proc in the system are complement with each other. If the duality properties are satisfied, the computation is guaranteed to be deadlock-free. This safety guarantee is useful and powerful in the application of parallel code generation. In this domain, SPar is considered to be the intermediate language. Passing duality check for the intermediate representation means that as long as we preserve types and communication pattern carefully in later stages of code generation pipeline, the generated code obtained will share the same non-trivial properties: communication safety, protocol fidelity and deadlock-freedom.

The work done by [5] constructed the theoretic foundation of algorithms for checking dualities, and we will give an overview of the implementation in the Section 5.1.

4.4 Conclusions

In this section, we have introduced our intermediate language. It is human-friendly to use thanks to the monadic interface. In addition, communication and computation are

independent in SPar. We can parameterize Proc with the type that represents sequential computation so that users can simply use their construction for sequential computation. More importantly, our strategy for parallelism is clear now. In a nutshell, we achieve parallelism by message-passing concurrency: spawning a group of threads on a multi-core CPU where each thread executes its corresponding Proc program.

Before jumping into the code generation, we will use the next chapter to give an overview of some implementation challenges related to SPar first.

Chapter 5

SPar: Implementation

5.1 Session types

Haskell does not support session types natively. Other encodings of session types in Haskell is too much for this project since Proc does not support all actions that can be typed by session types, i.e. channel delegation. We decided to create our representation of session types in Haskell corresponding to a set of actions supported by Proc. We will introduce two methods for session typing the group of Proc programs followed by duality checking. One is at the value level, and another is at the type level.

Essentially, what we do is to infer a collection of local types from a set of SPar expressions. The collection not only can be used to check the duality compatibility of the overall computation but also can work with external tool [20] for additional analysis and verification

5.1.1 Representations of session types in Haskell

Session types belong to the family of behavioral types. We can learn from Section 2.3 that behavioral types have a correspondence between types and operations that will be typed. We exploit this similarity to defined our session types in terms of a free monad, the same method we have used in defining Proc.

```
data STypeF a next where
    S :: Nat -> a -> next -> STypeF a next
    R :: Nat -> a -> next -> STypeF a next
    B :: Nat -> SType a c -> SType a c -> next -> STypeF a next
    Se :: Nat -> SType a c -> SType a c -> next -> STypeF a next

type SType a next = Free (STypeF a) next
```

Listing 18: Session types in Haskell

The Listing 18 shows the definition in Haskell. `SType` is the session type in Haskell. It is parameterized by a type variable `a` so that the value-level session types and the type-

level session types can share the same basic definition of session types. `S` is mapped to $!\langle p, S\rangle.T$. `R` is mapped to $?(p, S).T$. `B` is mapped to Branch type, and `Se` is mapped to Select type.

5.1.2 Value-level duality check

For the value-level session types, the type variable `a` is instantiated with type `TypeRep`. `TypeRep` reifies types to some extent by associating type representations to types [22]. Due to session types are represented as value expressions in Haskell, session typing a Proc program is the same as writing an interpreter which can be easily done since Proc is a free monad.

We traverse Proc programs converting each operation to its corresponding type in `STypeF` and convert the value to its `TypeRep`. For output actions, we recursively call the substructure to build the rest of the session types. For input actions like `Recv`, we will apply the continuation with the Core value constructed by `Var` and recursively call the function on the result. The trick to applying `Var` to the continuation makes it possible to inspect the static structure of every Proc programs because `b -> Proc a` is not inspectable, i.e., we cannot pattern match on it, while `Proc a` can be inspected. We will also use this for code generation, which will be introduced in Chapter 7.

5.1.3 Type-level duality check

The value-level duality check works well in checking duality at runtime, but as programmers, we aim to eliminate problems earlier. Hence, we propose a solution that makes use of Haskell's powerful type systems to check the duality of the system at the compile time. Besides, we will introduce some combinators to helps us build a group of Proc to form parallel computation, and this mechanism can act as an extra safety guard to make sure the correctness of these combinators.

The general approach of type-level duality checks can be summarized as the following steps.

1. Create a type-level representation of session types.
2. Modify the algebra of Proc to make it indexed by session types so that we can session type a proc while building it at the same time. Unlike the above method, we can only session type a Proc by interpretation after it has been constructed.
3. Gather the indexed session types of each Proc in the system and check the duality pair-wise at the type level.

The first step is achieved by reusing the definition in Listing 18 and use Haskell DataKind extension to promote data constructors of `STypeF` and data constructors of Free monad to type constructors. At this stage, type parameter `a` has been promoted to a kind parameter, and it will be instantiated with kind `*` representing the kind of all types that have values, i.e. Int, List of float. Also, we should also create a type-level function that is

```
type family (>*>) (a :: SType * c) (b :: SType * c) where
    'Free ('S r v n) >*> b = 'Free ('S r v (n >*> b))
    'Free ('R r v n) >*> b = 'Free ('R r v (n >*> b))
    'Free ('B r n1 n2 n3) >*> b = 'Free ('B r n1 n2 (n3 >*> b))
    'Free ('Se r n1 n2 n3) >*> b = 'Free ('Se r n1 n2 (n3 >*> b))
    'Pure _ >*> b = b
```

Listing 19: Implementations of type level bind

equivalent to bind in Free monad to help us compose session types. The implementation of type level bind `>*>` can be seen in Listing 19. It is similar to bind in Free monad but defined as a type family.

```
data ProcF (i :: SType * *) (j :: SType * *) next where
    Send :: Sing (n :: Nat) -> Core a -> next
            -> ProcF ('Free ('S n a j)) j next
    Recv :: Sing (n :: Nat) -> (Core a -> next)
            -> ProcF ('Free ('R n a j)) j next
    Branch :: Sing (n :: Nat) ->
              Proc' left ('Pure ()) c ->
              Proc' right ('Pure ()) c ->
              next ->
              ProcF ('Free ('B n left right j)) j next
    Select :: Sing (n :: Nat) ->
              Core (Either a b) ->
              (Core a -> Proc' left ('Pure ()) c) ->
              (Core b -> Proc' right ('Pure ()) c) ->
              next ->
              ProcF ('Free ('Se n left right j)) j next

type Proc (i :: SType * *) a =
    forall j . IxFree ProcF (i >*> j) j (Core a)
```

Listing 20: The algebra of Proc indexed by session types and the definition of indexed Proc

The second step is challenging since we need to find a way to make the Proc indexed by our free monad. Obviously, the original definition of Free monad and Proc does not provide any extra type parameters to be indexed by session types. Hence we use the indexed free monad. It is indexed by two parameters `i` and `j`. In the context of this project, you can treat `i` as the session type for the current proc and `j` as the continuation of session types. Accordingly, we will modify the definition of the algebra of Proc as well as Proc (see in Listing 20). The main operations remain unchanged, and the main difference is 1) makes the algebra of Proc indexed by its corresponding session type (`i`) and continuation (`j`) 2) For the role identifier, we use the type level identifier: the type

whose kind is `Nat` instead of values because, in the later stage, we have to check duality at the type level. The definition of Proc use Haskell's RankNType and type family `>*>` to extract its corresponding indexed session type `i` from the continuation. RankNType allows any type of `j` hence models any continuation. It represents $\forall l$ in the session types of Proc (see in Section 4.3.2) in Haskell. By the definition of `>*>`, session type `i` must end with the Pure type constructor which is mapped to end in the session type. The basic helper functions for constructing Proc expressions are also indexed by session types. An example can be in seen in Listing 21. We will omit details of some of the helper functions. Observing the function signatures is crucial in understanding how it works.

```
liftF :: ProcF i j a -> IxFree ProcF i j a

(>>=) ::  IxFree ProcF i j a
          -> (a -> IxFree ProcF j k b)
          -> IxFree ProcF i k b

send ::
    Sing n
    -> Core a
    -> Proc ( 'Free ( 'S n a ( 'Pure ()))) a
send role value = liftF $ Send role value value

recv :: Sing n -> Proc ( 'Free ( 'R n a ( 'Pure ()))) a
recv role = liftF (Recv role id)
```

Listing 21: Implementations of helper functions

```
example = do
    x :: Core Int <- recv zero
    send one x

ghci:> :type example
example
  :: Proc
     ('Free
        ('R 0 Int
            ('Free ('S 1 Int ('Pure ())))))
     Int
```

Listing 22: An example of session type

We will conclude the implementation of the second step with an example in Listing 22. It represents a simple proc that receives an int from role zero and send the int to role one. Haskell's type system infers its session type, `'Free ('R 0 Int ('Free ('S 1 Int (Pure ())))))`, which corresponds to the behavior of the example. Session typing can

be done simultaneously and automatically while users are building the Proc processes thanks to Haskell's type inference.

For the third step, we can assume we have already gathered a type level list of session types paired with its role identifier. The duality check algorithm is still the same. We match different Proc programs pair-wise and check whether the projection of both session types is complementary. The algorithm is easy to implement as the Haskell function, but lifting the computation into type level is tricky. One of the obstacles is that type family does not support higher-order type-level functions. We divide the problem into sub-problems. We need to implement 1) a type family that converts a list of Session type to a list of Session type pair 2) a type family that maps a function to list and combine the result 3) a type family that includes projection, checking whether a pair of session types are complementary. We combine the solutions to these problems and encapsulate them into type class constraint. The constraint is satisfied only if the duality checked is passed at the compile time.

5.2 SPar interpreter

5.2.1 Overview

SPar interpreter is a simulator that simulates a group of Proc programs in Haskell. It can be considered as the simplest backend for evaluating SPar expressions. It records traces of the executions and the final output values of each Proc programs in the system.

It focused on providing a reference implementation explaining the semantics of SPar expressions, not on performance. Besides, the SPar interpreter served as a very useful tool in the development of this project. We use it as a prototype to quickly verify whether the computation produces the expected results. This feature is useful, especially in the early stage of implementation or during debugging.

5.2.2 Implementation

The implementation of the SPar interpreter is standard. It is similar to the implementation of the scheduler we explained in the free monad section of the background chapter. In essence, it is a round robin scheduler for a group of Proc programs.

A partial implementation can be seen in Listing 23. It takes a list of Proc as a parameter, and it maintains a state which is the combination of a message queue, a trace and a list of output values. For the base case, the list is empty which means all processes has exited, it returns the current trace and a list of output values. For the recursive case, its pattern matches the first process at the beginning of the list. If the operation is Pure, it will update the list of output values and call itself recursively on the tail of the list. If the operation is an output action, i.e., `Send`, `Select` or `Broadcast`, it will update the message queue with the corresponding message containing the sender, the receiver and the value, then pop the head Proc and append its next step to the end of list of Procs, and call the list recursively. If the operation is an input action, i.e., `Recv` or `Branch`, it

```haskell
data InterpState = InterpState
    {
        outputValues :: [(Nat, String)],
        trace :: [String],
        messageQueue :: [(Nat, Nat, String)]
    }

interpret :: [(Proc (), Nat)] -> State InterpState ()
interpret []       = return ()
interpret (x : xs) = interp x xs

interp :: (Proc (), Nat) -> [(Proc (), Nat)] -> State InterpState ()
interp (Pure value, role) xs = do
    updateOutputValue role value
    interpret xs
interp (Free (Send receiver value next), role) xs = do
    updateMessageQueue (role, receiver, show $ interpCore value)
    updateTrace "send"
    interpret $ xs ++ [(next, role)]
interp p@(Free (Recv sender cont), role) xs = do
    (x, y, v) <- getTopMessage
    if x == sender && y == role
        then do
            popMessageQueue
            updateTrace "recv"
            let value = Lit (read v)
            interpret $ xs ++ [(cont value, role)]
        else
            interpret $ xs ++ [p]
```

Listing 23: Partial implementation of the SPar interpreter

will first examine whether sender and receiver pairs match the pair from the message at the top of the message queue. If so, it applies the continuation with the value in the message, removes the message from the queue, removes the head of the list, moves the result of applying continuation to the end of the list and calls the list recursively. If not, it simply moves the head of the list to the end of the list without changing the value of the head process and calls the list recursively. Because we have checked the duality of the processes in the system, this guarantees that for any input action, the required message will eventually appear at the head of the message queue in finite steps.

Also, the interpreter has a helper function that evaluates Core expressions. This helper function is easy to write because operation on product or sum type has its own mapping function in Haskell and for the `Prim` and `Lit`, we can get their underlying Haskell implementation directly. As for `Var`, this constructor is not intended to be used

externally so we will not encounter it.

Chapter 6

SArrow: An Arrow interface for writing SPar expressions

When trying to express more complex and interesting parallel patterns, such as the map or reduce pattern, we realize SPar is too low-level. It is difficult to express simple computations because of the overheads of expressing communication patterns by hand.

To solve this issue, we draw inspirations from the Arrow interface, in particular, [4] where they use the Arrow interface to do parallel programming in Haskell. We introduce SArrow; an Arrow interface for writing SPar expressions.

SArrow is an Arrow interface for writing SPar expressions. With help from SArrow, users can use canonical arrow combinators to write algorithms without writing any explicit communication, and gain parallelized algorithms for free.

6.1 Syntax

```
data Pipe a b = Pipe
    { start :: Nat
    , cont  :: a -> Proc
    , env   :: Map Nat Proc
    , end   :: Nat
    }

type SArrow a b = Nat -> Pipe a b

instance Arrow SArrow where
instance ArrowChoice SArrow where
```

Listing 24: Definition of SArrow

The simplified syntax of SArrow can be found in Listing 24. `Pipe a b` data structures are the essential component of SArrow. It regards computation as a pipe where

data with type `a` goes into the pipe and data with type `b` get out of the pipe. Internally, it's a record type of four fields. `start` field identifies the process where the input data is received. `cont` field has the type `a -> Proc`, which is a continuation waiting for the input data produced by the last pipe. `env` represents a group of Procs interacting inside the pipe to produce the output data; in other words, it is the parallel computation. `end` is the identification of the process that produces the output data in the end. We can retrieve the corresponding process by a lookup in `env` with the key `end`. The returned Proc returns data with type `b`.

SArrow is a type synonym of `Nat -> Pipe a b`. It consumes `Nat` which means the identifier of a process and output `Pipe a b`. The reason why we use `Nat` as the only parameter is to ensure that processes names are not duplicated. It will be explained more thoroughly in Section 6.3.

6.1.1 Arrow interface

`SArrow` is an instance of the Arrow typeclass as well as ArrowChoice type class. For example, the type signature of the combinators `>>>`, `|||`, `&&&` and `arr` are shown below. The main difference between their type signatures and the usual Arrow interface is that in the `arr`, the function is wrapped with Core. In general, it captures the same meaning as the usual Arrow interfaces. Implementation details of these combinators will be explained in Section 6.2.

```
(>>>) :: SArrow a b -> SArrow b c -> SArrow a c
arr :: Core (a -> b) -> SArrow a b
(|||) :: SArrow a c -> SArrow b c -> SArrow (Either a b) c
(&&&) :: SArrow b c -> SArrow b c' -> SArrow b (c, c')
(***) :: SArrow b c -> SArrow b' c' -> SArrow (b, b') (c, c')
```

As an example, we will illustrate some typical computation patterns used in parallel computing.

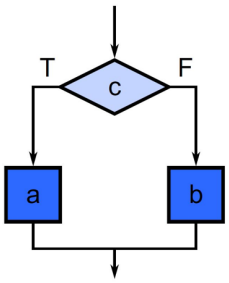
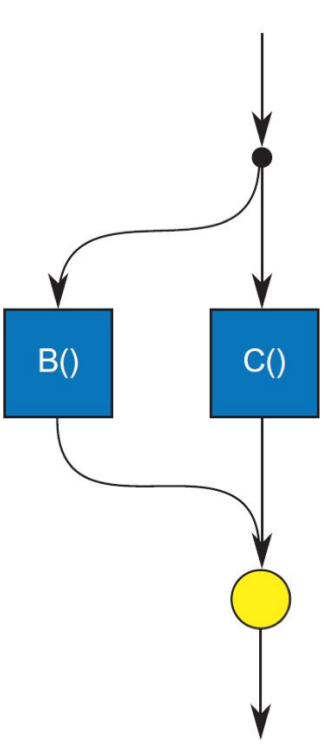
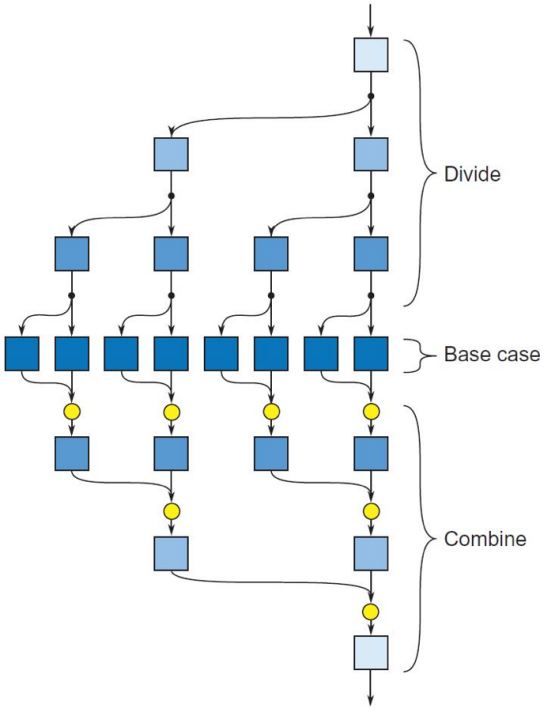


Figure 6.1: Visualization of the branching pattern [23]

First of all, the branching pattern illustrated by Figure 6.1 is equivalent to an expression formed by `|||` combinators, where the data constructor `Left` leads to one computation path and the data constructor `Right` leads to another computation path. It seems to be a simple pattern, but it is useful when composed with other more complex patterns. We will use an example to illustrate this at the end of this section.



(a) Visualization of the fork-join pattern [23]



(b) Fork-Join pattern for divide-and-conquer algorithms [23]

Figure 6.2: Fork-join pattern and divide-and-conquer algorithms

Secondly, the fundamental building block, the fork-join pattern illustrated by Figure 6.2a can be expressed by `&&&` combinator. The SArrow produced by `&&&` produces a tuple as output by collecting the computation result of the main thread and the forked thread and also acts as a synchronization point.

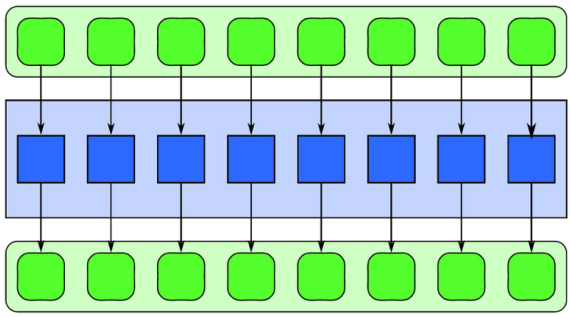


Figure 6.3: Visualization of parallel map [23]

Thirdly, the familiar parallel map pattern illustrated in Figure 6.3 is also a candidate to be expressed in SArrow. The code sample is in Listing 25. `pmap` splits the input `a` into 4 chunks using the splitting function `s`, applied the elemental function `f` and the arrow combinator `***` in parallel and finally use the collecting function `c` to collect the results. Usually, the input `a` is a list and `s` splits the list into four equal chunks. The size of the tuple determines the degree of parallelism.

Fourthly, we can apply a similar logic to express the parallel reduce pattern shown in Figure 6.4. The code sample is in Listing 26. The result of parallel reduce has similar type

```
pmap :: SArrow a b
     -> SArrow a (a, (a, (a, a)))
     -> SArrow (b, (b, (b, b))) b
     -> SArrow a b
pmap f s c = s >>> (f *** (f *** (f *** f))) >>> c
```

Listing 25: Parallel map in SArrow

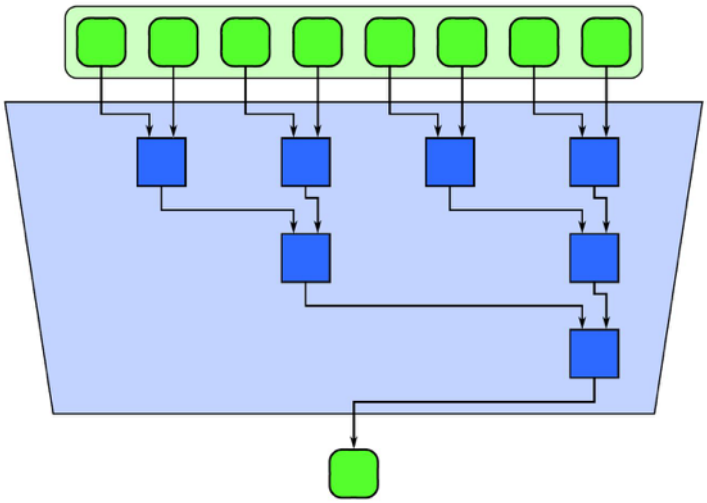


Figure 6.4: Visualization of parallel reduce in SArrow [23]

signature as the collecting function in `pmap` so it is often used with the `pmap` function. We use nested tuple `(a, (a, (a,a)))` to represent a sized array of data. The `helper` function transforms the array representation of data into a form so that we apply the reduce function `r` to the elements pair-wise and parallel.

Finally, a more complex pattern can be expressed compositionally from simpler patterns expressed in SArrow. We use a typical divide-and-conquer algorithm implemented with fork-join as an example. Figure 6.2b shows divide-and-conquer algorithms with 2-ways and 3-levels of fork-join. The algorithm in SArrow is in Listing 27. The divide-and-conquer pattern can be built recursively in Haskell. For the base case, we simply apply the basic computation. Otherwise, we first call split and then call the function recursively with the level decremented by one and, in the end, call the merge to combine the results. Every expression in the function definition is connected using arrow combinators. A 3-level divide-and-conquer algorithm is constructed by passing 3 to the function resulting in an algorithm with $2^3 = 8$-way parallelism.

Also, the divide-and-conquer parallel pattern can be optimized when combining with the branching pattern. The branching pattern allows us to add shortcuts to the pattern (illustrated in the Figure 6.5, red lines represent the alternative computation path provided by branching patterns). The shortcut gives us the ability to decide whether to do local computation or split into multiple subtasks depending on the input size. When the input size is small, the overhead of the latter usually outweighs its parallelism. Adding the simple branching pattern results in a pattern that is adaptive to various input sizes

```haskell
preduc :: SArrow (a, a) a -> SArrow (a, (a, (a, a))) a
preduc r = assoc >>> (r *** r) >>> r
 where
  assoc = (arr Id *** arr Fst) &&& (arr Snd >>> arr Snd)
```

Listing 26: Parallel reduce in SArrow

```haskell
divConquer
  :: Int
  -> SArrow a b
  -> SArrow a (a, a)
  -> SArrow (b, b) b
  -> SArrow a b
divConquer 0 baseFunc _split _merge = baseFunc
divConquer level baseFunc split merge =
  split
    >>> (   divConquer (level - 1) baseFunc split merge
        *** divConquer (level - 1) baseFunc split merge
        )
    >>> merge

twoWayThreeLevelDq = divConquer 3
```

Listing 27: 2-ways and 3-levels divide-and-conquer algorithm in SArrow

with better performance.

The implementation demonstrates the power of implementing SArrow as a domain-specific language embedded in Haskell. We make full use of Haskell features, i.e. high order functions and polymorphic functions to construct expressive, composable and generic computation patterns.

More examples of algorithms formed by SArrow, e.g. dot product or merge sort are shown in the Section 8.

6.2 Implementation of arrow combinators

In this chapter, we will present naive implementation, and the optimized solution is introduced in the next section.

Kleisli arrow has the same type as the second parameter of the monadic bind. The intuition why SArrow is an instance of Arrow comes from the Kleisli arrow of a monad is an instance of Arrow class (shown in Listing 28). The `cont` field in the `Pipe` has a similar type signature as the `runKleisli` field in the Kleisli arrow. From the previous section, we have shown that Proc is a monad, so `Pipe` is just an extended version of Kleisli arrow where computations in `Pipe` usually finish in one of the processes stored in

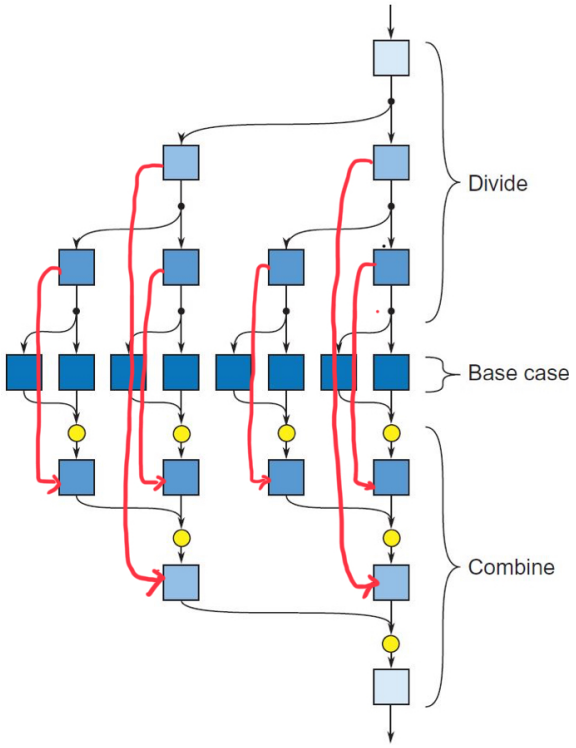


Figure 6.5: Combination of branching pattern and divide-and-conquer pattern

`env` instead of finishing at `cont` like Kleisli arrow. Intuitively, SArrow, a function from the role to `Pipe`, should be an instance of Arrow since `Pipe` can be made into an arrow instance and functions compose.

The essential issue when implementing arrow combinators is how to connect one `Pipe` by another `Pipe`. The first problem we need to address is how to deal with the `cont` in the tail `Pipe`. We know that only one `cont` field exists in the resulting `Pipe` and it must be that from the head `Pipe`. For the `cont` field in the tail `Pipe`, we append the action: receive from end in the first `Pipe` to the beginning of it by monadic binding. We then store the resulting Proc expression in the new `env`. We also extend the Proc corresponding to the end in the head `Pipe` with action: send to the `start` role defined in the tail `Pipe`. Finally, the new env is formed by merging the env from the head `Pipe` and the `env` from the tail `Pipe`. When there are duplications of role identifications, we compose them using the monadic bind. The `start` field in the resulting `Pipe` is the same as that from the head `Pipe` and the `end` field will be set the same as that in the tail `Pipe`.

We can use the `Pipe` composing function to implement arrow combinators for SArrow. The implementation just applies the first SArrow to the input role and the second SArrow to a new role. Usually, to avoid duplication of roles, the new role is set to be the maximum role in the first `Pipe` + 1 and finally apply the `Pipe` composing functions to both `Pipe`. A simplified code explanation can be seen in Listing 29. The rest of the combinators can be implemented in a similar fashion.

```haskell
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Arrow (Kleisli m) where
    id = Kleisli return
    (Kleisli f) . (Kleisli g) = Kleisli (\b -> g b >>= f)
    arr f = Kleisli (return . f)
    first (Kleisli f) =
        Kleisli (\ ~(b,d) -> f b >>= \c -> return (c,d))
    second (Kleisli f) =
        Kleisli (\ ~(d,b) -> f b >>= \c -> return (d,c))
```

Listing 28: The implementation of arrow instance for Kleisli arrow of a monad

```haskell
(>>>) :: (SArrow a b) -> (SArrow b c) -> (SArrow a c)
(>>>) leftArrow rightArrow start = compose firstPipe secondPipe
 where
  firstPipe  = leftArrow start
  secondPipe = rightArrow (end leftP + 1)
```

Listing 29: The simplified implementation of >>>

6.3 Strategies for optimized role allocation

From the last section, we know the number of roles in the system is directly related to the number of processes in the final generated code. Hence, role allocation is an essential part of generating efficient parallel programs.

In this section, we propose strategies for optimizing role allocation. We have two goals in mind when optimizing; (1) we would like to reduce the number of roles (processes) in the computation, since the overhead of thread creation and data transmission has negative impact on performance; (2) we want to avoid roles duplication when we compose SArrows since role duplication means the different computations must be merged in the same role and computations in the same thread are sequential.

An easy solution for the first goal would be to setup an upper bound on the number of roles, and then cycle through this fixed bound when allocating new roles. Processes corresponding to duplicated roles can be simply merged using binds since Proc is a monadic EDSL and duality check ensures binding will not cause deadlocks. However, this strategy is not ideal since the duplication of roles will decrease the degree of parallelism in the system.

The naive strategy used in Section 6.2 already satisfies the second goal. However, the number of channels required and the number of roles in the system will grow exponentially. In a divide-and-conquer algorithm, the number of channels increases from 10 to 120 and the number of roles increases from 6 to 36 when the level is increased from 1 to 3.

$$\text{id}\,\frac{x : \text{Role}, \quad a : \text{Type}}{id : \text{SArrow a a}, x \Rightarrow x}$$

Figure 6.6: Role allocation for id

For illustration, we use inference rules to explain our proposed strategy for optimized role allocations when composing SArrows. Please see Figure 6.6 as an example. $x \Rightarrow x$ means the computation starts with role x and ends with role x.

$$\text{compose}\,\frac{e1 : \text{SArrow a b}, x \Rightarrow y, \quad e2 : \text{SArrow b c}, y \Rightarrow z}{e1 \texttt{ >>> } e2 : \text{SArrow a c}, x \Rightarrow z}$$

$$\text{arr}\,\frac{f : \text{Core (a} \rightarrow \text{b)}, \quad x : \text{Role}}{\text{arr } f : \text{SArrow a b}, x \Rightarrow x}$$

$$\text{arrow choice: } |||\,\frac{e1 : \text{SArrow a c}, x \Rightarrow y, \quad e2 : \text{SArrow b c}, x \Rightarrow z}{e1 \texttt{ ||| } e2 : \text{SArrow (Either a b) c}, x \Rightarrow \max(y, z)}$$

$$\text{arrow choice: } +++\,\frac{e1 : \text{SArrow a c}, x \Rightarrow y, \quad e2 : \text{SArrow b d}, x \Rightarrow z}{e1 \texttt{ +++ } e2 : \text{SArrow (Either a b) (Either c d)}, x \Rightarrow \max(y, z)}$$

$$\text{arrow: } \&\&\&\,\frac{e1 : \text{SArrow a b}, x \Rightarrow y, \quad e2 : \text{SArrow a c}, (y + 1) \Rightarrow z}{e1 \texttt{ \&\&\& } e2 : \text{SArrow a (b, c)}, x \Rightarrow z}$$

$$\text{arrow: } ***\,\frac{e1 : \text{SArrow a b}, x \Rightarrow y, \quad e2 : \text{SArrow a' c}, (y + 1) \Rightarrow z}{e1 \texttt{ *** } e2 : \text{SArrow (a, a') (b, c)}, x \Rightarrow z}$$

Figure 6.7: Rules fo role allocations of different combinators

The rule for the rest of combinators is shown in Figure 6.7. Notice that for compose, id, arr and ArrowChoice, we do not introduce any new roles; in other words, there is no parallelization for these combinators. The reader may find it strange that we do not intend on parallelizing arr combinator which lifts a sequential computation represented by Core (a $\rightarrow$ b) into SArrow. It makes sense to introduce a new role to execute the computation and hence parallelize computationally heavy tasks. We use this strategy in the first place, but later, we found a more suitable strategy exists. This strategy is not ideal because introducing new roles for simple functions will damage performance. Also, another reason not to introduce a new role when encountering arr combinators is that we gained function fusion for free. Simple function, i.e. fst, inject left, or snd, are automatically fused into more complex user-defined functions.

For the class of combinators belonging to arrow choice, we do not introduce any new role. The expressions at the lhs and at the rhs start with the same role x because when only one code path will be executed as the name choice suggested so we should not use

separate roles for two expressions that will never be executed simultaneously. In the end, we decided the computation end in the role $\max(y, z)$. Max guarantees that there will not be role duplications when we compose expressions formed by ArrowChoice combinators with other combinators. For the implementation, all processes in both left and right SArrow expressions are wrapped inside a branch operation separately. Assume $\max(y, z) = y$, the process at the role y will be extended with actions that receive data from $\min(y, z) = z$ role at its right branch. Finally, applying inject left and inject right at left and right branches gives us an Either type as the output.

Finally, we decided that the right place to allocate new roles is `&&&` combinator. As shown in the type signature, product types mean computation at both branches will both be executed, and they are independent. To make sure that both computations are executed simultaneously, we constraint that the right SArrow expression must start with a role greater than the end role of the left SArrow expression. This ensures no role duplications, hence maximize parallelism. The combined expression ends in the end role of the right SArrow expression instead of introducing an unnecessary new role. For the implementation, the process corresponding to end role z is extended with actions that receive data from the end role y of the left process finally outputs a pair.

Even though from the implementation point of view, optimized role allocation is ad-hoc and more difficult to implement because we need to consider composition by send-and-receive as well as composition by local monadic bind. In contrast, the naive solution in Section 6.2 only consider send-and-receive, so the code is symmetry for both left and right roles. We believe the effort is worthy because, for a n-level divide-and-conquer algorithms, the optimized role allocation strategies allocate 2^n roles in total, which is the same as the degree of parallelism. All the roles are used to maximize parallelism instead of wasting valuable resources to create roles that merely transmit data.

Finally, the optimized strategy presented in this chapter is not the only solution. For example, there is a strategy that only introduces new roles into the computation graph when encountering a specific atomic function. Different strategies result in different communication structures hence different kind of parallelisms. We should choose the right strategy depending on the specific task.

6.4 Satisfaction of arrow laws

We have provided the implementation of SArrow combinators that is similar to the Arrow interface. However, the implementation is not enough to state that SArrow is an Arrow. We will present a justification on of basic arrow laws.

Figure 6.8 shows the rules of arrow laws. We include the subset of the laws that contain the $first$ combinator. The other half of the laws that contain the `second` combinator can be proved by symmetry. `first` combinator is implemented as `first = (*** id)` and $\alpha :: ((x, y), z) \to (x, (y, z))$.

We argue above equation holds if both sides of equation express the same computation. Equation (1), (2) holds because lifting Id from Core to SArrow will not modify the

$$arr(Id) \ggg a = a \tag{1}$$
$$a \ggg arr(Id) = a \tag{2}$$
$$(a \ggg b) \ggg c = a \ggg (b \ggg c) \tag{3}$$
$$arr(f; g) = arr(f) \ggg arr(g) \tag{4}$$
$$first(a) \ggg arr(Fst) = arr(Fst) \ggg a \tag{5}$$
$$first(a) \ggg arr(\alpha) = arr(\alpha) \ggg first(first(a)) \tag{6}$$
$$first(a \ggg b) = first(a) \ggg first(b) \tag{7}$$

Figure 6.8: Arrow laws [24]

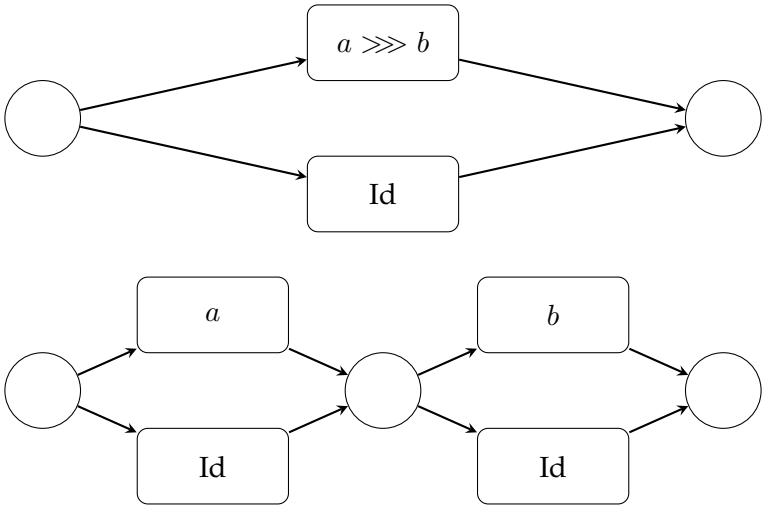


Figure 6.9: Graphical representation of equation (7)

result of computation due to the semantics of Id. Equation (3) is the associativity rule of $\ggg$. Left side of the equation sends the output from $a \ggg b$ to the input of c while right side sends the output of a to the input $b \ggg c$. The communication structure might change, but they both represent the same computation. Equation (4) is valid because applying the input value to the composition of f;g is the same as applying the input to f followed by a message passing (data communication could be local which means the communication structure for both sides of equations are the same in this case) and then applying it to g. One might claim that the left side is more efficient than the right side due to the fusion depending on the role allocation strategy. Equation (5) holds because they both represent a computation that takes an input pair and applies the function a to its first position and returns the result. Right hand side of equation (7) represents a SArrow expression that applies a on left position of input pair and applies Id on the right position of input pair in parallel, collects the results and applies b on left position of input pair and applies Id on the right position of input pair while left side fused a and b together and applied them in one step hence there is only one join point. Figure 6.9 is a graphical representation of equation (7). They represent the same computation. We can also derive

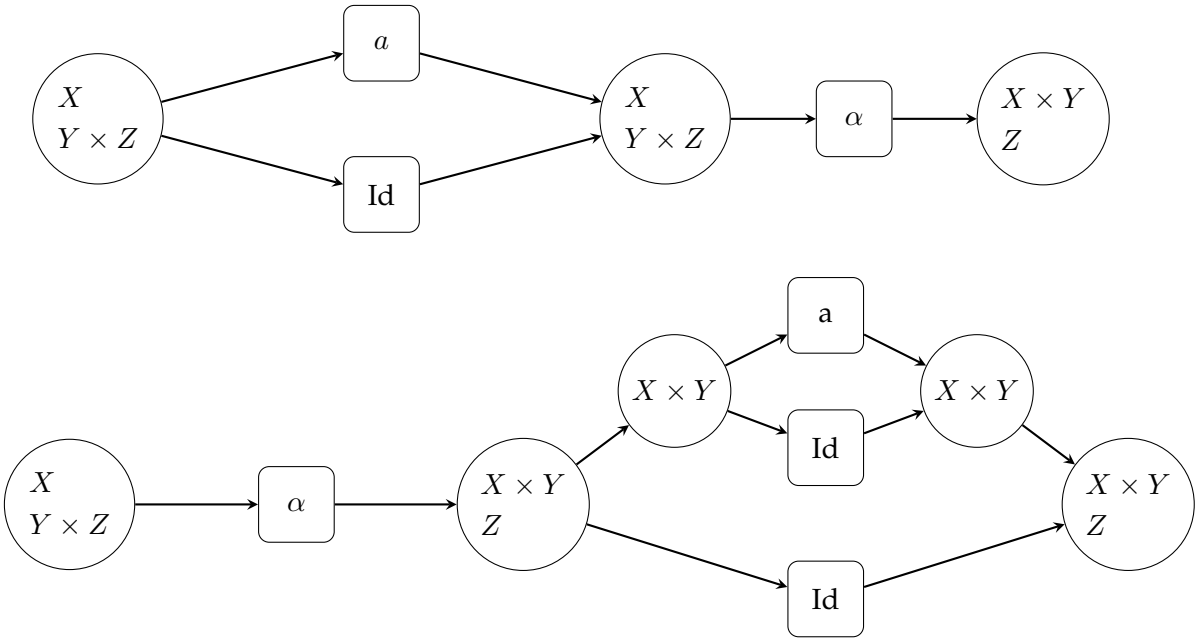


Figure 6.10: Graphical representation of equation (6)

equation (6) from its Visualization (see in Figure 6.10).

From the above explanation, we gain intuition why the left and the right sides of equations produce the same computation result. More importantly, since SArrow is an abstract layer on top of a message-passing intermediate language, we should argue that the underlying communication structures for both sides of an equation do not have error, i.e. there is no communication mismatched and computations are deadlock-free. Progressing work from [10] can be adapted to show that there exists a global type for an SArrow expression. The consequence is that SPar expressions for all participants beneath a SArrow expression are typeable against the projection of this global type and therefore, they are dual pair-wise indicating no communication errors.

6.5 Conclusions

The Arrow interface is the perfect interface to express general computation for this project, not only because it is intuitive to understand and visualize, but also because its combinators `***` and `&&&` have a natural parallel interpretation.

In addition, SArrow makes hassle-free compilation from Par-Alg to SPar possible because Par-Alg is also an arrow expression and simply interpreting arrow combinators by the SArrow implementations fills the gap between Par-Alg and SPar.

So far, we have introduced our interface for expressing computations. The remained challenge is the code generation from SPar to a target platform. In the next chapter, we will introduce one code generation backend to C. Once we achieve this, every computation in SArrow can be transformed into parallel C code automatically.

Chapter 7

Type-safe code generation from SPar

SPar has two components: Core representing the unit of computation and Proc as a skeleton of the communication. Naturally, the process of code generation from SPar should be divided into two parts correspondingly. We choose to make the two parts independent of each other so that it is possible to swap the code generation strategy of one component without modifying another one.

The procedure of code generation is standard: transformation. We start our programs in a high-level EDSL and run a series of transformations to a low-level EDSL. SPar expressions are converted to a low-level EDSL which is then transformed to an abstract syntax tree (AST) of C [25]. The generated code is obtained by pretty printing the AST.

7.1 Instr: A low-level EDSL for channel communication

In Proc, we have high-level actions like select, broadcast and branch abstracting implementation details such as variable declarations, variable assignments, channel initializations, channel communication and channel deletion. Hence, we need to define an EDSL containing instructions related to these low-level operations. We name it Instr. Programs will be translated to sequences of Instr.

When we design Instr, we keep the simplicity in mind, so Instr is not dependent on any specific target language. Any reasonable target language with a channel communication library can be easily used as a target from Instr.

7.1.1 Syntax and semantic

The definition of Instr is seen in Listing 30. `Channel` is our abstract representation of channels in Instr. It is indexed by a type a from the reified type `ReprType a`. Reified types give us information about the types of expressions at runtime. This type parameter makes sure the value sent or received in this channel are of the correct type. This is necessary because, for some target languages, the channels are typed. Similarly, type parameters in `Exp` have the same functionality. `Exp` is just a wrapper of Core expressions. In later stages, we will take care of the code generation of `Exp`. `Instr` defines the set of

```haskell
data Channel a where
    Channel :: CID -> ReprType a -> Channel a

data Exp a where
    Exp :: Core a -> ReprType a -> Exp a

data Instr where
    CInitChan :: Channel a -> Instr
    CDeleteChan :: Channel a -> Instr
    CSend :: Channel a -> Exp a -> Instr
    CRecv :: Channel a -> Int -> Instr
    CEnd :: Exp a -> Instr
    CDecla :: Int -> ReprType a -> Instr
    CAssgn :: Int -> Exp a -> Instr
    CBranch :: Int -> Seq Instr -> Seq Instr -> Instr
    CSelect :: Int -> Int -> Seq Instr -> Seq Instr -> Instr
```

Listing 30: The syntax of Instr in Haskell with accompanying low-level data types

statements that will be generated, and `Exp` represents the sequential computation, which is a value that will be generated.

The semantics of Instr is similar to what its names suggest. `CInitChan` represents operations that initialize a channel according to the given type and cid. `CDeleteChan` will destroy a channel. `CSend` operation sends the value `Exp a` through the Channel. `CRecv` action means the value received in the channel will be assigned to a variable whose postfix name is the int field. `CEnd` means the instruction exits with the value `Exp a`. `CDecla` and `CAssgn` are instructions for variable declaration and assignment. The type of the variable is determined by `ReprType a`, and the value is `Exp a`. `CBranch` and `CSelect` are used to express conditional control flow of the Instr language. SPar actions like broadcast are built on top of these operations. For `CBranch`, the first field represents the value of the Either type to be received via the channel, and two `Seq Instr`s represents the sequence of `Instr`s in the left branch and the right branch. For `CSelect`, the first field represents the variable containing the Either value, and the second field represents the variable whose value is assigned by the end result of the instructions from either the left branch or the right branch. The third and fourth fields represent the instructions in the left and right branches respectively.

7.1.2 Representation types

SPar programs cannot be fully parametric since the target language for code generation from SPar are usually less expressive, i.e., they do not treat function type `a->b` as a value, and are less efficient in dealing with some certain forms of data, e.g. languages targeting GPUs are usually more productive in dealing with arrays of floating point numbers while slow when working with aggregate structures [26].

```haskell
data ReprType a where
    NumReprType :: NumType a -> ReprType a
    LabelReprType :: ReprType Label
    SumReprType :: ReprType a -> ReprType b -> ReprType (Either a b)
    UnitReprType :: ReprType ()
    ProductReprType :: ReprType a -> ReprType b -> ReprType (a, b)
    ListReprType :: ReprType a -> ReprType [a]
```

Listing 31: The definition of representation types

```haskell
constToCExpr :: ReprType a -> a -> CExpr
constToCExpr (NumReprType numType) v = numTypeToCExpr numType v
constToCExpr LabelReprType         v = case v of
  Le -> cVar "LEFT"
  Ri -> cVar "RIGHT"
constToCExpr s@(ProductReprType a b) v = defCompoundLit
  (show s)
  [ ([], initExp $ constToCExpr a (fst v))
  , ([], initExp $ constToCExpr b (snd v))
  ]
```

Listing 32: An example usage of reified type in the code generation

Hence, we need to restrict the set of types allowed in SPar. We achieve this using the typeclass `Repr` and corresponding reified type `ReprType` (shown in Listing 31). `Repr` determines the set of types allowed in SPar. Reified type `ReprType` will be used to alter the behavior of code generation based on the type. This can be simply done by pattern matching because reified types are values in Haskell [27]. To be more concrete, Listing 32 gives an example. `constToCExpr` is a function that handles code generation from constant values to expressions in the C programming language. By pattern matching, we vary the behaviors of code generation so that constants with different types have their own way to be represented in C.

In conclusion, we allow the following types: numerical type like `Float` and `Int`, the unit type `()`, the label type which is used in the code generation of select and branch and the aggregate type: list, product and sum that are built recursively, to be expressed in SPar.

7.2 Compilation from SPar to Instr

7.2.1 Transformation from Proc to Instr

As described in the previous section, Instr contains a data type called `Exp` which is a wrapper of Core expressions. Compiling Core to Instr is hence not difficult. The challenge of compiling Core is mainly how to compile it to a specific target language. This

will be discussed in the next subsection.

In this section, we will explain how we transform operations in Proc to Instr. Generally speaking, each Proc operation is mapped to a sequence of actions in Instr. The transformation algorithm from a Proc expression to a sequence of Instr can be implemented easily by traversing Proc expressions, applying the mapping and collecting the results by concatenation. This is an advantage of using the free monad technique to build the AST because Proc expressions can be treated as data structures and traversing recursive data structures can be easily done in Haskell. Also, operations like `Recv` which involves continuations whose type is `Core a -> next` in their constructors are treated differently than those operations whose constructors only have a value type `next`. The latter is easy to implement; we can simply call the traversing function. For the former, we have to pass an expression whose type is `Core a` to the continuation to call the traversing function on the result of applying a value to the continuation. The exact value of that Core expression is not known at this stage, so we use the Var constructor to build the expression. Passing a unique variable to the continuation gives us `next` inexpensively, and we will define where the value of variables come from, for each operation in Proc.

We have introduced the general principle to the readers. Now let us dig into details of the translation rules for each operation.

- **Pure.** It is the base case in a free monad. Hence it is mapped to the `CEnd` instruction.

- **Send.** It is mapped to a sequence of three instructions. First of all, we declared a temporary variable using `CDecla` and then assign the value that will be sent to this variable using `CAssgn` and send the content of the variable via the specific channel. The problem of how to make sure the same channel is used in a send-and-receive pair will be discussed in the next sub section.

- **Recv.** It is the reverse of the send operation. Firstly, it declares a new variable with `CDecla` and uses `CRecv` to assign the value received from the sender to this variable. Notice that `Recv` has a continuation, we will pass the variable declared in the first `Instr` to the continuation to traverse the Proc expression recursively as discussed above.

- **Select.** It is a more complicated operation. Its constructor contains two continuations: one for left branch and one for the right branch. Hence, for this instruction, we need to declare two variables to be passed into the continuations. The value of both variables is assigned by the Core expression, whose value is Either type. Besides, we need to send a label indicating whether the execution of the left branch or the right branch of the receiver is selected. The value of the label is determined by the either value. Sending of the label is done by `CSend`. Finally, we call the transforming function recursively on the left branch and right branch and combine the results using `CSelect`.

- **Broadcast.** The mapping from Broadcast is similar to that of Select. The only difference is that the former sends to a list of receivers while the latter sends to a

single receiver. So in this operation, we will have multiple `CSend` corresponding to each receiver.

- **Branch.** It is the reverse of the Select operation. So it will use `CRecv` to receive a label from the sender and call on two branches and finally use `CBranch` to collect results.

7.2.2 Strategies for channel allocation

Channel allocation is important because correct allocation is essential in making sure the correctness and deadlock-freedom of generated code. Besides correctness concerns, we are also want to reduce the number of channels hence increase performance.

In the first iteration, inspired by the linearity of channels in the π calculus, we allocate a one-time channel for each send-and-receive pair. All channels' buffer size is one because of the linearity. Send actions will initialize a channel and receive actions will destroy this channel once they receive the value. We ensure the same channel is used for a pair of a send action and a receive action. During the transformation, we use a map of queues whose key is a pair of sender id and receiver id. When visiting the send action, it will push the channel into the queue, and the corresponding receive operation will pop the channel from the queue. Because we've ensured the duality of all processes in the system, we can claim that the channel is right for each pair of sender id and receiver id. However, we realize this strategy is complicated to implement and not resource efficient since too many channels are initialized.

In the second iteration, we defined a more efficient and simple strategy. The buffer size of all channels is still one due to the same reason about linearity. However, we decided to only allocate one channel for a pair of sender and receiver. We will not destroy the channel after the value has been received, and we will reuse the channel for the next communication. When all processes have returned, we will destroy all channels at once. For this strategy, we have simplified the state from a map of queues of channels to a map of channels.

7.2.3 Monad for code generation

From the last two subsections, we need to maintain several states during the compilation process. Hence we define a state monad to be used during the traversal. The `CodeGenState` is the collection of states and it is shown in Listing 33. Each state plays its own role in the code generation process. `chanTable` is the map we required during the channel allocation. `varNext` represents the next variable id to used. It will increment by one every time we declare a new variable. It helps us make sure the variable names are unique. `chanNext` has a similar functionality ensuring the uniqueness of channel names. `dataStructCollect` is the set of compound types we defined by traversal.

```haskell
data CodeGenState = CodeGenState
  {
    chanTable :: Map ChanKey CID,
    varNext   :: Int,
    chanNext  :: CID,
    dataStructCollect :: Set AReprType
  }

newtype CodeGen m a =
  CodeGen { runCodeGen :: StateT CodeGenState m a }
    deriving (Functor,
              Applicative,
              Monad,
              MonadState CodeGenState,
              MonadIO)
```

Listing 33: States required during the traversal

7.3 Code generation to C: from Instr to C

The last piece of the jigsaw is compilation from a sequence of Instr to C. This is done by transforming the sequence of Instr to a C AST. We used an open source library: language-c [25] to represent C AST in Haskell and pretty-printing the C AST gives us the generated code. This method can be generalized to any target language. As for the implementation of channel communication in C, we used another open source library: chan [28] whose internals are based on shared memory. In the following subsections, we will present some of our design choices during this last step.

7.3.1 Representations of Core data type in C

The first challenge we face is how to represent data structures in C. For primitive data types like Int or Float, a simple one-to-one mapping is sufficient. It is hard to deal with algebraic data types in C. First of all, C does not support polymorphic types. Hence, we choose to generate specific data types for every different compound data type even though they have the same structure. The drawback of this approach is an explosion on generated code size due to the duplication on data type declarations. We have a way to name the generated data type to avoid name duplication. The naming simply reflects the structure of the data types with its elemental type. For example, (Int, (Int, ())) is converted to Prod-Int-Prod-Int-Unit and Either (Either () (Int, Int)) (Int, Int) is convert to Sum-Sum-unit-Prod-int-int-Prod-int-int. In this project, all compound types are formed by sum types and product types. The product type will be converted to a struct with two fields in C. The sum type is represented by the tagged union type. The tagged union is a struct with two fields. The value of the first field indicates whether it is a left value or right value and the value of the second field is a union type containing either the

```c
typedef enum Label {
            LEFT, RIGHT
        } Label;

typedef struct Prod_int_int {
            int fst; int snd;
        } Prod_int_int;

typedef struct Sum_unit_Prod_int_int {
            Label label;
            union {
                int left; Prod_int_int right;
            } value;
        } Sum_unit_Prod_int_int;
```

Listing 34: Compound data type in C

left value or right value. We also implement a sorting algorithm based on the depth of compound types so that all necessary data types have been defined before the definition of the compound type. An example of what Either Int (Int, Int) will be converted to is shown in Listing 34.

Another challenge is the representation of recursive types. From type theory, we learn that a list of integers can be expressed as $\mu a.() + \text{Int} \times a$. We might reuse the idea from the last paragraph to generated recursive types in terms of sums and products. Hence a list of Int will look like the code below.

```c
typedef struct Sum_unit_Prod_int_a {
    Label label;
    union {
        int left;
        Sum_unit_Prod_int_a *right;
    } value;
} Sum_unit_Prod_int_a
```

However, we believe expressing typical recursive data structures like a list of integers in this way is bad for performance. C has a more efficient way to represent lists of integers using arrays. So we decided to have two ways of representing recursive data structures. For a set of specific recursive data structures, users can write their own representation to exploit the advantages of the target language. For example, a list of integers is encoded in C using a wrapper of pointer types (shown in Listing 35). This way is not very generic but better for performance. For other types of recursive data structures where user does not specify their optimized versions in C, we simply apply the method in the last paragraph to encode them in C. This way is generic but not efficient.

```c
typedef struct List_int {
        size_t size; int * value;
    } List_int;
```

Listing 35: Optimized representation of List in C

7.3.2 Compiling from Core to C

$$\text{Var}\frac{}{[\![\text{Var n}]\!] = \text{vn}} \qquad \text{Lit}\frac{}{[\![\text{Lit val}]\!] = \text{toC(val)}}$$

$$\text{Fst}\frac{[\![\text{a}]\!] = c}{[\![\text{Fst `ap` a}]\!] = c.\text{fst}} \qquad \text{Snd}\frac{[\![\text{a}]\!] = c}{[\![\text{Snd `ap` a}]\!] = c.\text{snd}}$$

$$\text{Inl}\frac{[\![\text{a}]\!] = c}{[\![\text{Inl `ap` a}]\!] = \{\text{LEFT}, c\}} \qquad \text{Inr}\frac{[\![\text{a}]\!] = c}{[\![\text{Inr `ap` a}]\!] = \{\text{RIGHT}, c\}}$$

$$\text{Pair}\frac{[\![\text{a}]\!] = c_1, \quad [\![\text{b}]\!] = c_2}{[\![\text{Pair a b}]\!] = \{c_1, c_2\}}$$

$$\text{Id}\frac{[\![\text{a}]\!] = c}{[\![\text{Id `ap` a}]\!] = c} \qquad \text{Const}\frac{[\![\text{a}]\!] = c, \quad [\![\text{v}]\!] = b}{[\![\text{(Const v) `ap` a}]\!] = b}$$

$$\text{Prim}\frac{[\![\text{a}]\!] = c}{[\![\text{(Prim fname fimpl) `ap` a}]\!] = \text{fname}(c)}$$

Figure 7.1: Rules for compilation from Core to C

Core has a concise syntax so it does not require too much work to write a function that generates C expressions from Core expressions. Not surprisingly, the compilation is a traversal of the Core expressions. The `ap` (apply) constructor is used with an expression whose type is `Core (a -> b)` and another expression whose type is `Core b`. The code generation for `ap` depends on the what the function expression is. The code generation rule is explained by the inference rules shown in Figure 7.1. $[\![a]\!]$ means the C code generated by Core expression `a`. toC is the function that converts constant values in Haskell to constant values in C.

- **Var, Lit:** Code generation of Lit simply converts Haskell values to its corresponding C values. As for Var, it will be converted to a string literal composed of the variable identity prefixed by 'v'.
- **Fst, Inl, Pair...** For Inl and Inr and Pair, we used C99 style to initialize struct and union. The rule for generating the corresponding struct are explained in the previous subsection. For Fst and Snd, we simply access the specific value using the designator.

- **Id, Const v:** Code generation of Id constructor, is the same as the code generation of the argument of the Id. Code generation of Const ignores the argument and use the code generation of v expression instead.
- **Prim:** Prim constructor represents user-defined functions. The code generation for function call of Prim is converted to function call by the name of the primitive function. Users can implement those functions in C and include them in the main generated file.

7.3.3 The structure of generated C code

We have covered the code generation algorithm for each process. We will now tackle how the generated code is structured from a group of interacting processes representing a parallel computation. For each process, we will generate its own C functions that take no argument. We generate an additional C function for users to interact with generated code by calling the function. This function will take a parameter as input data and return the computation result. Inside, the function, it will spawn the same number of threads as the number of roles in the system. Each thread will execute the code which is generated from its corresponding Proc expression. In addition to that, the function will send the input parameter to the starting role in the group of Proc to kick off the computation and waiting for the ending role to send back the computation results. After it receives the result and all the threads have returned, it will return the result. We called this function `proc0`.

An example of the generated code is shown in Listing 36. The code contains one process which received a list of int from `proc0`, sort the input by a user-defined function and then send the result to `proc0`. We omit global channels declarations and data type declaration for simplicity.

7.4 Conclusion

With the completion of code generation, we deliver the results we promised in the introduction section. We have implemented an end-to-end procedure that will generate low-level deadlock-free parallel code from an expressive high-level language embedded with a flexible backend that can target multiple languages with ease. Now, it is time to evaluate the performance of our achievement with quantitative measurements.

```
void proc1Rt()
{
    List_int v0;
    chan_recv_buf(c1, &v0, sizeof(List_int));
    List_int v1;
    v1 = sort(v0);
    chan_send_buf(c2, &v1, sizeof(List_int));
}
void * proc1()
{
    proc1Rt();
    return NULL;
}
List_int proc0(List_int v0)
{
    c1 = chan_init(1);
    c2 = chan_init(1);
    pthread_t th1;
    pthread_create(&th1, NULL, proc1, NULL);
    chan_send_buf(c1, &v0, sizeof(List_int));
    List_int v1;
    chan_recv_buf(c2, &v1, sizeof(List_int));
    pthread_join(th1, NULL);
    chan_dispose(c1);
    chan_dispose(c2);
    return v1;
}
```

Listing 36: An example of generated code

Chapter 8

Parallel algorithms and evaluation

In this section, we give an overview of how to use SPar for implementing parallel algorithms, benchmarking the performance of the generated code for various computations and analyzing the design choices of the project.

8.1 Parallel algorithms

The biggest advantage of writing parallel programs in SArrow is that the user can express the computation similar to the sequential one, without worrying about any low-level primitives for parallel computation. We will give a recommended recipe of expressing computations in SArrow and a concrete example for the explanation.

8.1.1 Four steps to write parallel algorithms in SArrow

Usually, divide-and-conquer algorithms are the best candidates for SArrow to parallelize. The recipe is below:

1. Understand the algorithm: We recommend programmers to express the algorithms using recursion schemes (see Section 2.2) Recursion schemes is recommended because it separates the split function, the merge function and the structure of divide-and-conquer from each other, so it helps you familiarize with the building blocks of the divide-and-conquer algorithms, i.e., the data structures involved, the type signature of the split function, the type signature of merge functions and their implementation.

2. Systematic parallelization: In the first iteration, we can express the split, merge functions and other necessary helper functions in terms of SArrow, by combing `arr` constructor and `Prim` constructor. Every computation wrapped by `arr` with `Prim` is considered to be sequential. We will substitute them into high-level parallel patterns provided by SArrow. For example, divide-and-conquer algorithms can be parallelized by `divConquer` (see Listing 27) helper functions. Notice that the number determining the level of a parallelized divide-and-conquer algorithm should be set accordingly by the number of cores of the execution machine.

3. Specific parallelization: The above step is generic and can be applied to any divide-and-conquer algorithm. In this stage, what will be done is determined by the specific implementation. The programmer should inspect the implementation of these functions wrapped by `arr` with `Prim`, to see whether there is any parallelism to exploit. If so, the programmer should rewrite these functions using the parallel patterns provided by SPar or arrow combinators. For example, the split function of the Quickhull [29] to solve convex hull problems will use a for-loop to find the point whose distance to the line is at the maximum. This step can be expressed by the parallel map-and-reduce pattern (see Listing 26). Programmers can apply this step iteratively until all possible sources of parallelism are exploited, or programmers think it is enough. All sequential computation is left in the form: `arr` with `Prim`.

4. Wrap up: Before code generation, the programmer needs to implement all the `Prim` functions in terms of the target language. In the scope of this project, programmers will write them in C and create a header file. The generated code will include the header file, and from then on, programmers obtain the parallelized version of the algorithm that is guaranteed to be deadlock-free.

8.1.2 Example: Merge sort

We will show how to use the framework to generate parallel code by expressing merge sort on a list of integers.

```
split :: SArrow [Int] (Either (Either () Int) ([Int], [Int]))
split = arr $ Prim "split" undefined

merge :: SArrow (Either (Either () Int) ([Int], [Int])) [Int]
merge = arr $ Prim "merge" undefined

sort :: SArrow [Int] [Int]
sort = arr $ Prim "sort" undefined
```

Listing 37: The code for atomic functions

1. The first step is to understand the algorithm. The merge sort is a famous divide-and-conquer algorithm. It splits the list into two halves, applies the algorithm to sub-lists recursively, and finally merges the sub-lists in order. From the above description, we identify three atomic functions `split`, `merge` and the base function `sort`. We need to define the type signatures for these functions to understand what data structures are involved. For the `split` function, it will take a list of integers `[Int]` as an input and output a pair of lists of integers `(Int, Int)` as an output. To deal with the case where the input list cannot be split i.e. when the list is empty or a singleton list, we wrap the output pair with an Either type to deal with these situations. So the output type is `Either (Either () Int) ([Int], [Int])` Accordingly, the type signature of `merge` is the reverse of the `split`. `sort` type

signature will be the same as the merge sort: `[Int] -> [Int]`. Finally, we wrap them using Prim and arr. The code written for the first step is shown in Listing 37.

```
mergesort :: Int -> SArrow [Int] [Int]
mergesort = divConq sort split merge

divConq ::
     SArrow a b
  -> SArrow a (Either c (a, a))
  -> SArrow (Either c (b, b)) b
  -> Int
  -> SArrow a b
divConq baseFunc _ _ 0 = baseFunc
divConq baseFunc alg coalg x =
  alg
    >>> (   arr Inl
        ||| (   (   (arr Fst >>>
                     divConq baseFunc alg coalg (x - 1))
                &&& (arr Snd >>>
                     divConq baseFunc alg coalg (x - 1))
                )
            >>> arr Inr
            )
        )
    >>> coalg
```

Listing 38: Construction of the algorithm using the parallel pattern and atomic functions

2. The second step is combining these atomic functions in the first step in a parallel pattern that we define. Since we are using our framework to parallelize a merge sort, we will use the divide-and-conquer parallel pattern. The result of applying the pattern is an expression whose type is `Int -> SArrow [Int] [Int]` where the first parameter determines the number of levels of the divide-and-conquer algorithm. In this case, we will use our refined version of the divide-and-conquer parallel pattern that supports shortcut. The code written for the second step is shown Listing 38. For the completeness, we also include the implementation of the parallel pattern. The polymorphic parallel pattern can be used to generate non-polymorphic code.

3. The third step is optimizing the atomic functions. Since split and merge are not very intensive computation. We will not modify anything for this step.

4. Up to this step, we have finished everything we need to do in the Haskell side to express computation. First of all, we will show how to generate C code from a SArrow expression using the framework. Secondly, we will talk about how to complete the implementation for the atomic functions. In the library, we have defined a

function `codeGen` that takes an SArrow expression as an input and generates three files in the specified path. The first file is called func.h, which contains declarations of all atomic functions. The second file is data.h, which includes the definition of data structures involved in C. The third file is code.c, which includes all the necessary headers like the standard library and channel library and generated code. The structure is similar to what we described in Section 7.3.3. The last job to do is to complete the implementation of the functions declared in the func.h. After that, we use the generated code in whatever way we want. The structures are shown in Table 8.1.

```
0: !<1, [Int]>.?(4, [Int]).end
1: ?(0,[Int]).Br<[2,3,4],
    {L: !<4,Either (Either () Int) ([Int],[Int])>.end,
     R: !<3,([Int],[Int])>.Br<[2],
        {L: !<2,Either (Either () Int) ([Int],[Int])>.end,
        R: !<2,([Int],[Int])>.!<2,[Int]>.end}>
        .end}>
    .end
2: &(1,
    {L: end,
    R: &(1,
        {L: ?(1,Either (Either () Int) ([Int],[Int])).end,
        R: ?(1,([Int],[Int])).?(1,[Int]).end}).!<4,[Int]>
        .end})
    .end
3: &(1,{
    L: end,
    R: ?(1,([Int],[Int])).Br<[4],
        {L: !<4,Either (Either () Int) ([Int],[Int])>.end,
        R: !<4,([Int],[Int])>.!<4,[Int]>.end}>
        .end})
    .end
4: &(1,
    {L: ?(1,Either (Either () Int) ([Int],[Int])).end,
    R: &(3,
        {L: ?(3,Either (Either () Int) ([Int],[Int])).end,
        R: ?(3,([Int],[Int])).?(3,[Int]).end})
        .?(2,[Int]).end})
    .!<0,[Int]>.end
```

Listing 39: Inferred session types

In addition to the generated code, we also show the inferred session types of each role in the system for the merge sort at level two in Listing 39. We can call the function `runType` on any SArrow expression to get a list of session types. The session types are

pretty printed. The original representation of session types is a free monad as explained in Chapter 4. `br` is a syntax sugar for a series of select (see in Section 4.3.2).

8.2 Benchmarks

```
int main()
{
    int * tmp = randomList(1048576);
    List_int a = (List_int) {1048576, tmp};
    double start = get_time();
    proc0(a);
    double end = get_time();
    printf("%lf\n", end - start);
    return 0;
}
```

Listing 40: The main function for benchmark

We have defined a specific code generation function for benchmarking the parallel algorithms. The main difference from the normally generated code is the main function. The main function will create a random source data by the specified input size, record the execution time and output the execution time. The main file is shown in Listing 40.

For each benchmark of a divide-and-conquer SArrow, we will generate the code for a range of sizes and a different number of recursion unrollings representing the level of the algorithm. We will record the execution time on high-performance computer.

The compiler we used is gcc (version 9.1.0) with the default optimizations. The platform on which we run our benchmarks is a 32-core high throughput computing machine provided by Imperial College London ICT services.

We have implemented three benchmarks to run.

1. `MergeSort`: It is one of the most classic divide-and-conquer algorithms. Its details are shown in the above subsection.

2. `DotProduct`: It computes the inner product of two vectors whose sizes are the same and of the form 2^n.

3. `IntCount`: It counts the number of occurrences of integer given a list of integer ranging from 0 to 50. `split` divides the list by halves. `count` will count the occurrence of integers and output a list of tuple where the integer is the value and the second integer is the number of occurrences of that value. `union` will union the resulting of sub-lists by summing up the occurrence.

8.2.1 Evaluation

We will demonstrate the execution time against different sizes for different levels of divide-and-conquer algorithms as well as the speedup of the parallel algorithms against the sequential algorithm.

Figure 8.1 shows the result for different cases running on a 32-core machine. For each level k, we will generate a total number of 2^k threads to execute the parallel computation. The x-axis indicates the size of the input where 22 means the input is a list of 2^{22} integers or a pair of a list of 2^{22} integers. The execution time is measured in seconds and the speedup is computed by $\frac{t_{\text{sequential}}}{t_{\text{parallel}}}$.

For `MergeSort`, the speedup increases as the size increases, which is shown by the increasing graph from Figure 8.1a. Different levels will have a varying degree of a performance boost on growing sizes. For `DotProduct`, this trend holds for up to size 26. When the size is 26, we witness a sudden decrease in speedup at various levels. The reason for this abnormal behavior is yet to be studied. In term of `IntCount`, the speedup increases against increasing sizes when the level is big enough. The degree of speedup increase at the large level is also the most obvious for `IntCount` among the three benchmarks. This can be observed by the two slopes when the level is equal to seven and eight. Unexpected behavior is the line when the level is equal to six. Its speedup is nearly as small as level = 1. Also, the overproduction of threads has positive impacts on the speedup. The best level is seven which contains 128 threads. It gives a 7.5X speedup for `MergeSort`, 8X speedup for `DotProduct` and nearly 12X speed up for `IntCount` when the size is 2^{26}. In general, a greater number of threads has better speedup for greater sizes. The level that gives the best performance depends on the number of cores of the machine. The relationship is not merely one-to-one, and from the experiment, we recommend to overproduce the number of threads compared to the number of cores. However, the performance will not be significantly increased and even slower if the level is too big (see the comparison of speedup line when the level is equal to seven and eight). Also, one may argue that speedup around 10X is not a great achievement on a 32-core machine. However, the expectation of a 30X speedup is not realistic because of Amdahl's law: the sequential part of the algorithm ends up dominating the execution time. For example, the split and merge functions in `MergeSort` are not parallelized in our implementation. What the law tells us is that if these operations take up $p\%$ of the sequential execution time, the theoretical speedup is limited to at most $\frac{1}{p\%}$ [30]. To be more specific, if these operations account for 5% of total time, then the limit of speedup is at most $\frac{1}{0.05} = 20$X no matter how many processors are used. The sequential execution becomes the bottleneck. We have not investigated too much on optimization of sequential code since this is not the scope of the project. But the speedup could be more notable once optimizations for the sequential code were done. In general, an algorithm where the inherently sequential part take less time will achieve higher speedups.

```c
typedef enum Label {
            LEFT, RIGHT
        } Label;
typedef struct List_int {
            size_t size; int * value;
        } List_int;
typedef struct Sum_unit_int {
            Label label;
            union {
                int left; int right;
            } value;
        } Sum_unit_int;
typedef struct Prod_List_int_List_int {
            List_int fst; List_int snd;
        } Prod_List_int_List_int;
typedef struct Sum_Sum_unit_int_Prod_List_int_List_int {
            Label label;
            union {
                Sum_unit_int left; Prod_List_int_List_int right;
            } value;
        } Sum_Sum_unit_int_Prod_List_int_List_int;
```

(a) data.h

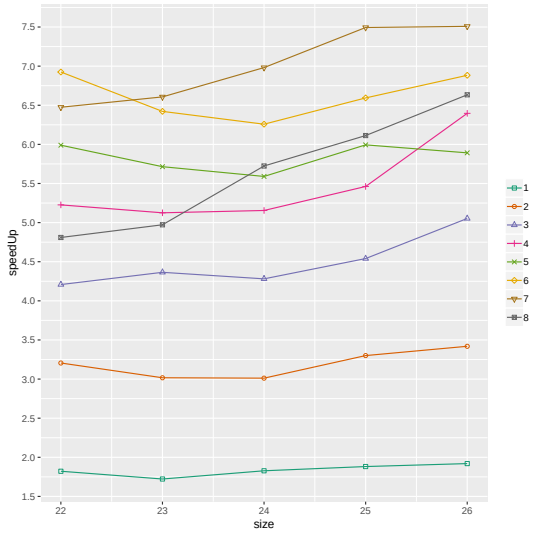
```c
#include "data.h"
List_int merge(Sum_Sum_unit_int_Prod_List_int_List_int);
Prod_List_int_List_int split(List_int);
List_int sort(List_int);
```

(b) func.h

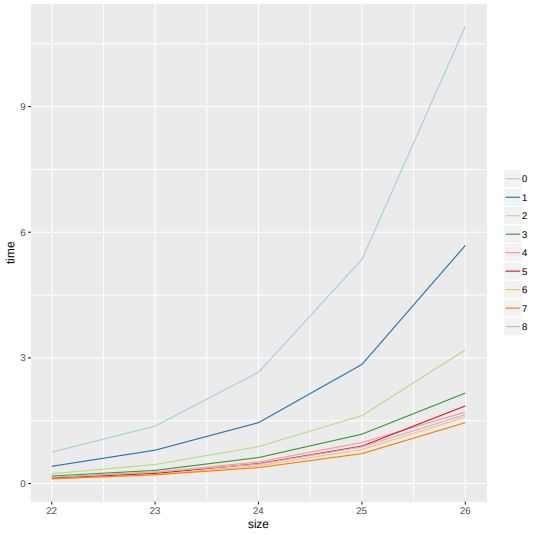
```c
#include<stdint.h>
#include<stdlib.h>
#include<chan.h>
#include<pthread.h>

#include"data.h"
#include"func.h"

List_int proco(List_int a) {

}
```

(c) code.c

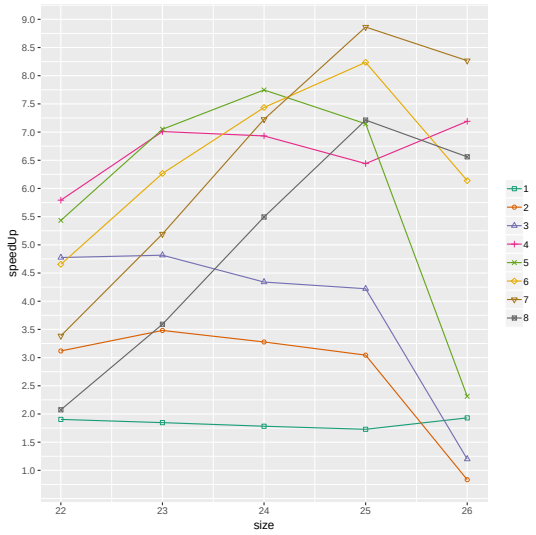
Table 8.1: The complete structure of the generated c code. Omit the implementation of code.c



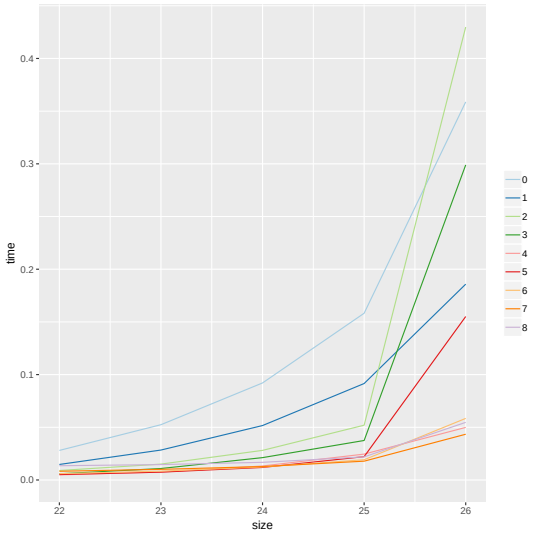
(a) Merge sort: speedup



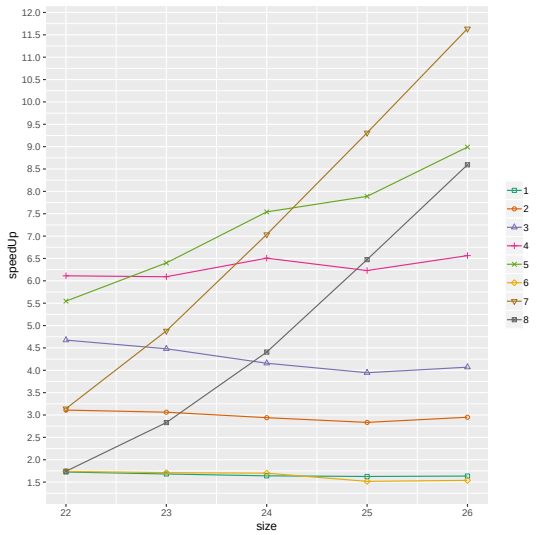
(b) Merge sort: execution time



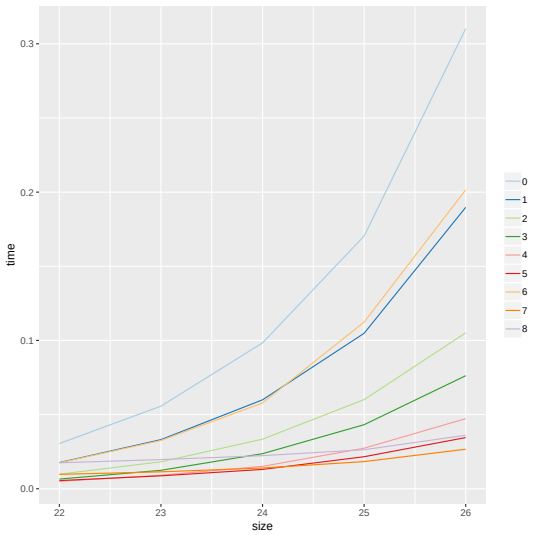
(c) Dot product: speedup



(d) Dot product: execution time



(e) Int count: speedup



(f) Int count: execution time

Figure 8.1: Benchmarks results

Chapter 9

Conclusions and future works

9.1 Conclusions

Our goal was to implement a backend for code generation for Alg parallel language using a session-typed intermediate language. We not only achieved this but also developed a high-level framework for parallel computation. The most important result of the project is a framework that generates parallel C code along with local types describing the communication patterns by interpreting data-flow as communication, from Arrow based high-level expressions that can be easily formed, composed and manipulated by users with the help of the host language: Haskell.

Specifically, we developed SPar: a session-typed free monad EDSL for message passing concurrency (see in Chapter 4) as our intermediate language. Also, we developed tools like local type inferrer to help us reason about the underlying communication structure with external tools by inferring session types from SPar expressions as well as a simulator to aid experimenting and act as a reference semantics (see in Chapter 5). On top of this, we draw inspirations from the Arrow interface and developed SArrow: an interface for writing SPar expressions (see in Chapter 6) to form complex computation patterns such as parallel divide-and-conquer and parallel map in a composite way. We designed a code generation backend from SPar to C. The core of the backend is Instr : a low-level EDSL for channel communication we created in Section 7.1. Code generation pipeline benefits from Instr's ease of transformation to a C AST. Finally, our benchmarks (see in Chapter 8) show that our framework can generate efficient parallel code and gain a notable performance boost compared to the sequential code. The best case gives us a speedup of nearly 12X when the input size is 2^{26} integers. In Section 8.1, we use a concrete example: expressing parallel merge sort in our framework to demonstrate that users can express computation in SArrow in a similar way as to how they would write for the sequential version and gain high-performance parallel code automatically.

In conclusion, with the recent release of AMD's latest generation of consumer CPUs featuring a processor with sixteen physical cores, Moore's law will be replaced by the addition of cores. No need to mention the area of high-performance computing where CPU with 64 cores are common. In contrast, most of the programmers have only written sequential code, and most of the algorithms about which students learn are not parallel.

We hope this project can contribute its force on parallelism on CPUs, encouraging more and more programmers to take advantages of modern computing architectures.

9.2 Future work

There are many interesting future works that we would like to implement. We will select some of them to introduce:

- **Optimization for benchmark.** Because of the time constraints, there are lots of space to optimize the generated code. We should do more fine-grained profiling on the generated code. It is interesting to use tools like EzTrace to trace and visualize the execution of all the threads. More importantly, reducing the size of the generated code by eliminating common sub-expressions will be useful. At the moment, there is much code duplication for communication among different roles. The only difference is that the role of participating in the communication. The size of generated code can be reduced a lot if we can extract the common part to a function parameterized by the roles participating.

- **Integrated user experience.** As demonstrated in the evaluation chapter, users need to write the computation using the EDSL in Haskell and then generate C code. From then on, they need to finish the implementation of their atom functions in C. Finally, they can run the generated code with their data in C. The user experience is isolated when you have to write Haskell first and manually completed the generated code and run them in C. Instead, it will be great if we can provide an integrated user experience where the user does everything in Haskell from writing the high-level expression to collect computation results. This is possible thanks to packages like inline-C and foreign language interface in Haskell. User experience will be greatly improve if we can offer an interface in Haskell that looks like `run :: SArrow a b -> (a -> b)`. This function will take a SArrow expression and produces a function that will convert a Haskell value into C data and execute the computation in C and copy back the C output by foreign language interface to Haskell. From the user pointer of way, it can be used the same as a normal Haskell function with type `a -> b`. Forming a closed loop in Haskell would give us the best user interface and automate a large amount of boilerplate work.

- **Fine-grained control for strategies in role allocation.** We talked about how different role allocation strategies give us different parallel computation. It will be great if we parameterize the SArrow with role allocation strategies and adding ways to specify what strategy will be used at a different stage of the computation. This also opens the possibility for users to implement their strategies to customize their parallel computation tasks.

- **More customizations.** Similar to customized role allocation strategies, we can even have customized representation of sequential computation since the separation of the communication EDSL and the sequential computation EDSL. This can be

done by parameterizing `Core` in `Proc`. This kind of work requires well-designed interfaces.

Bibliography

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

[2] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A Heterogeneous Parallel Framework for Domain-Specific Languages," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, (Galveston, TX, USA), pp. 89–100, IEEE, Oct. 2011.

[3] R. Li, H. Hu, H. Li, Y. Wu, and J. Yang, "MapReduce Parallel Programming Model: A State-of-the-Art Survey," *International Journal of Parallel Programming*, vol. 44, pp. 832–866, Aug. 2016.

[4] M. Braun, O. Lobachev, and P. Trinder, "Arrows for Parallel Computation," *arXiv:1801.02216 [cs]*, Jan. 2018.

[5] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida, "A Gentle Introduction to Multiparty Asynchronous Session Types," in *Formal Methods for Multicore Programming* (M. Bernardo and E. B. Johnsen, eds.), vol. 9104, pp. 146–178, Cham: Springer International Publishing, 2015.

[6] J. Hughes, "Generalising monads to arrows," *Science of Computer Programming*, vol. 37, pp. 67–111, May 2000.

[7] "Haskell/Understanding arrows - Wikibooks, open books for an open world." https://en.wikibooks.org/wiki/Haskell/Understanding_arrows.

[8] "Tacit programming," *Wikipedia*, Jan. 2019. Page Version ID: 879102751.

[9] C. Elliott, "Generic functional parallel algorithms: Scan and FFT," *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1–25, Aug. 2017.

[10] D. Castro and N. Yoshida, "Algebraic Multiparty Protocol Programming," *Under submission*, p. 37.

[11] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, pp. 1–40, Sept. 1992.

[12] N. Ng, J. G. de Figueiredo Coutinho, and N. Yoshida, "Protocols by Default," in *Compiler Construction* (B. Franke, ed.), Lecture Notes in Computer Science, pp. 212–232, Springer Berlin Heidelberg, 2015.

[13] "Message Passing Interface." https://www.mcs.anl.gov/research/projects/mpi/.

[14] D. Orchard and N. Yoshida, "Session types with linearity in Haskell," *Behavioural Types: from Theory to Tools*, p. 219, 2017.

[15] K. Claessen, "A poor man's concurrency monad," *Journal of Functional Programming*, vol. 9, no. 3, pp. 313–323, 1999.

[16] "Control.Monad.Cont." http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Cont.html.

[17] S. Marlow, R. Newton, and S. Peyton Jones, "A monad for deterministic parallelism," in *ACM SIGPLAN Notices*, vol. 46, pp. 71–82, ACM, 2011.

[18] "Free monad in nLab." https://ncatlab.org/nlab/show/free+monad.

[19] C. contributors, "Cats: FreeMonads." https://typelevel.org/cats/datatypes/freemonad.html.

[20] J. Lange and N. Yoshida, "Verifying Asynchronous Interactions via Communicating Session Automata," *arXiv:1901.09606 [cs]*, Jan. 2019.

[21] J. Svenningsson and E. Axelsson, "Combining deep and shallow embedding of domain-specific languages," *Computer Languages, Systems & Structures*, vol. 44, pp. 143–165, Dec. 2015.

[22] "Data.Typeable." http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Typeable.html.

[23] M. D. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programing: Patterns for Efficient Computation*. Amsterdam: Elsevier, Morgan Kaufmann, 2012.

[24] R. Atkey, "What is a Categorical Model of Arrows?," *Electronic Notes in Theoretical Computer Science*, vol. 229, pp. 19–37, Mar. 2011.

[25] "Language.C." http://hackage.haskell.org/package/language-c-0.8.2/docs/Language-C.html.

[26] T. L. McDonell, M. M. Chakravarty, V. Grover, and R. R. Newton, "Type-safe runtime code generation: Accelerate to LLVM," *ACM SIGPLAN Notices*, vol. 50, no. 12, pp. 201–212, 2016.

[27] "Reified type - HaskellWiki." https://wiki.haskell.org/Reified_type.

[28] T. Treat, "Pure C implementation of Go channels. Contribute to tylertreat/chan development by creating an account on GitHub." https://github.com/tylertreat/chan, June 2019.

[29] C. B. Barber, D. P. Dobkin, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.

[30] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pp. 483–485, ACM, 1967.

Appendix A

Examples of generated code

A.1 Merge sort

This is the generated code.c at level 3 for merge sort.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<chan.h>
4  #include<pthread.h>
5  #include"data.h"
6  #include"func.h"
7  chan_t * c1;
8  chan_t * c2;
9  chan_t * c3;
10 chan_t * c4;
11 chan_t * c5;
12 chan_t * c6;
13 chan_t * c7;
14 chan_t * c8;
15 chan_t * c9;
16 chan_t * c10;
17 chan_t * c11;
18 chan_t * c12;
19 chan_t * c13;
20 chan_t * c14;
21 chan_t * c15;
22 chan_t * c16;
23 chan_t * c17;
24 void proc1Rt()
25 {
26     List_int v0;
27     chan_recv_buf(c1, &v0, sizeof(List_int));
28     Sum_Sum_unit_int_Prod_List_int_List_int v1;
29     v1 = split(v0);
30     Label v2;
31     v2 = v1.label;
32     chan_send_int(c2, v2);
33     chan_send_int(c3, v2);
34     chan_send_int(c4, v2);
35     chan_send_int(c5, v2);
36     chan_send_int(c6, v2);
37     chan_send_int(c7, v2);
38     chan_send_int(c8, v2);
39     Sum_unit_int v3;
40     Prod_List_int_List_int v4;
41     int v5;
42     if (v1.label == LEFT)
43     {
44         v3 = v1.value.left;
45         Sum_Sum_unit_int_Prod_List_int_List_int v6;
46         v6 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v3 }};
47         chan_send_buf(c8,
48                       &v6,
49                       sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
50         v5 = 0;
51     }
52     else
53     {
54         v4 = v1.value.right;
55         Prod_List_int_List_int v7;
```

```
56          v7 = v4;
57          chan_send_buf(c5, &v7, sizeof(Prod_List_int_List_int));
58          Sum_Sum_unit_int_Prod_List_int_List_int v8;
59          v8 = split(v4.snd);
60          Label v9;
61          v9 = v8.label;
62          chan_send_int(c2, v9);
63          chan_send_int(c3, v9);
64          chan_send_int(c4, v9);
65          Sum_unit_int v10;
66          Prod_List_int_List_int v11;
67          int v12;
68          if (v8.label == LEFT)
69          {
70              v10 = v8.value.left;
71              Sum_Sum_unit_int_Prod_List_int_List_int v13;
72              v13 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v10 }};
73              chan_send_buf(c4,
74                            &v13,
75                            sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
76              v12 = 0;
77          }
78          else
79          {
80              v11 = v8.value.right;
81              Prod_List_int_List_int v14;
82              v14 = v11;
83              chan_send_buf(c3, &v14, sizeof(Prod_List_int_List_int));
84              Sum_Sum_unit_int_Prod_List_int_List_int v15;
85              v15 = split(v11.snd);
86              Label v16;
87              v16 = v15.label;
88              chan_send_int(c2, v16);
89              Sum_unit_int v17;
90              Prod_List_int_List_int v18;
91              int v19;
92              if (v15.label == LEFT)
93              {
94                  v17 = v15.value.left;
95                  Sum_Sum_unit_int_Prod_List_int_List_int v20;
96                  v20 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v17 }};
97                  chan_send_buf(c2,
98                                &v20,
99                                sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
100                 v19 = 0;
101             }
102             else
103             {
104                 v18 = v15.value.right;
105                 Prod_List_int_List_int v21;
106                 v21 = v18;
107                 chan_send_buf(c2, &v21, sizeof(Prod_List_int_List_int));
108                 List_int v22;
109                 v22 = sort(v18.snd);
110                 chan_send_buf(c2, &v22, sizeof(List_int));
111                 v19 = 0;
112             }
113             v12 = 0;
114         }
115         v5 = 0;
116     }
117 }
118 void proc2Rt()
119 {
120     int v24;
121     Label v23;
122     chan_recv_int(c2, &v23);
123     if (v23 == LEFT)
124     {
125         v24 = 0;
126     }
127     else
128     {
129         Label v25;
130         chan_recv_int(c2, &v25);
131         if (v25 == LEFT)
132         {
133         }
134         else
135         {
136             Sum_Sum_unit_int_Prod_List_int_List_int v27;
137             Label v26;
138             chan_recv_int(c2, &v26);
139             if (v26 == LEFT)
140             {
141                 Sum_Sum_unit_int_Prod_List_int_List_int v28;
```

```
142                 chan_recv_buf(c2,
143                               &v28,
144                               sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
145                 v27 = v28;
146             }
147             else
148             {
149                 Prod_List_int_List_int v29;
150                 chan_recv_buf(c2, &v29, sizeof(Prod_List_int_List_int));
151                 List_int v30;
152                 v30 = sort(v29.fst);
153                 List_int v31;
154                 chan_recv_buf(c2, &v31, sizeof(List_int));
155                 v27 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v30, v31} }};
156             }
157             List_int v32;
158             v32 = merge(v27);
159             chan_send_buf(c9, &v32, sizeof(List_int));
160         }
161         v24 = 0;
162     }
163 }
164 void proc3Rt()
165 {
166     int v34;
167     Label v33;
168     chan_recv_int(c3, &v33);
169     if (v33 == LEFT)
170     {
171         v34 = 0;
172     }
173     else
174     {
175         int v36;
176         Label v35;
177         chan_recv_int(c3, &v35);
178         if (v35 == LEFT)
179         {
180             v36 = 0;
181         }
182         else
183         {
184             Prod_List_int_List_int v37;
185             chan_recv_buf(c3, &v37, sizeof(Prod_List_int_List_int));
186             Sum_Sum_unit_int_Prod_List_int_List_int v38;
187             v38 = split(v37.fst);
188             Label v39;
189             v39 = v38.label;
190             chan_send_int(c10, v39);
191             Sum_unit_int v40;
192             Prod_List_int_List_int v41;
193             int v42;
194             if (v38.label == LEFT)
195             {
196                 v40 = v38.value.left;
197                 Sum_Sum_unit_int_Prod_List_int_List_int v43;
198                 v43 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v40 }};
199                 chan_send_buf(c10,
200                               &v43,
201                               sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
202                 v42 = 0;
203             }
204             else
205             {
206                 v41 = v38.value.right;
207                 Prod_List_int_List_int v44;
208                 v44 = v41;
209                 chan_send_buf(c10, &v44, sizeof(Prod_List_int_List_int));
210                 List_int v45;
211                 v45 = sort(v41.snd);
212                 chan_send_buf(c10, &v45, sizeof(List_int));
213                 v42 = 0;
214             }
215             v36 = v42;
216         }
217         v34 = v36;
218     }
219 }
220 void proc4Rt()
221 {
222     Label v46;
223     chan_recv_int(c4, &v46);
224     if (v46 == LEFT)
225     {
226     }
227     else
```

```
228     {
229         Sum_Sum_unit_int_Prod_List_int_List_int v48;
230         Label v47;
231         chan_recv_int(c4, &v47);
232         if (v47 == LEFT)
233         {
234             Sum_Sum_unit_int_Prod_List_int_List_int v49;
235             chan_recv_buf(c4,
236                           &v49,
237                           sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
238             v48 = v49;
239         }
240         else
241         {
242             Sum_Sum_unit_int_Prod_List_int_List_int v51;
243             Label v50;
244             chan_recv_int(c10, &v50);
245             if (v50 == LEFT)
246             {
247                 Sum_Sum_unit_int_Prod_List_int_List_int v52;
248                 chan_recv_buf(c10,
249                               &v52,
250                               sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
251                 v51 = v52;
252             }
253             else
254             {
255                 Prod_List_int_List_int v53;
256                 chan_recv_buf(c10, &v53, sizeof(Prod_List_int_List_int));
257                 List_int v54;
258                 v54 = sort(v53.fst);
259                 List_int v55;
260                 chan_recv_buf(c10, &v55, sizeof(List_int));
261                 v51 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v54, v55} }};
262             }
263             List_int v56;
264             v56 = merge(v51);
265             List_int v57;
266             chan_recv_buf(c9, &v57, sizeof(List_int));
267             v48 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v56, v57} }};
268         }
269         List_int v58;
270         v58 = merge(v48);
271         chan_send_buf(c11, &v58, sizeof(List_int));
272     }
273 }
274 void proc5Rt()
275 {
276     int v60;
277     Label v59;
278     chan_recv_int(c5, &v59);
279     if (v59 == LEFT)
280     {
281         v60 = 0;
282     }
283     else
284     {
285         Prod_List_int_List_int v61;
286         chan_recv_buf(c5, &v61, sizeof(Prod_List_int_List_int));
287         Sum_Sum_unit_int_Prod_List_int_List_int v62;
288         v62 = split(v61.fst);
289         Label v63;
290         v63 = v62.label;
291         chan_send_int(c12, v63);
292         chan_send_int(c13, v63);
293         chan_send_int(c14, v63);
294         Sum_unit_int v64;
295         Prod_List_int_List_int v65;
296         int v66;
297         if (v62.label == LEFT)
298         {
299             v64 = v62.value.left;
300             Sum_Sum_unit_int_Prod_List_int_List_int v67;
301             v67 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v64 }};
302             chan_send_buf(c14,
303                           &v67,
304                           sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
305             v66 = 0;
306         }
307         else
308         {
309             v65 = v62.value.right;
310             Prod_List_int_List_int v68;
311             v68 = v65;
312             chan_send_buf(c13, &v68, sizeof(Prod_List_int_List_int));
313             Sum_Sum_unit_int_Prod_List_int_List_int v69;
```

```
314             v69 = split(v65.snd);
315             Label v70;
316             v70 = v69.label;
317             chan_send_int(c12, v70);
318             Sum_unit_int v71;
319             Prod_List_int_List_int v72;
320             int v73;
321             if (v69.label == LEFT)
322             {
323                 v71 = v69.value.left;
324                 Sum_Sum_unit_int_Prod_List_int_List_int v74;
325                 v74 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v71 }};
326                 chan_send_buf(c12,
327                               &v74,
328                               sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
329                 v73 = 0;
330             }
331             else
332             {
333                 v72 = v69.value.right;
334                 Prod_List_int_List_int v75;
335                 v75 = v72;
336                 chan_send_buf(c12, &v75, sizeof(Prod_List_int_List_int));
337                 List_int v76;
338                 v76 = sort(v72.snd);
339                 chan_send_buf(c12, &v76, sizeof(List_int));
340                 v73 = 0;
341             }
342             v66 = 0;
343         }
344         v60 = v66;
345     }
346 }
347 void proc6Rt()
348 {
349     int v78;
350     Label v77;
351     chan_recv_int(c6, &v77);
352     if (v77 == LEFT)
353     {
354         v78 = 0;
355     }
356     else
357     {
358         Label v79;
359         chan_recv_int(c12, &v79);
360         if (v79 == LEFT)
361         {
362         }
363         else
364         {
365             Sum_Sum_unit_int_Prod_List_int_List_int v81;
366             Label v80;
367             chan_recv_int(c12, &v80);
368             if (v80 == LEFT)
369             {
370                 Sum_Sum_unit_int_Prod_List_int_List_int v82;
371                 chan_recv_buf(c12,
372                               &v82,
373                               sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
374                 v81 = v82;
375             }
376             else
377             {
378                 Prod_List_int_List_int v83;
379                 chan_recv_buf(c12, &v83, sizeof(Prod_List_int_List_int));
380                 List_int v84;
381                 v84 = sort(v83.fst);
382                 List_int v85;
383                 chan_recv_buf(c12, &v85, sizeof(List_int));
384                 v81 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v84, v85} }};
385             }
386             List_int v86;
387             v86 = merge(v81);
388             chan_send_buf(c15, &v86, sizeof(List_int));
389         }
390         v78 = 0;
391     }
392 }
393 void proc7Rt()
394 {
395     int v88;
396     Label v87;
397     chan_recv_int(c7, &v87);
398     if (v87 == LEFT)
399     {
```

```
400             v88 = 0;
401         }
402         else
403         {
404             int v90;
405             Label v89;
406             chan_recv_int(c13, &v89);
407             if (v89 == LEFT)
408             {
409                 v90 = 0;
410             }
411             else
412             {
413                 Prod_List_int_List_int v91;
414                 chan_recv_buf(c13, &v91, sizeof(Prod_List_int_List_int));
415                 Sum_Sum_unit_int_Prod_List_int_List_int v92;
416                 v92 = split(v91.fst);
417                 Label v93;
418                 v93 = v92.label;
419                 chan_send_int(c16, v93);
420                 Sum_unit_int v94;
421                 Prod_List_int_List_int v95;
422                 int v96;
423                 if (v92.label == LEFT)
424                 {
425                     v94 = v92.value.left;
426                     Sum_Sum_unit_int_Prod_List_int_List_int v97;
427                     v97 = (Sum_Sum_unit_int_Prod_List_int_List_int) {LEFT, { .left = v94 }};
428                     chan_send_buf(c16,
429                                   &v97,
430                                   sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
431                     v96 = 0;
432                 }
433                 else
434                 {
435                     v95 = v92.value.right;
436                     Prod_List_int_List_int v98;
437                     v98 = v95;
438                     chan_send_buf(c16, &v98, sizeof(Prod_List_int_List_int));
439                     List_int v99;
440                     v99 = sort(v95.snd);
441                     chan_send_buf(c16, &v99, sizeof(List_int));
442                     v96 = 0;
443                 }
444                 v90 = v96;
445             }
446             v88 = v90;
447         }
448     }
449     void proc8Rt()
450     {
451         Sum_Sum_unit_int_Prod_List_int_List_int v101;
452         Label v100;
453         chan_recv_int(c8, &v100);
454         if (v100 == LEFT)
455         {
456             Sum_Sum_unit_int_Prod_List_int_List_int v102;
457             chan_recv_buf(c8,
458                           &v102,
459                           sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
460             v101 = v102;
461         }
462         else
463         {
464             Sum_Sum_unit_int_Prod_List_int_List_int v104;
465             Label v103;
466             chan_recv_int(c14, &v103);
467             if (v103 == LEFT)
468             {
469                 Sum_Sum_unit_int_Prod_List_int_List_int v105;
470                 chan_recv_buf(c14,
471                               &v105,
472                               sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
473                 v104 = v105;
474             }
475             else
476             {
477                 Sum_Sum_unit_int_Prod_List_int_List_int v107;
478                 Label v106;
479                 chan_recv_int(c16, &v106);
480                 if (v106 == LEFT)
481                 {
482                     Sum_Sum_unit_int_Prod_List_int_List_int v108;
483                     chan_recv_buf(c16,
484                                   &v108,
485                                   sizeof(Sum_Sum_unit_int_Prod_List_int_List_int));
```

```
                v107 = v108;
            }
            else
            {
                Prod_List_int_List_int v109;
                chan_recv_buf(c16, &v109, sizeof(Prod_List_int_List_int));
                List_int v110;
                v110 = sort(v109.fst);
                List_int v111;
                chan_recv_buf(c16, &v111, sizeof(List_int));
                v107 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v110, v111} }};
            }
            List_int v112;
            v112 = merge(v107);
            List_int v113;
            chan_recv_buf(c15, &v113, sizeof(List_int));
            v104 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v112, v113} }};
        }
        List_int v114;
        v114 = merge(v104);
        List_int v115;
        chan_recv_buf(c11, &v115, sizeof(List_int));
        v101 = (Sum_Sum_unit_int_Prod_List_int_List_int) {RIGHT, { .right = (Prod_List_int_List_int) {v114, v115} }};
    }
    List_int v116;
    v116 = merge(v101);
    chan_send_buf(c17, &v116, sizeof(List_int));
}
void * proc1()
{
    proc1Rt();
    return NULL;
}
void * proc2()
{
    proc2Rt();
    return NULL;
}
void * proc3()
{
    proc3Rt();
    return NULL;
}
void * proc4()
{
    proc4Rt();
    return NULL;
}
void * proc5()
{
    proc5Rt();
    return NULL;
}
void * proc6()
{
    proc6Rt();
    return NULL;
}
void * proc7()
{
    proc7Rt();
    return NULL;
}
void * proc8()
{
    proc8Rt();
    return NULL;
}
List_int proc0(List_int v0)
{
    c1 = chan_init(1);
    c2 = chan_init(1);
    c3 = chan_init(1);
    c4 = chan_init(1);
    c5 = chan_init(1);
    c6 = chan_init(1);
    c7 = chan_init(1);
    c8 = chan_init(1);
    c9 = chan_init(1);
    c10 = chan_init(1);
    c11 = chan_init(1);
    c12 = chan_init(1);
    c13 = chan_init(1);
    c14 = chan_init(1);
    c15 = chan_init(1);
    c16 = chan_init(1);
```

```
572         c17 = chan_init(1);
573         pthread_t th1;
574         pthread_create(&th1, NULL, proc1, NULL);
575         pthread_t th2;
576         pthread_create(&th2, NULL, proc2, NULL);
577         pthread_t th3;
578         pthread_create(&th3, NULL, proc3, NULL);
579         pthread_t th4;
580         pthread_create(&th4, NULL, proc4, NULL);
581         pthread_t th5;
582         pthread_create(&th5, NULL, proc5, NULL);
583         pthread_t th6;
584         pthread_create(&th6, NULL, proc6, NULL);
585         pthread_t th7;
586         pthread_create(&th7, NULL, proc7, NULL);
587         pthread_t th8;
588         pthread_create(&th8, NULL, proc8, NULL);
589         chan_send_buf(c1, &v0, sizeof(List_int));
590         List_int v1;
591         chan_recv_buf(c17, &v1, sizeof(List_int));
592         pthread_join(th1, NULL);
593         pthread_join(th2, NULL);
594         pthread_join(th3, NULL);
595         pthread_join(th4, NULL);
596         pthread_join(th5, NULL);
597         pthread_join(th6, NULL);
598         pthread_join(th7, NULL);
599         pthread_join(th8, NULL);
600         chan_dispose(c1);
601         chan_dispose(c2);
602         chan_dispose(c3);
603         chan_dispose(c4);
604         chan_dispose(c5);
605         chan_dispose(c6);
606         chan_dispose(c7);
607         chan_dispose(c8);
608         chan_dispose(c9);
609         chan_dispose(c10);
610         chan_dispose(c11);
611         chan_dispose(c12);
612         chan_dispose(c13);
613         chan_dispose(c14);
614         chan_dispose(c15);
615         chan_dispose(c16);
616         chan_dispose(c17);
617         return v1;
618     }
619     int main()
620     {
621         int * tmp = randomList(1024);
622         List_int a = (List_int) {1024, tmp};
623         double start = get_time();
624         proc0(a);
625         double end = get_time();
626         printf("%lf\n", end   start);
627         return 0;
628     }
```