Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

TODO

Author:
Shuhao Zhang

Supervisor:
Prof. Nobuko Yoshida
David Castro-Perez

Second Marker:
Dr Iain Phillips

May 30, 2019

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 written in low level languages with verbose and non-idiomatic decorations, hard to debug because machines, where code is written, are usually different from machines where code is intended to run and hard to maintain and reuse because even though the underlying algorithms are not changed, multiple version of parallel code is needed to tackle various platform and evolution of architectures.

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

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 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, e.g. MPI [1], Cuda. Moreover, it can be interpreted with a simulator to aid debugging parallel programs.

With the help of this intermediate languages, the implementation complexity is reduced from $O(M \times N)$, where each of the M high-level languages needs to implement N compilers to generate parallel code in N different platforms, to $O(M + N)$, where each compiler of a high-level language implements a translation rule to the intermediate language which implements one compiler and N backend to generate different target languages.

In addition, it couples with multiparty session type (MPST) [5]. It takes advantages of properties of MPST to enable aggressive optimisation but ensuring code correctness and allow more meaningful static analysis; e.g. cost modelling for parallel programming.

1.2 Objectives

1. **Design**: Design the intermediate languages, argue its generality and build its connection with MPST.

2. **Tranlsation**: Define a translate rule from the language in a high-level parallel framework to our language.

3. **Simulator**: Build a simulator to prove the correctness of code generation and act as a playground for experiments.

[1] Message Passing Interface (MPI) is a standardised and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures [4].

4. **Code generation**: Generate parallel code in C.
5. **Static Analysis**: Session typing the language to obtain properties guaranteed by session types

More details of the objectives will be explained in the project planning at Section **??**

Chapter 2

Background

This section is an overview of techniques that influence the design choices of our monadic language for parallel computation. First of all, we give an overview of techniques applied in the high-level parallel programming framework: 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 parallelising. It also demos 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.

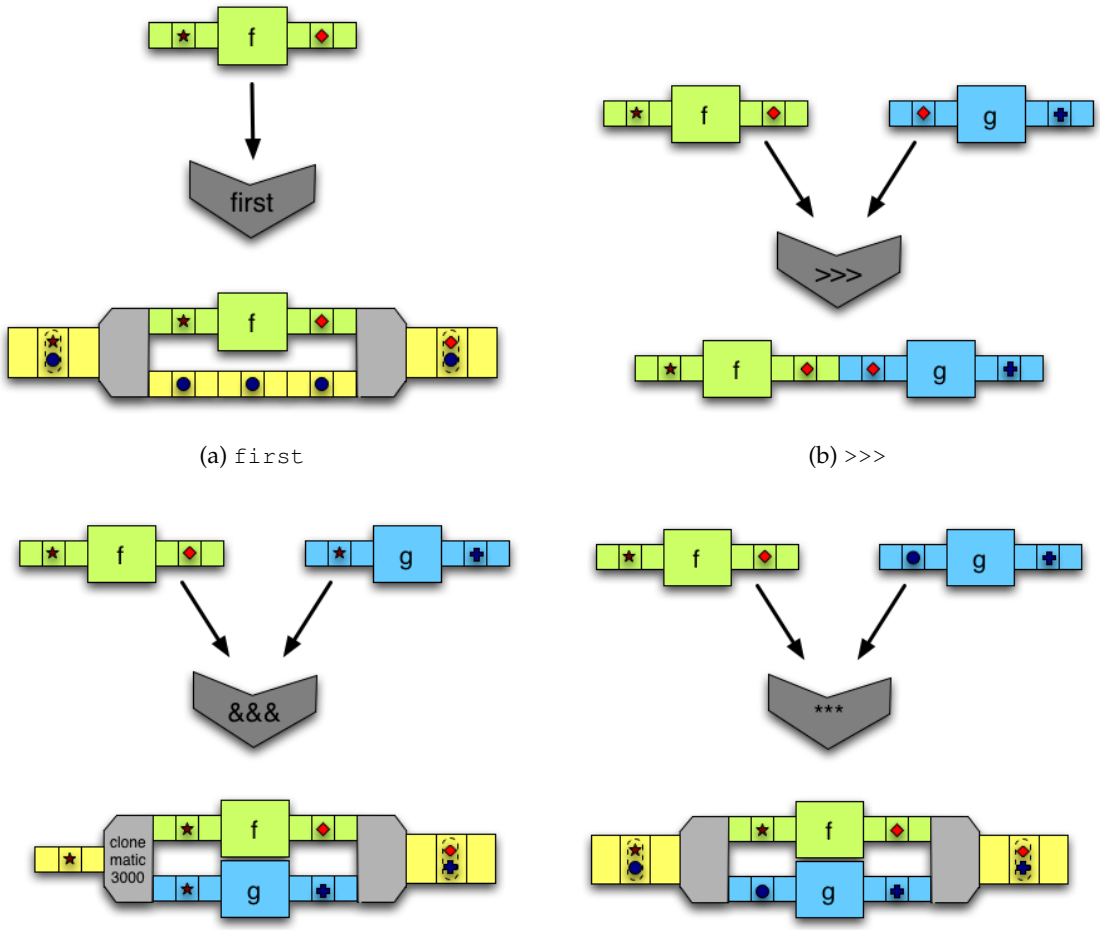
2.1.1 Definition

Listing 1 shows the Arrow definition in Haskell. Intuitively, an arrow type `y a b`(that is, the application of the parameterised 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 reset of function 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)`

```
1  class Arrow y where
2    arr :: (a -> b) -> y a b
3    first :: y a b -> y (a, c) (b, c)
4    second :: y a b -> y (c, a) (c, b)
5    (***) :: y a c -> y b d -> y (a, b) (c, d)
6    (&&&) :: y a b -> y a c -> y a (b, c)
```

Listing 1: Arrow class in Haskell



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

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

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

```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 a function to calculate the mean from a list of floating number, we will compare the usual, arrows implementations. 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].

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

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

The arrows implementation can be visualised in Figure 2.2.

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

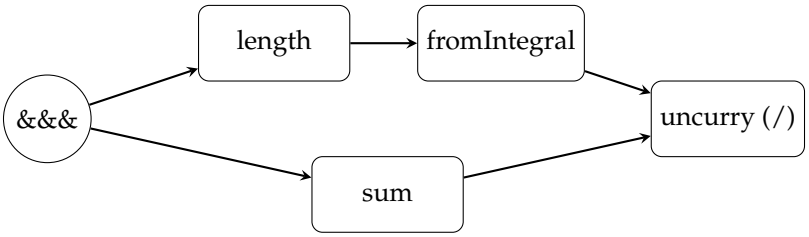


Figure 2.2: Visualization of mean'

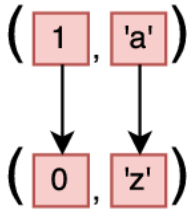
Arrows are not the only way to form point-free programs. The above code snippet is the more traditional approach form of point-free mean function in Haskell. We can argue this form of point-free function is more difficult to understand compared to arrows because it involves knowledge of monads (`liftM2`)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

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

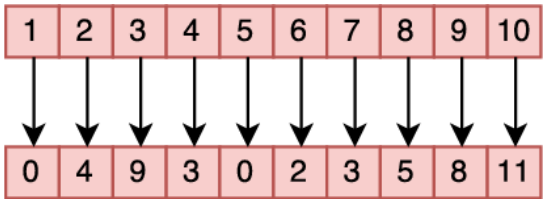
2.1.3 Application in parallel computation

From the previous example, the data flow of programs written regarding arrow combinators can be easily visualised (shown in Figure 2.1). It is intuitive to recognise 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 [9, 10, 11] has been done to generate parallel code from algebraic patterns. In particular, details of [11] are introduced in later section.

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` happened in parallel. Figure 2.3b shows that it can be extending to parallel map in terms of arrows, taking an arrow computation `arr ab` and returning a list of computation in parallel (`arr [a] [b]`).



(a) Visualization of parallel `***`[9]



(b) Visualization of parMap [9]

2.2 Recursion Schemes

Recursion schemes are patterns for expressing general computation. In particular, they are like high order function abstracting recursion so that programmer can express any kind of recursion by data structures combined with recursion schemes instead of writing explicit recursive functions.

2.2.1 Definition

We will introduce three typical recursion schemes: catamorphisms, anamorphisms and hylomorphisms (seen in Listing 4). As mentioned before, recursion schemes express recursion with the help of data structures, in particular, the fixed point of data structures (seen in Listing 3)

```
1 newtype Fix f = Fix { unfix :: f (Fix f) }
2
3 data TreeF a =
4     Node a a
```

```haskell
    | Leaf int
    | Empty
  deriving Functor

type Tree = Fix TreeF
```

Listing 3: Definition of fix point of data structures

Anamorphisms takes a function from a to f a (called the co-algebra) and a value a and return the Fix f. Used Tree as an example, anamorphisms takes a single value a and applies the co-algebra to the value. It continues to apply itself to the branches 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 top-down.

Catamorphisms is the reverse of anamorphisms, folding a data structure to a single value bottom-up. It takes a function from f a to a (called the algebra) and Fix f to fold and return a single value a. Catamorphisms and anamorphisms describe the process globally (from a to Fix f and from Fix f to a) 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 consumes recursive data structure while anamorphism builds them.

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 like co-algebra while merging is like algebra. We use hylomorphisms to combine them hence getting a sorted list (seen in Listing 5).

```haskell
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 type and direction of data exchanged. Multiparty session types are a branch of behavioural 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 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 type contains three main elements. Global types (seen in Section 2.3.1), local (session) types and processes. Processes are the concrete descriptions of the behaviour of the peers involved in the distributed system [5] using a formal language. Usually, the most used and the original language is π-calculus [12]. However, for the simplicity, we will not introduce π-calculus The coming sections are an intuitive introduction of session types by examples.

2.3.1 Global types and local types

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 the global types is shown in Table 2.1 and an example of global types is shown in Table 2.3.

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 local type. The syntax of local types is shown in Table 2.2 and an example of local type is shown in Table 2.4.

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

Table 2.1: Global types

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

$T ::=$	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

Table 2.2: Session types/local types

1. Customer(0) sends an order number to Agency(1), and 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, Agency notify both Customer and Agency2(2).
5. Customer sends an address to Agency2 and Agency2 sends back a delivery date.

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

Table 2.3: An example of a protocal described by global types G

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

Table 2.4: Session types of client and server end point of a ATM service

2.3.1.1 Projection between global types and local types

Projection is the formalisation 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 Table 2.5.

As an example, a projection of global type in Table 2.3 is

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

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

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

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

Table 2.5: 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 Table 2.6.

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

Table 2.6: Inductive definition of duality

Duality is essential for checking type compatibility. Compatible types mean that each common channel k is associated with complementary behaviour: 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[13] has shown how to generate MPI[1] 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`,

[1] Message Passing Interface (MPI) is a standardised and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures [4].

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)[2]. 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.

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

newtype C a = C { runC :: (a -> Action) -> Action } ⑥

instance Monad C where
    (>>=) :: C a -> (a -> C b) -> C b ⑨
    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 programmes 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).

```
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

An example of programme written in the concurrency monad is shown 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 programmes 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]`.

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

```haskell
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
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 the later section).

```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.

In summary, many techniques and ideas like continuation presented in the implementation of this monad afford us inspirations in designing 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 free monad to a given AST, we can follow a routine [18].

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 DSL operations.
5. Build interpreters for Free ADT by interpreting
6. Interpret the program by the interpreter.

We will demo 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.

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

Then we can easily implement an interpreter for the example program

```
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 examplePrograe
Tom
Good morning Tom
Mary
Good morning Mary
```

2.5.3 Applications

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

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

Chapter 3

Alg and ParAlg: An overview

Chapter 4

SPar: A session typed free monad EDSL for concurrency

4.1 Computation: The Core EDSL

4.1.1 Syntax

4.1.2 Representation of recursive data structures

4.1.3 Semantics

4.2 Communication: The Proc EDSL

4.2.1 Syntax

4.2.2 Representation in Haskell

4.2.3 Semantics

4.2.4 Session typing

4.3 Parallel computation: A group of Procs

4.3.1 Duality check

Chapter 5

SPar: Implementation

5.1 Session type

5.1.1 Representations of session types in Haskell

5.1.2 Type-indexed Free Monad

5.1.3 Type-level duality check

5.1.4 Value-level duality check

5.2 SPar interpreter

5.2.1 Overview

5.2.2 Implementation

Chapter 6

ArrowPipe: An arrow interface for writing SPar expressions

When trying to express more complicated and interesting parallel patterns, e.g map or reduce, We realize SPar is too low-level so that it is difficult to express simple computation because of overheads of expressing communication patterns by hand. In addition, compilation from Par-Alg to SPar is hard since they are very different domain specific languages.

To solve both issues, we draw inspirations from the Arrow interface (in particular, work done by [9] where they use arrow interface to express parallel computation) and introduce ArrowPipe.

ArrowPipe is an arrow interface for writing SPar expressions. Withe the help from ArrowPipe, Users can use canonical arrow combinators to write algorithms in Arrow without writing any explicit communication and gain parallelized algorithms for free. Similarly, ArrowPipe makes hassle-free compilation from Par-Alg to SPar possible because Par-Alg Proto is also an arrow expression and simply interpreting arrow combinators by the ArrowPipe implementations fills the gap between Par-Alg and SPar.

6.1 Syntax

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

type ArrowPipe a b = Nat -> Pipe a b

instance Arrow ArrowPipe where
    -- implementation omit
instance ArrowChoice ArrowPipe where
    -- implementation omit
```

Listing 12: Definition of ArrowPipe

The simplified syntax of ArrowPipe can be found in Listing 12. ArrowPipe 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 reson why we use `Nat` as the only parameter is to ensuring no duplication of processes name since in most of the time, duplication is bad for parallelization. It will be explained more thoroughly in Section 6.3.

`Pipe a b` data structures is the essential component of ArrowPipe. 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` indicates the process that produces the output data in the end. We can retrieve the corresponding process by a look up on the env with the key `end`. The returned Proc returns a data with type `b`.

6.1.1 Arrow interface

`ArrowPipe` is an instance of 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.

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

6.1.2 Example: Parallel programming patterns

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

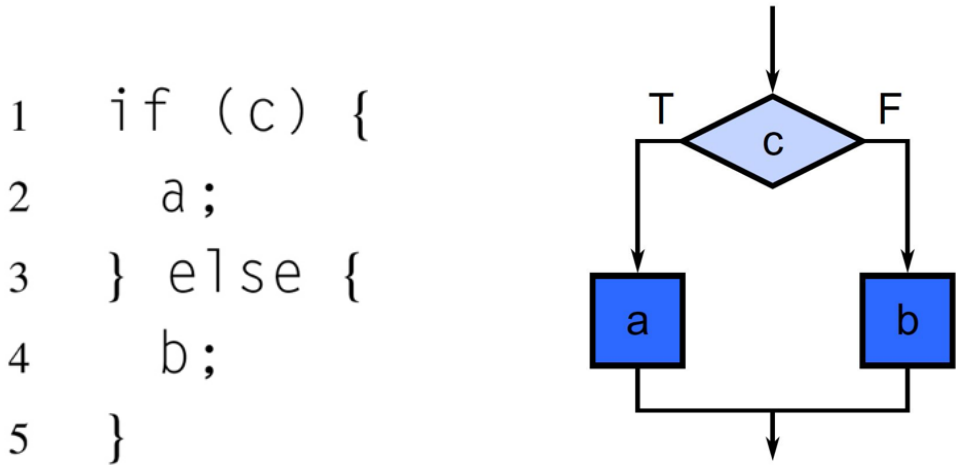
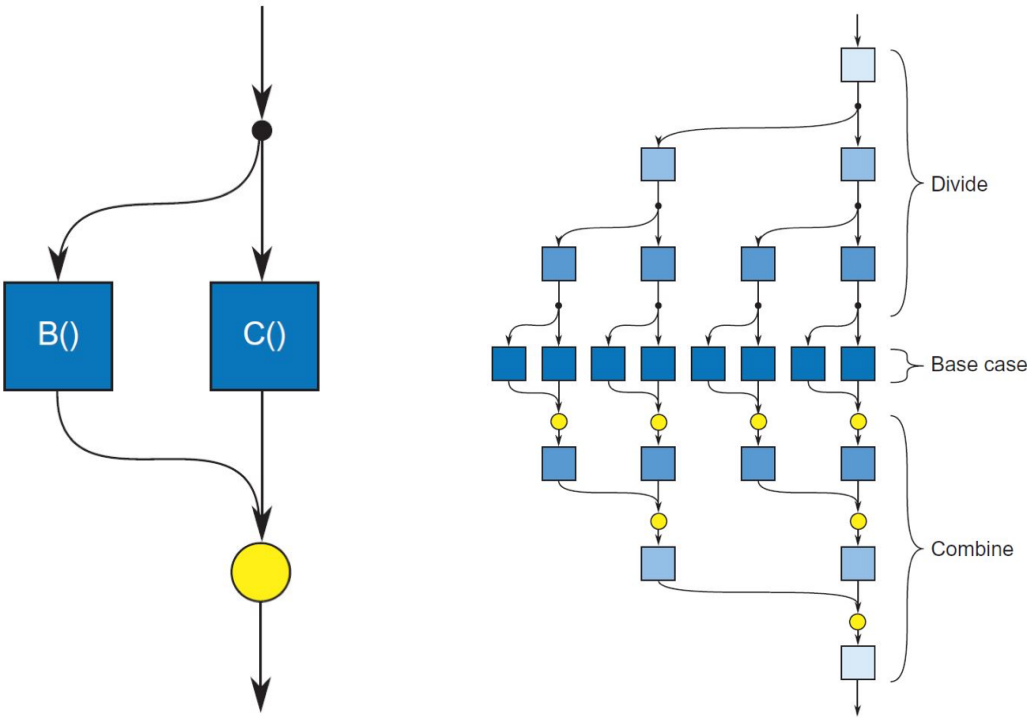


Figure 6.1: Visualization of the select pattern [19]

First of all, the select pattern illustrated by Figure 6.1 is equivalent to an expression formed by `|||` combinators, where the data constructor `Left` means True and the data constructor `Right` means False for the sum type `Either`. Secondly, the fundamental building block of parallel pattern, the fork-join pattern illustrated by Figure 6.2a can be expressed by `&&&` combinator. The arrowPipe produced by `&&&` has the two-ary tuple as the output type collecting the computation result of the main thread and the forked thread and also acts as a synchronization point.

Thirdly, the familiar parallel map pattern illustrated in Figure 6.3 is also a candidate to be expressed in ArrowPipe. The code sample is in Listing 13. `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 4 equal chunks. The number of function `f` applied decides the number of ways of parallelism. We will describe a possible solution to make `pmap` more generic so that we do not need write a new functions for every number of ways of parallelism. (TODO).



(a) Visualization of the fork-join pattern [19]

(b) Fork-Join Pattern for Divide-Conquer [19]

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

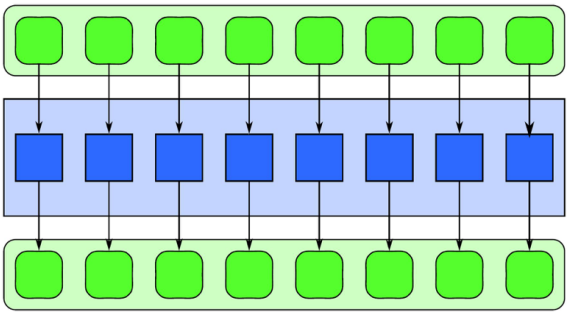


Figure 6.3: Visualization of parallel map [19]

Fourthly, we can apply the similar logic to express parallel reduce pattern shown in Figure 6.4. The code sample is in Listing 14. The result of parallel reduce has similar type 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, more complicated pattern can be expressed compositely from simpler pattern expressed in ArrowPipe. We can use a typical divide-and-conquer algorithms implemented with fork-join as an example. Figure 6.2b shows a divide-and-conquer algorithms with 2-ways and 3-levels of fork-join. The algorithm can be expressed in arrowPipe shown in Listing 15. 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 expressions in the function definition are connected using arrow combinators. A 3-level divided-and-conquer algorithm is constructed by passing 3 to the function resulting a algorithm with $2^3 = 8$-way parallelism.

The implementation demos the power of implementing ArrowPipe as a domain specific language embedded in Haskell. We make full use of Haskell features, i.e high order functions and

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

Listing 13: Parallel map in ArrowPipe

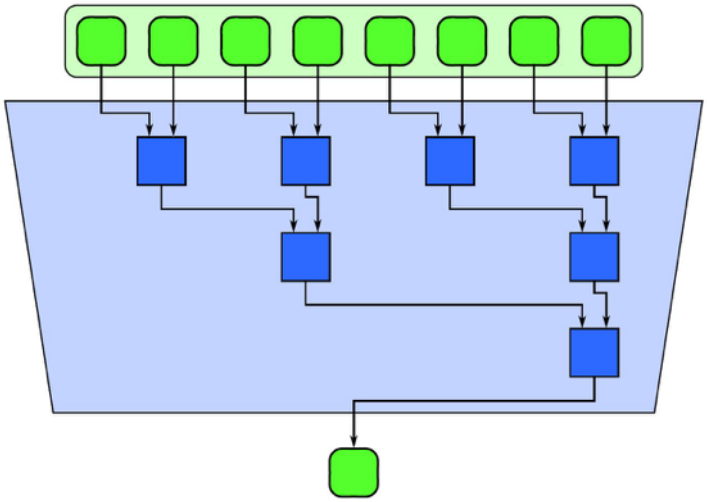


Figure 6.4: Visualization of parallel reduce in ArrowPipe [19]

polymorphic functions to construct expressive, composable and generic computation patterns.

More examples of algorithms formed by ArrowPipe, 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.

The intuition why ArrowPipe is an instance of Arrow comes from the Kleisli arrow of a monad is an instance of Arrow class (shown in Listing 16). The `cont` field in the Pipe has similar type signatures as the `runKleisli` field in the Kleisli arrow. From the previous section, we 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 process stored in `env` instead of finishing at `cont` like Kleisli arrow. Intuitively, ArrowPipe, a function from the role to the Pipe, should be an instance of Arrow since Pipe looks like an arrow instance and functions are composite.

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. Hence the only option is to convert it to a Proc expression and store the converted expression in the updated env in the resulting Pipe. A right way to do it bind the cont with the action: receive from end in the first Pipe. Also, extend the proc related to the end in the head Porc with action: send to start in the second Pipe. Besides addition of the converted `cont` expression, the new env is formed by merging the env from the head Pipe and the env from the tail Pipe. Merging two envs is trivial. When there are duplication, we simply use monadic bind to combine them so that the actions belonging to head Pipe followed by the actions belonging to the tail Pipe. 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 from the tail Pipe. We can apply the Pipe composing function to implement arrow combinators for ArrowPipe. The implementation is just apply the first ArrowPipe with the input role (don't forget ArrowPipe is a function) to get the Pipe and apply the second ArrowPipe with a new role to get the second Pipe (usually in order to avoid duplication of roles, the new role is

```
preduc :: ArrowPipe (a, a) a -> ArrowPipe (a, (a, (a, a))) a
preduc r = helper >>> (r *** r) >>> r
 where
  helper = (arr Id *** arr Fst) &&& (arr Snd >>> arr Snd)
```

Listing 14: Parallel reduce in ArrowPipe

```
divConquer
  :: Int
  -> ArrowPipe a b
  -> ArrowPipe a (a, a)
  -> ArrowPipe (b, b) b
  -> ArrowPipe 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 15: 2-ways and 3-levels divided-and-conquer algorithm in arrowPipe

set to be 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 17. The rest of combinators can be implemented in a similar fashion.

6.3 Strategies for optimized role allocations

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 allocating is an essential part in generating efficient parallel programs.

In this section, we propose strategies for optimizing role allocations. We have two goals in mind when optimizing; The first one is 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. The second one is we do not want roles duplication when we try to compose ArrowPipes since role duplications means the different computation must be merged in the same role and computations in the same thread is sequential hence role duplications has negative impact on degree of parallelization.

If we only put the first goal in mind, an easy solution will be setup an upper bound of the number roles, and then we cycle through a fixed bound when allocating new roles. Processes corresponding to duplicated roles can be simply merged using binds since Proc is a monadic DSL and duality check ensures binding will not cause deadlocks. However, this strategies is not ideal since duplications of roles will decrease the degree of parallelization in the system.

If we only consider the second goal, naive strategies used in the previous section will satisfy the goal. However, the number of channels required and the number of roles in the system will grow exponentially. In a divided-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.

For the purpose of illustration, we use inference rules to explain our proposed strategies for optimized role allocations when composing ArrowPipes. Please see Listing 18 as an example. $x \Rightarrow x$ means the computation start with role x and end with role x.

The rule for the rest of combinators are shown in Listing 19. 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. Reader may find it strange that we do not intent to parallelize arr combi-

```
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 16: The implementation of arrow instance for Kleisli arrow of a monad

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

Listing 17: The simplified implementation of >>>

nator which lifts a sequential computation represented by Core (a $\rightarrow$ b) into ArrowPipe. It makes sense to introduce a new role to execute the computation and hence parallelize computational heavy tasks. We use this strategy in the first place but later we found a more suitable strategy exists which will be introduced in the later paragraph. Also, another reason not to introduce 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 complicated user defined functions. Introducing new roles for these simple functions will damage performance.

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 starts 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 process in both left and right ArrowPipe expression 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 Either type as the output.

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

In conclusions, even though from the implementation point of view, arrowPipe composition with the optimized role allocations is ad-hoc and less elegant to implement because we need to consider composition by send-and-receive and composition by local monadic bind and more edge cases to be dealt with compared to the naive solution in the last section where composition is done solely by send-and-receive and role allocations is mindless. We believe the effort is worthy because for a n-level divided-and-conquer algorithms, the optimized role allocation strategies allocate 2^n roles in total which is the same as the number of way of parallelization in theory. All the roles are used to maximize parallelism instead of wasting the valuable resources to create roles that merely transmit data.

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

Listing 18: Role allocation for id

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

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

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

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

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

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

Listing 19: Rules fo role allocations of different combinators

6.4 Satisfaction of arrow laws

6.5 Conclusions

Arrow interface is the perfect interface to express general computation for this project because not only is it intuitive to understand and visualize but also its combinators $***$ and $\&\&\&$ has built-in parallel natural.

So far, we've used ArrowPipe to help us compile Par-Alg to SPar. The remained challenge is the code generation from SPar to a target platform. In the next chapter, we will introduce our methods for code generation and specifically code generation to C. Once we achieved this, every computation in ArrowPipe can be transformed into parallel C code in one step.

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 computation, describing the communication patterns. Naturally, the process of code generation from SPar should be divided into two parts correspondingly. We choose to make two parts independent of each other so that it's 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 DSL and run a series of transformation to low-level DSL. SPar expressions are converted to a low-level EDSL which is then transformed to an abstract syntax tree (AST) of C (TODO cite the package). 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, i.e variable declaration, variable assignment, channel initialization, channel communication and channel deletion. Hence, we need to define a EDSL containing instructions related to these low-level operations. We name it Instr. A Spar programs will be translated to a sequence of Instr.

When we design Instr, we keep the simplicity in mind so Instr is not coupled with any fancy language feature related to some specific target languages. Any reasonable target language with a channel communication library can be easily converted from Instr.

7.1.1 Syntax and semantic

The definition of Instr is seen in Listing 20. `Channel` is our abstract representation of Channel in Instr. It is indexed by a type a from the reified type `ReprType a`. More details of this reified type will be introduced in Section 7.1.2. This type parameter preserves type in channel initialization hence make sure the value to be sent or received in this channel has the same type as this channel. This is necessary because for some target languages, the channel are typed. Similarly, type parameters in `Exp` have the same functionality. `Exp` is just a wrapper of the expression in Core. In later stages, we will take care of code generation of `Exp` along with `Instr`. `Instr` defines the set of statements that will be generated and `Exp` represents the sequential computation, which is a value that will be generated.

The semantic of Instr operations are similar to what their 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 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 languages. SPar action like broadcast are built on top of these operations. For `CBranch`, the first field represents the value of 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 Either value and the second field field

```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 20: The syntax of Instr in Haskell with accompanying low-level data types

```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 21: The definition of representation types

represents the variable whose value is assigned by the end results of instructions from either the left branch or the right branch. The third and fourth fields represent the instructions in the left branch and the right branch.

7.1.2 Representation types

SPar programs cannot be fully parametric since the target languages of 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 when processing with some specific form of data, i.e, languages targeting GPUs are usually more productive in dealing with array of floating point number while slow in working with aggregate structures [20]. Hence, we need to restrict the set of types allowed in SPar. We achieve this using the type class `Repr` and corresponding reified type `ReprType` (shown in Listing 21). `Repr` determines the set of type allow in SPar. Reified type `ReprType` will be used

```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))
  ]
-- omit other cases
```

Listing 22: An example usage of reified type in the code generation

to alter the behaviors of code generation based on the type. This can be simply done by pattern matching because reified types are values in Haskell [21]. To be more concrete, Listing 22 gives a demo. `constToCExpr` is function that handle code generation from constant value to expression in C programming languages. By pattern matching, we vary the behaviors of code generation so that constants with different types has their own way to be represented in C.

In conclusion, we allow the following type: 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 expression. 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 expression, applying the mapping and collecting the results by concatenation. This is an advantage of using free monad technique to build the AST because Proc expression can be treated as a data structures and traversing recursive data structures can be easily done in Haskell. In addition, operations like `Recv` which involves continuation 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 recursively. For the former, we have to pass an expression whose type is `Core a` to the continuation so that we can recursively call the traverse function recursively on the results of applying value to the continuation. The answer to what Core expression should be used is `Var` denoting variable. Passing a unique variable to the continuation gives us `next` inexpensively and we will define where does values of variables comes from for each operations in Proc.

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

Pure: It is the base case in 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 make sure the same channel is used in a send-and-receive pair will be discussed in the next sub chapter.

Recv: It is the reverse of send operation. Firstly, it declares a new variable with `CDecla` and use `CRecv` to assign the value received from the sender to this variable. Notice that the 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 operations. Its constructor contains two continuation: one for left branch and one for right branch. Hence for this instruction, we need to declared two variable to passed into continuations. The value of both variables is assigned by the Core expression whose value is a Either type. Besides, we need to send label indicating whether the execution of the left branch or the right branch of the receiver is selected. Its value is assigned by the either value as well and the sending operation 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 recursively on two branches and finally use `CBranch` to collect results.

7.2.2 Strategies for channel allocations

Channel allocations is important because correction allocation is essential in making sure the correctness and deadlock-free of generated code. Besides correctness concerns, we are also want to reduce the amount of channels creations hence increase performances.

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

In the second iteration, we come up with a more efficient and simple strategy. The buffer size of all channels is still one due to the same reason about linearity. However, we decide to only allocate one channel for a pair of sender and receiver. We will not destroy the channel after the value has received and 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

```
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 23: States required during the traversal

From the last two subsections, we need to maintain a number of states during the compilation process. Hence we define a state monad to be used during the traversal. The `CodeGenState` is the collection of states with different purposes and it is shown in Listing 23. `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 the similar functionality ensuring the uniqueness of channel names. `dataStructCollect` is the set of compound type we encounter during the traversal. With this states, we know what kind of data structures whose definitions will be generated before the generation of executing code.

7.3 Code generation to C: from Instr to C

The last piece of jigsaw is compilation from a sequence of Instr to C. This process is rather simple but trivial. This is simply done by transforming the sequence of Instr to a C AST. We used an open source library: language-c [22] 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 [23]

whose internal is based on shared memory. In the following subsections, we will present some design choices during this final step.

7.3.1 Representations of Core data type in 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 24: Compound data type in C

The first challenge we face is how to represent data structure in C. For primitive data type like Int or Float, a simple one-to-one mapping is sufficed. It is hard to deal with compound data type in C. First of all, C does not support polymorphic type. Hence, we choose to generate specific data type for every different compound data type even though they have the same structure. 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 type and product type. The product type will be converted to a struct with two field in C. The sum type is represented by the tagged union type. Tagged union is a struct with two field. The value of the first field indicates whether it is a left value or right value and the value of second field is a union type containing either the left value or right value. We also implement a sorting algorithm based on the depth of compound type so that all necessary data types have been defined before the definition of the compound types. An example of what `Either Int (Int, Int)` will be converted to is shown in Listing 24.

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

```
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 list of int in this way is bad for performances. Obviously, C has a more efficient way to represent list of int using an array. So we decided to has two ways to represent recursive data structures. For a set of specific recursive data structure, users can write their own representation to exploit the advantages of the target language. For example, a list of int are encoded in C using a wrapper of pointer type (shown in Listing 25). This way is not very generic but friendly for performances. For other types of recursive data structures where user do 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 25: Optimized represent of List in C

7.3.2 Compiling from Core to C

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

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

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

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

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

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

Listing 26: 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 generate C expressions from Core expressions. Not surprisingly, the compilation is a traversal of the Core expression. Pattern matching on the structure of the core expression alters the behaviors of compilation. `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 Listing 26. $[\![a]\!]$ means the C code generated by Core expression `a`. toC is function that convert constant value in Haskell to constant value in C.

Var, Lit:

Fst, Inl, Pair... For Inl and Inr and Pair, we used C99 style to initialize struct and union. The rule for generating corresponding struct are explained in the previous subsection. For Fst and Snd, we simply access the specific value using the designator.

Id, Const:

Prim:

7.3.3 The structure of generated C code

7.4 Conclusion

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

Chapter 8

Parallel algorithms and evaluation

8.1 Parallelized algorithms

8.1.1 Merge sort

8.1.2 Quick sort

8.2 Benchmarks

8.2.1 Benchmarks against generated Haskell code

8.2.2 Benchmarks against C implementation

8.3 Evaluation

8.3.1 Design choices: Why Haskell?

Chapter 9

Conclusions and future work

9.1 Future work

Bibliography

[1] M. I. Cole, "Algorithmic Skeletons: Structured Management of Parallel Computation," p. 137.

[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] "Message Passing Interface," *Wikipedia*, Oct. 2018. Page Version ID: 863149040.

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

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

[11] "Algebraic Multiparty Protocol Programming," p. 37.

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

[13] N. Ng, J. G. F. Coutinho, and N. Yoshida, "Safe MPI Code Generation based on Session Types," p. 22.

[14] D. Orchard and N. Yoshida, "Session Types with Linearity in Haskell," p. 24.

[15] K. Claessen, *Functional Pearls: A Poor Man's Concurrency Monad*. 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. P. Jones, "A Monad for Deterministic Parallelism," p. 12.

[18] C. contributors, "Cats: FreeMonads." http://typelevel.org/cats/.

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

[20] T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton, "Type-safe Runtime Code Generation: Accelerate to LLVM," p. 12.

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

[22] "Language-c: Analysis and generation of C code." //hackage.haskell.org/package/language-c.

[23] T. Treat, "Pure C implementation of Go channels. Contribute to tylertreat/chan development by creating an account on GitHub," May 2019.