



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 23, 2019

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Objectives	4
2	Background	6
2.1	Arrows	6
2.1.1	Definition	6
2.1.2	Example: Calculate the mean	7
2.1.3	Application in parallel computation	8
2.2	Recursion Schemes	8
2.2.1	Definition	8
2.2.2	Example: Merge sort	9
2.3	Multiparty session types	10
2.3.1	Global types and local types	10
2.3.2	Applications in parallel computing	12
2.4	Message passing concurrency	12
2.4.1	Primitives for message-passing concurrency	12
2.4.2	Concurrency Monads	12
2.5	Free monad	14
2.5.1	Definition	14
2.5.2	Example	15
2.5.3	Applications	16
3	Alg and ParAlg: An overview	17
3.1	Syntax	17
3.2	Compilation from Alg to ParAlg	17
3.3	Multiparty session types for ParAlg	17
3.4	Global types and protocols	17
3.5	Example: Parallel merge sort	17
4	SPar: A session typed free monad EDSL for concurrency	18
4.1	Computation: The Core EDSL	18
4.1.1	Syntax	18
4.1.2	Representation of recursive data structures	18
4.1.3	Semantics	18
4.2	Communication: The Proc EDSL	18
4.2.1	Syntax	18
4.2.2	Representation in Haskell	18
4.2.3	Semantics	18
4.2.4	Session typing	18
4.3	Parallel computation: A group of Procs	18
4.3.1	Duality check	18

5	SPar: Implementation	19
5.1	Session type	19
5.1.1	Representations of session types in Haskell	19
5.1.2	Type-indexed Free Monad	19
5.1.3	Type-level duality check	19
5.1.4	Value-level duality check	19
5.2	SPar interpreter	19
5.2.1	Overview	19
5.2.2	Implementation	19
6	ArrowPipe: An arrow interface for writing SPar expressions	20
6.1	Syntax	20
6.1.1	Arrow interface	21
6.1.2	Example: Parallel programming patterns	21
6.2	Implementation of arrow combinators	22
6.3	Strategies for optimized role allocations	23
6.4	Satisfaction of arrow laws	25
6.5	Conclusions	25
7	Type-safe code generation from SPar	26
7.1	Instr: A low-level EDSL of channel communication	26
7.1.1	Syntax	26
7.1.2	Reified type	26
7.2	Compilation from SPar to Instr	26
7.2.1	Strategies for channel allocation	26
7.2.2	Type preservation	26
7.3	Code generation to C: from Instr to C	26
7.3.1	Representations of Core data type in C	26
8	Parallel algorithms and evaluation	27
8.1	Parallelized algorithms	27
8.1.1	Merge sort	27
8.1.2	Quick sort	27
8.2	Benchmarks	27
8.2.1	Benchmarks against generated Haskell code	27
8.2.2	Benchmarks against C implementation	27
8.3	Evaluation	27
8.3.1	Design choices: Why Haskell?	27
9	Conclusions and future work	28
9.1	Future work	28

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 compile 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. MPI¹, 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. **Translation:** 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.

¹Message 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



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

```

1 instance Arrow (->) where
2   arr f = f
3   (***) 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].

```

1 mean :: [Float] -> Float
2 mean xs = sum xs / (fromIntegral . length) xs
3
4 mean' :: [Float] -> Float
5 mean' = (sum &&& (length >>> fromIntegral)) >>> uncurry (/)

```

The arrows implementation can be visualised in Figure 2.2.

```

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

```

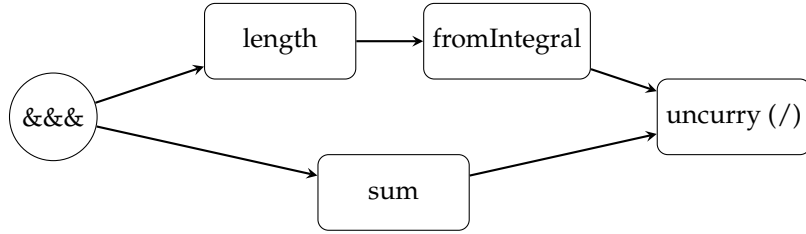


Figure 2.2: Visualization of mean'

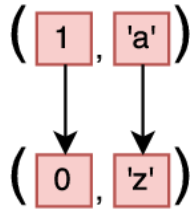
Arrows are not the only way to form point-free programs. The above code snippet is the more traditional approach form of point-free mean function in Haskell. We can argue this form of point-free function is more difficult to understand compared to arrows because it involves knowledge of monads (`liftM2`)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

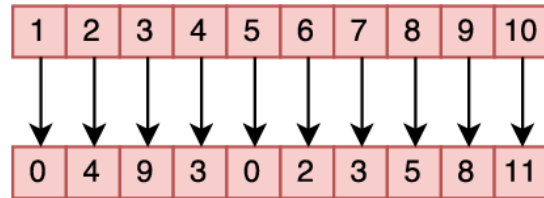
2.1.3 Application in parallel computation

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
  
```



```

5 | Leaf int
6 | Empty
7 deriving Functor
8
9 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 $\text{Fix}\ f$. 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 $\text{Fix}\ f$ to fold and return a single value a . Catamorphisms and anamorphisms describe the process globally (from a to $\text{Fix}\ f$ and from $\text{Fix}\ f$ 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.

```

1 ana :: Functor f => (a -> f a) -> a -> Fix f
2 ana coalg = Fix . fmap (ana coalg) . coalg
3
4 cata :: Functor f => (f a -> a) -> Fix f -> a
5 cata alg = alg . fmap (cata alg) . unfix
6
7 hylo :: (f b -> b) -> (a -> f a) -> b -> a
8 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).

```

1 mergeSort :: [Int] -> [Int]
2 mergeSort = hylo merge split where
3   merge Empty      = []
4   merge (Leaf c)   = [c]
5   merge (Node l r) = usualMerge l r
6
7   split [] = Empty
8   split [x] = Leaf x
9   split xs = Node l r where
10      (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
$p \rightarrow q : \langle S \rangle . G$	Value exchange
$p \rightarrow q : \langle T \rangle . G$	Channel exchange
$p \rightarrow q : \{l_i : G_i\}_{i \in I}$	Branching
$\mu t . G \mid t \mid \text{end}$	Recursion/End

Table 2.1: Global types

$S ::=$	Sorts	$T ::=$	Session types/local types
$\text{bool} \mid \text{nat} \mid \text{string}$		$! \langle p, S \rangle . T$	Send value
\dots		$! \langle p, T \rangle . T$	Send channel
		$? \langle p, T \rangle . T$	Channel Receive
		$? \langle p, S \rangle . T$	Sorts Receive
		$\oplus \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Selection
		$\& \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Branching
		$\mu t . T \mid t \mid \text{end}$	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.

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

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

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

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

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

$$\begin{aligned}
(\mathbf{p} \rightarrow \mathbf{p}' : \langle U \rangle. G') \upharpoonright \mathbf{q} &= \begin{cases} !\langle \mathbf{p}', U \rangle. (G' \upharpoonright \mathbf{q}) & \text{if } \mathbf{q} = \mathbf{p}, \\ ?\langle \mathbf{p}, U \rangle. (G' \upharpoonright \mathbf{q}) & \text{if } \mathbf{q} = \mathbf{p}', \\ G' \upharpoonright \mathbf{q} & \text{otherwise.} \end{cases} \\
(\mathbf{p} \rightarrow \mathbf{p}' : \{ l_i : G_i \}_{i \in I}) \upharpoonright \mathbf{q} &= \begin{cases} \oplus \langle \mathbf{p}', \{ l_i : T_i \}_{i \in I} \rangle & \text{if } \mathbf{q} = \mathbf{p} \\ \&(\mathbf{p}, \{ l_i : G_i \upharpoonright \mathbf{q} \}_{i \in I}) & \text{if } \mathbf{q} = \mathbf{p}' \\ G_{i_0} \upharpoonright \mathbf{q} & \text{where } i_0 \in I \text{ if } \mathbf{q} \neq \mathbf{p}, \mathbf{q} \neq \mathbf{p}' \\ & \text{and } G_i \upharpoonright \mathbf{q} = G_j \upharpoonright \mathbf{q} \text{ for all } i, j \in I. \end{cases} \\
(\mu t. G) \upharpoonright \mathbf{q} &= \begin{cases} \mu t. (G \upharpoonright \mathbf{q}) & \text{if } G \upharpoonright \mathbf{q} \neq t, \\ \text{end} & \text{otherwise.} \end{cases} \quad t \upharpoonright \mathbf{q} = t \quad \text{end} \upharpoonright \mathbf{q} = \text{end}.
\end{aligned}$$

Table 2.5: The definition of projection of a global type G onto a participants \mathbf{q} [5]

2.3.1.2 Duality of session types

In binary session types where all protocols 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.

$$\begin{array}{lcl}
\overline{?[S]; T} & = & ![\tilde{S}]; \overline{T} \\
\overline{![S]; T} & = & ?[\tilde{S}]; \overline{T} \\
\overline{\text{end}} & = & \text{end}
\end{array}
\quad
\begin{array}{lcl}
\overline{\oplus\{l_i : T_i\}_{i \in I}} & = & \&\{l_i : \overline{T_i}\}_{i \in I} \\
\overline{\&\{l_i : T_i\}_{i \in I}} & = & \oplus\{l_i : \overline{T_i}\}_{i \in I} \\
\overline{\mu t. T} & = & \mu t. \overline{T}
\end{array}
\quad
\begin{array}{lcl}
\overline{?[T]; T'} & = & ![\overline{T}]; \overline{T'} \\
\overline{![T]; T'} & = & ?[\overline{T}]; \overline{T'} \\
\overline{\bar{t}} & = & \bar{t}
\end{array}$$

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 MPI¹ 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.

```

1 data Chan a
2 newChan :: IO (Chan a)
3 writeChan :: Chan a -> a -> IO ()
4 readChan :: Chan a -> IO a
5 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`,

¹Message 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`.

```

1 data Action =
2     Atom (IO Action)
3   | Fork Action Action
4   | Stop
5
6 newtype C a = C { runC :: (a -> Action) -> Action } ⑥
7
8 instance Monad C where
9     (>=) :: C a -> (a -> C b) -> C b ⑨
10    m >= f = C $ \k -> runC m (\v -> runC (f v) k)
11    return :: a -> C a
12    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).

```

1 atom :: IO a -> C a
2 atom m = C $ \k -> Atom $ do
3     r <- m
4     return $ k r
5
6 fork :: C () -> C ()
7 fork m = C $ \k -> Fork (runC m (const Stop)) (k ())
8
9 action :: C a -> Action
10 action m = m (\_.
```

Listing 8: Helper functions

An example of programme written in the concurrency monad is shown below.

```

1 example :: C ()
2 example = do
3     atom $ putStrLn "Hello"
4     name <- atom getLine
5     fork $ atom $ putStrLn "World"
6     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]`.

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

```

1 schedule :: [Action] -> IO ()
2 schedule [] = return ()
3 schedule (a:as) = sched as a
4
5 sched :: [Action] -> Action -> IO ()
6 sched as (Atom ioa) = do
7     a <- ioa ⑦
8     schedule $ as ++ [a] ⑧
9 sched as (Fork a1 a2) = schedule $ as ++ [a2, a1] ⑨
10 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).

```

1 newtype IVar a = IVar (IORef (IVarContents a)) ①
2 data IVarContents a = Full a | Blocked [a -> Action.]
3
4 data Action . =
5     Fork Action Action
6     | Stop
7     | forall a . Get (IVar a) (a -> Action) ⑦
8     | forall a . Put (IVar a) a Action ⑧
9     | 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 IVarContents 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`.

```

1 data Free f a
2   = Pure a
3   | Free f (Free f a)
4
5 instance Functor f => Monad (Free f) where
6   return = pure
7   (Pure x) >>= fab = fab x
8   (Free fx) >>= fab = Free $ fmap (>>= fab) fx

```

Listing 10: Free monad in Haskell

```

1 liftF :: Functor f => f a -> Free f a
2 liftF = Free . fmap Pure
3
4 freeM :: (Functor f, Functor g) => (f a -> g a) -> (Free f a) -> (Free g a)
5 freeM phi (Pure x) = Pure x
6 freeM phi (Free fa) = Free $ phi (fmap (freeM phi) fa)
7
8 monad :: Monad m => Free m a -> m a
9 monad (Pure x) = pure x
10 monad (Free mfx) = mfx >>= monad
11
12 interpret :: (Functor f, Monad m) => (f a -> m a) -> (Free f a -> m a)
13 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`.

```

1 data GreetingF next
2   = Getname (String -> next)
3   | Greet String next
4   deriving Functor
5
6 type Greeting = Free GreetingF

```

Then we create helper functions of `Greeting` using `liftF`.

```

1 getName = liftF $ Getname id
2 greet str = liftF $ Greet str ()

```

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

```
1 exampleProgram :: Greeting ()
2 exampleProgram = do
3     a <- getName
4     greet a
5     b <- getName
6     greet b
```

Then we can easily implement an interpreter for the example program

```
1 goodMorningInterpreter :: Greeting a -> IO a
2 goodMorningInterpreter = interpret helper
3     where
4         helper (Getname next) = fmap next getLine
5         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

3.1 Syntax

3.2 Compilation from Alg to ParAlg

3.3 Multiparty session types for ParAlg

3.4 Global types and protocols

3.5 Example: Parallel merge sort

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 reason why we use `Nat` as the only parameter is to ensure 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.

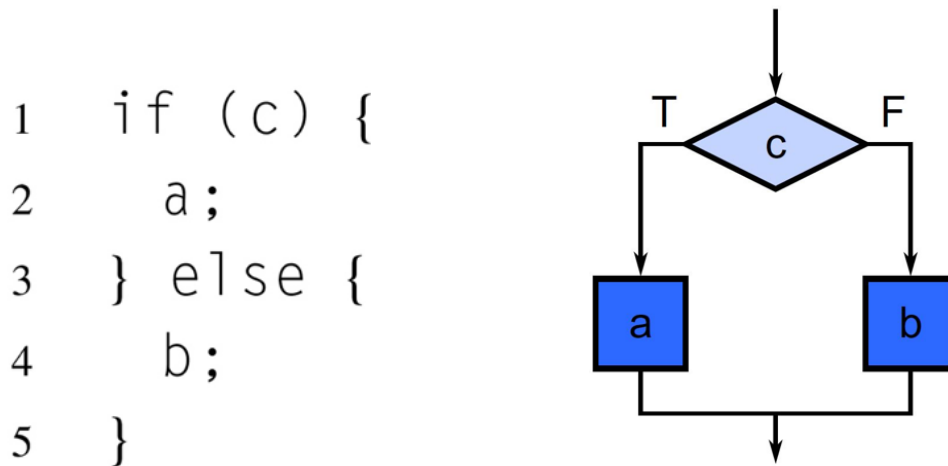
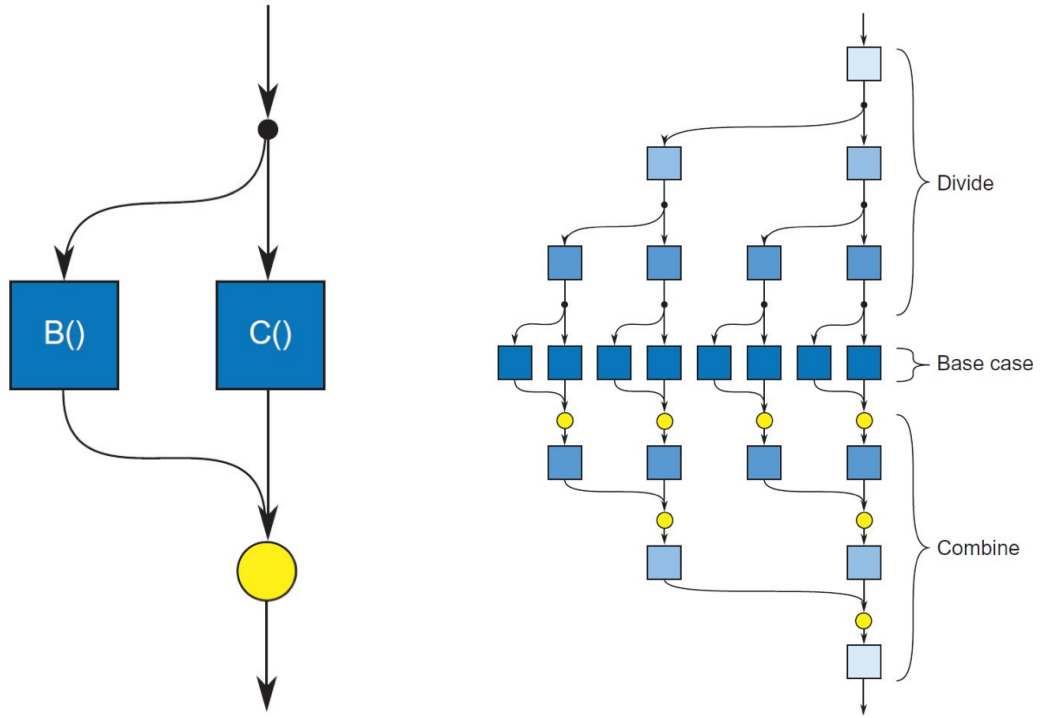


Figure 6.1: Visualization of the select pattern [19]

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.

In the end, 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 13. 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



(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

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 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 optimized solution is introduced in next section.

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 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 14. The rest of combinators can

```

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 13: 2-ways and 3-levels divided-and-conquer algorithm in arrowPipe

```

(>>>) :: (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 14: The simplified implementation of >>>

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.

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

Listing 15: Role allocation for id

For the purpose of illustration, we use inference rules to explain our proposed strategies for

optimized role allocations when composing ArrowPipes. Please see Listing 15 as an example. $x \Rightarrow x$ means the computation start with role x and end with role x .

$$\begin{array}{c}
\text{compose} \frac{e1 : \text{ArrowPipe } a \ b, x \Rightarrow y, \quad e2 : \text{ArrowPipe } b \ c, y \Rightarrow z}{e1 >>> e2 : \text{ArrowPipe } a \ c, x \Rightarrow z} \\
\\
\text{arr} \frac{f : \text{Core } (a \rightarrow b), \quad x : \text{Role}}{\text{arr } f : \text{ArrowPipe } a \ b, x \Rightarrow x} \\
\\
\text{arrow choice: } ||| \frac{e1 : \text{ArrowPipe } a \ c, x \Rightarrow y, \quad e2 : \text{ArrowPipe } b \ c, x \Rightarrow z}{e1 ||| e2 : \text{ArrowPipe } (\text{Either } a \ b) \ c, x \Rightarrow \max(y, z)} \\
\\
\text{arrow choice: } +++ \frac{e1 : \text{ArrowPipe } a \ c, x \Rightarrow y, \quad e2 : \text{ArrowPipe } b \ d, x \Rightarrow z}{e1 +++ e2 : \text{ArrowPipe } (\text{Either } a \ b) \ (\text{Either } c \ d), x \Rightarrow \max(y, z)} \\
\\
\text{arrow: } \&\&\& \frac{e1 : \text{ArrowPipe } a \ b, x \Rightarrow y, \quad e2 : \text{ArrowPipe } a \ c, (y + 1) \Rightarrow z}{e1 \&\&\& e2 : \text{ArrowPipe } a \ (b, c), x \Rightarrow z} \\
\\
\text{arrow: } *** \frac{e1 : \text{ArrowPipe } a \ b, x \Rightarrow y, \quad e2 : \text{ArrowPipe } a' \ c, (y + 1) \Rightarrow z}{e1 *** e2 : \text{ArrowPipe } (a, a') \ (b, c), x \Rightarrow z}
\end{array}$$

Listing 16: Rules fo role allocations of different combinators

The rule for the rest of combinators are shown in Listing 16. 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` combinator which lifts a sequential computation represented by `Core (a → 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.

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 the target platform which will be illustrated in the next chapter.

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 the standard method: tree transformation. The SPar expression 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 of channel communication

7.1.1 Syntax

7.1.2 Reified type

7.2 Compilation from SPar to Instr

7.2.1 Strategies for channel allocation

7.2.2 Type preservation

7.3 Code generation to C: from Instr to C

7.3.1 Representations of Core data type in C

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.