Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

The Interim Report

Author:
Shuhao Zhang

Supervisor:
Prof. Nobuko Yoshida
David Castro-Perez

Second Marker:
Dr Iain Phillips

January 25, 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 3

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.

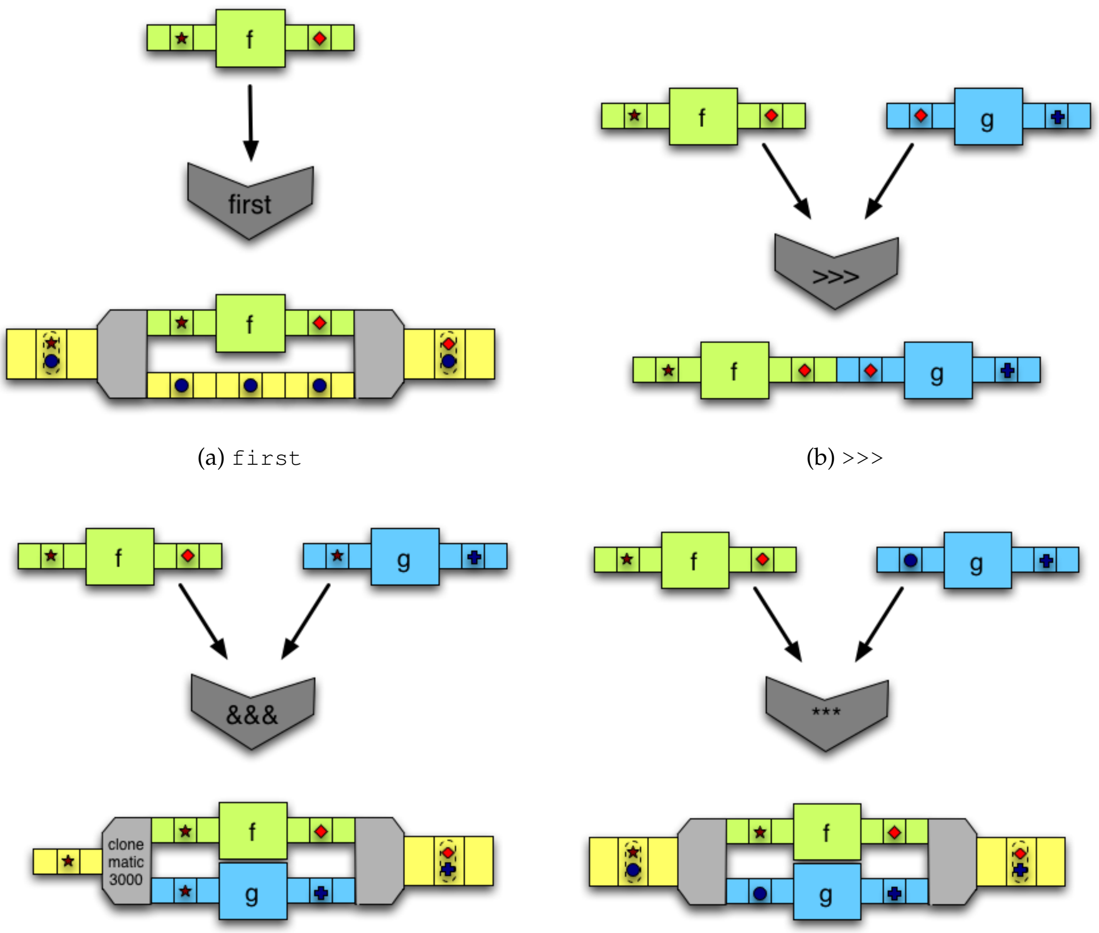
2.1.1 Definition

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)`

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

Listing 1: Arrow class in Haskell



(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: (→) 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)
```

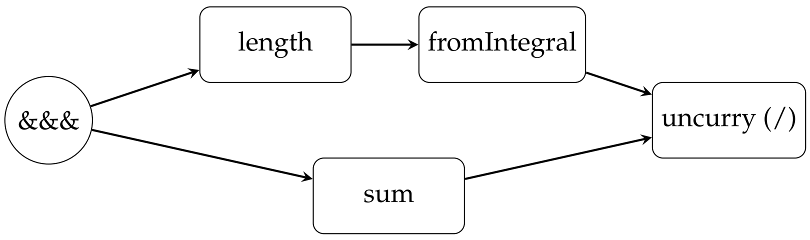


Figure 2.2: Visualization of mean′

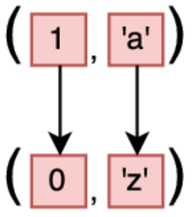
Arrows are not the only way to form point-free programs. The above code snippet is the more traditional approach form of point-free mean function in Haskell. We can argue this form of point-free function is more difficult to understand compared to arrows because it involves knowledge of monads (`liftM2`)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

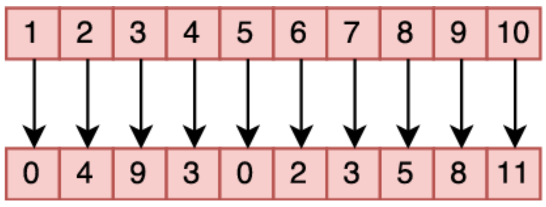
2.1.3 Application in parallel computation

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)

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

data TreeF a =
    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
$\mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end}$	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
$\mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end}$	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.

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

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

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

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

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

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

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

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

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.

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

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.

```haskell
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).

```haskell
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.

```haskell
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.

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

Then we can easily implement an interpreter for the example program

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

Finally, execute the program.

```
ghci:> goodMorningInterpreter 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

Project plan

This section is the proposed plan for the project. It is organised into milestones with the corresponding deadlines attached.

1. Week 5 - 6: Specification of the intermediate language.
    1.1. Week 5 - Jan 31st: Define the syntax.
    1.2. Week 6 - Feb 8th: Define the operation semantics
2. Week 7 - 8: A simulator that reflects the operation semantics of the language
    2.1. Week 8 - Feb 22nd: Implement a simulator that gathers all possible traces of execution of programs
3. Week 9: Session typing the language
    3.1. Week 9 - Mar 1st: Use session type to type check the language. There're two possible approaches
    - Encode session type into the Haskell type system to type check
    - Write our own type checker to type check the programs
4. Week 10: Translation rule
    4.1. Week 10- Mar 8th: Adapt a translation rule from a high-level language in parallel framework PAL to the language
5. Week 11 - 18: Code generation in C
    5.1. Week 13- Mar 29th: Specify the target language constructors (a subset of C)
    5.2. Week 16- Apr 19th: Translation scheme to the target language
    5.3. Week 18- May 3rd: Refine the implementations and debug
6. Week 19 - 23: Evaluation
    6.1. Week 20- May 17th: Implement example algorithms for benchmarking
    6.2. Week 22- May 31st: Benchmark the performance of the generated code
    6.3. Week 23- Jun 7th: Benchmark the tool-chain performance like compile time or size of generated code

Chapter 4

Evaluation

Completing the objectives in the introduction section is the first step of this project. In order to evaluate whether we have accomplished the objectives, first of all, to show the generality and expressibility of our languages, we will implement some algorithms like mergesort, Cooley-Tukey FFT or N-Body simulations in the high-level language and compile it to our language. Also, we will run code generation to the target low-level parallel code in C and measure the performance of generated code against sequential implementations as well as the parallel code generated by the original method used in the high-level framework. We would like to observe there will be no significant loss of efficiency or even better performance by adding one extra layer in the process of parallel code generations.

Also, not only will we benchmark the performance of execution of generated code, but also we will benchmark the performance of our tool-chain; e.g. measure the compile time against different source code size.

Finally, we will focus on measuring the quality of generated code regarding the size of the generated code and its readability. We hope we will not witness an exponential growth of code size against input data.

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/.