Department of Computer Science

Master's Thesis

A Process Calculus for Parallel and Distributed Programming in Haskell

Submitted:

March 2, 2015

Submitted by:
Christopher Blöcker
Mozartstraße 9
22941 Bargteheide
Phone: (04532) 24 834
E-mail: chrisbloecker@googlemail.com

Supervisor:
Prof. Dr. Ulrich Hoffmann
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-41
E-mail: uh@fh-wedel.de

Co-Supervisor:
Prof. Dr. Uwe Schmidt
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-45
E-mail: si@fh-wedel.de

> If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.
>
> *(Donald E. Knuth)*

Preface

In the very beginning it was my idea to build a distributed system that offers optimisation as a service, motivated by the omnipresence of computationally hard problems in theory and practice. I started investigating the idea in the context of a software project and by building a prototype of the system I had in mind.

My first attempt started off in elixir, a functional, dynamically typed programming language that could be described as a Ruby-fied version of Erlang and equally well suited for distributed programming as Erlang. But I found myself missing Haskell and its type system very soon after I started to design my data model and tried to express it in elixir. The main problem was the question of how to constrain the user to only submit valid process structures that obey the rules concerning static semantics of process composition. I took the decision to switch to Haskell and started over from scratch.

I finished the software project with a working prototype of an optimisation system implemented in Haskell and learned some lessons from it. There was more fundamental work to do than I had expected in the beginning. Of course there were tools for distributed programming, but I found they were mixing technical details about process management and communication with problem-specific code of algorithms. Furthermore, it was my idea to describe algorithms as a composition of processes and have them interpreted and executed in a distributed system. I decided to build the foundation that would allow to do so: a process calculus.

While designing the model for my process calculus and working on the implementation, I found myself running into the situation that Donald Knuth advises to avoid. In turns, I spent too much time only thinking about theory and only working on the implementation. As a result, I ended up with a gap between model and implementation. I had to take a step back and take a look at the whole picture. In the end, this led to an elegant model (if I may say so) and an implementation closely resembling it.

Once the calculus was designed and the implementation done, I realised that distribution is not the only thing the calculus can be used for. Parallelisation of algorithms on one computer is feasible as well and of interest for practical applications. It went fast to implement a second interpreter for processes, running on only one computer, not involving distribution. I was a bit surprised to see that the variant for parallelisation performs so much better than the distributed one that had been my goal from the beginning. Unfortunately, there was no time left to investigate in the reasons for this and to improve the performance of the distributed system.

Bargteheide, March 2, 2015 Christopher Blöcker

Contents

Preface . . . III

Nomenclature **VI**

1. Introduction **1**

1.1. Motivation . . . 1
1.2. Goal . . . 2
1.3. Results . . . 2

2. Related Work **3**

2.1. Process calculi . . . 3
2.1.1. Communicating Sequential Processes . . . 3
2.1.2. Calculus of Communicating Systems . . . 4
2.2. Programming languages . . . 5
2.2.1. Occam . . . 6
2.2.2. Haskell . . . 7

3. The Calculus **9**

3.1. Syntax . . . 9
3.2. Static Semantics . . . 10
3.3. Semantics . . . 12
3.4. Semantic Equivalence and Substitution . . . 16
3.5. Laws . . . 17
3.5.1. Associativity . . . 17
3.5.2. Distributivity . . . 17
3.5.3. Neutral elements . . . 17
3.5.4. Idempotence . . . 17
3.5.5. Further Laws . . . 17

4. Implementation **18**

4.1. Implementation using Concurrent Haskell . . . 19
4.1.1. Data Model . . . 19
4.1.2. Process Interpreter . . . 21
4.2. Implementation using Cloud Haskell . . . 24
4.2.1. Cloud Haskell . . . 24
4.2.2. Data Model . . . 25
4.2.3. Architecture of the Distribution Infrastructure . . . 26
4.2.4. Process Interpreter . . . 27
4.3. Example: a Parallel Interpreter for Arithmetic Expressions . . . 29

5. A Real World Example **31**

5.1. The Travelling Salesman Problem . . . 31

5.2. Meta-heuristics . . . 32
5.3. Artificial Ant Systems . . . 33
5.4. A Prototype of a Distributed Ant System . . . 35

6. Test **39**
6.1. Setup . . . 39
6.2. Results . . . 40
6.3. Interpretation . . . 42

7. Conclusion **43**

8. Future Work **44**

Appendices **45**

A. Proofs **46**

B. Code **52**
B.1. Implementation of the Process Interpreter using Cloud Haskell . . . 53
B.2. Ant System Processes . . . 54

C. DVD with source code **55**

List of Figures **56**

List of Listings **57**

Bibliography **59**

Index **60**

Nomenclature

$(_ \to _ \vee _)$ .... Choice composition of processes
$(_ \leftarrow _ | _)$ .... Parallel composition of processes
$(_ \triangleright _)$ .... Sequential composition of processes
$(_ \propto _)$ .... Repetition of a process

$\mathcal{P}$ .... Set of all processes
B, C, P, Q, R, S, T .... Processes
$\mathcal{T}$ .... Set of types
T_i .... Data type i
a, b, c, d, e .... Type variables
ρ .... Function that assigns a type signature to a process
$\perp$ .... The undefined value

sem .... Function that gives the semantics of a process
f_P .... Intrinsic function of a basic process P
$\circ$ .... Function composition
$B(x)$ .... Shorthand notation for $sem \langle B \rangle (x) = True$
$\overline{B(x)}$ .... Shorthand notation for $sem \langle B \rangle (x) = False$
Id .... The identity process
Err .... The error process

$\equiv$ .... Process equivalence
$P_{[Q/R]}$ .... Substitution of sub-process Q with R in P
$\overline{P}$ .... Process that is inverse to P

G .... A graph
V .... Set of nodes of a graph
E .... Set of edges of a graph
δ .... Labelling function for graph edges
ϕ .... Function that gives the length of a path in a graph

$N(i)$ .... Set of unvisited neighbour nodes of node i
$\eta_{i,j}$ .... Heuristic value of edge from node i to j
$\tau_{i,j}$ .... Pheromone concentration on edge from node i to j

τ_0 .... Initial pheromone concentration
τ_i^+ .... Pheromone amount to be deposited by ant i on used edges
α .... Value to weigh the influence of pheromones
β .... Value to weigh the influence of heuristic information
p_j .... Probability of visiting node j next
s_i .... Solution candidate i
c_s .... Costs of solution s

iff .... if and only if
ghc .... The Glasgow Haskell Compiler
JSON .... JavaScript Object Notation
TSP .... Travelling Salesman Problem

1 Introduction

In the introduction we start off with a motivation: what are the reasons for what we are doing? After that we set our goals and give a brief overview about our results.

1.1. Motivation

Both in theory and practice, computer scientists encounter computationally hard problems in a large variety of fields. This includes, e.g. register allocation in compilers, timetable layout and tour planning, all of which are formalised in a graph theoretic [GJ79] representation.

The limiting factor in these problems is the huge number of possible solutions, which **all** have to be examined in order to find the optimal solution [GJ79]. Even for small problem instances, the number of possible solutions is so big that this approach is rendered impractical. To tackle this difficulty, usually approximation algorithms, parallel algorithms, distributed algorithms or a combination of these is employed. Approximation algorithms speed up finding solutions by searching for approximate solutions, as their name suggests. Parallel and distributed algorithms break problems down into smaller pieces which then are solved in parallel.

Many of the currently widely used programming languages, such as Haskell, Java and Python, come with libraries for development of parallel and distributed programs. These libraries contain tools for running code remotely, serialising data, sending data to remote processes as well as concepts for synchronisation. Some languages, e.g. Erlang and elixir, are specially designed for the development of distributed software. However, unrelated to the particular problem, the code produced in this context always looks very similar and involves rather technical details concerning communication between processes. Its re-usability is limited insofar that code for process communication and synchronisation is mixed with the actual algorithm. A toolbox that hides these technical details and thereby makes it possible to describe algorithms on a higher level of abstraction is both desirable and would allow the developer to focus more on solving the actual problem than on the communication between processes.

1.2. Goal

Our goal is to design a toolbox that allows for the development of parallel and distributed programs on a higher level of abstraction. Ideally, we want to be able to define computations, to compose and to execute them. This could happen locally on one computer as well as in a distributed system. No matter which option is chosen, we do not want the user of this toolbox to be concerned about communication between processes, serialisation of data or synchronisation. It should only be apparent **that** parallelism is introduced, not **how** this is done. Other than by modelling computations to be carried out in parallel, the involvement of multiple threads, processes or a distributed system should be hidden entirely from the user.

We aim for a calculus on processes as easy to use as arithmetic operations on natural numbers. The calculus should only allow composition of compatible computations, i.e. only if their types match, respectively to the used combinator. We want to be able to check the validity of a described computation already at compile time to eliminate runtime errors that would arise from mismatching types.

The technique we plan to use in order to achieve an "automated" distribution of a process structure is the interpreter pattern [GHJV95]. The developer has to identify the pieces of a program and formulate them in a form of basic processes our calculus can operate on. Then, he has to combine these basic processes with the process combinators provided by our calculus. When the interpreter interprets a process, it examines its structure and takes care of its execution, "automatically" introducing parallelism wherever modelled accordingly. This leaves the developer with the responsibility to identify and mark parallel pieces of the program. Note, however, that we are **not** providing a tool that can parallelise any given algorithm automatically, since this is impossible due to the undecidability[1] of function equivalence.

Once we have developed the calculus and made an implementation, we want to assess its applicability in practice. We are interested in whether it is possible to achieve a speedup in execution of a program simply by modelling parts of it for parallel execution and using a suitable interpreter, both in case of parallelism on one computer and in a distributed system.

1.3. Results

We introduce a process calculus that can be used to describe parallel programs on a high level of abstraction and use Haskell for its implementation. The process calculus hides technical details concerning process management and communication between processes. Tests show that it is generally possible to achieve a speedup by parallelising programs using our process calculus. However, in the case of distribution of processes across a network of computers, we are not able to achieve a speedup and future work remains to be done.

[1] The decidability of function equivalence would imply the decidability of the halting problem, which has been shown to be undecidable [GJ79].

2
Related Work

In this chapter, we give an overview on related work that has been done in this field. First, we briefly introduce two process calculi, i.e. formalisms that allow to describe the behaviour of processes. After that, we discuss the application of these formal concepts in a selection of programming languages.

2.1. Process calculi

Process calculi are formalisms that can be used to describe processes and their behaviour in a concurrent system. The description is done in a mathematical way, similar to algebraic structures, and independently from an implementation. Usually process calculi operate on abstract processes and provide a set of operators and combinators which can be used to compose processes. By introducing this kind of formalism, process calculi make it possible to reason about processes and perform (equivalence) transformations on them, e.g. to obtain a more optimised[1] representation. In [HvS12], an algebraic model, which can be used to derive different process calculi, is discussed. The common, generalised model for various process calculi shows that these calculi are essentially the same and equivalent in their expressiveness.

2.1.1. Communicating Sequential Processes

Communicating Sequential Processes, or short CSP, was developed by C.A.R. Hoare and appeared in its first version in 1978. It was one of the first process calculi in the history of computer science. A few years later, in 1985, Hoare published a book about CSP [Hoa85]. We introduce a subset of CSP and show how to build processes using CSP, however, we don't go too deep into detail, neither do we claim to provide a complete description.

[1] Optimised according to a defined criterion, e.g. the minimal representation of a process.

In CSP, processes are composed of two types of primitives: sequential processes $\mathcal{P}$ and events $\mathcal{E}$. A process $P \in \mathcal{P}$ can observe an event $e \in \mathcal{E}$ and react to it. It cannot take influence on events or manipulate them as they are indivisible, but it can itself give raise to events that are observable by other processes. In the following, let $P, Q \in \mathcal{P}$ be processes and $a, b \in \mathcal{E}$ be events.

The prefix combinator "$\rightarrow$" takes an event a and a process P. $(a \rightarrow P)$ creates a new process that waits until it observes a and then behaves like P. A process that repeatedly waits to observe a and then behaves like P can be described using recursion: $P = (a \rightarrow P)$.

With the choice operator "|" we can construct a process that allows two flows of control. The process $(a \rightarrow P \,|\, b \rightarrow Q)$ waits until it either observes a and then behaves like P or observes b and then behaves like Q. Note that the prefix operator binds stronger than the choice operator and that the choice operator has commutative property.

The parallel operator "||" allows us to put processes together such that they run in parallel. The process created by $(P \,||\, Q)$ waits for an event that both P and Q can observe and then behaves like P and Q concurrently. Let $P = (a \rightarrow P \,|\, b \rightarrow a \rightarrow P)$ and $Q = (a \rightarrow Q)$ then $(P \,||\, Q) = (a \rightarrow P \,||\, a \rightarrow Q)$, since Q cannot observe b and thus the second choice from P has to be omitted.

In contrast to the parallel operator "||", the interleaving operator "|||" can be used to put processes together to execute concurrently, independently from what events they are able to observe. The process $(P \,|||\, Q)$ behaves like P and Q at the same time. If an event occurs that exactly one of the two processes can observe, the according process does so. If an event occurs that both P and Q are able to observe, the choice for which of them exclusively observes it is made non-deterministically.

Communication between processes is done by using channels: a channel exists between exactly two processes and is uni-directional. Let c be a channel, let P be the process at the sending end of c and Q be the process at the receiving end of c. Then P can send a message m over the channel using the sending operator "!": $c!m$. Q can receive this message using the receiving operator "?": $c?m$. Sending and receiving messages are ordinary events, created and observed by processes. A process that reads messages from channel c_{in} and forwards them to channel c_{out} may look like this: $P = (c_{in}?m \rightarrow c_{out}!m \rightarrow P)$.

We have seen some basic combinators of CSP and how to use them to compose processes based on processes and events. We can construct processes that run in parallel and make their execution dependent on observable events. The formalism of CSP allows reasoning about processes and transformations on them. CSP comes with many more operators and combinators than the ones we have introdcued, but as already mentioned, we do not intend to give a complete description.

2.1.2. Calculus of Communicating Systems

The Calculus of Communicating Systems, short CCS, is a process calculus introduced by Robin Milner in 1982 [Mil82]. CCS arose around the same time as CSP and belongs to the circle of the first process calculi. It comprises a relatively small set of combinators which are used to describe process behaviour. We give an overview over some of the combinators and illustrate how to use them. Note that our introduction to CCS is not complete and we only want to show the general idea.

In CCS, processes (also called *agents*) have a name and are equipped with a set of labelled ports. A port can be seen as an endpoint of a communication channel and can be used for either sending or receiving messages, depending on which end it represents. For a channel with name c, the sending port is denoted by c and the receiving port is denoted by $\overline{c}$. Processes can perform internal actions, e.g. a calculation, and external actions, i.e. sending or receiving messages over ports.

Processes are defined inductively: $\emptyset$ is the empty process that does not perform any action. Using the prefixing operator ".", processes can be prefixed with an action. Let P be a process and a be an action, then $a.P$ is a process that performs action a and then behaves like P. Repetition can be expressed using recursive process definitions: a process that repeatedly performs action a can be described by $P := a.P$.

Other operators for process composition are, e.g. the choice operator "+" and the parallel operator "|". Let P, Q be processes. $P + Q$ creates a process that behaves like P or like Q, where the choice between P and Q is made non-deterministically. $P \mid Q$ creates a process that concurrently executes P and Q.

Processes created using the parallel combinator can share channels and use them to send messages to each other. In CCS, communication over channel requires participation of processes at both endpoints at the same time. When a process sends a message over a channel, it must wait for the process on the other side to be ready to receive the message and vice versa. As a consequence, exchanging messages always includes mutual synchronisation of the participating processes. Let a, b be actions, let c be a channel and let x be a value to be sent over c. Then two processes P, Q that participate it the exchange of x over c may be described by $P := a.c\left(x\right).P$ and $Q := b.\overline{c}\left(x\right).Q$, where P is in control over the sending port of c and Q is in control over the receiving port of c. The composition $P \mid Q$ enables them to use c as a channel to synchronise and exchange information.

2.2. Programming languages

Most programming languages come with support for parallel and concurrent programming, either built into the language or available as some sort of extension. This makes it possible to transform the description of processes made in e.g. CCS or CSP into executable programs. This transformation is fairly straight-forward: sequential composition can be expressed by chaining together a sequence of processes, executing exactly one of them at a time in the order of composition. Parallel execution of processes can be done by executing each of the processes in a separate thread and waiting for all of them to terminate. The choice between processes can usually be expressed using *if-then-else* compositions.

2.2.1. Occam

The programming language occam[2], which first appeared in 1983, resembles the ideas of CSP closely. It incorporates direct implementations of the operations of choice between processes as well as parallel and sequential combination of processes. Furthermore, it introduces named channels and both sending and receiving operations on them.

Sending data on a channel can be done using the sending operator `!`, receiving data from a channel can be done using the receiving operator `?`, see listing 2.1.

```
1 channel ! out
2 channel ? in
```

Listing 2.1: Sending data over a channel and receiving data from a channel in occam.

In contrast to many other programming languages where sequential composition is implicitly the standard behaviour, occam has an operator for sequential composition of expressions: `SEQ`. It takes a list of processes and executes them sequentially as shown in listing 2.2.

```
3 SEQ
4   x := 3 * 7
5   y := x * 2
```

Listing 2.2: Sequential composition of processes in occam.

Parallel composition can be done using the `PAR` combinator in occam. Like `SEQ`, it takes a list of processes and evaluates them concurrently as shown in listing 2.4.

```
6 PAR
7   x := f(a)
8   y := g(b)
```

Listing 2.3: Parallel composition of processes in occam.

A non-deterministic choice between different alternatives of processes can be made using the `ALT` combinator. It takes a list of guarded processes and non-deterministically selects one of them for execution, provided its guard signals readiness for the execution of the process. If none of the guards signals readiness, `ALT` waits until one of them does. A guard can be either a boolean expression, the action of reading from a channel or the combination of both.

```
9  ALT
10   x > 3 & chan1 ? msg
11     SEQ
12       ...
13   chan2 ? msg
14     PAR
15       ...
16   ...
```

Listing 2.4: Choice between process alternatives in occam.

[2] The reference manual for occam can be found at http://www.wotug.org/occam/documentation/oc21refman.pdf

2.2.2. Haskell

In Haskell, there are a number of different modules available for expressing concurrency, parallelism[3] and distribution. Typically, these modules introduce a monad that comes with different operations for the expression of computations, such as sequential or parallel execution. Two of these modules are Concurrent Haskell in `Control.Concurrent` for concurrency, and Parallel Haskell in `Control.Parallel` for parallelism. A third module for distribution, Cloud Haskell in `Control.Distributed.Process`, is discussed later in chapter 4.2.1.

Concurrent Haskell

Concurrent Haskell introduces the operation `forkIO` that creates a new lightweight thread that executes concurrently in the `IO` monad. Furthermore, a communication mechanism for threads is introduced: mutable variables, or `MVars`. An `MVar` can either be full or empty and can be used by threads to communicate information. `MVars` can be read when they hold information, and they can be used to store information when they are empty. When a thread tries to read from an empty `MVar` or to write to an `MVar` that currently holds information, it is suspended until the `MVar`'s state is changed by another thread. Hence, `MVars` can be used for synchronisation [Mar13].

Assume there are operations `a`, `b`, `c` where `a` and `b` take an `MVar` and can be executed concurrently. `c` consumes the results produced by `a` and `b` and therefore has to wait for both these results to be available. Using `Concurrent Haskell`, this can be expressed as shown in listing 2.5.

```
1  import Control.Concurrent
2  import Control.Concurrent.MVar
3
4  main :: IO ()
5  main = do
6    -- create mvars for communication and synchronisation
7    mvarA <- newEmptyMVar
8    mvarB <- newEmptyMVar
9
10   -- run a and b in new threads
11   _ <- forkIO (a mvarA)
12   _ <- forkIO (b mvarB)
13
14   -- wait for a and b to return
15   ra <- takeMVar mvarA
16   rb <- takeMVar mvarB
17
18   res <- c ra rb
19   ...
```

Listing 2.5: Parallel and sequential composition in Concurrent Haskell.

[3] Note that there is a difference between concurrency and parallelism: while parallelism aims at performing a computation more quickly by executing computations on different processors, concurrency is a concept to structure programs and express actions that **conceptually** happen at the same time by executing them in different threads of control. However, it is an implementation detail if these threads actually run at the same time [Mar13].

Parallel Haskell

Parallel Haskell introduces the `Eval` monad along with the operations `rpar`, `rseq` and `runEval`. `rpar` is used to create a parallel computation while `rseq` is used to create a computation that is executed sequentially. `runEval` is used to execute computations created with `rpar` and `rseq`. In contrast to Concurrent Haskell, `Parallel Haskell` does not build on the `IO` monad, computations in the `Eval` monad are pure. In addition to that, Parallel Haskell introduces the concept of *strategies*. Strategies allow for a better modularisation of code and make the decoupling of algorithms and their parallelisation possible. The concrete way of parallelistaion can then easily be changed by using a different evaluation strategy [Mar13].

Analogously to before, assume there are function `a`, `b` and `c`. Assume that `a` and `b` can be evaluated in parallel and `c` consumes the values created by `a` and `b`, but cannot start before both these values are available. Using Parallel Haskell, this can be expressed as shown in listing 2.6

```
1  import Control.Parallel
2  import Control.Parallel.Strategies
3
4  foo :: a
5  foo = runEval $ do
6    -- execute a and b in parallel
7    ra <- rpar a
8    rb <- rpar b
9
10   -- wait for both a and b to return
11   rseq ra
12   rseq rb
13
14   res <- rseq (c ra rb)
15   ...
```

Listing 2.6: Parallel and sequential composition in Parallel Haskell.

3
The Calculus

In the previous chapter, we have seen two examples of process calculi and how they work. In this chapter, we develop our own calculus and define its semantics. In contrast to CCS and CSP, our model does not involve non-determinism in the choice between processes. Furthermore, it does not involve any explicit notation of communication between processes.

First, let us define what we understand as a *process*: a process is (a piece of) a computer program. It can run concurrently with other processes. A process receives an input and produces an output. Essentially, a process can be seen as a function and just like that, it is pure and free of side-effects, it does not have any other state than the input it receives upon creation.

We are only interested in **what** a process does, not **how** it does that. We treat processes as black boxes that receive an input, perform an action and eventually produce an output. Our point of interest is the composition of processes and the definition of the semantics of the resulting processes, which we give in denotational form.

3.1. Syntax

Let $\mathcal{P}$ be the set of processes and let $P, Q, R \in \mathcal{P}$ be processes. Then syntactically correct processes can be formed by using the following construction rules:

- $(R \rightarrow P \vee Q)$ (Choice)
- $(R \leftarrow P \mid Q)$ (Parallel)
- $(P \triangleright Q)$ (Sequence)
- $(R \propto P)$ (Repetition)

3.2. Static Semantics

In this section, we define the static semantics of the process combinators in our calculus. We define what a **basic** and what a **composed** process is and we equip processes with a type signature. The **type signatures** of processes are used to determine whether they can be composed using a certain combinator to yield a valid process with respect to their types.

As before, let $\mathcal{P}$ be the set of processes. A basic process is a process that performs a basic operation an is no further divisible insofar, that it does not involve any of the introduced process combinators. A composed process is a process that is composed of other processes, using one or many process combinators.

Intuitively, we call values that we present to processes their **input** and values produced by processes their **output**.

Definition 3.1 (Types)**.** *Let $\mathcal{T}$ be the set of types and T_i a type for every i with*

$$\mathcal{T} = \bigcup_i \{T_i\}$$

Furthermore, let lowercase letters, e.g. a, b, c be type variables, i.e. placeholders for types.

□

A type represents a set of values, all of which share some common property and therefore, are attributed to the same type. Examples include the type of natural numbers $T_{\mathbb{N}}$, commonly referred to as *integers*, and the type of boolean values $T_{Boolean} = \{False, True\}$.

Definition 3.2 (Undefined values)**.** *For every $T_i \in \mathcal{T}$, let $\bot_i \in T_i$ be the undefined value of type T_i that is distinguishable from every other value of type T_i. Every type implicitly contains an undefined value, even if not explicitly mentioned[1]. When talking about the undefined value $\bot$, we omit its type and only mention it explicitly when necessary.*

□

Definition 3.3 (Type signature)**.** *The type signature of a process states of which types the input and the output of the respective process are. Let $\rho\colon \mathcal{P} \to \mathcal{T} \times \mathcal{T}$ be the function that assigns a type signature to processes.*

□

Definition 3.4 (Identity and error process)**.** *Let $Id \in \mathcal{P}$ be the identity process, i.e. the process that outputs its input, and let $Err \in \mathcal{P}$ be the error process, i.e. the process that always returns the undefined value.*

□

To be precise, there is an identity process Id_i with $\rho(Id_i) = (T_i, T_i)$ and an error process Err_i with $\rho(Err_i) = (T_i, T_i)$ for every $T_i \in \mathcal{T}$. For simplicity, we refer to them as Id and Err, but keep in mind that for every type, there is a specific identity and error

[1] That means that the type of natural numbers $T_{\mathbb{N}}$ contains the undefined value $\bot_{\mathbb{N}}$ and the type $T_{Boolean}$ actually contains the three values $False$, $True$ and $\bot_{Boolean}$.

process. Let their type signatures be $\rho(Id) = (a, a)$ and $\rho(Err) = (a, a)$, where a is a type variable resembling the fact that Id and Err represent a family of processes.

In the following, we also call processes $B \in \mathcal{P}$ with output type $T_{Boolean}$, i.e. with the property $\rho(B) = (a, T_{Boolean})$ *predicates*. They are used to express deterministic choice between processes and termination of repetition.

Definition 3.5 (Static semantics of choice composition)**.** *Let B be a predicate and let $P, Q \in \mathcal{P}$ be processes and let a, b be type variables. Then the composition $(B \to Q \vee P)$ is valid iff*

$$\begin{aligned} \rho(B) &= (a, T_{Boolean}) \\ \rho(P) &= (a, b) \\ \rho(Q) &= (a, b) \end{aligned}$$

and invalid otherwise. The type signature of the resulting process is $\rho((B \to P \vee Q)) = (a, b)$.

□

Definition 3.6 (Static semantics of parallel composition)**.** *Let $P, Q, R \in \mathcal{P}$ be processes and let a, b, c, d be type variables. Then the composition $(R \leftarrow P \mid Q)$ is valid iff*

$$\begin{aligned} \rho(P) &= (a, c) \\ \rho(Q) &= (a, d) \\ \rho(R) &= ((c, d), b) \end{aligned}$$

and invalid otherwise. The type signature of the resulting process is $\rho((R \leftarrow P \mid Q)) = (a, b)$.

□

Definition 3.7 (Static semantics of sequential composition)**.** *Let $P, Q \in \mathcal{P}$ be processes and let a, b, c be type variables. Then the composition $(P \triangleright Q)$ is valid iff*

$$\begin{aligned} \rho(P) &= (a, c) \\ \rho(Q) &= (c, b) \end{aligned}$$

and invalid otherwise. The type signature of the resulting process is $\rho((P \triangleright Q)) = (a, b)$.

□

Definition 3.8 (Static semantics of repetition)**.** *Let B be a predicate, let $P \in \mathcal{P}$ be a process and let a be a type variable. Then the composition $(B \propto P)$ is valid iff*

$$\begin{aligned} \rho(B) &= (a, T_{Boolean}) \\ \rho(P) &= (a, a) \end{aligned}$$

and invalid otherwise. The type signature of the resulting process is $\rho((B \propto P)) = (a, a)$.

□

3.3. Semantics

In chapter 3.1 we have introduced the syntactical rules for process construction and in chapter 3.2 we have introduced static semantics of process composition. In this section, we define the semantics of processes by giving them a denotation as it is common in the design of programming languages [Sch86].

For the definition of process semantics, we introduce the polymorphic function

$$sem\colon \mathcal{P} \to a \to b.$$

sem takes a process $P \in \mathcal{P}$ and returns its meaning, i.e. the function computed by P. We write $sem\,\langle P\rangle$ for that. Since $sem\,\langle P\rangle$ is a function, it can be applied to a concrete input value x for P, written as $sem\,\langle P\rangle\,(x)$. The expression $sem\,\langle P\rangle\,(x)$ is valid iff x is of appropriate type to be used as input for P.

For every basic process, there is a function that is intrinsic to that particular process: for a basic process $P \in \mathcal{P}$, let its intrinsic function be called f_P. Composed processes do not involve an intrinsic function and the notation f_P does not make sense if P is not a basic process. The semantics of a basic process $P \in \mathcal{P}$ is directly given by its intrinsic function.

Definition 3.9 (Semantics of a basic process)**.** *Let P be a basic process, then the semantics $sem\,\langle P\rangle$ is given by P's intrinsic function f_P:*

$$sem\,\langle P\rangle = f_P.$$

□

Example 3.10 (Semantics of basic processes)**.** *Let $\sigma\colon T_{\mathbb{N}} \to T_{\mathbb{N}}, x \mapsto x+1$ be the function that is intrinsic to the basic process $S \in \mathcal{P}$. Then the semantics $sem\,\langle S\rangle$ of S is given by its intrinsic function σ:*

$$sem\,\langle S\rangle = \sigma = x \mapsto x+1$$

Let $\delta\colon T_{\mathbb{N}} \to T_{\mathbb{N}}, x \mapsto 2x$ be the function that is intrinsic to the basic process $D \in \mathcal{P}$. Then the semantics $sem\,\langle D\rangle$ of D is given by its intrinsic function δ:

$$sem\,\langle D\rangle = \delta = x \mapsto 2x$$

Definition 3.11 (Semantics of the identity process)**.** *Let $Id \in \mathcal{P}$ be the identity process, i.e. the process that always outputs its input. Its semantics is:*

$$sem\,\langle Id\rangle = x \mapsto x.$$

□

The semantics of the error process Err, no matter which input it receives, is always the undefined value $\bot$ of the respective type.

Definition 3.12 (Semantics of the error process)**.** *Let $Err \in \mathcal{P}$ be the error process, i.e. the process that always outputs the undefined value $\bot$. Its semantics is:*

$$sem\,\langle Err\rangle = x \mapsto \bot.$$

□

The semantics of composed processes is determined by their structure, i.e. the used process combinator, and the semantics of the involved sub-processes. The semantics of processes created by choice composition, parallel composition and sequential composition is defined inductively. The semantics of processes created by repetition composition is given by a recursive equation.

Definition 3.13 (Sub-process)**.** *Let $P \in \mathcal{P}$ be a composed process. Then the processes P is composed of are called the sub-processes of P.*

□

Definition 3.14 (Shorthand notation for predicate processes)**.** *Let B be a predicate. Then $B(x)$ is a shorthand notion for $sem\langle B\rangle(x) = True$ and $\overline{B(x)}$ is a shorthand notation for $sem\langle B\rangle(x) = False$. For non-predicates this notation is not defined.*

□

The semantics of a process with the structure of choice composition is that of one of its sub-processes. A predicate is employed to inspect the input and decide which of the sub-processes determines the semantics.

Definition 3.15 (Semantics of choice composition)**.** *Let B be a predicate, let $P, Q \in \mathcal{P}$ be processes and let them be composed using choice composition, i.e. $(B \to P \vee Q)$. Then, based on $sem\langle B\rangle$, the resulting process behaves either like P or Q. The semantics of $(B \to P \vee Q)$ is given by:*

$$sem\langle(B \to P \vee Q)\rangle = x \mapsto \begin{cases} sem\langle P\rangle(x) & \text{if } B(x) \\ sem\langle Q\rangle(x) & \text{if } \overline{B(x)} \\ \bot & \text{otherwise.} \end{cases}$$

□

Example 3.16 (Semantics of choice composition)**.** *Consider the definitions of S and D from example 3.10. Furthermore, let $Even \in \mathcal{P}$ with $\rho(Even) = (\mathbb{N}, T_{Bool})$ be the process that outputs $True$ if its input is an even number and $False$ otherwise. If we compose S and D, guarded by $Even$, using choice composition, the semantics of the resulting process is as follows:*

$$sem\langle(Even \to S \vee D)\rangle = x \mapsto \begin{cases} sem\langle S\rangle(x) & \text{if } x \text{ is even} \\ sem\langle D\rangle(x) & \text{if } x \text{ is odd} \\ \bot & \text{otherwise} \end{cases}$$

The semantics of a process with the structure of parallel composition is given by the semantics of its sub-processes, combined by a combinator process.

Definition 3.17 (Semantics of parallel composition)**.** *Let $P, Q, R \in \mathcal{P}$ be processes and let them be composed using parallel composition, i.e. $(R \leftarrow P \mid Q)$. Then the semantics of the resulting process $(R \leftarrow P \mid Q)$ is given by:*

$$sem\langle(R \leftarrow P \mid Q)\rangle = x \mapsto sem\langle R\rangle(sem\langle P\rangle(x), sem\langle Q\rangle(x)).$$

□

Example 3.18 (Semantics of parallel composition)**.** *Consider the definitions of* σ, δ, S *and* D *from example 3.10. Furthermore, let* $sum\colon T_{\mathbb{N}} \times T_{\mathbb{N}} \to T_{\mathbb{N}}, (x, y) \mapsto x + y$ *be the function that is intrinsic to the basic process* $Sum \in \mathcal{P}$*. Then the semantics of the process created by parallel composition of* S *and* D*, combined with* Sum*, is:*

$$\begin{aligned} sem\left\langle (Sum \leftarrow S \mid D)\right\rangle &= x \mapsto sem\left\langle Sum\right\rangle \left(sem\left\langle S\right\rangle (x), sem\left\langle D\right\rangle (x)\right) \\ &= x \mapsto sem\left\langle Sum\right\rangle \left(\sigma (x), \delta (x)\right) \\ &= x \mapsto sum\left(\sigma (x), \delta (x)\right) \\ &= x \mapsto sum\left(x + 1, 2x\right) \\ &= x \mapsto (x + 1) + (2x) \\ &= x \mapsto 3x + 1 \end{aligned}$$

The semantics of a process with the structure of sequential composition is given by applying the function defined by its sub-processes sequentially. This can be expressed using function composition.

Definition 3.19 (Semantics of sequential composition)**.** *Let* $P, Q \in \mathcal{P}$ *be processes and let them be composed using sequential composition, i.e.* $(P \triangleright Q)$*. Then the semantics of the resulting process is given by function composition:*

$$sem\left\langle (P \triangleright Q)\right\rangle = sem\left\langle Q\right\rangle \circ sem\left\langle P\right\rangle = x \mapsto sem\left\langle Q\right\rangle \left(sem\left\langle P\right\rangle (x)\right)$$

□

Example 3.20 (Semantics of sequential composition)**.** *Consider the definitions of* σ, δ, S *and* D *from example 3.10 and use sequential composition to compose* D *and* S*. Since function composition* $\circ$ *does not have commutative properties,* $(S \triangleright D)$ *and* $(D \triangleright S)$ *yield different results.*

$$\begin{aligned} sem\left\langle (S \triangleright D)\right\rangle &= sem\left\langle D\right\rangle \circ sem\left\langle S\right\rangle \\ &= \delta \circ \sigma \\ &= x \mapsto \delta\left(\sigma (x)\right) \\ &= x \mapsto 2\left(x + 1\right) \\ &= x \mapsto 2x + 2 \end{aligned}$$

$$\begin{aligned} sem\left\langle (D \triangleright S)\right\rangle &= sem\left\langle S\right\rangle \circ sem\left\langle D\right\rangle \\ &= \sigma \circ \delta \\ &= x \mapsto \sigma\left(\delta (x)\right) \\ &= x \mapsto 2x + 1 \end{aligned}$$

The semantics of a process with the structure of repetition composition is given by a recursive equation. It involves repeated sequential composition of a process with itself. A predicate is employed to test the process's input for a property and determine the length of the process composition sequence.

Definition 3.21 (Semantics of repetition composition)**.** *Let B be a predicate, let $P \in \mathcal{P}$ be processes and let them be composed using repetition composition, i.e. $(B \propto P)$. Then the semantics of $(B \propto P)$ is given by the following recursive equation:*

$$sem\,\langle(B \propto P)\rangle = sem\,\langle(B \to (P \triangleright (B \propto P)) \vee Id)\rangle\,.$$

□

The existence of a solution satisfying definition 3.21 is not immediately obvious. However, this is a well-known and well-studied problem. The theory of *least fixed point semantics* from domain theory guarantees the existence of a solution and gives information about how to find it [Sch86].

Example 3.22 (Semantics of repetition composition)**.** *Consider the definition of δ and D from example 3.10 and let B be a predicate that returns $True$ if its input is smaller than 1024 and $False$ if its input is equal to 1024 or bigger.*

$$\begin{aligned}
& sem\,\langle(B \propto D)\rangle \\
= \; & sem\,\langle(B \to (D \triangleright (B \propto D)) \vee Id)\rangle \\
= \; & x \mapsto \begin{cases} sem\,\langle(D \triangleright (B \propto D))\rangle\,(x) & \text{if } x < 1024 \\ sem\,\langle Id\rangle\,(x) & \text{if } x \geq 1024 \\ \bot & \text{otherwise} \end{cases} \\
= \; & x \mapsto \begin{cases} sem\,\langle(D \triangleright (B \to (D \triangleright (B \propto D)) \vee Id))\rangle\,(x) & \text{if } x < 1024 \\ sem\,\langle Id\rangle\,(x) & \text{if } x \geq 1024 \\ \bot & \text{otherwise} \end{cases} \\
= \; & \ldots
\end{aligned}$$

$(B \propto D)$ yields a process that takes its input, doubles it and passes it on recursively to itself as long as its input is smaller than 1024. If its input is bigger than 1024, it outputs its input.

3.4. Semantic Equivalence and Substitution

Based on the definition of process semantics from chapter 3.3, we have a tool at hand to reason about processes and perform transformations on them.

Definition 3.23 (Equivalence of processes)**.** *Let $P, Q \in \mathcal{P}$ be processes. Then P and Q are equivalent iff their semantics are the same. We write $P \equiv Q$ to indicate equivalence of P and Q.*

$$P \equiv Q \Leftrightarrow sem\langle P\rangle = sem\langle Q\rangle$$

□

Definition 3.24 (Process substitution)**.** *Let $P, Q, R \in \mathcal{P}$ be processes and let Q be a sub-process of P. Then P can be transformed into a new process by replacing one or more occurrence of Q in P with R. This is denoted by $P_{[Q/R]}$.*

□

Theorem 3.25 (Substitution with an equivalent process leaves semantics unchanged)**.** *Let $P, Q, R \in \mathcal{P}$ be processes and let $Q \equiv R$. Then, substituting one or more occurrences of Q in P with R leaves $sem\langle P\rangle$ unchanged:*

$$Q \equiv R \rightarrow P \equiv P_{[Q/R]}.$$

□

The property stated in theorem 3.25 allows us to transform processes into a different representation without altering their semantics and to optimise them according to some measurable property. E.g. one might want to find a more "simple" representation of a process or reduce its complexity by replacing sub-processes. A proof of theorem 3.25 can be found in appendix A, proof A.16.

Definition 3.26 (Inverse process)**.** *Let $P \in \mathcal{P}$ be a process. Then an inverse process $\overline{P}$ for P is a process that satisfies the following property:*

$$\left(P \triangleright \overline{P}\right) \equiv Id.$$

□

As a final remark, we point out that, for a process P, an inverse process as defined in definition 3.26 does **not** exist in general since not every computable function is invertible[2].

[2] Take, e.g. the function $f\colon x \mapsto x^2$. Clearly, f is not injective since $f(x) = f(-x)$ and therefore not invertible. However, there is an inverse relation $\overline{f_r}\colon x \mapsto \{\overline{x} \mid f(\overline{x}) = x\}$ that maps a value x to the set of values that, if f is applied to them, yield x.

3.5. Laws

In chapter 3.4, definition 3.23, we have defined under which circumstances two processes are equivalent. This definition, together with the definitions for process semantics given in chapter 3.3, allows us to find a set of laws regarding process equivalence in our calculus that can be used by the user of the calculus to transform processes into different representations, satisfying his needs.

In the following, let B be a predicate and let $C, P, Q, R \in \mathcal{P}$ be processes with suitable type signatures for composition in the respective cases. Proofs for the stated laws can be found in appendix A.

3.5.1. Associativity

The associative property holds for sequential composition.

$$(P \triangleright (Q \triangleright R)) \quad \equiv \quad ((P \triangleright Q) \triangleright R)$$

3.5.2. Distributivity

Parallel and sequential composition distribute over choice composition. Also, sequence composition distributes over parallel composition.

$$\begin{aligned} (C \leftarrow P \,|\, (B \rightarrow Q \vee R)) \quad &\equiv \quad (B \rightarrow (C \leftarrow P \,|\, Q) \vee (C \leftarrow P \,|\, R)) \\ (P \triangleright (B \rightarrow Q \vee R)) \quad &\equiv \quad (B \rightarrow (P \triangleright Q) \vee (P \triangleright R)) \\ (P \triangleright (C \leftarrow Q \,|\, R)) \quad &\equiv \quad (C \leftarrow (P \triangleright Q) \mid (P \triangleright R)) \end{aligned}$$

3.5.3. Neutral elements

The identity process Id is both a left and right neutral element for sequential composition.

$$\begin{aligned} (Id \triangleright P) \quad &\equiv \quad P \\ (P \triangleright Id) \quad &\equiv \quad P \end{aligned}$$

3.5.4. Idempotence

The identity process Id and the error process Err are idempotent[3] elements regarding sequential composition.

$$\begin{aligned} (Id \triangleright Id) \quad &\equiv \quad Id \\ (Err \triangleright Err) \quad &\equiv \quad Err \end{aligned}$$

3.5.5. Further Laws

Sequential composition with the error process Err on the right always yields the error process.

$$(P \triangleright Err) \quad \equiv \quad Err$$

[3] Let S be a set and $*\colon S \times S \rightarrow S$ be a binary operation on the elements of S. An element $x \in S$ is considered to be idempotent with respect to $*$ if $x * x = x$. If for all $x \in S$ the idempotence property $x * x = x$ holds, the operation $*$ is considered idempotent.

4 Implementation

In this chapter, we describe two implementations of the calculus from chapter 3 in Haskell. We design our implementation in Haskell to closely resemble the structure and semantics of our calculus, so it is straightforward to transform process structures formulated in Haskell into a representation in our calculus and reason about them. As a useful consequence, the laws stated in chapter 3.5 transform from the calculus to processes in our implementation and remain valid.

For one of the implementations, we use Concurrent Haskell and fork local threads in the `IO` monad for parallelisation on one computer. For the other one, we use Cloud Haskell to distribute processes into a distributed system and run them in Cloud Haskell's `Process` monad. Other implementations based on, e.g. Parallel Haskell are imaginable.

Ideally, we want the two implementations to be usable interchangeably by simply importing the desired implementation and without changing any of the productive code, however, we cannot achieve this at the moment. When using the implementation based on Cloud Haskell, we need to require additional properties of the data types that are transmitted over the network to other processes. Furthermore, in order to run code remotely, the code has to be known on the remote node since ghc[1] does not support serialising and transmitting functions over the network [EBPJ11]. To solve this problem, a trick is used that requires us to change the model of basic processes slightly. However, we are optimistic that in future versions of ghc, it will be possible to achieve full interchangeability of both (and possibly more) implementations.

[1] The Glasgow Haskell Compiler, c.f. `https://www.haskell.org/ghc/`.

4.1. Implementation using Concurrent Haskell

In the implementation for local parallelisation on one computer, we make use of Concurrent Haskell. Basic processes are represented as computations in the `IO` monad. Parallelisation is achieved by running processes that are modelled for parallel execution concurrently in lightweight threads using `forkIO`, synchronisation is done using `MVar`s.

An introduction to Concurrent Haskell and Haskell's `IO` monad can be found in standard Haskell literature such as [Hut07], [Bir98] or [Mar13]. The fact that we use the `IO` monad has immediate consequences: we cannot force processes to be free of side-effects, hence they can perform actions like, e.g. reading from or writing to the file system.

4.1.1. Data Model

Our data model directly resembles the structure of the syntactic rules of process composition from chapter 3.1. We use the power of Haskell's type system and a generalised algebraic data type to assure the model meets the requirements concerning static semantics formulated in chapter 3.2 and prevent the creation of invalid processes with respect to static semantics.

As mention before, basic processes are computations in the `IO` monad. They receive an input of type `a` and output a value of type `b`.

```
1 type BasicProcess a b = a -> IO b
```

Listing 4.1: Representation of basic processes as computations in the `IO` monad.

Predicates are represented as processes that receive an input and output a value of type `Bool`, as introduced in chapter 3.2.

```
2 type Predicate a = Process a Bool
```

Listing 4.2: Representation of predicates as processes.

The data type for processes involves two type parameters, `a` and `b`. They reflect the process' input and output types where `a` is the input type and `b` is the output type.

```
3 data Process a b where
```

Listing 4.3: Data type for the representation of processes.

The identity process and the error process are modelled using the `Id` and the `Err` data constructors. Their type signatures are as stated in definition 3.4.

```
4 Id  :: Process a a
5 Err :: Process a a
```

Listing 4.4: Signatures of the `Id` and `Err` data constructors.

Basic processes are turned into processes by wrapping them using the `Basic` data constructor. The developer is cautioned about using basic processes that incorporate side-effects as they potentially render the laws given in chapter 3.5 invalid. `Basic` takes a basic process of type `BasicProcess a b`, i.e. a computation in the `IO` monad that takes an input of type `a` and outputs a value of type `b`, and creates a process from it.

```
6  Basic :: BasicProcess a b
7        -> Process a b
```

Listing 4.5: Signature of the `Basic` data constructor.

The `Choice` data constructor is used to build a process that makes a choice between two processes. Its type signature satisfies definition 3.5: `Choice` takes a predicate of type `Predicate a`, two processes of type `Process a b` and results in a process of type `Process a b`.

```
8  Choice :: Predicate a
9         -> Process a b
10        -> Process a b
11        -> Process a b
```

Listing 4.6: Signature of the `Choice` data constructor.

The `Parallel` data constructor is used to model parallel composition of processes, its type signature reflects definition 3.6. It takes two processes for parallel composition with type signatures `Process a c` and `Process a d` as well as a third process of type `Process (c, d) b` that is used to combine the results of the other two processes.

```
12 Parallel :: Process (c, d) b
13          -> Process a c
14          -> Process a d
15          -> Process a b
```

Listing 4.7: Signature of the `Parallel` data constructor.

The `Sequence` data constructor takes two processes for sequential composition. Just as for conventional function composition, the output type of the first process and the input type of the second process must coincide, as stated in definition 3.7.

```
16 Sequence :: Process a c
17          -> Process c b
18          -> Process a b
```

Listing 4.8: Signature of the `Seqeuence` data constructor.

With the `Repetition` data constructor, a process can be wrapped for repeated execution. `Repetition` takes a predicate of type `Predicate a` and a process of type `Process a a`, as defined in definition 3.8.

```
19 Repetition :: Predicate a
20            -> Process a a
21            -> Process a a
```

Listing 4.9: Signature of the `Repetition` data constructor.

In addition to the combinators introduced in chapter 3.1, the implementation contains the `Multilel` combinator. `Multilel` provides a convenient short-hand notation for the parallel composition of a list of processes instead of exactly two as is the case for the `Parallel` combinator. It takes a list of processes of type `Process a c` and a process of type `Process (a, [c]) b`. The list of processes contains the processes that should be composed in parallel, the additional process is used to combine the results of the

processes together into one value. `Multilel` does not change the expressiveness of the model, in fact it can be expressed in terms of parallel and sequential composition.

```
22 Multilel :: [Process a c]
23          -> Process (a, [c]) b
24          -> Process a b
```

Listing 4.10: Signature of the `Multilel` data constructor.

4.1.2. Process Interpreter

In this section, we discuss how the Process interpreter works in detail by taking a look at its implementation.

Processes can be executed by providing them with an input using the `runProcess` function. `runProcess` runs in the `IO` monad, it takes a process of type `Process a b` and an input for that process of matching type. The execution of a process yields a result of type `b`. `runProcess` is the implementation of the function *sem* introduced in chapter 3.3 that defines the semantics of processes.

```
1 runProcess :: Process a b -> a -> IO b
```

Listing 4.11: Signature of the process interpreter implemented using Concurrent Haskell.

The semantics of an `Id` process is as given in definition 3.11: it simply outputs its input.

```
2 runProcess Id x =
3   return x
```

Listing 4.12: Implementation of the interpreter for `Id` processes.

The semantics of an `Err` process is as stated in definition 3.12. The error process outputs the undefined value regardless of its input.

```
4 runProcess Err _ =
5   return undefined
```

Listing 4.13: Implementation of the interpreter for `Err` processes.

A `Basic` process is simply a wrapper for a computation `f` in the `IO` monad, which is the equivalent of an intrinsic function of a basic process as described in chapter 3.3. The interpretation of a `Basic` process is done by applying the process's intrinsic function `f` to the process's input `x`, as given in definition 3.9.

```
6 runProcess (Basic f) x =
7   f x
```

Listing 4.14: Implementation of the interpreter for `Basic` processes.

For the interpretation of `Choice` processes, the output of `predicate`, when supplied with input `x`, has to be obtained first by making use of the interpreter function `runProcess`. Then, based on the predicate's output, the choice between executing either `p1` or `p2` is made as defined in definition 3.15.

```
 8 runProcess (Choice predicate p1 p2) x = do
 9   b <- runProcess predicate x
10   runProcess (if b then p1 else p2) x
```

Listing 4.15: Implementation of the interpreter for `Choice` processes.

A `Parallel` process involves two process for concurrent execution, namely `p1` and `p2`, as well as a combinator process `combinator` to combine the outputs of `p1` and `p2`. In order to execute two processes at the same time, a second thread needs to be forked and the interpretation of one of the processes has to be done there. To do so, the auxiliary function `runProcessHelper` is implemented: `runProcessHelper` takes a process `p` that should be interpreted, an input `x` for `p` and an `MVar`. `runProcessHelper` determines the output of `p`, applied to `x`, by making use of `runProcess` and saves the output to the `MVar`.

```
11 runProcessHelper :: Process a b -> a -> MVar b -> IO ()
12 runProcessHelper p x mvar = putMVar mvar =<< runProcess p x
```

Listing 4.16: Auxiliary function for the interpretation of `Parallel` processes.

To interpret a `Parallel` process, an empty `MVar` is created and, together with `p1` and input `x`, passed on to `runProcessHelper` that is forked in a new thread using `forkIO`. While the output of `p1` is determined in the new thread, the interpreter determines the output of `p2` in its local thread. The previously created `MVar` is used to comfortably wait for the output of `p1` that `runProcessHelper` saves to the `MVar` once it has been obtained. Finally, `combinator` is used to combine the outputs of `p1` and `p2` into a single value, as given in definition 3.17.

```
13 runProcess (Parallel p1 p2 combinator) x = do
14   mvar <- newEmptyMVar
15   _    <- forkIO $ runProcessHelper p1 x mvar
16   r2   <- runProcess p2 x
17   r1   <- takeMVar mvar
18   runProcess combinator (r1, r2)
```

Listing 4.17: Implementation of the interpreter for `Parallel` processes.

For the interpretation of a `Sequence` process, `runProcess` is employed to obtain the output of `p1` when supplied with input `x`. Then, the output of `p1` if used as input for the interpretation of `p2` with `runProcess`, as defined in definition 3.19. The implementation of this is straightforward, using the bind operator `>>=`.

```
19 runProcess (Sequence p1 p2) x =
20   runProcess p1 x >>= runProcess p2
```

Listing 4.18: Implementation of the interpreter for `Sequence` processes.

The `Repetition` process is used to express repeated execution of a process. As stated in definition 3.21, repetition is expressed in terms of a combination of choice and sequential composition.

```
21 runProcess rep@(Repetition predicate p) x =
22   runProcess (Choice predicate (p `Sequence` rep) Id) x
```

Listing 4.19: Implementation of the interpreter for `Repetition` processes.

Finally, there is the `Multilel` process that can be expressed as a combination of `Parallel` and `Sequence` processes. The list of processes that should all be executed in parallel is transformed into a hierarchy of parallel processes, composed using the `Parallel` data constructor, by `foldr`. For the combination of the processes' results, two helper processes are employed: `emptyListP` and `consP`. `emptyListP` is a process that ignores its input and returns the empty list. `consP` is the process variant of the cons operator `(:)` and is used together with `emptyListP` to produce the list of result values. A third helper process `pairP` is used to create a pair of the original input value `x` and the list of result values from the parallel composition of processes. The final result of the `Multilel` process is produced by folding the result values together using the process `fold`.

```
23 runProcess (Multilel ps fold) x =
24   flip runProcess x $
25                foldr (Parallel consP) emptyListP ps
26     `Sequence` pairP x
27     `Sequence` fold
28   where
29     -- a process that creates a pair with first component x
30     -- and second component given by its input
31     pairP :: a -> Process b (a, b)
32     pairP x = Basic (\y -> return (x,y))
33
34     -- process variant of the cons operator on lists
35     consP :: Process (c, [c]) [c]
36     consP = Basic (return . uncurry (:))
37
38     -- a process that ignores its input and returns the empty list
39     emptyListP :: Process a [c]
40     emptyListP = Basic (return . const [])
```

Listing 4.20: Implementation of the interpreter for `Multilel` processes.

4.2. Implementation using Cloud Haskell

In the implementation for distributed parallelisation of process interpretation, we use Cloud Haskell and execute processes in Cloud Haskell's `Process` monad. The involvement of network communication requires instances of specific type classes for the types that are sent to other processes. Furthermore, the representation of basic processes needs to be adapted to the specifics of Cloud Haskell.

4.2.1. Cloud Haskell

Cloud Haskell [EBPJ11] is a domain specific language for distributed programming in Haskell. It is highly inspired by Erlang and its message passing mechanism for communication between processes, with no implicit shared memory involved.

A Cloud Haskell process is essentially a function that is evaluated in the Cloud Haskell `Process` monad and can be spawned on a local or remote node. Processes can send messages to other processes if they have knowledge about their process identifier, which serves as an address. The `Process` monad builds on top of the `IO` monad and thus, as mentioned in chapter 4.1, we cannot force processes to be free of side-effects.

While Erlang uses atoms as tags for messages, Cloud Haskell uses data types that need to have an instance of `Serializable`. `Serializable` itself is only a combination of both `Binary` and `Typeable`. `Binary` is neccessary to serialise a message into a `ByteString`, `Typeable` is used to identify the type of a value. This way, serialisation is made explicit, in contrast to Erlang where it is implicit [EBPJ11].

In Haskell, functions can only be executed, composed and passed as arguments, they cannot be serialised. However, this is necessary in order to send a function to a remote node and execute it there. Cloud Haskell avoids this problem by using fully qualified top level names of functions that are known at compile time in a table of static code pointers to refer to functions by their name. For remote execution, a function's name is put into a closure, together with its serialised environment, i.e. its argument, and sent to a remote node where it is deserialised and executed. A closure is a function (name) together with its argument [EBPJ11] and can be created using Cloud Haskell's function `mkClosure` and Template Haskell[2]. `mkClosure` takes the top level name of a 1-ary function of some type `a → Process b` and returns a process closure generator. For a function `f` with type `a → Process b`, the type of `$(mkClosure 'f)` resolves to `a → Closure (Process b)`, i.e. it is a closure generator that generates a closure when supplied with a value of type `a`.

After a closure has been executed, the result is serialised and sent back to the caller. However, the type system cannot infer the type and serialisability of the result automatically and therefore additional information needs to be provided[3]. For a type `a`, serialisation information can be provided with a value of type `Static (SerializableDict a)`. Basically, this is an explicit type tag that enables the selection of the correct serialisation function for type `a`.

[2] The function that should be executed remotely is actually combined with a matching decoder for its input.

[3] Specifically, the problem is that the type constructors can be constrained with required type class instances for parameters. On deconstruction, this information is not available and therefore needs to be added explicitly again.

4.2.2. Data Model

The data model for the representation of processes stays almost unchanged: only the definition of basic processes and the `Basic` data constructor for processes have to be adapted to meet specific requirements that arise with the use of Cloud Haskell. Basic processes are now functions that take an input of type `a` and return a closure that contains a Cloud Haskell process producing an output of type `b` when executed.

```
1 import qualified Control.Distributed.Process as CH
2
3 type BasicProcess a b = a -> CH.Closure (CH.Process b)
```

Listing 4.21: Representation of basic processes as computations in the `CH.Process` monad.

The form of the data type for the representation of processes remains unchanged.

```
4 data Process a b where
```

Listing 4.22: Data type for the representation of processes.

`Basic` reflects the adaptations that are necessary to use Cloud Haskell for remote interpretation of processes: the output type of the wrapped basic process must have an instance of `Serializable` and a static `SerializableDict` for the same type must be provided. As mentioned in chapter 4.2.1, this is necessary so remotely[4] spawned processes know how to serialise their output and send it back to the caller.

```
5 Basic :: (Serializable b)
6       => CH.Static (SerializableDict b)
7       -> BasicProcess a b
8       -> Process a b
```

Listing 4.23: Signature of the `Basic` data constructor.

A new processes type, `Local`, is added to the model. `Local` is a simple decorator [GHJV95], with the purpose of giving an indication to the process interpreter that the wrapped process should be executed locally. Typically, the reason for this arises from the expectation that serialising a closure, sending it to a remote node, executing it there and obtaining the result is more expensive[5] than executing the respective process locally. Since there is no general approach to estimate the necessary amount of time to run a process, the decision has been made to equip the developer with a tool to force local execution of a process and burden him with the obligation to make appropriate use of it.

```
 9 Local :: Process a b
10       -> Process a b
```

Listing 4.24: Signature of the additional `Local` data constructor.

The signatures of the rest of the data constructors, i.e. `Parallel`, `Sequence`, `Repetition` and `Multilel`, remain unchanged.

[4] This is true for locally spawned processes as well, as we cannot distinguish between them: Cloud Haskell transparently takes care of that we can use both local and remote processes in the same way. Technically, however, we could inspect the process identifier and tell whether the respective process is running on the local node or not.

[5] Expensive in terms of the necessary amount of time to run the process.

4.2.3. Architecture of the Distribution Infrastructure

For the interpretation of process structures in a distributed system, some kind of infrastructure and management of involved nodes is needed, so process execution can be delegated to them. In this section, we describe the architecture and functioning of our system.

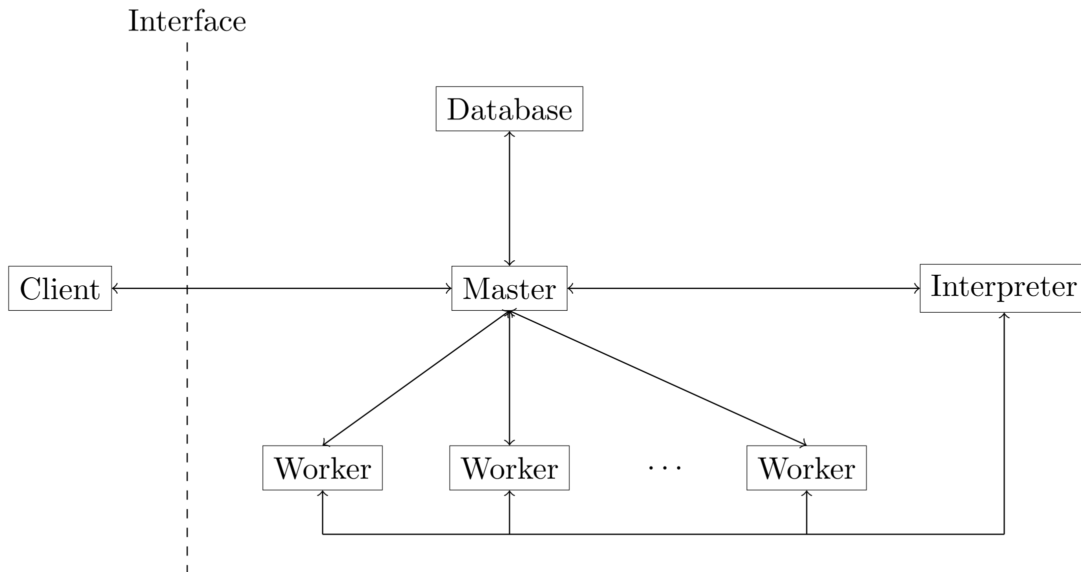


Figure 4.1.: Architecture of the distribution infrastructure.

The architecture of the system's infrastructure is intentionally kept simple and is as shown in figure 4.1. It involves a designated **master** node and a collection of **worker** nodes. The master node provides an interface through which processes can be inserted into the system by a client.

When a client wants to submit a request, he has to make use of the interface and supply a problem description in an appropriate format, e.g. a JSON[6] document. A parser reads the submitted JSON document, generates a corresponding process structure from its content and passes it on to the process interpreter if the document contains a valid process description. For simplicity, we assume in the following that a request equals an interpretable process description.

The master node is responsible for handling client requests, logging information related to the received requests and managing connected nodes. When a request is received, the master spawns a new Cloud Haskell process that runs the process interpreter and passes the request to the new process where it is interpreted. At the same time, the master assigns a ticket ID to the request, logs the request together with its ID to a database[7] and reports the ticket ID to the client. When the interpreter finishes interpretation of the process structure from the request, it reports the result to the master. The master saves the result to the database and associates it with the according ticket ID. At any time, the client can try to retrieve the result by supplying the request's ticket ID to the master

[6] JavaScript Object Notation, see `http://www.json.org/`

[7] For this purpose, the `acid-state` library is used. `acid-state` allows to save Haskell values into a file-based database if their type has an instance of `SafeCopy`. Further information on `acid-state` can be found at `http://acid-state.seize.it/` and `https://github.com/acid-state/acid-state`

through the interface. The master then tries to read the result from the database, but only replies to the client's satisfaction after the interpretation of the request has been finished and the result logged to the database.

Worker nodes are kept very simple. When started and supplied with the master's address, i.e. its process identifier, they report their availability to the master and wait for further instruction.

The process interpreter is run in a new process by the master for every client request that is received. The interpreter takes the process description from the request and distributes the incorporated sub-processes to worker nodes or executes them locally if they have been marked for local execution using the `Local` wrapper. For every basic process, the interpreter asks the master for a worker node on which it can run the basic process. When receiving a request for a worker node, the master fetches a worker node from a FIFO queue and returns it to the interpreter as soon as there is one available. After a basic process has finished execution on a worker node, the interpreter returns the respective worker node to the master for future allocation to other interpreters.

The source code of the distribution system can be found in appendix C.

4.2.4. Process Interpreter

Now that a distributed system is involved, the implementation of the process interpreter needs to be adapted to this situation. Where there was only local parallelisation in the `IO` monad in Concurrent Haskell involved before, process interpretation has to be adapted to the specifics of the Cloud Haskell `Process` monad. This particularly involves execution of processes on remote nodes.

When the master node receives a client request, it spawns a process interpreter on its local node and passes the process structure from the request to the interpreter. The interpreter then inspects the process structure and distributes the incorporated sub-processes to connected worker nodes accordingly. To do so, the interpreter asks the master node for an available worker node for every sub-process of the form `Basic` that has to be executed[8]. The master node is represented by a `ProcessId`, wrapped into a value of type `Master`.

```
1 newtype Master = Master ProcessId
```

Listing 4.25: Data type for the representation of the address of the master node.

The signature of `runProcess` has to be extended by an additional parameter: a value of type `Master` that carries information about the master node's address.

```
2 runProcess :: Master -> Process a b -> a -> CH.Process b
```

Listing 4.26: Signature of the process interpreter implemented using Cloud Haskell.

For a `Basic` process in a process structure, the process interpreter asks the master node for an available worker node on which it can run the wrapped basic process. This operation blocks until the master node is able to satisfy the request for an available worker node and supplies it to the interpreter. The interpreter then uses the closure generator `closureGen` to generate a closure that is serialised and sent to the worker node

[8] Except for processes wrapped into a `Local` process, of course.

for execution. This operation blocks until execution of the remotely spawned process has terminated and a result is obtained. The interpreter gives the worker node back to the master and returns the remotely calculated result to its caller.

```
3  runProcess master (Basic sDict closureGen) x = do
4    node <- getNode master
5    res  <- call sDict node (closureGen x)
6    returnNode master node
7    return res
```

Listing 4.27: Implementation of the interpreter for `Basic` processes.

Processes that are wrapped into a `Local` wrapper are supposed to be executed locally on the same node by the process interpreter instead of being distributed to remote nodes. This includes all sub-processes. To accomplish this behaviour, a little trick is applied: a fake master that always returns the local node when asked for an available worker node is created and used for all sub-processes of the `Local` process. When interpretation of all sub-process has finished, the fake master is terminated.

```
8  runProcess _ (Local p) x = do
9    fakeMaster <- getFakeMaster
10   res <- runProcess fakeMaster p x
11   terminateMaster fakeMaster
12   return res
```

Listing 4.28: Implementation of the interpreter for `Local` processes.

A problem arises in the interpretation of `Multilel` processes. Where it was possible to create auxiliary processes for the representation of `Multilel` processes in terms of `Parallel` and `Sequence` in the implementation based on Concurrent Haskell, this is not possible in Cloud Haskell at the moment. Processes of type `Basic` require a **specific** `SerializableDict` for the type of their output, which must be known at compile time. This prevents the use of polymorphic processes and makes a direct implementation of the parallelisation necessary, analogous to the interpretation of `Parallel` processes. For each of the processes that should be executed in parallel, `runProcessHelper` is spawned in a new thread[9], receiving an `MVar` to take up the output of the respective process. When all processes have terminated and their outputs have been obtained from the `MVars`, the process `fold` is used to combine the results together into a single value.

```
13 runProcess master (Multilel ps fold) x = do
14   mvars <- forM ps $ \_ -> liftIO newEmptyMVar
15   mapM_ (\(p, m) -> spawnLocal $ runProcessHelper master p x m)
        (ps `zip` mvars)
16   ress  <- forM mvars $ \mvar -> liftIO . takeMVar $ mvar
17   runProcess master fold (x, ress)
```

Listing 4.29: Implementation of the interpreter for `Multilel` processes.

The adaptations that have to be made for the interpretation of the remaining process types, i.e. `Choice`, `Parallel`, `Sequence` and `Repetition` are minimal. They involve adding the necessary `master` parameter and adaptations specific to the Cloud Haskell `Process` monad, i.e. using `spawnLocal` instead of `forkIO` and lifting actions on `MVars` into the Cloud Haskell `Process` monad using `liftIO`. The implementation can be found in appendix B.1 and appendix C.

[9]Spawning a process in Cloud Haskell is actually implemented using the `forkIO` functionality from Concurrent Haskell and creates a lightweight thread on a node.

4.3. Example: a Parallel Interpreter for Arithmetic Expressions

After discussing the process model and the interpreter that takes care of process interpretation, we give an example of how to apply the process calculus. To do so, we develop the typical hello world program for interpreters, i.e. an interpreter for arithmetic expressions with parallel evaluation of sub-expressions. Except for the import of our process calculus, this example is self-contained and is intentionally kept very simple. The necessity of a parallelised interpreter for arithmetic expressions is certainly not given. However, an interpreter for arithmetic expressions is fairly simple and thus well suited to illustrate how to apply the process calculus to it.

An arithmetic expression `Expr` is either a value `Val` containing an `Int` or a combination of two arithmetic expressions. The combinators for arithmetic expressions are addition `Add`, subtraction `Sub`, multiplication `Mul` and division `Div`.

```
1  data Expr = Val Int
2            | Add Expr Expr
3            | Sub Expr Expr
4            | Mul Expr Expr
5            | Div Expr Expr
```

Listing 4.30: Data model for the representation of arithmetic expressions.

The semantics of an `Expr` is a value of type `Int` and can be obtained by interpreting it. The semantics of a `Val` expression is simply its wrapped `Int` value. The semantics of a more complicated expression can be obtained by recursively interpreting the involved sub-expressions and combining the results with the appropriate operator, i.e. `+` for `Add`, `-` for `Sub`, `*` for `Mul` and `div` for `Div`.

```
6  eval :: Expr -> Int
7  eval (Val i) = i
8  eval (Add x y) = eval x + eval y
9  eval (Sub x y) = eval x - eval y
10 eval (Mul x y) = eval x * eval y
11 eval (Div x y) = eval x `div` eval y
```

Listing 4.31: Implementation of an interpreter for arithmetic expressions.

For parallel interpretation of arithmetic expressions using the process interpreter `runProcess`, the expressions need to be transformed and represented as processes.

An expression of type `Val` wraps a value of type `Int` and its semantic is exactly that of the wrapped value. The equivalent in form of a process is a process that disregards its input and returns a constant value. The function `val` creates a process like that: it takes a value of type `Int` and creates a process that always returns this value, regardless of its input.

```
12 val :: Int -> Process () Int
13 val = Basic . const . return
```

Listing 4.32: A function that generates basic processes for the representation of `Val` expressions.

Arithmetic expressions of the form `Add`, `Sub`, `Mul` and `Div` can be represented as processes by representing their sub-expressions as processes and combining them in a `Parallel` process using an appropriate combinator. For each of the arithmetic operations, a combinator process resembling the semantics of the arithmetic operation has to be defined. This is done by uncurrying the exact same arithmetic operations used in `eval` and wrapping them into basic processes.

```
14 add :: Process (Int, Int) Int
15 add = Basic (return . uncurry (+))
16
17 subtract :: Process (Int, Int) Int
18 subtract = Basic (return . uncurry (-))
19
20 multiply :: Process (Int, Int) Int
21 multiply = Basic (return . uncurry (*))
22
23 divide :: Process (Int, Int) Int
24 divide = Basic (return . uncurry div)
```

Listing 4.33: Basic processes for the combination of results from parallel interpretation of sub-expressions.

The transformation of expressions to processes is done using an interpreter [GHJV95] that takes an arithmetic expression and returns a process. The created process takes an input of type unit `()` and outputs an `Int`. Expressions of kind `Val` are simply represented by the basic process `val`. Expressions of other kind, i.e. the ones representing arithmetic operations, are represented as a parallel composition of their transformed sub-expression, combined using the matching basic process for the respective arithmetic operation.

```
25 transform :: Expr -> Process () Int
26 transform (Val i)   = val i
27 transform (Add x y) = Parallel add      (transform x) (transform y)
28 transform (Sub x y) = Parallel subtract (transform x) (transform y)
29 transform (Mul x y) = Parallel multiply (transform x) (transform y)
30 transform (Div x y) = Parallel divide   (transform x) (transform y)
```

Listing 4.34: Transformation of arithmetic expressions into processes.

Using `transform` and the process interpreter `runProcess`, arithmetic expressions can now be evaluated in parallel. To achieve this, it was necessary to represent arithmetic expressions as processes while preserving their semantics. This has been done by using the exact same arithmetic operators and wrapping them into processes. Most notably, it was not necessary to explicitly implement parallelisation by forking new threads and utilising synchronisation. By modelling processes to be executed in parallel, `runProcess` automatically takes care of the parallelisation.

The source code for this example can be found in appendix C.

5
A Real World Example

In this chapter, we use the previously developed process calculus to build a distributed program that finds solutions for the travelling salesman problem. Since the travelling salesman problem is considered to be a computationally hard optimisation problem, we do not have extended hopes in finding optimal solutions. This is a common case in practical applications. We apply the meta-heuristic approach of artificial ant systems to find approximate solutions for the travelling salesman problem.

5.1. The Travelling Salesman Problem

The travelling salesman problem (or short: TSP) is a graph theoretic optimisation problem. For a graph $G = (V, E, \delta)$ where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of nodes, $E \subseteq V \times V$ is the set of edges, i.e. the connections between the nodes, and $\delta \colon E \to \mathbb{N}$ is a function that assigns a length to every edge. The TSP asks for the **shortest** round trip through the nodes of G.

A round trip is a permutation of the nodes of G, hence every node appears exactly once in a round trip. The length of a round trip is given by the sum of the lengths of its pieces, which are defined by δ. Let $i = (i_1, i_2, \ldots, i_n)$ be a permutation of the natural numbers from $[1, n] \subset \mathbb{N}$. i defines a numbering of the nodes of G and represents a permutation of them. Let $r_i = (v_{i_1}, v_{i_2}, \ldots v_{i_n})$ be the round trip defined by the indices given by i. The length $\phi(r_i)$ of r_i is implied by δ and can be calculated by

$$\phi(r_i) = \delta((v_{i_n}, v_{i_1})) + \sum_{j=1}^{n-1} \delta\left(\left(v_{i_j}, v_{i_{j+1}}\right)\right).$$

Note that, in order to resemble the cyclic nature of a permutation, the length of the edge from the last node back to the first node must not be forgotten. This is necessary to make the path induced by the permutation a round trip.

The TSP is considered to be a computationally hard optimisation problem and belongs to the class of $\mathcal{NP}$-hard problems [GJ79]. This means that there is no known efficient

algorithm that finds the solution to an arbitrary, non-restricted instance of the TSP. Furthermore, it is widely believed that no such algorithm can exist at all. Except for some special cases where there are constraints put on the structure of the graph, the only way to find the optimal solution is to check **all** possible solutions for optimality. Since for an arbitrary graph the number of possible solutions is exponential in its number of nodes, the approach of exploring **every** possible solution is impractical even for graphs with reasonably few nodes. For an extended discussion of the TSP, see [LLKS85].

A real world application of the TSP is just as the name suggests a scenario where there is a travelling salesman who has to visit a number of houses or cities each exactly once to sell his goods and then return home. It is naturally in his interest to choose the shortest way in order to minimise travelling costs and/or time. Another application is, e.g. the problem of determining the drilling order for holes in mining operations in order to minimise the moving time for the drill.

5.2. Meta-heuristics

Meta-heuristics[1] are an approach to tackle the problem that arises with huge sets of possible solutions, which, like in the case of the TSP, is common in practical applications. The idea of meta-heuristics is to explore the set of possible solutions in a more intelligent way than checking **every** possible solution for optimality. Meta-heuristics try to make an "intelligent guess" about which solution might be close to optimal and can be based on different, problem-specific criteria. As a direct consequence of "intelligent guessing", it is very well possible to miss the optimal solution.

One thing different meta-heuristics have in common is a solution finding principle called **local search**. In **local search**, first an initial solution candidate is generated using a suitable construction algorithm. The meta-heuristic aims to improve this candidate iteratively: based on defined criteria, modifications are made to the solution candidate in the hope of finding a closer to optimal solution. The modifications yield a set of solutions similar to the solution candidate, called its **local neighbourhood**. From the local neighbourhood, the closest to optimal solution is chosen to replace the previous solution candidate [DS04].

When applying this approach, it might happen that in the local neighbourhood of a solution candidate, there is no closer to optimal solution than the candidate itself. In this case, the candidate is considered to be a so-called **local optimum**. A local optimum might be the global optimum, however in general this is highly unlikely. Therefore, if the meta-heuristic runs into such a situation, it must allow to escape local optima and execute steps towards farther from optimal solutions. As a direct consequence, the meta-heuristic must have a way to prevent from directly running back into the same local optimum as this is likely to happen in the step after proceeding towards a farther from optimal solution. A common approach to achieve this is the use of a so-called **tabu list**: a tabu list keeps track of the most recently visited solutions candidates [DS04]. As long as a candidate is in the tabu list, it must not be chosen again. Unfortunately, this does not entirely eliminate the possibility of running into the same solution again. The solution space might have cycles of higher length than the length of the tabu list, causing the meta-heuristic to return to an undesirable solution.

[1] Heuristic, from Greek *Ευρισκω*: "find" or "discover".

The term meta-heuristic is chosen because it involves heuristics to guess solutions and is applicable to a wide field of problems. With a meta-heuristic, we have a “recipe” of how to build a heuristic algorithm to solve arbitrary problems, as long as we can define the local neighbourhood of a solution.

5.3. Artificial Ant Systems

Artificial ant systems are a meta-heuristic approach that can be applied to a big variety of computationally hard optimisation problems that can be represented as a graph, like the TSP. An overview about different types of ant systems and example applications can be found in [DS04]. Here, we only briefly discuss one form of ant systems and use the developed process calculus to implement a prototype of an ant system that finds approximate solutions for instances of the TSP. Without further mentioning, [DS04] serves us as a reference throughout this chapter.

Artificial ant systems are inspired by the swarm behaviour of ants from ant colonies in nature. When in search for new food sources, natural ants explore their environment by performing random walks. Once they discover a new food source, they return to their colony and mark the way to the food source with so called *pheromones*. These pheromones can be sensed by other ants to guide their way to the food source. When an ant encounters a pheromone trail, it makes a decision between following it or continuing exploring its environment randomly. The probability of following a trail gets higher as it gets covered more densely with pheromones. If the ant decides to follow the pheromone trail, it continues up to the food source, picks up some food and carries it back to its colony. On the way back, it reinforces the pheromone trail by depositing additional pheromones. Over time, more and more ants follow the pheromone trail and an ant trail emerges. A natural phenomenon called *evaporation* counters the built-up of trails: a small fraction of pheromones disappears continuously, so the trail vanishes eventually if there are no ants left reinforcing it.

Artificial ant systems resemble the described situation from nature and employ a graph to do so. Depending on the problem at hand, one node of the graph is selected to represent the ant colony. At this special node, ants are generated to perform an exploration of the graph and construct a solution to the problem represented by the graph. In case of the TSP, a valid solution is one that represents a permutation of the graph's nodes. An ant starts its round trip through the graph at the colony, selects one of the unvisited nodes and proceeds to it. It continues doing so until it has visited all of the nodes, then it returns to the colony. It does not matter which node is selected to represent the colony since every node has to appear in the solution exactly once.

When constructing a round trip for a graph $G = (V, E, \delta)$, ants are guided by two different values. On the one hand, they use a heuristic value based on the value of an edge to estimate its desirability: for an edge $(i, j) \in E$, its heuristic value is given by

$$\eta_{i,j} = \frac{1}{\delta\left((i,j)\right)}. \tag{5.1}$$

An edge with a smaller value, i.e. a short edge, has a higher heuristic value than a long edge. For ants, higher heuristic values are desirable, i.e. short edges. On the other hand, pheromone trails τ are used. The pheromone concentration on the edge from node i to j

is denoted by $\tau_{i,j}$. In the beginning, an initial pheromone concentration τ_0 is deposited on every edge of the graph. As the ant system preforms its work, the pheromone values are updated and changed over time.

In each step, an ant picks the next node to visit non-deterministically based on a visiting probability obtained from a combination of heuristic values and pheromone values. Let $N(i)$ be the set of unvisited neighbour nodes for an ant that is currently at node i. The probability of visiting node j is given by

$$p_j = \frac{\tau_{i,j}^{\alpha}\eta_{i,j}^{\beta}}{\sum_{j\in N(i)} \tau_{i,j}^{\alpha}\eta_{i,j}^{\beta}}. \tag{5.2}$$

where α and β are values to weigh the influence of τ and η. A random value $r \in [0, 1]$ is generated and used to choose a node based on the visiting probabilities. However, to save runtime, the approach from [BI12] is applied and only the nominators from equation 5.2 are calculated. Then, instead of generating a random value $r \in [0, 1]$, a value $r' \in \left[0, \sum_{j\in N(i)} p_j\right]$ is generated and used for the selection of a node. The result is left unchanged by this modification.

As a direct consequence of how ants construct their way through the graph, they need a complete graph to operate on, otherwise they might run into situations where they get stuck in one node because there are no more reachable, unvisited nodes left from where they are. Fortunately, every graph can be extended to be a complete graph without altering the optimal solution to the TSP: Let $G = (V, E, \delta)$ be a graph. For every edge $\overline{e}$ that does not exist in the graph, i.e. $\overline{e} \in (V \times V) \setminus E$, we introduce a new edge and assign a suitably large value $\delta(\overline{e})$ to it, e.g. $\delta(\overline{e}) = 1 + \sum_{e\in E} \delta(e)$. We make sure that none of the added edges can possibly be part of an optimal solution by assigning a value to them that is guaranteed to be higher than the value of the optimal solution. For other problems, the process of adapting the graph to a suitable form may look differently.

When all ants have finished the construction of a round trip, each of them deposits new pheromones on all the edges it has used for the construction of its solution. The longer the round trip is, the less desirable it is and hence, the less pheromones should be added by that ant. Let s_i be the solution candidate, i.e. the round trip, that has been constructed by ant i and let c_{s_i} be its costs, induced by δ and summation over the used edges in s_i. Then ant i is allowed to deposit additional pheromones of $\tau_i^+ = \frac{1}{c_{s_i}}$ on every used edge and $\tau_i^+ = 0$ on unused edges. Note that there are various other ways to handle pheromone updates, many of which can be found in [DS04].

After ants have deposited new pheromones, evaporation takes place on all edges. An evaporation factor ρ specifies which fraction of the pheromones survives evaporation:

$$\tau'_{i,j} = (1 - \rho)\,\tau_{i,j}. \tag{5.3}$$

Ant systems perform their work in three phases:

1. initialisation
   initial pheromones of concentration τ_0 are deposited on all edges of the graph
2. construction
   a predefined number of ants is created, each of which constructs a solution candidate
3. update
   ants deposit new pheromones and evaporation takes place

Phase (1) takes place exactly once. The combination of phases (2) and (3) is called a *cycle*. Cycles take place repeatedly until either a predefined maximum execution time is reached or a solution of sufficient quality has been found. Other criteria to terminate the computation can be applied as well.

5.4. A Prototype of a Distributed Ant System

Having introduced the principles of ant systems, we use our process calculus to implement a distributed ant system. Most notably, we do so without specifying any explicit communication between processes, illustrating that the process calculus fulfils its purpose.

The model for the configuration of the ant system, including the previously mentioned necessary parameters, can be found in listing 5.1. The actual problem input is given by `graph`, `cycles` specifies how many cycles should be performed by the ant system, `alpha` and `beta` are used by the ants when exploring solutions and `rho` is the evaporation factor. The configuration also includes the current pheromones and the best solution to the problem that has been found so far, both kept up to date by the ant system. Furthermore, it must be possible to send the configuration over the network to ants running on other nodes in a distributed system. Therefore instances for the type classes `Generic`, `Typeable` and `Binary` are included.

```
1  import Process
2  import Graph
3  import Pheromones
4
5  type Solution = (Path, Int)
6
7  data Configuration = Configuration { graph      :: !(Graph Int)
8                                     , pheromones :: !Pheromones
9                                     , solution   :: !Solution
10                                    , cycles     :: Int
11                                    , alpha      :: Double
12                                    , beta       :: Double
13                                    , rho        :: Double
14                                    }
15   deriving (Generic, Typeable)
16
17 instance Binary Configuration where
```

Listing 5.1: Imports and configuration for the ant system.

Next, the basic processes which are later wrapped into processes are defined. The basic processes can be found in listing 5.2, their implementation details are not shown here, but can be found in appendix B.2 and appendix C. For now, only the existence of these processes and what they do is important.

```
18 ant             :: Configuration -> CH.Process Solution
19 combinePaths    :: (Configuration, [Solution])
20                 -> CH.Process Configuration
21 evaporations    :: Configuration -> CH.Process Configuration
22 cycle           :: Configuration -> CH.Process Configuration
23 continue        :: Configuration -> CH.Process Bool
24 extractSolution :: Configuration -> CH.Process Solution
```

Listing 5.2: Signatures of the basic processes for the ant system.

`ant` represents an ant that is supplied with a configuration and returns a solution candidate for the presented problem.

`combinePaths` receives a configuration and a list of candidate solutions and returns a new configuration. To do so, it takes list of solution candidates and updates the pheromone concentrations for the graph stored in the configuration and the best known solution so far.

`evaporations` takes the configuration and updates the pheromone values by reducing the pheromone concentration on every edge according to the evaporation parameter `rho`.

`cycle` decreses the value of `cycles` in the configuration by 1, signalling that one less cycle should be performed by the ant system.

`continue` receives a configuration and determines whether the ant system should perform another cycle, depending on the value of `cycles` in the configuration. If another cycle should be performed, `continue` returns `True` and `False` otherwise.

`extractSolution` is fairly simple and does not do anything else than extracting the best known solution from the configuration it is supplied with.

As discussed in chapter 4.2.1, a `SerializableDict` for every data type that should be sent over the network to another process has to be provided. This is the case for `Solution`, `Configuration` and `Bool` since values of these types are returned by the basic processes. As a brief reminder: a `SerializableDict` is basically just an explicit type tag and does not involve anything more than using a type constructor, as can be seen in listing 5.3.

```
25 boolDict :: SerializableDict Bool
26 boolDict = SerializableDict
27
28 solutionDict :: SerializableDict Solution
29 solutionDict = SerializableDict
30
31 configurationDict :: SerializableDict Configuration
32 configurationDict = SerializableDict
```

Listing 5.3: Dictionaries for data serialisation.

In order to make the basic Cloud Haskell processes and the dictionaries remotely usable, their names have to be passed to `remotable`, as shown in listing 5.4. `remotable` creates the closures for the given functions, their decoders and meta information and adds the information to a table of functions that can be used remotely.

```
33 remotable [ 'ant
34           , 'combinePaths
35           , 'evaporations
36           , 'cycle
37           , 'continue
38           , 'extractSolution
39           , 'boolDict
40           , 'solutionDict
41           , 'configurationDict
42           ]
```

Listing 5.4: Making processes and dictionaries remotable.

Next, the basic processes can be wrapped into processes using the `Basic` constructor. For that, Template Haskell is used to generate the necessary dictionaries for serialisation and the function closures. As modelled in chapter 4.2.2, the input and output type of the respective basic process determine the signature of the resulting process and the type of the necessary `SerliazableDict`. The processes are shown in listing 5.5.

```
43 antP :: Process Configuration Solution
44 antP = Basic $(mkStatic  'solutionDict)
45              $(mkClosure 'ant)
46
47 combinePathsP :: Process (Configuration, [Solution]) Configuration
48 combinePathsP = Basic $(mkStatic  'configurationDict)
49                       $(mkClosure 'combinePaths)
50
51 evaporationP :: Process Configuration Configuration
52 evaporationP = Basic $(mkStatic  'configurationDict)
53                      $(mkClosure 'evaporations)
54
55 cycleP :: Process Configuration Configuration
56 cycleP = Basic $(mkStatic  'configurationDict)
57                $(mkClosure 'cycle)
58
59 continueP :: Predicate Configuration
60 continueP = Basic $(mkStatic  'boolDict)
61                   $(mkClosure 'continue)
62
63 extractSolutionP :: Process Configuration Solution
64 extractSolutionP = Basic $(mkStatic  'solutionDict)
65                          $(mkClosure 'extractSolution)
```

Listing 5.5: Processes, built up from previously defined basic processes.

Now, all the necessary pieces to put together an ant system are prepared: the processes (listing 5.5), the data model (listing 5.1) and the process combinators from the developed process calculus. The function `mkAntSystem` takes an integer that defines how many ants should be used in the ant system and returns a processes that, when executed using the process interpreter `runProcess`, behaves like an ant system.

```
66 mkAntSystem :: Int -> Process Configuration Solution
67 mkAntSystem ants =
68   let antsP   = Multilel (replicate ants antP) (Local combinePathsP)
69       innerP  = antsP `Sequence` Local (evaporationP `Sequence` cycleP)
70       cyclesP = Repetition continueP innerP
71   in cyclesP `Sequence` Local extractSolutionP
```

Listing 5.6: Transformation of a configuration for an ant system into a process hierarchy.

The ant system constructed by `mkAntSystem` consists of a sequential composition of all the cycles that have to be executed, namely `cyclesP`, and a process that extracts the solution in the end, namely `extractSolutionP`. `cyclesP` is created using the `Repetition` constructor and is guarded by the `continueP` predicate. As long as there are cycles left that should be executed, the process `innerP` is executed. `innerP` is the sequential composition of a set of `antP` processes that all run in parallel, the evaporation process `evaporationP` and the process `cycleP` that decreases the number of cycles that are left to be executed by 1. Parallel execution of the ant processes is achieved by composing them using the `Multilel` constructor and combining their results using the `combinePathsP` process. `mkAntSystem` takes one parameter, `ants`, that specifies how many ants should

be used in the ant system. All other information, including the number of cycles to be executed, the input graph and the pheromones are provided later in form of a configuration when the ant system is executed using the process interpreter `runProcess`.

As we can see, it is possible to describe an ant system without programming parallelisation and communication between processes explicitly. The developed process calculus is employed to model the structure and interplay of processes. Parallelisation and communication between processes is added automatically by the process interpreter `runProcess`.

6 Test

In this chapter, we conduct some tests in order to assess if using our process calculus for program parallelisation can help to yield a speedup. Furthermore, we are interested in whether there is a difference between the two implementations based on Concurrent Haskell and Cloud Haskell.

6.1. Setup

For the tests, we use three variants of an artificial ant system that differ in their way of parallelisation, but implement the same algorithm otherwise. One of the variants is the distributed ant system developed in chapter 5.4, we call it A_{dis}. A_{con} is a slightly modified variant of A_{dis} and uses the implementation of the developed process calculus that is based on Concurrent Haskell instead of Cloud Haskell. The third variant, A_{seq}, performs all computations sequentially and does not involve any parallelisation. The source code of all three variants can be found in appendix C.

As input for the ant systems, we use a set of 10 instances from the TSPLIB[1] `bays29`, `berlin52`, `st70`, `pr76`, `gr96`, `eil101`, `pr107`, `pr124`, `bier127` and `ch130`. The number included in the instance names give information about their size, i.e. the number of nodes in the graph described by the instance.

We configure the ant systems using parameters based on suggestions given in [DS04]: n ants for a graph of size n, 100 cycles, initial pheromone concentration $\tau_0 = 2$, evaporation factor $\rho = 0.1$ and $\alpha = 2, \beta = 5$.

Since A_{seq}, A_{con} and A_{dis} are implementations of the same meta-heuristic and only differ in their parallelisation, we expect them to produce similar solutions in all cases. However, we are not interested in the proposed solutions but rather in the necessary time to obtain them. We use the runtime of A_{seq} as a reference and compare the runtimes of A_{con} and A_{dis} against that.

[1] The TSPLIB is a collection of instances of the TSP that have been solved to optimality and can be found at http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/.

The tests for comparing A_{con} to A_{seq} are carried out on an Intel® Core™ i7-3770 CPU @ 3.4GHz with 16GB RAM. We use `ghc` to compile the programs and the compiler flags `-O2` for A_{seq} and `-threaded -O2` for A_{con}. We run A_{con} with 1, 2, 3 and 4 cores active, respectively, using the `+RTS -N` option. Each test case is repeated 10 times.

The tests for comparing A_{dis} and A_{seq} are carried out using a network of computers with Intel® Core™ i5-2400 CPUs @ 3.1GHz and 4GM RAM. We use `ghc` to compile the programs and the compiler flags `-O2` for A_{seq} and `-threaded -O2` for A_{dis}. We run A_{dis} with the dedicated master node on one computer and with 4, 8, 16, 32 and 64 worker nodes connected, respectively. Note that, as the used CPUs have 4 physical cores, each of them can be used to run 4 worker nodes. Each test case is repeated 3 times.

6.2. Results

The results of the test runs for the comparison of A_{seq} and A_{con} are shown in table 6.1 and visualised in figure 6.1. Analogously, the results for the comparison of A_{seq} and A_{dis} are shown in table 6.2 and visualised in figure 6.2. The data shows the average runtime on each configuration.

	A_{seq}	A_{con}			
scenario		1 core	2 cores	3 cores	4 cores
`bays29`	0.45s	0.511s	0.424s	0.379s	0.375s
`berlin52`	2.253s	2.561s	2.016s	1.657s	1.521s
`st70`	5.292s	6.484s	4.911s	3.894s	3.416s
`pr76`	6.4s	8.546s	6.142s	4.867s	4.236s
`gr96`	12.855s	20.767s	13.703s	9.881s	8.209s
`eil101`	15.126s	25.893s	16.682s	11.949s	9.773s
`pr107`	16.244s	26.738s	16.482s	11.817s	10.521s
`pr124`	26.277s	50.024s	30.203s	22.587s	18.33s
`bier127`	28.468s	54.311s	33.37s	24.948s	20.225s
`ch130`	32.039s	61.328s	37.042s	27.866s	22.909s

Table 6.1.: Runtimes of A_{seq} and A_{con} on an Intel® Core™ i7-3770 CPU @ 3.4GHz with 16GB RAM.

	A_{seq}	A_{dis}				
scenario		4 worker	8 worker	16 worker	32 worker	64 worker
`bays29`	0.558s	57s	39s	28s	30s	29s
`berlin52`	2.939s	146s	97s	72s	63s	61s
`st70`	6.844s	282s	189s	137s	119s	108s
`pr76`	10.751s	355s	233s	162s	138s	128s
`gr96`	16.692s	629s	448s	306s	247s	223s
`eil101`	19.515s	771s	515s	353s	281s	252s
`pr107`	21.474s	866s	586s	407s	325s	295s
`pr124`	34.469s	1,342s	864s	603s	480s	427s
`bier127`	39.341s	1,364s	937s	647s	515s	460s
`ch130`	41.225s	1,472s	1,002s	699s	547s	495s

Table 6.2.: Runtimes of A_{seq} and A_{dis} in a network of computers with Intel® Core™ i5-2400 CPUs @ 3.1GHz and 4GM RAM.

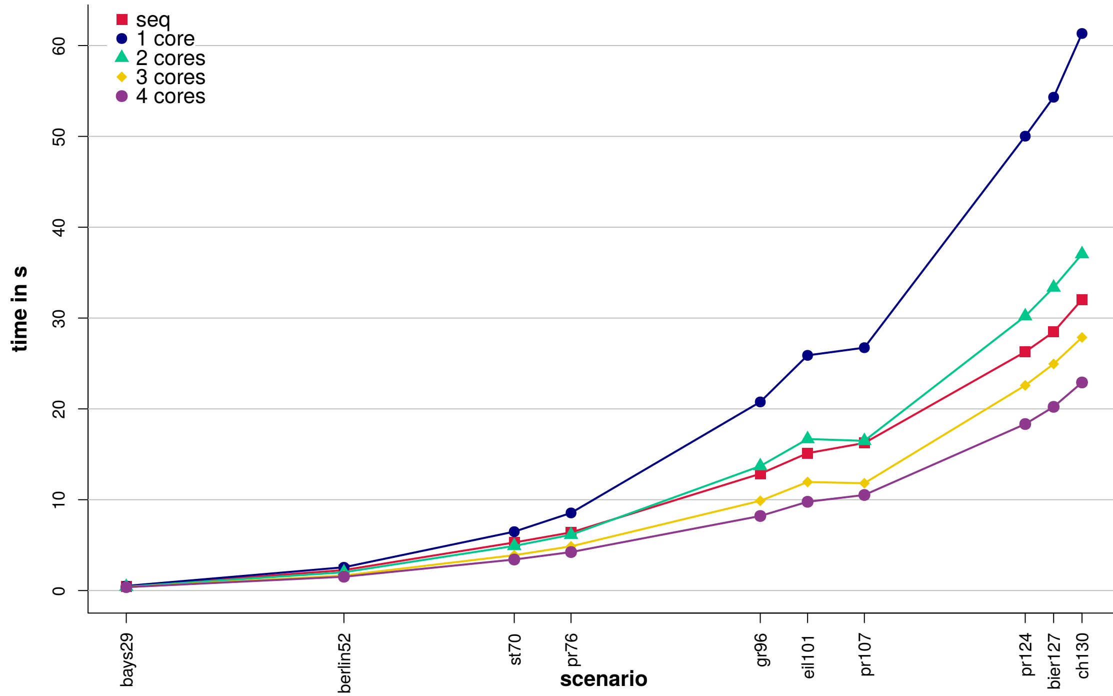


Figure 6.1.: The runtimes of A_{seq} and A_{con} based on the data from table 6.1.

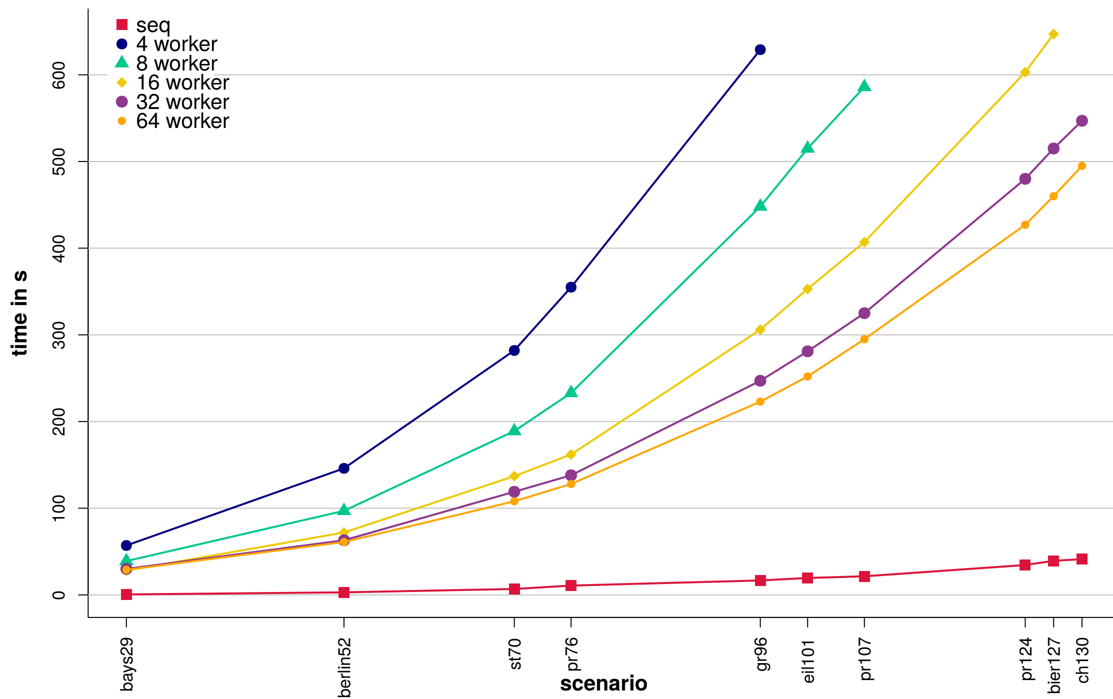


Figure 6.2.: The runtimes of A_{seq} and A_{dis} based on the data from table 6.2. Runtimes over 650s are not shown in the plot.

6.3. Interpretation

The test results show that in general, a speedup can be achieved when using the developed process calculus for program parallelisation. The runtime of A_{con} decreases as there are more physical cores made available to carry out the computations. A similar behaviour can be observed for A_{dis}: the more worker nodes are connected, the faster a solution can be computed, the system scales.

However, when put into relation to the runtime of A_{seq} on the respective hardware, A_{con} and A_{dis} show different properties. Depending on the scenario, A_{con} achieves a speedup between 1.2 and 1.57 compared to A_{seq} when 4 cores are used. Contrary to the expectation, A_{dis} is not able to achieve a speedup compared to A_{seq}, but takes between 11.69 and 51.97 times as long as A_{seq}, even with 64 worker nodes connected, depending on the scenario.

The data shows that A_{con} takes between 1.14 and 1.91 times as long as A_{seq} to find solutions when only using one core. This may be explained with the introduced overhead for expressing parallelism using our process calculus and interpreting the process structures. Depending on the scenario, A_{con} needs 2 or 3 cores to be able to outperform A_{seq}. We expect that A_{con} could perform even faster when run on more than 4 cores, however one should not forget that the sequential part of the program gives an upper bound for the maximum achievable speedup as stated by AMDAHL's law.

In our tests, A_{dis} was not able to outperform A_{seq}, no matter how many workers were connected. In order to explain this, different things may have to be taken into account. On the one hand, the process interpreter has to request a worker node for every process from the master node and wait for the reply. Since network communication takes remarkably longer than forking a local thread, a much higher overhead is introduced in A_{dis} than in A_{con}. On the other hand, for every process that is spawned remotely, a function closure has to be serialised, sent to a remote node and deserialised there. Before termination, the remote process has to serialise its result and send it back to the caller. This way of making data available to the processes takes remarkably longer than shared access to the memory as is the case in A_{con}. It is left open for future work to investigate in the reasons for this.

7 Conclusion

Our goal was the design of a process calculus that allows the description of parallel programs on a high level of abstraction while hiding technical details about parallelisation and process communication, which we have reached. We have taken a look at two prominent process calculi, namely CCS and CSP, but found that they are unsuitable for our purposes: both of them involve a notation of process communication and non-determinism, which is an undesired property for application in practice.

We have designed our own process calculus and used Haskell for a direct implementation. The process calculus introduces a small set of process combinators that are used to compose processes. Each of the process combinators involves rules concerning its static semantics. These rules have been represented in Haskell using a generalised algebraic data type that only allows composition of suitable processes. The semantics of processes has been defined inductively, based on the used process combinator and the semantics of the sub-processes. For the implementation of process semantics in Haskell, given by an interpreter, we have chosen two variants: Concurrent Haskell and Cloud Haskell. The implementation has been straightforward, directly reflecting the formal definition. We found that Haskell was particularly well suited for this.

To illustrate the usability of our process calculus, we have given two examples. One of them being a parallel interpreter for arithmetic expressions and the other one a prototype of an artificial ant system. We have created three variants of the ant system: a parallelised variant using our implementation based on Concurrent Haskell, a distributed variant using our implementation based on Cloud Haskell and a sequential variant.

Our tests have shown that the ant system using our implementation based on Concurrent Haskell achieves a speedup compared to the sequential version. The ant system using our implementation based on Cloud Haskell has not achieved a speedup but taken significantly longer than the sequential variant.

We conclude that our process calculus is mature enough for practical application and to lead to a speedup when used for parallelisation. However, our tests have shown that our implementation based on Cloud Haskell introduces a high overhead, making it impossible to achieve a speedup in its current form. For future work, it remains to be seen if this is the case in general or only in our test case.

8 Future Work

In this chapter, we discuss open research questions and ideas on implementation improvements.

In the case of ant systems, we have seen that our process calculus can be used to generate a speedup in execution. We are wondering if this is the case for other algorithms as well and how big the achievable speedup for different algorithms is. A related question is whether our distributed implementation is unsuitable for practical application in general or if it shows a different behaviour concerning the speedup in case of other algorithms.

The technical details of how processes receive their input differ between the implementations based on Concurrent Haskell and Cloud Haskell. Where in Concurrent Haskell the process interpreter can simply pass a reference to the data structure to the respective process, in Cloud Haskell this involves serialising data and sending it to a remote node. This has to be done every time a new process is spawned, even for constant data that never changes. Introducing a data distribution layer that takes care of making data available to the nodes where it is needed, should help reducing this overhead. The data distribution layer could involve a centralised data store running on the master node. Process interpreters could then submit data to the data store and receive a key that identifies the submitted data. For passing data to a process, only the key associated to the data would be needed. When a process wants to access data based on a key, the data distribution layer would take care of copying the data into a local cache on the relevant worker node, making it available for future use without the need to transmit it over the network again.

In the current implementation of the scheduling strategy for assigning worker nodes to process interpreter, the presence of a single slow computer could slow the whole system down. The master node keeps the available worker nodes in a FIFO queue. This way, every node gets to do some work eventually. However, this also means that situations are likely to occur where a worker node running on a slow computer is assigned, although a worker node running on a faster computer is available. Introducing additional information about worker nodes and employing a more sophisticated scheduling strategy should help to improve the situation.

Appendices

A
Proofs

The proofs in this appendix establish the laws proposed in chapter 3.5 as well as the theorem on process equivalence after substitution of a sub-process with an equivalent sub-process from chapter 3.4.

Theorem A.1 (Associative property of sequential composition)**.** *Sequential composition has associative property, i.e. for any three processes $P, Q, R \in \mathcal{P}$ with suitable type signatures, the following equivalence holds:*

$$(P \triangleright (Q \triangleright R)) \equiv ((P \triangleright Q) \triangleright R)\,.$$

□

Proof A.2 (Associative property of sequential composition)**.** *Let $P, Q, R \in \mathcal{P}$ be processes with type signatures $\rho(P) = (a, b)\,, \rho(Q) = (b, c)\,, \rho(R) = (c, d)$.*

Then $sem\,\langle(P \triangleright (Q \triangleright R))\rangle = sem\,\langle((P \triangleright Q) \triangleright R)\rangle$.

$$\begin{aligned}
& sem\,\langle(P \triangleright (Q \triangleright R))\rangle \\
\overset{definition\ 3.19}{=}\ & sem\,\langle(Q \triangleright R)\rangle \circ sem\,\langle P\rangle \\
\overset{definition\ 3.19}{=}\ & (sem\,\langle R\rangle \circ sem\,\langle Q\rangle) \circ sem\,\langle P\rangle \\
\overset{associativity\ of\ \circ}{=}\ & sem\,\langle R\rangle \circ (sem\,\langle Q\rangle \circ sem\,\langle P\rangle) \\
\overset{definition\ 3.19}{=}\ & sem\,\langle R\rangle \circ sem\,\langle(P \triangleright Q)\rangle \\
\overset{definition\ 3.19}{=}\ & sem\,\langle((P \triangleright Q) \triangleright R)\rangle
\end{aligned}$$

Thus, theorem A.1 is valid. ■

Theorem A.3 (Distributivity of parallel composition over choice composition)**.** *Parallel composition distributes over choice composition, i.e. for any predicate B and processes $C, P, Q, R \in \mathcal{P}$ with suitable type signatures, the following equivalence holds:*

$$(C \leftarrow P \,|\, (B \rightarrow Q \vee R)) \equiv (B \rightarrow (C \leftarrow P \,|\, Q) \vee (C \leftarrow P \,|\, R))\,.$$

□

Proof A.4 (Distributivity of parallel composition over choice composition)**.** *Let B be a predicate and let $C, P, Q, R \in \mathcal{P}$ be processes with type signatures $\rho(B) = (a, T_{Boolean})$, $\rho(C) = ((b, c), d), \rho(P) = (a, b), \rho(Q) = (a, c), \rho(R) = (a, c)$.*

Then $sem\,\langle(C \leftarrow P \,|\, (B \rightarrow Q \vee R))\rangle = sem\,\langle(B \rightarrow (C \leftarrow P \,|\, Q) \vee (C \leftarrow P \,|\, R))\rangle$.

$$\begin{aligned}
& sem\,\langle(C \leftarrow P \,|\, (B \rightarrow Q \vee R))\rangle \\
\overset{definition\ 3.17}{=}\ & x \mapsto sem\,\langle C\rangle\,(sem\,\langle P\rangle\,(x)\,, sem\,\langle(B \rightarrow Q \vee R)\rangle\,(x)) \\
\overset{definition\ 3.15}{=}\ & x \mapsto sem\,\langle C\rangle \left(sem\,\langle P\rangle\,(x)\,, x \mapsto \begin{cases} sem\,\langle Q\rangle\,(x) & \text{if } B\,(x) \\ sem\,\langle R\rangle\,(x) & \text{if } \overline{B\,(x)} \\ \bot & \text{otherwise} \end{cases} \right) \\
=\ & x \mapsto \begin{cases} sem\,\langle C\rangle\,(sem\,\langle P\rangle\,(x)\,, sem\,\langle Q\rangle\,(x)) & \text{if } B\,(x) \\ sem\,\langle C\rangle\,(sem\,\langle P\rangle\,(x)\,, sem\,\langle R\rangle\,(x)) & \text{if } \overline{B\,(x)} \\ \bot & \text{otherwise} \end{cases} \\
\overset{definition\ 3.17}{=}\ & x \mapsto \begin{cases} sem\,\langle(C \leftarrow P \,|\, Q)\rangle\,(x) & \text{if } B\,(x) \\ sem\,\langle(C \leftarrow P \,|\, R)\rangle\,(x) & \text{if } \overline{B\,(x)} \\ \bot & \text{otherwise} \end{cases} \\
\overset{definition\ 3.15}{=}\ & sem\,\langle(B \rightarrow (C \leftarrow P \,|\, Q) \vee (C \leftarrow P \,|\, R))\rangle
\end{aligned}$$

Thus, theorem A.3 is valid. ■

Theorem A.5 (Distributivity of sequential composition over parallel composition)**.** *Sequential composition distributes over parallel composition, i.e. for any processes $C, P, Q, R \in \mathcal{P}$ with suitable type signatures, the following equivalence holds:*

$$(P \triangleright (C \leftarrow Q \,|\, R)) \equiv (C \leftarrow (P \triangleright Q) \mid (P \triangleright R))\,.$$

□

Proof A.6 (Distributivity of sequential composition over parallel composition)**.** *Let $C, P, Q, R \in \mathcal{P}$ be processes with type signatures $\rho(C) = ((c, d), e), \rho(P) = (a, b)$, $\rho(Q) = (b, c), \rho(R) = (b, d)$.*

Then $sem\,\langle(P \triangleright (C \leftarrow Q \,|\, R))\rangle = sem\,\langle(C \leftarrow (P \triangleright Q) \mid (P \triangleright R))\rangle$.

$$\begin{aligned}
& sem\,\langle(P \triangleright (C \leftarrow Q \,|\, R))\rangle \\
\overset{definition\ 3.19}{=}\ & sem\,\langle(C \leftarrow Q \,|\, R)\rangle \circ sem\,\langle P\rangle \\
\overset{definition\ 3.17}{=}\ & x \mapsto sem\,\langle C\rangle\,(sem\,\langle Q\rangle\,(x)\,, sem\,\langle R\rangle\,(x)) \circ sem\,\langle P\rangle \\
\overset{function\ composition}{=}\ & x \mapsto sem\,\langle C\rangle\,(sem\,\langle Q\rangle\,(sem\,\langle P\rangle\,(x))\,, sem\,\langle R\rangle\,(sem\,\langle P\rangle\,(x))) \\
\overset{definition\ 3.19}{=}\ & x \mapsto sem\,\langle C\rangle\,(sem\,\langle(P \triangleright Q)\rangle\,(x)\,, sem\,\langle(P \triangleright R)\rangle\,(x)) \\
\overset{definition\ 3.17}{=}\ & sem\,\langle(C \leftarrow (P \triangleright Q) \mid (P \triangleright R))\rangle
\end{aligned}$$

Thus, theorem A.5 is valid. ■

Theorem A.7 (Distributivity of sequential composition over choice composition)**.** *Sequential composition distributes over choice composition, i.e. for any predicate B and processes $P, Q, R \in \mathcal{P}$ with suitable type signatures, the following equivalence holds:*

$$(P \triangleright (B \to Q \vee R)) \equiv (B \to (P \triangleright Q) \vee (P \triangleright R)).$$

□

Proof A.8 (Distributivity of sequential composition over choice composition)**.** *Let B be a predicate and let $P, Q, R \in \mathcal{P}$ be processes with type signatures $\rho(B) = (a, T_{Boolean})$, $\rho(P) = (a, b)$, $\rho(Q) = (b, c)$, $\rho(R) = (b, c)$.*

Then $sem\langle(P \triangleright (B \to Q \vee R))\rangle = sem\langle(B \to (P \triangleright Q) \vee (P \triangleright R))\rangle$.

$$\begin{aligned}
& sem\langle(P \triangleright (B \to Q \vee R))\rangle \\
\overset{definition\ 3.19}{=}\ & sem\langle(B \to Q \vee R)\rangle \circ sem\langle P\rangle \\
\overset{definition\ 3.15}{=}\ & \left(x \mapsto \begin{cases} sem\langle Q\rangle(x) & if\ B(x) \\ sem\langle R\rangle(x) & if\ \overline{B(x)} \\ \bot & otherwise \end{cases}\right) \circ sem\langle P\rangle \\
=\ & x \mapsto \begin{cases} (sem\langle Q\rangle \circ sem\langle P\rangle)(x) & if\ B(x) \\ (sem\langle R\rangle \circ sem\langle P\rangle)(x) & if\ \overline{B(x)} \\ \bot & otherwise \end{cases} \\
\overset{definition\ 3.19}{=}\ & x \mapsto \begin{cases} sem\langle(P \triangleright Q)\rangle(x) & if\ B(x) \\ sem\langle(P \triangleright R)\rangle(x) & if\ \overline{B(x)} \\ \bot & otherwise \end{cases} \\
\overset{definition\ 3.15}{=}\ & sem\langle(B \to (P \triangleright Q) \vee (P \triangleright R))\rangle
\end{aligned}$$

Thus, theorem A.7 is valid. ■

Theorem A.9 (Id is a left and right neutral element of sequential composition)**.** *Sequential composition of any process $P \in \mathcal{P}$ with the identity process Id on the left or on the right yields P.*

$$(Id \triangleright P) \equiv P \equiv (P \triangleright Id)$$

□

Proof A.10 (Id is a left and right neutral element of sequential composition)**.** *Let $P \in \mathcal{P}$ be a process and Id be the identity process. Id is a left and right neutral element of sequential composition.*

$$\begin{aligned}
& sem\langle(Id \triangleright P)\rangle \\
\overset{definition\ 3.19}{=}\ & sem\langle P\rangle \circ sem\langle Id\rangle \\
\overset{definition\ 3.11}{=}\ & sem\langle P\rangle \circ (x \mapsto x) \\
\overset{composition\ with\ identity\ function}{=}\ & sem\langle P\rangle \\
\overset{composition\ with\ identity\ function}{=}\ & (x \mapsto x) \circ sem\langle P\rangle \\
\overset{definition\ 3.11}{=}\ & sem\langle Id\rangle \circ sem\langle P\rangle \\
\overset{definition\ 3.19}{=}\ & sem\langle(P \triangleright Id)\rangle
\end{aligned}$$

Thus, theorem A.9 is valid. ■

Corollary A.11 (Idempotence of the identity process with respect to sequential composition)**.** *Since Id is both a left and right neutral element of sequential composition, Id is an idempotent element respecting sequential composition. The following equivalence follows directly from proof A.10:*

$$sem\,\langle(Id \triangleright Id)\rangle = sem\,\langle Id\rangle\,.$$

■

Theorem A.12 (Sequential composition with the error process on the right)**.** *Sequential composition of any process $P \in \mathcal{P}$ with the error process Err on the right yields the error process, i.e.*

$$(P \triangleright Err) \equiv Err.$$

□

Proof A.13 (Sequential composition with the error process on the right)**.** *Let $P \in \mathcal{P}$ be a process and let Err be the error process. Let P be sequentially composed with Err on the right, then $sem\,\langle(P \triangleright Err)\rangle = sem\,\langle Err\rangle$.*

$$\begin{aligned}
& sem\,\langle(P \triangleright Err)\rangle \\
\overset{definition\ 3.19}{=}\ & sem\,\langle P\rangle \circ sem\,\langle Err\rangle \\
\overset{definition\ 3.19}{=}\ & x \mapsto sem\,\langle Err\rangle\,(sem\,\langle P\rangle\,(x)) \\
\overset{definition\ 3.12}{=}\ & x \mapsto \bot \\
\overset{definition\ 3.12}{=}\ & sem\,\langle Err\rangle
\end{aligned}$$

Thus, theorem A.12 is valid.

■

Corollary A.14 (Idempotence of the error process with respect to sequential composition)**.** *The error process Err is an idempotent element of sequential composition, i.e. sequential composition of the error process with itself yields the error process. The following equivalence follows directly from proof A.13:*

$$(Err \triangleright Err) \equiv Err.$$

■

Theorem A.15 (Substitution with an equivalent process leaves semantics unchanged)**.** *Let $P, Q, R \in \mathcal{P}$ be processes and let $Q \equiv R$. Then, substituting one or more occurrences of Q in P with R leaves $sem\,\langle P\rangle$ unchanged:*

$$Q \equiv R \rightarrow P \equiv P_{[Q/R]}.$$

□

Proof A.16 (Substitution with an equivalent process leaves semantics unchanged)**.** *In the following, let $P, Q, R, S, T \in \mathcal{P}$ be processes with $Q \equiv R$.*

Consider the case that Q is not a sub-process of P. Then, trivially $P \equiv P_{[Q/R]}$ holds.

Now consider the case that Q is a sub-process of P. The claimed property can be shown by structural induction:

1. *$P = Q$ is a basic process with intrinsic function f_Q. Replacing Q with R gives $P_{[Q/R]} = R$ with intrinsic function f_R. Since $Q \equiv R$, it must be $f_Q = f_R$ and thus $P \equiv P_{[Q/R]}$. This establishes the basic case.*
2. *$P = (Q \triangleright S)$ has the structure of sequential composition. Then:*

$$\begin{aligned} sem\langle P\rangle &= sem\langle (Q \triangleright S)\rangle \\ &\overset{definition\ 3.19}{=} sem\langle S\rangle \circ sem\langle Q\rangle \\ &\overset{sem\langle Q\rangle = sem\langle R\rangle}{=} sem\langle S\rangle \circ sem\langle R\rangle \\ &\overset{definition\ 3.19}{=} sem\langle sequenceRS\rangle \\ &= sem\left\langle P_{[Q/R]}\right\rangle \end{aligned}$$

   And thus $P \equiv P_{[Q/R]}$.
3. *Analogously to (2), $P \equiv P_{[Q/R]}$ can be shown for $P = (S \triangleright Q)$.*
4. *$P = (Q \rightarrow S \vee T)$ has the structure of choice composition. Then:*

$$\begin{aligned} sem\langle P\rangle &= sem\langle (Q \rightarrow S \vee T)\rangle \\ &\overset{definition\ 3.15}{=} x \mapsto \begin{cases} sem\langle S\rangle (x) & \text{if } Q(x) \\ sem\langle T\rangle (x) & \text{if } \overline{Q(x)} \\ \bot & \text{otherwise} \end{cases} \\ &\overset{sem\langle Q\rangle = sem\langle R\rangle}{=} x \mapsto \begin{cases} sem\langle S\rangle (x) & \text{if } R(x) \\ sem\langle T\rangle (x) & \text{if } \overline{R(x)} \\ \bot & \text{otherwise} \end{cases} \\ &\overset{definition\ 3.15}{=} sem\langle (R \rightarrow S \vee T)\rangle \\ &= sem\left\langle P_{[Q/R]}\right\rangle \end{aligned}$$

   And thus $P \equiv P_{[Q/R]}$.
5. *$P = (S \rightarrow Q \vee T)$ has the structure of choice composition. Then:*

$$\begin{aligned} sem\langle P\rangle &= sem\langle (S \rightarrow Q \vee T)\rangle \\ &\overset{definition\ 3.15}{=} x \mapsto \begin{cases} sem\langle Q\rangle (x) & \text{if } S(x) \\ sem\langle T\rangle (x) & \text{if } \overline{S(x)} \\ \bot & \text{otherwise} \end{cases} \\ &\overset{sem\langle Q\rangle = sem\langle R\rangle}{=} x \mapsto \begin{cases} sem\langle R\rangle (x) & \text{if } S(x) \\ sem\langle T\rangle (x) & \text{if } \overline{S(x)} \\ \bot & \text{otherwise} \end{cases} \\ &\overset{definition\ 3.15}{=} sem\langle (S \rightarrow R \vee T)\rangle \\ &= sem\left\langle P_{[Q/R]}\right\rangle \end{aligned}$$

And thus $P \equiv P_{[Q/R]}$.

6. *Analogously to (5),* $P \equiv P_{[Q/R]}$ *can be shown for* $P = (S \rightarrow T \vee Q)$.
7. $P = (Q \leftarrow S \,|\, T)$ *has the structure of parallel composition. Then:*

$$\begin{aligned} sem\langle P\rangle &= sem\langle (Q \leftarrow S \,|\, T)\rangle \\ &\overset{definition\ 3.17}{=} x \mapsto sem\langle Q\rangle\left(sem\langle S\rangle(x), sem\langle T\rangle(x)\right) \\ &\overset{sem\langle Q\rangle = sem\langle R\rangle}{=} x \mapsto sem\langle R\rangle\left(sem\langle S\rangle(x), sem\langle T\rangle(x)\right) \\ &\overset{definition\ 3.17}{=} sem\langle (R \leftarrow S \,|\, T)\rangle \\ &= sem\left\langle P_{[Q/R]}\right\rangle \end{aligned}$$

And thus $P \equiv P_{[Q/R]}$.

8. $P = (S \leftarrow Q \,|\, T)$ *has the structure of parallel composition. Then:*

$$\begin{aligned} sem\langle P\rangle &= sem\langle (S \leftarrow Q \,|\, T)\rangle \\ &\overset{definition\ 3.17}{=} x \mapsto sem\langle S\rangle\left(sem\langle Q\rangle(x), sem\langle T\rangle(x)\right) \\ &\overset{sem\langle Q\rangle = sem\langle R\rangle}{=} x \mapsto sem\langle S\rangle\left(sem\langle R\rangle(x), sem\langle T\rangle(x)\right) \\ &\overset{definition\ 3.17}{=} sem\langle (S \leftarrow R \,|\, T)\rangle \\ &= sem\left\langle P_{[Q/R]}\right\rangle \end{aligned}$$

And thus $P \equiv P_{[Q/R]}$.

9. *Analogously to (8),* $P \equiv P_{[Q/R]}$ *can be shown for* $P = (S \leftarrow T \,|\, Q)$.
10. $P = (Q \propto S)$ *has the structure of repetition. Then* $P \equiv P_{[Q/R]}$ *since repetition is expressed in terms of choice and sequential composition and process equivalence for these cases has been shown in (2), (3), (4), (5), (6).*
11. $P = (S \propto Q)$ *has the structure of repetition. Then* $P \equiv P_{[Q/R]}$ *since repetition is expressed in terms of choice and sequential composition and process equivalence for these cases has been shown in (2), (3), (4), (5), (6).*

In all cases $P \equiv P_{[Q/R]}$ *holds. Thus, theorem A.15 is valid.*

■

B
Code

B.1. Implementation of the Process Interpreter using Cloud Haskell

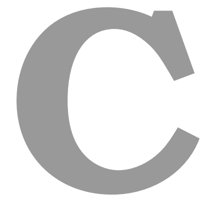
```
1  runProcess :: Master -> Process a b -> a -> CH.Process b
2  runProcess _ Id x =
3    return x
4
5  runProcess _ Err _ =
6    return undefined
7
8  runProcess master (Basic sDict closureGen) x = do
9    node <- getNode master
10   res  <- call sDict node (closureGen x)
11   returnNode master node
12   return res
13
14 runProcess _ (Local p) x = do
15   fakeMaster <- getFakeMaster
16   res <- runProcess fakeMaster p x
17   terminateMaster fakeMaster
18   return res
19
20 runProcess master (Choice pr p1 p2) x = do
21   b <- runProcess master pr x
22   runProcess master (if b then p1 else p2) x
23
24 runProcess master (Sequence p1 p2) x =
25   runProcess master p1 x >>= runProcess master p2
26
27 runProcess master (Parallel combinator p1 p2) x = do
28   mvar <- liftIO newEmptyMVar
29   _    <- spawnLocal $ runProcessHelper master p1 x mvar
30   r2   <- runProcess master p2 x
31   r1   <- liftIO $ takeMVar mvar
32   runProcess master combinator (r1, r2)
33
34 runProcess master (Multilel ps fold) x = do
35   mvars <- forM ps $ \_ -> liftIO newEmptyMVar
36   mapM_ (\(p, m) -> spawnLocal $ runProcessHelper master p x m)
        (ps `zip` mvars)
37   ress  <- forM mvars $ \mvar -> liftIO . takeMVar $ mvar
38   runProcess master fold (x, ress)
39
40 runProcess master rep@(Repetition pr p) x =
41   runProcess master (Choice pr (p `Sequence` rep) Id) x
42
43 runProcessHelper :: Master -> Process a b -> a -> MVar b -> CH.
      Process ()
44 runProcessHelper master p x mvar = do
45   r <- runProcess master p x
46   liftIO $ putMVar mvar r
```

Listing B.1: Implementation of runProcess using Cloud Haskell.

B.2. Ant System Processes

```
ant :: Configuration -> CH.Process Solution
ant Configuration {..} = do
  path <- runAnt graph pheromones [1] (nodes graph \\ [1])
  return (path, pathLength' graph path)
    where
      runAnt :: Graph Int -> Pheromones -> [Node] -> [Node] -> CH.Process Path
      runAnt _ _ visited        [] = return visited
      runAnt g p visited unvisited = do
        let tau   = distance' p (last visited)
        let eta   = (1.0/) . fromIntegral . distance' g (last visited)
        let probs = [tau u**alpha * eta u**beta | u <- unvisited]
        rand <- liftIO $ randomRIO (0, sum probs)
        let next  = fst . head . dropWhile ((< rand) . snd) $ zip unvisited (scanl1 (+) probs)
        runAnt g p (visited ++ [next]) (unvisited \\ [next])

combinePaths :: (Configuration, [Solution]) -> CH.Process Configuration
combinePaths (conf@(Configuration {..}), ss) = return conf { pheromones = pheromones', solution = solution' }
  where
    pheromones' = depositPheromones ss pheromones
    solution'   = minimumBy (compare `on` snd) (solution:ss)

evaporations :: Configuration -> CH.Process Configuration
evaporations (conf@Configuration {..}) = return conf { pheromones = evaporation rho pheromones }

cycle :: Configuration -> CH.Process Configuration
cycle conf@Configuration {..} = return conf { cycles = cycles - 1 }

continue :: Configuration -> CH.Process Bool
continue Configuration {..} = return (cycles > 0)

extractSolution :: Configuration -> CH.Process Solution
extractSolution (Configuration {..}) = return solution
```

Listing B.2: Implementation of basic processes for the ant system.



DVD with source code

List of Figures

4.1. Architecture of the distribution infrastructure. 26

6.1. The runtimes of A_{seq} and A_{con} based on the data from table 6.1. 41
6.2. The runtimes of A_{seq} and A_{dis} based on the data from table 6.2. Runtimes over 650s are not shown in the plot. 41

List of Listings

2.1. Sending data over a channel and receiving data from a channel in occam. 6
2.2. Sequential composition of processes in occam. 6
2.3. Parallel composition of processes in occam. 6
2.4. Choice between process alternatives in occam. 6
2.5. Parallel and sequential composition in Concurrent Haskell. 7
2.6. Parallel and sequential composition in Parallel Haskell. 8

4.1. Representation of basic processes as computations in the `IO` monad. . . . 19
4.2. Representation of predicates as processes. 19
4.3. Data type for the representation of processes. 19
4.4. Signatures of the `Id` and `Err` data constructors. 19
4.5. Signature of the `Basic` data constructor. 20
4.6. Signature of the `Choice` data constructor. 20
4.7. Signature of the `Parallel` data constructor. 20
4.8. Signature of the `Seqeuence` data constructor. 20
4.9. Signature of the `Repetition` data constructor. 20
4.10. Signature of the `Multilel` data constructor. 21
4.11. Signature of the process interpreter implemented using Concurrent Haskell. 21
4.12. Implementation of the interpreter for `Id` processes. 21
4.13. Implementation of the interpreter for `Err` processes. 21
4.14. Implementation of the interpreter for `Basic` processes. 21
4.15. Implementation of the interpreter for `Choice` processes. 22
4.16. Auxiliary function for the interpretation of `Parallel` processes. 22
4.17. Implementation of the interpreter for `Parallel` processes. 22
4.18. Implementation of the interpreter for `Sequence` processes. 22
4.19. Implementation of the interpreter for `Repetition` processes. 22
4.20. Implementation of the interpreter for `Multilel` processes. 23
4.21. Representation of basic processes as computations in the `CH.Process` monad. 25
4.22. Data type for the representation of processes. 25
4.23. Signature of the `Basic` data constructor. 25
4.24. Signature of the additional `Local` data constructor. 25
4.25. Data type for the representation of the address of the master node. . . . 27
4.26. Signature of the process interpreter implemented using Cloud Haskell. . . 27
4.27. Implementation of the interpreter for `Basic` processes. 28
4.28. Implementation of the interpreter for `Local` processes. 28
4.29. Implementation of the interpreter for `Multilel` processes. 28
4.30. Data model for the representation of arithmetic expressions. 29
4.31. Implementation of an interpreter for arithmetic expressions. 29
4.32. A function that generates basic processes for the representation of `Val` expressions. 29

4.33. Basic processes for the combination of results from parallel interpretation of sub-expressions. 30
4.34. Transformation of arithmetic expressions into processes. 30

5.1. Imports and configuration for the ant system. 35
5.2. Signatures of the basic processes for the ant system. 35
5.3. Dictionaries for data serialisation. 36
5.4. Making processes and dictionaries remotable. 36
5.5. Processes, built up from previously defined basic processes. 37
5.6. Transformation of a configuration for an ant system into a process hierarchy. 37

B.1. Implementation of `runProcess` using Cloud Haskell. 53
B.2. Implementation of basic processes for the ant system. 54

Bibliography

[BI12] Christopher Blöcker and Sebastian Iwanowski. Utilising an ant system for a competitive real-life planning scenario. IARIA, 2012.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell (2nd Edition)*. Prentice Hall, 1998.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[EBPJ11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[HvS12] Tony Hoare and Stephan van Staden. The laws of programming unify process calculi. In *Proceedings of the 11th International Conference on Mathematics of Program Construction*, MPC'12, pages 7–22, Berlin, Heidelberg, 2012. Springer-Verlag.

[LLKS85] E.L. Lawler, J.K. Lenstra, A.H.G.R. Kan, and D.B. Shmoys. *The Traveling Salesman Problem*. Wiley Interscience Series in Discrete Mathematics. John Wiley & Sons, 1985.

[Mar13] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'REILLY, 2013.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

Index

A
Algorithm . . . 32
Ant System . . . 33, 35
Architecture . . . 26
Arithmetic Expression . . . 29
Associativity . . . 17

C
Calculus . . . 9, 10
Calculus of Communicating Systems . . 4
Candidate . . . 32
CCS . . . 4
Client . . . 26, 27
Closure . . . 24
Cloud Haskell . . . 7, 18, 24, 27
Combinator . . . 4
Communicating Sequential Processes . 3
Conclusion . . . 43
Concurrent Haskell . . . 7, 18, 19
CSP . . . 3
Cycle . . . 35

D
Data constructor
- Basic . . . 19, 25
- Choice . . . 20
- Err . . . 19
- Id . . . 19
- Local . . . 25
- Multilel . . . 20, 25
- Parallel . . . 20, 25
- Repetition . . . 20, 25
- Sequence . . . 20, 25

Data Model . . . 19, 25
Decorator . . . 25
Distributivity . . . 17
Domain Theory . . . 15

E
Environment . . . 24
Erlang . . . 24
Error Process . . . 12
Evaporation . . . 33
Event . . . 4
Example . . . 29, 31

F
Function . . . 9, 24
Future Work . . . 44

G
Generalised Algebraic Data Type 19, 43
GHC . . . 18
Graph . . . 33
Graph Theory . . . 1, 31

H
Haskell . . . 7, 18, 24
- Cloud Haskell . . . 18, 24
- Concurrent Haskell . . . 7, 18, 19
- Parallel Haskell . . . 8

Hoare, C.A.R. . . . 3

I
Idempotence . . . 17
Identity Process . . . 12
Implementation . . . 18
Infrastructure . . . 26
Input . . . 9, 10
Interface . . . 26
Interpreter . . . 2, 26, 27, 29
Intrinsic Function . . . 12, 21
Introduction . . . 1
Inverse Process . . . 16

J
JSON . . . 26

L
Laws . . . 17
- Associativity . . . 17
- Distributivity . . . 17
- Idempotence . . . 17
- Neutral Element . . . 17

Least Fixed Point Semantics . . . 15

Local search 32

M

Message passing 24
Meta-heuristic 31–33
Milner, Robin 4

N

Neighbourhood 32
Neutral Element 17
Node
- Local 24
- Master 26, 27
- Remote 24
- Worker 26, 27

Non-determinism 4, 6, 9

O

Occam 6
Optimisation Problem 31
Optimum
- Global 32
- Local 32

Output 9, 10

P

Parallel Haskell 7, 8
Permutation 31
Pheromone 33
Predicate 11, 13, 19
Process 9
- Basic 2, 10, 12, 19
- Calculus 3, 9
- Combinator 10
- Composed 10, 13
- Equivalence 16
- Error 10, 12
- Identity 10, 12
- Inverse 16
- Substitution 16

Process Interpreter 21, 27
- Basic 21, 27
- Choice 21, 28
- Err 21
- Id 21
- Local 28
- Multilel 23, 28
- Parallel 22, 28
- Repetition 22, 28
- Sequence 22, 28

Prototype 33, 35

R

Request 26, 27
Round Trip 31, 33

S

Semantic Equivalence 16
Semantics 9, 12
- Basic Process 12
- Choice 13
- Error Process 12
- Identity Process 12
- Parallel 13
- Repetition 15
- Sequence 14
- Static 10

Serialisation 24
Side-effect 9, 19
State 9
Static Semantics
- Choice 11
- Parallel 11
- Repetition 11
- Sequence 11

Sub-process 13, 27, 28
Substitution 16
Syntax 9
- Choice 9
- Parallel 9
- Repetition 9
- Sequence 9

T

Tabu List 32
Test 39
Travelling Salesman Problem 31
TSP 31, 33
TSPLIB 39
Type 10
- Parameter 19
- Signature 10
- Variable 10

U

Undefined Value 10, 12

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum

Christopher Blöcker