AALBORG UNIVERSITY
STUDENT REPORT

TLDR

THE LANGUAGE DESCRIBED IN THIS REPORT

Project Group
SW404F15

Supervisor
THOMAS BØGHOLM

May 27th, 2015

This report was typeset using LaTeX.

AALBORG UNIVERSITY
STUDENT REPORT

Software - 4th semester
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
`http://www.cs.aau.dk/`

Title
TLDR - The Language Described in this Report

Theme
Design, Definition and Implementation of Programming Languages

Project Period
Spring Semester 2015

Project Group
sw404f15

Participants
Alexander Dalsgaard Krog
Christian Heider Nielsen
Jens Hegner Stærmose
Kasper Fuglsang Christensen
Kasper Kohsel Terndrup
Simon Vandel Sillesen

Supervisor
Thomas Bøgholm

Copies
2

Number of Pages
109

This report documents the creative process and general reasoning behind The Language Described in this Report, a high-level, modular, functional first, domain specific programming language intended for natural and social scientists to model objects and interactions between objects in concurrent real-world systems.

This page intentionally left (almost) blank.

Contents

List of Figures

List of Tables

1 Preface

This report was created by, and is a result of the work of, group sw404f15 at Aalborg University.
The intellectual property rights to all original material, brought forth in this report, belong to the authors named below.
Figures, data and other resources gathered from third party sources will be referenced according to the IEEE standard.
The software solution discussed in this report is open source, under the GNU GPLv3 license, and is freely available at `https://github.com/simonvandel/TLDR`.

The authors of this report would like to thank Thomas Bøgholm, for his supervision during the process of making this report.

Aalborg University, May 27th, 2015

Alexander Dalsgaard Krog
<akrog13@student.aau.dk>

Christian Heider Nielsen
<chrini13@student.aau.dk>

Jens Hegner Stærmose
<jstarm13@student.aau.dk>

Kasper Fuglsang Christensen
<kfch13@student.aau.dk>

Kasper Kohsel Terndrup
<kternd13@student.aau.dk>

Simon Vandel Sillesen
<ssille13@student.aau.dk>

This page intentionally left (almost) blank.

2 Introduction

This report covers the creation of The Language Described in this Report (TLDR), a programming language useful for creating simulations of real world systems.. The language is constructed with scientists, who wish to compute large simulations in different fields of science, as the primary user group. Large simulations refer to problems which are deemed infeasible to be run on regular computers such as laptops and desktops. Focusing on such large simulations effectively gives the language a subfocus on concurrent computing.

In the field of supercomputing, languages such as C and Fortran are heavily used. Although these languages are very powerful and efficient programming languages, they are beginning to show their age and both lack a lot of prominent functionalities, such as pattern matching, for-each loops and arbitrary precision numerals.

Instead TLDR tries to provide the computing power of a supercomputer to scientists, who do not have the extended experience with computer science that is required to take full advantage of concurrency and parallelism.

The Language Described in this Report, aims to modernise programming supercomputers and cluster computers.

This premise gives the following problem statement:

How can one design and implement a programming language which supports a programmer in modeling arbitrary scientific simulations, in a way that inherently encourages concurrent processing.

3 Problem Domain

In this chapter, the problem domain will be explained and analysed, in order to provide a better understanding of the intentions with The Language Described in this Report, and the reasoning upon which choices and delimitations will be made. The chapter will briefly cover reasons for modelling and different approaches to modelling a real world scenario. Following this is a section on simulations in general, and specific ways of doing theoretical simulations. This also addresses some problems with the creation of computer simulations, and how such a problem can be viewed.

3.1 Models

Model is a term often used in both natural and social sciences. Unfortunately the term is perhaps not as well defined as it should be, and it will therefore be specified here. The approach here is based on Mario Bunge's definition of a model [1]. According to his theory, a model consists of two parts:

- A general theory
- A special description of an object or system

This perception is a very general way of thinking of models and often works very well with natural and, to some degree, social sciences where experiments and models are based on an overall theory and consists of objects and systems. In social sciences, though, it often happens that the general theory is vaguely defined or even non-existent and the purpose of the model is only to observe interactions and behaviour of the systems and their objects. To extend Bunge's theory, models will here be defined as "a set of assumptions about a system" meaning everything we know about a system that we can observe or theorise about. Generally there are two different types of models, static and dynamic models.

3.1.1 Dynamic Models

Dynamic models include some form of evolution, meaning that the system changes due to some changing factor. This is often time, but can also be things like energy, as is often the case in chemistry, the only important thing is that the factor changes. Nearly all systems in natural and social sciences are dynamic models.

3.1.2 Static Models

A static model is a snapshot of a given system with objects with non-changing states. These systems are often not very representative of reality, since nearly all physical systems change, but they can still be very useful to construct, since it can be much easier to collect information and understanding through a static model.

3.2 Simulation

We often need to create and analyse models when trying to get a better understanding of a given situation. For a portion of these models, an analytical approach can be used, where one can analyse the model via mathematical methods and calculate how static models will look and dynamic models evolve. This approach only works on a limited portion of the models, where the system and behaviour of objects are well known and can be described mathematically. As the models become more stochastic, purely mathematical analysis begins to fall short and simulations are often used to be able to create the models.

Formally, a simulation is "an imitation of a process through another process" [2]. This is a very broad definition, since real world systems can be very different in their structure and the types of simulations also differ greatly. Generally one can distinguish between two different types of simulations, experimental and theoretical. An experimental simulation is where a real system is imitated by another real system, often smaller and/or simpler. For instance, when biologists simulate the creation of life on the back of crystals in their lab it is an experimental simulation. Theoretical simulations are simulations where a real system is imitated theoretically often on a computer or on paper.

This project will focus on theoretical simulations. To further narrow the definition of a simulation used in this project, the focus will mainly be put on natural and social sciences where simulations are most often used. Within these sciences, real systems can still vary a lot, but they nearly always share the property that they change state over, what is often, but not necessarily, time. This is close to the definition of a dynamic model. In this respect simulations are close to a dynamic model. How this change is modelled can also vary and generally we often speak of two different paradigms; discrete and continuous simulations.

3.2.1 Continuous Simulations

Continuous simulations are simulations that are continuously evaluated. A continuous simulation is in many ways the most direct representation of reality, where actions and reactions happen based on each other and the environment they are in. It can be expressed as a differential equation. Just as with differential equations, a high precision is obtainable, but extracting results can prove to be cumbersome. But continuous simulations suffer in places were a granularity is hard to determine. Imagine, as an example, a simulation of light reflected in a solar system. If we were to conduct an experimental simulation of the light in a solar system, we could have a regular lightbulb represent the sun, and observe how different levels of power to the bulb changes the reflections in the model. But if we were to give a true accounting of the amount of photons a given star emits, the amount of calculations in a theoretical simulation would quickly grow out of reach for even the greatest supercomputer. This is a reason why continuous simulations can be problematic when implemented as a computer simulation.

3.2.2 Discrete Simulations

Discrete simulations are simulations, where the state of the system is updated in discrete iterations. Observations in discrete simulations do not make sense during update but only before/after iterations. A discrete simulation tends to be more useful when certain abstractions can be made about the simulations. In the above mentioned example with the solar system, one might have observed not precisely when photons are emitted, but instead that every minute a certain number of photons are emitted on average. This could be modelled as a single emission of light every minute. This makes the calculation a lot simpler, but at the price of a loss of precision. Whenever a problem is discretisied, there will be a loss of detail.

Granularity

When creating a model for a theoretical simulation, there is usually a certain degree of information lost during modelling, due to either the process of isolation from the environment, or an omittance of details deemed insignificant. In the example of modelling the light reflected in a solar system, an example of isolation could be the omittance of light beeing emitted from other stars than the one residing in the solarsystem. In the same example, an omittance of details could be the interference of satellites orbiting a planet, effectively blocking some light from reaching the planet. Since the amount of light that is blocked by a satelite is miniscule in comparison, it could be deemed too insignificant to include this in the calculation, since it will likely not have any observable effect on the simulation. The choice of deciding the amount of details included, is known as choosing granularity.

Granularity, from the point of view of computer science, is a qualitative measure of the ratio of computation to communication. The granularities should be chosen, accordingly with the consideration of this ratio. Periods of computation are typically separated from periods of communication by synchronisation events.

One can say that *fine-grain parallelism* is:

- Relatively small amounts of computational work is done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronisation between tasks takes longer than the computation.

and *coarse-grain parallelism* is:

- Relatively large amounts of computational work is done between communication/synchronisation events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently

Decomposition

Decompositioning within computer science, is breaking a problem into discrete "chunks" of work that can be distributed as multiple tasks, for the computer to compute. In decompositioning the problem to multiple tasks, the designer of the solution can make use of two basic concepts: to partition the tasks in such a way that it can be parallelised by domain or functional decomposition.

Domain Decomposition is distributing similar tasks of a problem among multiple different processors. *Figure 3.1* illustrates multiple similar tasks that can be parallelised, each working on their own domain of the problem set.

This is useful when working with data traversals of sets of data, if the operations of each iteration is independent of each other, the traversal can easily be split up into multiple parallel tasks.

Functional Decomposition focuses on the operations performed rather than the data manipulated by the computation. Some tasks differ from others in that they manipulate different data of the problem domain. This is illustrated in *fig. 3.2*.

The fact that we can decompose the problems into multiple independent tasks means that we make use of the performance provided in high performance computing (HPC) computers. The systems used in the field of HPC, relies on their many processor units to gain tremendous computational speed as opposed to the regular high throughput computing (HTC) computers, that ensures a reliable throughput instead.

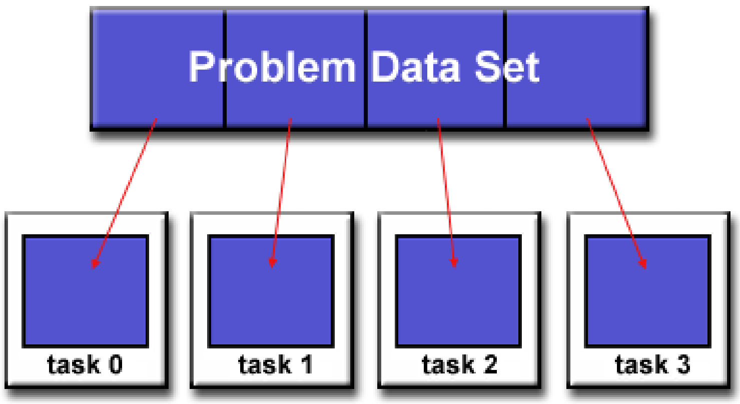


Figure 3.1: *Illustration of domain decompositioning. The problem data set is partitioned in chunks and then operated on by multiple tasks running in parallel. [3]*

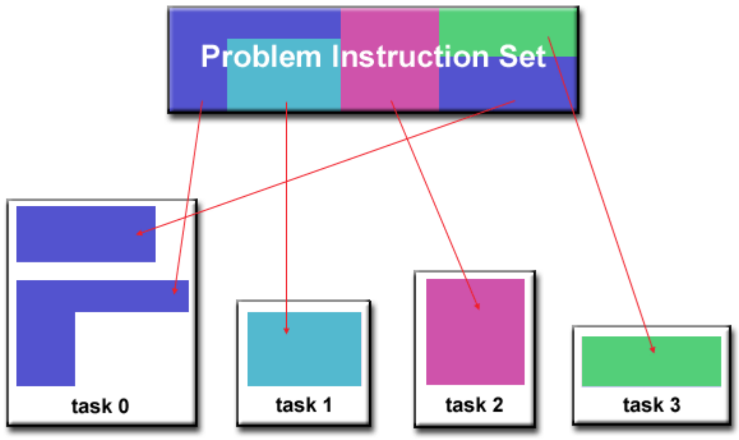


Figure 3.2: *Illustration of functional decompositioning in which each task performs operations on data in parallel. Each task has a predetermined operation to perform. [3]*

4 Conditions for a Simulation Language

In this chapter, we will examine certain technological tools for handling problems within the domain of computer simulations. This will include a section on high performance computing (HPC), which will cover the systems that typically compute simulations. Right after the HPC section will be a section on parallelism, based on theory from computer science. This section on parallelism covers the attractiveness of parallelism, as well as the problems which can occur in a parallel system. Then a section on the different memory architectures used in parallelism, and a section on the usage of the Message Passing Interface (MPI).

This all precedes a section on the actor model, and how this model can be used to do simulations, and where problems can arise with it.

4.1 High Performance Computing

High performance computing (from here on, HPC) is a field of computing, where the main objective is performance, often measured in floating point operations per second (FLOPS). The term HPC is typically associated with supercomputers and large distributed systems.

HPC systems are rarely used by the "average Joe", but more so by scientists and engineers who need to perform computations on very large data sets, model reality with a high level detail or calculate very large equations. It is this type of computing that TLDR, will focus on.

4.1.1 Supercomputers and Distributed Systems

Nearly all HPC systems are either supercomputers or distributed systems. Contemporary supercomputers are getting more and more powerful, with the current world leader, the Chinese Tianhe-2, having a theoretical peak performance of almost 55 petaFLOPS [4].

Distributed systems are generally a large group, called a "cluster", consisting of computers, referred to as "nodes". In these systems, the individual nodes might not be any more powerful than an ordinary desktop computer, but the collective processing power of the whole cluster can rival that of a supercomputer.

A simple cluster could consist of the following nodes:

Head The controller of the cluster. This is where the clusters interface is, and where the jobs that should be performed are first introduced to the system.

Compute A generic worker. A compute node receives a job and returns the result of the computation. In a normal cluster, you would have many compute nodes.

Scheduler A node with the sole purpose of sending jobs to compute nodes. A good scheduler will evenly distribute the load of jobs to the compute nodes, minimising the total computation time.

These systems' performance is high, mainly due to their extensive use of parallel computing. This parallelism is achieved, in the case of Tianhe-2, by using MPICH2, one of the most popular implementations of the MPI specification for message passing. This requires the programmer to consider parallelism when writing the programs.

4.1.2 Current Technology

Even though the hardware used in HPC systems evolve at a very high rate, unfortunately the same cannot be said for the programming languages used to program the systems. The programming languages almost solely used to program HPC systems are C and Fortran. At the time of writing, C is 43 years old and Fortran is 58 years old. For any technology related to software to survive this long is quite a feat, but also rather worrying considering its heavy use in HPC, which should be the milestone for efficient programming.

So why is such a critical performance-oriented field still dominated by technologies which are about half a century old? There are three very big factors in why these languages are dominant:

Tried and true Both C and Fortran have existed for such a long time, that both have had the time needed to mature. Every feature in the languages is almost guaranteed to be correct, as the languages have been extensively used and developed for such a long time.

Support Nearly all general computational problems have been solved in C and Fortran, meaning that for almost all trivial problems, someone else has already done all the work and you can merely use their code.

Near to the metal As C and Fortran are rather low level languages, there is a large amount of control in the hands of the developer. This means that there is a lot of potential for optimisation in the code.

These are all good features for a language, and especially being near to the metal becomes a desirable trait, when one is trying to increase the speed of the computation.

However, this trait also provides a barrier of entry for scientists, who are not well versed in programming. This is due to multiple problems, one being a low readability and another being low writeability, both due to the languages requiring you to consider problems which solely exist in the domain of computers, and not in the domain of the problem that was originally sought to be solved. The languages are also not accommodating towards modelling of simulations, due to their imperative paradigm. It is expected that some scientists would be willing to sacrifice some of the control and speed of the C and Fortran languages, in order to better understand the code they are running, and thereby be able to write new programs faster.

In light of this, TLDR will be focused on the areas of programming where C and Fortran fall short, and there will not be as great a focus on the areas of their strengths.

4.2 Parallelism

In the continued effort of trying to squeeze as much power out of computers as possible, the computer scientific community is at a point where increasing the clock speed of processors is no longer as viable a solution as it used to be. This has spawned an increased interest in increasing computational power in other ways. One of these ways is parallelism.

Parallelism is the act of dividing calculations into independently solvable parts, and then solving them on multiple processors before finally gathering the individual results and combining them. The main benefit of parallelising anything is to gain *speed up* in the computation time of problems. This will be described in *section 4.2.1*.

With parallelism you gain *speed up* through combining multiple processing units. This is seen in newer CPU's as multiple cores, but on a larger scale this principle can be used to create supercomputers, capable of performing immense calculations.

But even without a supercomputer, a distributed network of multiple computers can provide large amounts of parallel computing power. With this being a foreseeable future, we predict an increase in the access to, and need for, parallel systems.

4.2.1 Speed Up

For parallel computing, Gene Amdahl, an american computer scientist, defined a law for describing the theoretical maximum speed up using multiple processors. The maximum speed up a system can achieve by parallelisation is limited to at most 20×. The law can be used to describe the speed up achievable by a giving system, by the percentage of parallelisable code, in this equation [5].

The time an algorithm takes to finish with $n \in \mathbb{N}$ execution units and $B \in [0,1]$ the fraction of the algorithm that is strictly serial.

$$T(n) = T(1)(B + \frac{1}{n}(1 - B)) \tag{4.1}$$

The theoretical speed up to be made can then be described by:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)(B + \frac{1}{n}(1 - B))} = \frac{1}{B + \frac{1}{n}(1 - B)} \tag{4.2}$$

By this law, maximising the percentage of parallelisable, the highest possible speed up achievable is increased, thereby improving the scaling of the solution on multiple processor.

4.2.2 Types of Tasks

Tasks within a problem can be dependent on each other, in the sense that one task needs the output of a computation done by another task. This section will describe two types of problems relevant when doing parallel computations [3], [6].

Embarrassingly Parallel Problems

A problem can be describe as being *embarrassingly parallel* when the tasks within the problem are independent of each other, and can therefore be parallelised without any concern of the order in which the tasks can be executed. They are trivially parallel in nature because of the independency. An example of this type of problem is incrementation of a large matrix, the individual cells in the matrix are completely independent from each other and can therefore be incremented without regard of other cells in the matrix.

Serial Problems with Dependencies

Although multiple similar simulations can be observed as being independent of each other, as utilised by the Monte Carlo method, most simulations do not satify the condition of being independent. Instead these are inherently sequential. They form a class of problems that cannot be split into independent sub-problems. In some cases it is not possible to gain speed up at all, by trying to parallelise a problem, which is not parallelisable. The only thing that a simulation designer can achieve in this case, is to add overhead to the computations. An example is calculating the fibonacci series by f(n) = f(n-1)+f(n-2), where f(n) is dependent on first finding the previous values of f.

4.2.3 Parallel Data Access

When trying to solve problems that are not embarrasingly parallel, with a parallel approach, some problems arise when multiple processes try to access the same memory. Among these the most common are race conditions and deadlocks.

Race Conditions

This problem arises when multiple processes want to modify the same data in memory or a similar resource, and the outcome of the modification depends on the internal timing of processes. We call the resource "the shared resource" and the code which works with the shared resource "the critical region". An example of a race condition is two concurrent processes that want to raise the value of an integer by 1. In a normal modern day processor the process could be split into the three atomic operations[1]:

- Copy the current value of the integer from main memory into register A
- Calculate the value from register A, add 1 to it and place the result in register B
- Take the new value from register B and override the integer in memory

Since we do not know when each process will try to access the memory, the value can either be raised by one, if both processes access it before the other overrides it, or raised by two, if one process finished before the other copy the value from memory. This is a well known problem and exists in many situations, where multiple processes work with non-atomic operations on the same memory. Some software solutions, that ensure only one instance of the critical region has permission to access the shared resource at the time, have been developed; especially Gary L. Peterson's algorithm, published in 1981, is used today. Other solutions have also been developed by creating atomic assembly instructions that can set a flag, thereby ensuring that only one critical region access the shared resource at the time.

Deadlocks

Deadlocks are a type of problem that occur when two or more processes are waiting for at least one of the others to finish, before it itself finishes, thereby never finishing. This is a common problem within concurrent programming. Edward G. Coffman described four conditions that must be present if a deadlock is to occur [7]:

[1]Atomic operations are operations which the hardware manufacturer ensures happen without disruptions and that cannot be split into smaller operations

Mutual Exclusion Resources can not be shared simultaneously by two processes.
Hold and Wait A process is at some point holding one resource, while waiting for another.
No Preemption A resources can not be released externally from a process.
Circular Wait A process must be waiting for a resource which is held by another process, which is waiting for the first process to release a resource.

By knowing these four conditions, we can try prevent deadlocks by making sure at least one of the conditions is not present.

4.3 Memory Architectures

This section describes two memory architectures commonly used in parallel computing. Based on the characteristics of the architectures, a target architecture is chosen for TLDR.

4.3.1 Shared Memory

In this memory architecture, all processors in the computer have access to the same memory, that is, the memory address space is global. In addition the memory is in close proximity to the processors, providing lower access latency. This approach of having unified memory is very simplistic and eases the programmer's view of the memory layout. Because of this sharing of memory, a processor A can affect memory used by processor B. This leads to non-deterministic programs where a processor can not be sure what state the memory is in at a given time, without synchronisation between processors [3].

To summarise on the above, the architecture has the following advantages and disadvantages.

Advantages

- Simple unified memory model leads to simple understanding of the memory layout
- Fast access to memory

Disadvantages

- Sharing of memory between processes leads to non-deterministic processes if synchronisation between processes is not employed

4.3.2 Distributed Memory

In this memory architecture, processors have their own local memory, and it is not possible to directly access memory between processors. For two processors to share memory, the data must be sent via a connection. A typical connection method is Ethernet. A system of processors is called a distributed system once connected through a network. Depending on the proximity of processors and the bandwidth of the connection method, the speed and latency may be sub-par compared to the shared memory architecture, described above [3].

To summarise, the distributed architecture has the following advantages and disadvantages.

Advantages

- It is easy to add new processors to the network, since each processor only known about its own memory. The network therefore scales well

Disadvantages

- It is more complex to access data on another processor, and data located on a remote processor is slower to retrieve

4.3.3 Architecture Choice

TLDR is targeted at scientists wanting to perform simulations. Because of the need to perform these simulations quickly, even without super-computers, there should be support for connecting several computers in a network to create a distributed system. The distributed memory architecture is therefore chosen as the preferred architecture choice for this programming language.

4.4 Message Passing Interface

Message Passing Interface (MPI) is the de facto standard for message passing, when doing high performance computation. It was first released in 1994 as version 1.0. It was created in collaboration between 40 different organisations involving about 60 people in total. It was based in a preliminary proposal, known as MPI1, that was created by Jack Dongarra, Rolf Hempel, Tony Hey, and David Walker and finished in November 1992 and inspired from the, then, state of the art practice, such as PVM, NX, Express, p4, etc. It has since been iterated upon, with the latest version, MPI-3.0 being released in 2012.

The reason for creating this standard is to ensure portability and ease-of-use. In a communicating distributed memory environment, high-level routines and abstractions are often build upon lower level message passing routines, of which this standardisation ensures that these abstractions and routines are available on all systems. This also presents vendors with a few clearly defined base-routines they need to implement efficiently and, if possible, provide hardware support for, to enhance scalability.

MPI is designed to allow efficient communication between processes, use in heterogeneous environments and ensure thread-safety. The semantics of the interface itself should be independent from any language it is implemented in, however it does focus on providing convenient bindings for the C and Fortran languages. MPI also specifies how communication failures are dealt with, in the underlying systems.

The standard is designed for message passing only and does not include specifications for explicit operations on shared memory. Operations, that go beyond the current operating system support, are not included. These include interrupt-driven receives, remote execution and active messages, since the interface is not intended for kernel programming/embedded systems. Neither does it include support for debugging, explicit support for threads or task management nor support for I/O functions [8].

4.4.1 Message Types

Point-to-point Operations

Point-to-point communication is the the basic building block of message passing.

Send Here one process sends data in form of a message to a single other process.

Receive When a process receives a message, it enqueues the message in a queue called its message box. Each message in the message box is processed sequentially by dequeuing and handling them one at a time [8].

Collective Operations

Broadcast Broadcast is a one-to-many operation, where one process has some specific data that it sends to many other processes. The data is therefore multiplied, as opposed to being divided.

Scatter Scatter is a close relative to broadcast. It is also a one-to-many operation, but here the data is divided into equally large chunks and is distributed among multiple processes (including the process originally containing all the data). This could for instance be sending all entries in a collection to different processes that individually process their part.

Gather Gather is a many-to-one operation and is the inverse of scatter. Here data from many processes are sent to one of them. This operation often implies a hierarchy of the processes containing the data, where the process highest in the hierarchy receives all the data (also from itself).

Reduce Reduce is a many-to-one operation. Here one operation, called the reduction function, is done on data from multiple processes and the result is placed in one of them. As with gather, a hierarchy is also customary and the process highest in the hierarchy receives the result of the reduction. All reduction functions must be both associative and commutative, so results can be reduced without concern for the order of operations [8].

System Buffer

Consider the following: When sending messages, what happens if the receiver is not ready to process them? To solve this problem, MPI dictates that an implementation must be able to store messages, however the way this is done is up to the implementer.

One way to do this is with a system buffer. In short terms, a system buffer works as an inbox, and sometimes also an outbox, where messages are stored until they can be processed. A few things to note about this inbox, is that it is supposed to work "behind the scenes", not manageable by the programmer. However, what the programmer needs to realise, is that this buffer is a finite resource, which will be exhausted if one is not cautious [9].

Blocking and Non-blocking Sends

Messages can be divided into two distinct groups, the blocking sends and the non-blocking sends. The straight forward approach is the non-blocking send, where the sender assumes, or is certain, that the receiver is ready to handle the message. These types of messages return almost immediately, but there is no guarantee that the message was received, or how it was received.

On the other hand there is the blocking send. This only returns after it is safe to modify the application buffer again. The sent data could still be sitting in a buffer, but it is considered safe.

Generally, blocking sends are used for cases where it is crucial that the communication has been completed, and non-blocking sends are used to increase performance in cases where the communication is not crucial [9].

Order and Fairness

When sending messages, the order in which they are sent and received can matter a great deal. MPI gives a few guarantees as to how this can be expected to happen. One must note that these rules are not enforced if multiple threads are participating in the communication.

When using MPI, messages will not overtake each other. If one task sends out two messages in succession, the message sent out first, will be the first one to be received. Also, if a receiver has multiple requests looking for a matching message, the request that was made first, will get the first match.

MPI does not give any guarantees related to fairness. It is entirely possible to starve tasks. If this is not desired, the responsibility lies with the programmer [9].

4.5 The Actor Model

As described in *section 4.2* most simulations are very heavy and need to be computed on a HPC system to operate at resonable speeds. Furthermore HPC systems are parallel in nature, utilising the performance of multiple processors. On modern architectures parallelism is achieved via system threads. Threads are one of multiple ways to optain concurrency, but they are obligatory when trying to achieve parallelism, at least in current modern usage. As described in *section 3.1*, models are used as the conceptional still images of simulations and general dynamic behaviour. We therefore need a way to create models that can operate in parallel. Threads are not fitting to use to abstract over the definition of models. Since this language focuses heavily on models and simulations a more compatible abstraction is needed.

If we look at the definition given earlier about models, they consist of a general theory and a description of objects and systems. This is a definition is very similar to how object oriented programming works, where all programs consist of objects, with fields and methods, and systems are build around these. It is not surprising that this is a property of object oriented programming, since it has been designed to model the real world. Because of this it would be very beneficial to use a concurrent abstraction that has the same base where everything is objects. The actor model is often described as the object oriented concurrency model. The rest of this section will describe why this is the case, but because of its similarity to object oriented thinking, the actor model was found to be the best concurrency model for simulations and therefore also TLDR.

The actor model is a concurrency model, which first appeared in 1973 in the report "A Universal Modular ACTOR Formalism for Artificial Intelligence" by Carl Hewitt, Peter Bishop and Richard Steiger [10]. This section will explain the model generally, be compared to alternative concurrency models and a few implementations of the model will be briefly described.

4.5.1 Model

The actor model is focused on the fact that any computational behaviour, be it functions, processes or data structures, can be modelled as a single behaviour; sending messages to actors. An actor in this context is a process, which has its own isolated state, used to store values and modify its own behaviour, and a message box, for receiving messages. An actor can only share information by sending messages, asynchronously, to other actors. Actors are very similar to the communicating sequential processes (CSP) described in C.A.R. Hoare's publication "Communicating Sequential Processes" [11]. Actors, though, have a few well-defined behaviours in response to receiving a message, where CSPs are free to exhibit other behaviour. An actor can, in response to a message:

1. Send a finite number of messages to other actors.
2. Spawn a finite number of new actors.
3. Change its own behaviour for the next message that is received.

In figure 4.1, a model of an actor based program can be seen.

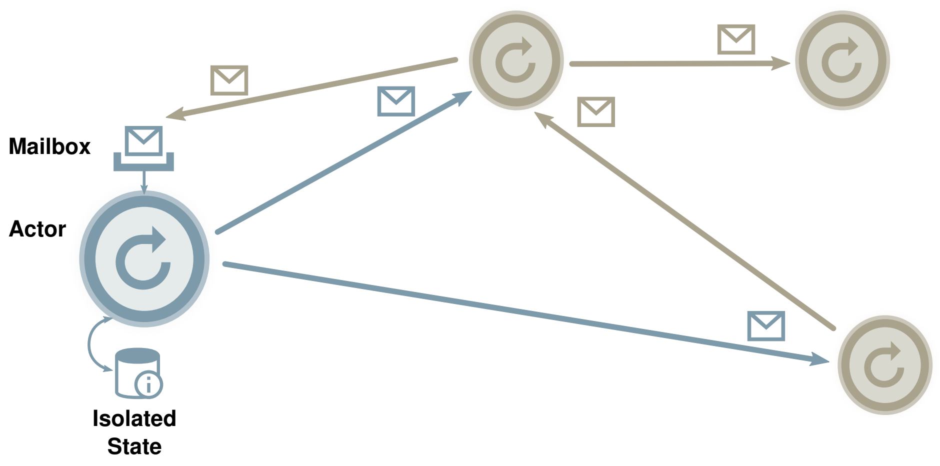


Figure 4.1: *A model of an example program using actors. The nodes in this graph represent actors and the edges represent messages being sent.*

Theoretical Basis

As mentioned, actors are very similar to, and have their theoretical roots in CSP. In CSP, a program is composed of processes that operate independently from each other. These processes share information by messages passing. In Hoare's original version of CSP, each process was defined as a sequential series of instructions, however later versions of CSP typically allow a process' logic to consist of other processes.

While many aspects of CSP and the actor model are identical, there are some differences. In CSP, all processes are anonymous, while in the actor model, actors have identity. Sending a message to an actor can be regarded as largely equivalent to invoking a method on an object in the object-oriented world view. Since CSP does not have named processes, communication is done via named channels, as seen in programming languages such as Go and OCaml. In CSP, communication is synchronous, i.e. the receiver of a message must be ready to receive the message, before it can be sent by the sender. In the Actor model, all communication is asynchronous, meaning a message can be sent, without considering whether the receiving end is ready or not.

Adding Logic

One of the main features of the actor model is, as Hoare describes it, the ability to add knowledge to a system, without rewriting the existing knowledge. This means that if a developer is interested in adding a logical step in the business logic of his system, this can be achieved by merely adding an actor, which can perform the logical operations, that are to be added, and then adding that actor as a link in the flow of logic.

Example We have a system which can read a CSV-file and print it to stdout. This can be implemented by letting *Reader* and *Printer* be actors.

Step	Reader	Printer
1	**Read CSV-file**	*Wait for message*
2	**Send data to Printer**	*Wait for message*
3	*Wait for message*	**Receive data from Reader**
4	*Wait for message*	**Print data to stdout**

Table 4.1: *A simple example showing two actors reading a file and printing the contents, respectively.*

At some point we decide that we wish to format our data before printing it. To do so, we create and incorporate a new actor, *Formatter*, which changes the flow of execution in this manner:

Notice that in adding the new logic, the only existing logic that was changed is the actor whom *Reader* sends data to.

Step	Reader	Formatter	Printer
1	**Read CSV-file**	*Wait for message*	*Wait for message*
2	**Send data to Formatter**	*Wait for message*	*Wait for message*
3	*Wait for message*	**Receive data**	*Wait for message*
4	*Wait for message*	**Format data**	*Wait for message*
5	*Wait for message*	**Send data to Printer**	*Wait for message*
6	*Wait for message*	*Wait for message*	**Receive data**
7	*Wait for message*	*Wait for message*	**Print data to stdout**

Table 4.2: *An example of extending the logic of* section 4.5.1 *by also formatting the contents of the file, using another actor.*

Concurrency

The actor model is, as previously stated, inherently concurrent.

Isolated State As actors only communicate via message passing, each actor can have its own isolated state. This means that no two actors share any data, meaning problems such as race conditions, starvation and deadlocks are avoided entirely. These problems would lead to programmers having to deal with the administration of serialising access to the data, by means of locks, mutexes, semaphores or similar concurrency control mechanisms.

Separated Logic In the actor model, each actor has its own subprogram, which is only known by the actor itself and can be run in an infinite loop. A consequence of this is that an arbitrary actor never depends on another actor, meaning an actor's logic is isolated from the rest of the system and can be regarded as an atomic concurrent part of the system.

Example To demonstrate the actor model's ability to deal with classical concurrency problems, an example will be shown here. This example, shows the problem of phantom reads, which can be a big problem in data-rich, highly parallel systems. In the example, a collection is being accessed by two concurrent processes, or threads. One tries to read the collection, and the other tries to delete it.

We start off, with the problem shown in a typically imperative way, using threads. See *table 4.3.*

As seen in this example, thread B is able to delete the collection, while thread A is still using it. This means that once thread A tries to access the second element of the collection, it runs into the phantom read problem and would likely terminate and throw an exception.

Step	Thread A	Thread B
1	**Read first element in collection**	*No operation*
2	**Process element**	**Delete collection**
3	**Read second element in collection**	*No operation*

Table 4.3: *An example of a phantom read using two threads.*

To demonstrate the same functionality that these threads are performing, using actors, we need three actors, *Holder*, *Reader* and *Deleter*. *Holder* contains the collection, *Reader* reads and processes the elements in the collection and *Deleter* wants to delete the collection.

Step	Holder	Reader	Deleter
1	*Wait for message*	**Send request to *Holder***	*Wait for message*
2	**Send collection to *Reader***	*Wait for message*	**Send “delete” to *Holder***
3	**Delete collection**	**Read first element in collection**	*Wait for message*
4	*Wait for message*	**Read second element in collection**	*Wait for message*
		...	
n	*Wait for message*	**Read nth element in collection**	*Wait for message*

Table 4.4: *An example of avoiding a phantom read by using actors.*

As can be seen in *table 4.4*, using actors to model this system ensures that there are no phantom reads. This is ensured, by the fact that actors never share data; therefore, an arbitrary actor can never corrupt, or delete, the data of another.

Supervisors

Most implementations of the actor model implement a form of a supervisor-worker relationship. This means that every actor has a supervisor, typically their parent, to whom they report failures. The supervisor then has to deal with the failure, by for example restarting the failed actor. Supervisors in the actor model works very well with the “fail fast”-principle of programming, meaning, in the actor model, that an actor will not try to continue operation or correct an error when encountering a failure. An actor that has failed will, as soon as the failure is detected, report the failure to its supervisor and halt operation, moving the responsibility of handling the failure up the supervision tree.

4.5.2 Implementations

TLDR is not the first language to implement the actor model. Two popular implementations, a mature one and a more current one, will be described here, to provide insight in the typical syntax and semantics of actors.

Erlang

When talking about the actor model, one cannot escape the subject of Erlang. Erlang was one of first languages to fully incorporate the actor model as the main model of concurrency. Erlang is a purely functional language, developed for the telecommunications industry, where a lot of concurrent processes have to be handled.

Actors in Erlang, or processes as they are called, are controlled by a few very simple constructs; actors can be spawned using the built-in spawn-function, the spawn-function takes three arguments, the module in which the actor is contained, the function that defines the behaviour for the actor and an initial message. You can send messages to actors using the bang-operator (!) and you can define an actor's behaviour in a receive-block.

In Erlang, an actor's parent is it's supervisor, if one chooses to use the supervisor module available. Using the supervisor module in Erlang gives the programmer the ability to choose different strategies to be used when a child actor fails. These strategies are:

1. Respawn the failed child
2. Respawn all children when one fails
3. Respawn all children after the child in the start order of the children

Example To demonstrate the use of actors in Erlang, a simple example is shown in listing 4.1. This program has only one actor, which counts the number of "incr"-messages it has sent.

Listing 4.1: A simple message-counter in Erlang.

```
-module(countMsgs).
-export([run/0, counter/1]).

run() ->
  S = spawn(countMsgs, counter, [0]), %spawn S as counter-actor
  sendMsgs(S, 10000), %send 10000 messages to S
  S.

counter(Sum) -> %function-definition for counter-actor
  receive
    value -> io:fwrite("Value is ~w~n", [Sum]);
    incr -> counter(Sum+1)
  end.

sendMsgs(_, 0) -> true; %base case
sendMsgs(S, N) -> %recursive function to send N messages to
   actor S
  S ! incr, %send incr to S
  sendMsgs(S, N-1).
```

Akka

A more modern approach to the actor model is taken in Akka, a concurrency toolkit for Scala and Java. In Akka, an actor is just an object, which extends Akka's Actor-class and implements a receive method. One difference in the receive-method from Erlang's receive-block is that Akka's receive-method is exhaustive, meaning that the programmer has to define behaviour for all possible messages that an actor can receive.

Creating an actor in Akka is done by calling the *actorOf*-method, on an actor or an instance of *ActorSystem*, which acts as the parent of top-level actors. The *actorOf*-method takes a *Probs* as an argument, *Probs* is just an object containing properties for an actor, such as the definition of the actor. Just like Erlang, messages are sent using the bang-operator.

In Akka, supervisors have the ability to handle failures in child actors in the following manner:

1. Ignore failure and resume operation
2. Restart the child from initial state
3. Terminate the child, without respawning it
4. Fail (move failure up supervision tree)

Example To demonstrate Akka's implementation of actors, the same example shown in listing 4.1 will be implemented in Scala, using Akka, in listing 4.2.

Listing 4.2: A simple message-counter in Scala.

```

import akka.actor.Actor
import akka.actor.Probs

class Counter extends Actor{ //definition of counter-actor
  var count = 0
  def receive = {
    case "incr" => count += 1 //if actor receives "incr",
       increment count
    case "value" => println(s"Value is $count")
    case other => println(s"Error: Cannot understand message
       $other") //default case
  }
}

object Main extends App {
  val mySystem = ActorSystem("MySystem")
  val myCounter = mySystem.actorOf(Probs[Counter], name="
     MyCounter") //create counter-actor as top-level actor
  def main(args:Array[String]){
    for (i <- 1 to 10000) { //send 10000 messages to myCounter
      myCounter ! "incr"
    }
  }
}
```

4.5.3 The Iteration Problem

The actor model makes it possible to create continuous simulations in the language, with a good abstraction for simulations and models in general. Unfortunately, the actor model does not support discrete simulations as well. When trying to implement a discrete simulation solely with actors, a problem arises that needs to be addressed.

The problem is that there is no way for the programmer in the language to tell whether a group of actors has stopped working or not. This is important since the programmer must to be able to implement iteration steps if he or she is to create discrete simulations. The programmer has to be able to find out when one step has ended to update the state of the whole system and start a new iteration. Due to the parallel nature of the actor model an unfortunate variant of race conditions arise.

For an actor to have stopped, the programmer needs to know that three following conditions are fulfilled:

- The actor is not currently executing any messages
- The actor's message queue is currently empty
- The actor will not receive any further messages

The two first conditions are rather trivial to check. But for the last condition to hold, one must make sure that all other actors that know about this actor, also have stopped. Otherwise they can send messages to the actor and thereby "reviving" it. This problem can be solved in various ways, but a few conditions are necessary for any solution:

- The programmer must be able to assure which actors can communicate with each other. Without this knowledge, it is impossible to know whether or not a given actor will receive future messages.
- The actors must keep track of and report their status.

To illustrate the solution we now look at the first condition. To keep track of which actors can communicate with other actors, there must be some way to group actors together. If we look to discrete simulations, each iteration illustrates an update for all objects and subsystems within a closed system. The initial idea was that one could create actors within actors.

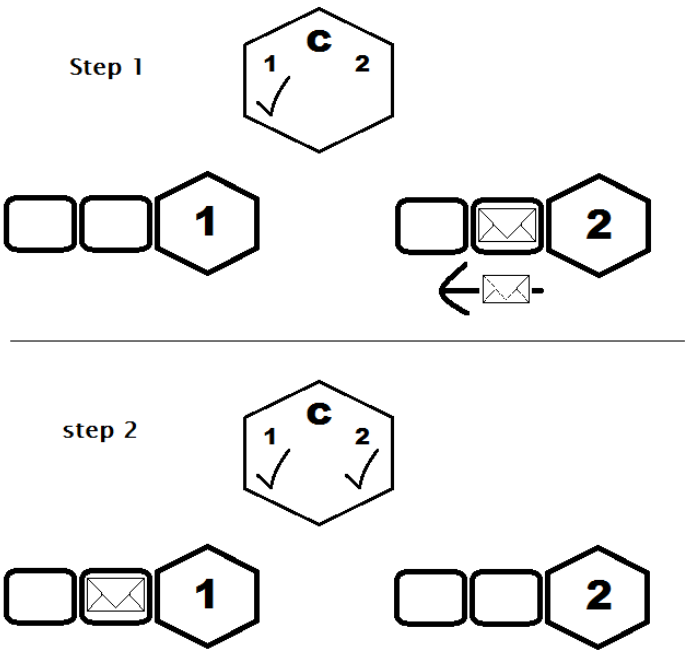
The problem with this solution is that actors can see symbols declared in the same scope as themselves. This means that if they are declared within another actor they can also see the symbols declared within that actor, thereby violating one of the principles of MPI and defying the definition of an actor. There could be made a special case for this, but this could result in complicated rules that can be hard to remember; especially for people that are not used to programming.

A possible solution to this would be to introduce what we will refer to as environments. An environment is a construct used to group actors and block communication to other actors outside of the environment. Like actors, environments can be created but only exist if they are instantiated by an actor. When an environment is instantiated the actor that created it can start an iteration with the keyword “go” and a message. The programmer is then able to create receive-methods to correspond to these messages along with actors, structs, functions and other environments within the environment. Actors are also able to instantiate actors and environments created within their own environment and can contain functions, variables, constants, structs and recieve-methods.

Instantiating an environment is a blocking action meaning that the actor instantiating the environment does not continue until the iteration is done, how this is determinated will be described later in this section. No environment, nor its actors, is allowed to receive messages from any actor outside the environment, except from the “go”-keyword.

This creates a tree structure of iterations where one iteration can spawn more environments inside it with iterations, meaning that the outer iteration cannot end before its inner iterations are completed, since the actor spawning them is not done yet. This enables the programmer to solve all the computations as simulations within simulations, mixing different layers of continuous and discrete simulations, as needed. This means that the programmer now has iterations with isolated actors that upholds MPI, thereby enabling both continuous and discrete simulations.

Now we look at the second condition; how actors should keep track of and report their status. The naïve solution would be to have all the actors report when they have no messages and when they are not running, and then reporting to a controller, assuming that when the controller has received a message from all actors, that they will be done running. The problem with this approach can be illustrated with the figure *section 4.5.3*.



Step	Actor 1	Actor 2
1	Has no message and is not running, so it sends a message to the controller	Evaluates a message, which causes it to send a message to actor 1
2	Receives the message from actor 2 and starts running	Has no message and is not running, so it sends a message to the controller

Now the controller will have received all reports, and will proceed with the next iteration, even though actor 1 is still running. Even if actor 1 was to send a message to the controller, revoking the original message, it could arrive too late, which would not give the security wanted by the iteration feature.

During the design of the language, two solutions where considered.

Pause and Check

This solution is the direct solution to the naïve approach. It solves the problem through a pause. Whenever the controller has received checks from all actors in the group, it will force an interrupt on the actors. This will allow the controller to go through the actors, directly checking whether their messagebox is empty or they are currently running. This solution however imposes a potential for inefficiency, especially in systems with many actors, which will cause the controller runthrough to become a significant overhead.

Semaphore

This solution builds on the idea of a central controlling semaphore shared by the group of actors. Whenever a message is sent, the semaphore is incremented, and whenever a message has been evaluated the semaphore is decremented. Since the increment happens before the decrement, it should effectively only become lower if an actor sent fewer messages than it received. This solution removes the overhead of checking actors, and deals with the problem of the race condition. However it requires a critical region with a semaphore, which could also potentially provide a significant slowdown by bottlenecking actors at the semaphore, even after optimisations are made. This solution was preferred due to the lack of directly halting running actors.

Delimitation

For this current project, built in support for discrete simulations will not be implemented due to time constraints. Even so, efforts have been made to ease a future implementation, in order to keep the language true to its goal of supporting both continuous and discrete simulations.

4.5.4 Summary

The actor model is a model for concurrency formalised by Carl Hewitt, based on the work of C.A.R. Hoare. The actor model provides an abstraction over concurrent processes, where locks, mutexes, semaphores and the like are unnecessary. The actor model's abstraction also solves some of the classical problems with concurrency, such as phantom reads. The actor model integrates very well with functional languages, as seen in *section 4.5.2*. Based on these observations and with confidence that the iteration problem can be solved, the actor model is chosen as the concurrency model for TLDR.

4.6 Criteria for Language Design

This section will, in light of the preceding two chapters, *chapters 3 and 4*, settle on some criteria for TLDR. The criteria are inspired by those found in [12]. However, only those characteristics which differentiate TLDR will be discussed, even though other characteristics will be present in the language.

4.6.1 Simplicity, Orthogonality and Syntax Design

In order to accommodate a user group with programming as a secondary skill set, TLDR should be simple. This would result in strict and straightforward rules regarding interactions. The language should try to keep simple rules regarding orthogonality as well, but not necessarily be highly orthogonal for that reason. The syntax design, should for these reasons, cater towards a mathematical perspective rather than a computer science perspective.

4.6.2 Data Types

Since the users of TLDR will typically have a background in mathematics, or require the use of some mathematics in order to asses the value of any given result of a simulation, the language should strive towards having data types which allow for representation of traditional mathematical numbers.

4.6.3 Type Checking and Exception Handling

Due to the language targeting high performance computation, the language should try to avoid run-time errors, and catch problems as early as possible. This suggests a strongly typed language, but this will be elaborated further on in *section 5.1.1*. For the same reasons, the language will not value exception handling, since edge cases should be fully encompassed in a simulation. However it is worth noticing that the actor model could require exception handling, if communication problems, caused by either race conditions or network conditions, should prove frequent.

4.6.4 Readability over Writability

TLDR values readability over writability, since the problems which the languages aims to support are usually big complex mathematical problems, which quickly can confuse the programmers, if they are not able to understand the code. The size of the problems will also likely popularise the use of external libraries, which will also benefit from readability, since an understanding of the functionalities supplied by a given library is useful when exploring which library best solves the problem.

5 Language

This chapter will describe the design of TLDR and explain the reasoning behind the choices made during the design of the language.

The chapter is split into four parts. The first part is the general properties and formal models. This includes the type system, general semantics and scope rules.

After this, three sections will cover the specifics of the language. The seperation is between expressions, statements, and actors. Even though actors are also statements, their importance for the characteristics of the language, was deemed worthy of a seperate section.

The different sections will contain explanations, which will consist of both formal and informal syntax, semantics and type rules of the construct and why this was chosen. Some constructs will merit more discussion and explanation than others.

The syntactical constructs will be formally presented with EBNF, but it will be kept close to BNF where EBNF will not provide further clarification.

To describe the semantics, the formal notation will follow the example set by [13] for small step semantics. The choice of small step semantics is done, since big step semantics can not describe parallelism, and a uniform use of semantics was prefered.

The formal semantic rules have the following convention of what specific symbols are used for:

- $n \in Num$ - Numerals
- $x \in Sym$ - Symbols
- $a \in Aexp$ - Arithmetic expressions
- $b \in Bexp$ - Boolean expressions
- $S \in Stm$ - Statements

The formal rules have three parts as illustrated in *fig. 5.1*

$$RULE-NAME\frac{[PREMISE]}{[CONCLUSION]}, [SIDE-CONDITION]$$

Figure 5.1: *A description of formal small-step semantics*

The conclusion always consists of a transition arrow, "⇒". On the left hand side, the initial code that the rule transitions from and the needed transition functions in their current state is placed inside angle brackets, for example $\langle S, sEnv, env_a\rangle$. On the right hand side of the transition arrow is the code that this can transition to and the possible updates in the transition functions inside angle brackets, for example $\langle S', sEnv', env'_a \rangle$. If any symbol is marked with an apostrophe, it has the possibility to be something different within the same set. This must be defined in either the premise or side condition. The premise is how something else must transition before the conclusion can be met. The side condition is other things that are side effects of the premise. The mapping of the transition functions are put inside square brackets, for instance "$sEnv = [x \mapsto \underline{4}, y \mapsto \underline{5}]$" denotes that $sEnv$ is the transition function where x maps to the numeral $\underline{4}$ and y to $\underline{5}$. A transition function can also be expanded using square brackets, for instance we can make a new transition function $sEnv'$ that expands on $sEnv$ so that it also maps z to the numeral $\underline{6}$; "$sEnv' = sEnv[z \mapsto \underline{6}]$".

5.1 General Language Properties and Formal Models

This section will describe the type system of TLDR, the scoping rules and some fundamental semantics for the language.

5.1.1 Type System

This section first defines what a type system is and presents two categories of type systems. The two type system categories are described and its properties are outlined.

Later in this section, the type system decisions made for TLDR are described, and formal type rules are given.

The goal of the type system is to minimise programming errors, while on the other hand not being too restrictive, such that the expressiveness of the language is not reduced.

Strictness

Type systems are often described as *strong* or *weak*. No formal definitions of such categorisations of type systems exist, so this report will use the following understanding of *weak* and *strong* type systems. A type system goes from *weak* to *strong* as the amount of undefined behaviour, unpredictable behaviour or implicit conversions between types approach zero.

Static Versus Dynamic Type Systems

Languages can generally be categorised into dynamically typed and statically typed languages. The difference between static and dynamic typing is the time of type checking. In static typing, types are checked at compile time. In dynamic typing, types are checked at run-time.

Statically typed languages have the following advantages over dynamically typed languages.

- Type errors are presented at the soonest possible time; compile-time. This makes it impossible to have a program stop unexpectedly because of a type error at run-time
- Because of the fact that types are checked at compile-time, no overhead is imposed on the compiled program at run-time

Of course dynamically typed languages have their uses. One can argue that prototyping a program is done faster in a dynamically typed language because the programmer does not need to think of types when programming. The mental overhead is usually lower. The trade-off of dynamic typing versus static typing is often expressiveness for safety and performance.

Type Inference

Type checking solves the problem of determining if a program is well-typed i.e. does not have any type errors. Type inference can be seen as solving the opposite problem; determine types in a program, thus making the program well-typed.

In order to reduce the mental overhead and the number of explicit type declarations to write in a language, type inference can be implemented in the compiler. Having types inferred by the compiler in a statically typed language makes the language as expressive as a dynamically typed language, whilst keeping the safety and performance of static typing.

Summary

A type system is defined by the type rules it enforces. Because the strictness of the type rules can vary, type systems can range from being very strict to being very weak.

Generally two different approaches exist to solve the goal of minimising errors in programs. Static typing checks types at compile-time and therefore is able to present type errors at a early stage. Because of this, static typing is often regarded as safer and more performant.

Dynamic typing checks types at run-time and therefore allows the programmer to not have the same mental overhead of thinking about types when programming. This however comes at the price of safety and performance.

Type inference removes the need to explicitly annotate programs with type annotations. This can reduce the mental overhead associated with static typing.

Type System Choice for TLDR

The type system for TLDR is *strong statically typed.* No implicit type conversions are made in the type system. The choice of static typing was made based on the domain in which TLDR is meant to be used. Having type errors presented early on in the process of developing programs in the language, means much stronger guarantees can be made about the final running program. This is important because running programs on large systems or distributed systems is costly; whenever a program is run on such systems, the program should produce the expected result without run-time failures.

Primitive Data Types

Since mathematics is a theoretical abstraction, it is rarely a problem to denote size of numbers, precision on numbers or even infinities. Computers, sadly, have physical limitations which makes it complicated or inefficient to express all these things as just "numbers". This the main reason for type declarations on numbers. In mathematics we can also denote a number to be of a certain "type", such as the natural numbers, the rational numbers and the real numbers. However in mathematics, this is done for a different reason, namely the different charateristics numbers have.

However, both computer science's type declarations and mathematics' number systems, serve to provide a set of characteristics for the number, and how it will behave. We were inspired by this similarity, and chose to preserve it in our language. However, The mathematical symbol for set membership, "$\in$", is not found on a regular keyboard, so instead the colon, ":", was chosen.

When initialising a symbol, the programmer must declare its type, as if the symbol belonged to a set, where the set must be a type in TLDR.

Primitives are the lowest abstraction of value types, that user can interact with, it is not divisible by the user, therefore atomic from their point of view. This means, as an example, that it is not possible for the user to express any deep meaning of what constitutes an integer, e.g. an integer composed of 4 bits.

The following primitives exist in the language. All primitives are lower-cased.

- int
- real
- bool
- char
- unit

Integer The integer primitive can have values of whole numbers, e.g. *2*. Integers in TLDR are arbitrary precision, meaning there are no bounds to the range of possible values.

Real The real primitive can have values of real numbers. The real primitive's literal representation is as decimal numbers with fractions e.g. *2.5* or *2.0*. As with integers, reals are arbitrary precision.

Bool The bool primitive's value can be either *true* or *false*.

Char The char primitive can have values of the ASCII standard. It is written literally as a character defined in the ASCII standard, surrounded by single quotation marks. For example: *'0'* and *'A'*.

Unit The unit primitive can have only one value: itself. The use of the primitive is to signal emptiness.

The prmitives will be elaborated upon in *section 5.2.2*.

Type Declaration

From the start of the language design, TLDR was supposed to have type inference, due to types not being a known artifact/property in the mathematics of physics and social sciences, which is the scope of the language. Due to prioritisation and time span of the project, it was decided that the language should initially have an explicit type system.

Types not being a known artifact in previously mentioned sciences and mathematics, a technique known from the functional paradigm of separating the type signature from the function definition, as shown here *section 5.1.1*, was chosen. In this signature, the function can take any number of arguments and return either a single typed symbol or a new function. The input and output is differenciated by the amount of input parameters (arg1, arg1, ... ,argN) where N would be the amount of inputs and M - N would be the size of the output.

```
functionName(arg1, arg2, ... ,argN) :: arg1Type -> arg2Type -> ...
   -> argMType
```

If the user wants to use functions instead of single typed symbols he or she simply puts these in parentheses as seen in *section 5.1.1*. Here the function, "f", takes another function that maps an int to an int and maps this to a real.

```
  f(x) : (int -> int) -> real
```

This can be nested in as many levels as the user desires, as illustrated in *section 5.1.1*, where the function takes a function typed the same way as in *section 5.1.1* and maps this to a list of chars. Note that both *section 5.1.1* and *section 5.1.1* still only take one argument (x) since this is a function and can be used as such in the body.

```
  f(x) : ((int -> int) -> real) -> [char]
```

Implicit Unit This approach posed a problem with the concept of "unit". This represents the non-existing value, for the type system. It means that something does not have a value and using it in any context does not make sense. For instance when a function returns unit i.e. nothing, it cannot be assigned to anything. The need for explicitness resides in that The Language Described in This Report has functions as, what is called, "first class citizens", meaning that functions can be used as arguments and essentially are treated in the same way as other symbols such as integers, structs etc. Here, the language becomes ambigious with implicit unit, i.e. the user not needing to explicitly tell when a function returns unit. For instance in *section 5.1.1* the programmer would not be able to tell whether the function takes a function that takes an int and returns an int or returns a function that goes from int to unit since the syntax is the same. Several proposed solutions will be described here after.

First, a type signature int the language looks as follows:

```
printint(a): int -> int
```

Where the first int is the parameter a of the function $printint$, and the second int of the signature is the return type.

So in the case that the function $printint$ should not return anything, the last type of the type signature will be decorated with a type of nothing, e.g. unit, as follows:

```
printint(a): int -> unit
```

The first proposal of a solution is to denote unit as "nothing", like this:

```
printint(a): int ->
```

If the type signature of a function is that it takes nothing as parameter but returns something, like this:

```
printint(): -> int
```

In the end, it was chosen to explicitly express unit both to avoid confusion for users of the language and since it does not make much sense to focus on improving the explicit type signatures, since type inference is a long term goal for the language. The explicit format ended up as follows:

```
printint(a): int -> unit
```

Formal Type Rules

The following primitive types are defined

$$\text{PT} ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{char}$$

A tuple is a finite set of ordered elements of arbitrary type T

$$\text{TUPLE} ::= T_1 \times T_2 \times \ldots \times T_n$$
$$\text{B} ::= [\text{T}] \mid \text{PT} \mid \text{TUPLE}$$

All types defined in the type system is defined in T.

$$\text{T} ::= \text{B} \mid x : B \rightarrow ok$$

5.1.2 Scoping

As with most of the choices made during the design of this language, it was desired to strive towards a natural mathematical syntax. This idea was pulling towards the use of linebreaks for denoting the end of a statement, and using indentation for denoting scopes. However since this is impossible to describe with context-free-grammar, and explicitly modifying the parser to support this was adding unnecessary complexity to the language implementation, it was decided upon other constructs. For denoting scopes, the bracket symbols "{ }" were chosen. This decision was made due to its similarity with the parentheses known from mathematics, and since it is a construct known from many other programming languages. For separating statements, a semi-colon ";" was chosen. This is also a known construct from other programming languages.

5.1.3 Transition System for Formal Semantics

The transition system used in the formal semantics of TLDR uses three different partial functions. These will be introduced at they are needed but here follows an explanation of each function in succession. The first transition rule, "at", is used to keep track of all actor types. The actor type is a form of prototype used to initialise actors. When created, all its statements are run inside a new environment consisting of aEnv, sEnv. This transition rule is there throughout the whole program meaning that all actor types can be initialised everywhere.

$$at = \text{ActorTypes} \rightharpoonup \text{Stm}$$

To describe a complete environment in TLDR one needs both the actor environment and the symbol environment. The actor environment is used to keep track of what handles, for actors, inside an actor maps to.

$$aEnv = \text{Anames} \cup next \rightharpoonup sEnv$$

The symbol environment is used to keep track of what statements and other symbols map to. The statements that a symbol maps to are the statements that are run in an invocation of that symbol. Note that all symbols in TLDR are invoked, even variables and constants which will always return the value which they are assigned, no matter the state of their environment. The symbols that a symbol maps to are used to keep track of what formal parameters an invocation needs, note that this set can be empty.

$$sEnv = \text{Symbols} \rightharpoonup \text{Stm} \times \text{Symbols}$$

5.1.4 Comments

Comments are declared in C style by "//" being a single line comment and "/**/" being a multi line comment.

They follow this grammar:

⟨*COMMENT_LINE*⟩ ::= '//' (.* - (.* ⟨*NEW_LINE*⟩ .*)) ⟨*NEW_LINE*⟩?

⟨*COMMENT_BLOCK*⟩ ::= '/*' (.* - (.* '*/' .*)) '*/'

A concrete example:

```
// this is a single line comment
/* this is
   a
   multi line comment */
```

5.2 Expressions

This section describes expressions. In The Language Described in This Report, expressions are defined as the constructs that have a value. These can be used together with specific operators to create larger expressions as one would do in mathematics. Apart from combining expressions, they are often used as the right hand side of an assignment, but can also be used for indexing, the condition in conditional statements, for return values and in general all places where a value is expected.

5.2.1 Operators

First we look at the mathematical operators that can combine expressions.

```
⟨Expression⟩    ::= ⟨Expression⟩ ⟨Psevenoperator⟩ ⟨OP6⟩
                 |  ⟨OP7⟩

⟨OP7⟩           ::= ⟨OP7⟩ ⟨Psixoperator⟩ ⟨OP6⟩
                 |  ⟨OP6⟩

⟨OP6⟩           ::= ⟨Pfiveoperator⟩ ⟨OP6⟩
                 |  ⟨OP5⟩

⟨OP5⟩           ::= ⟨OP5⟩ ⟨Pfouroperator⟩ ⟨OP4⟩
                 |  ⟨OP4⟩

⟨OP4⟩           ::= ⟨OP4⟩ ⟨Pthreeoperator⟩ ⟨OP3⟩
                 |  ⟨OP3⟩

⟨OP3⟩           ::= ⟨OP3⟩ ⟨Ptwooperator⟩ ⟨OP2⟩
                 |  ⟨OP2⟩

⟨OP2⟩           ::= ⟨OP2⟩ ⟨Poneoperator⟩ ⟨OP1⟩
                 |  ⟨OP1⟩

⟨OP1⟩           ::= ⟨Pzerooperator⟩ ⟨OP1⟩
                 |  ⟨OP0⟩

⟨OP0⟩           ::= ⟨Operand⟩
                 |  '(' ⟨Expression⟩ ')'

⟨PZEROOPERATOR⟩ ::= '("int' | 'real' | 'char' | 'bool")'

⟨PONEOPERATOR⟩ ::= '^' | '#'

⟨PTWOOPERATOR⟩ ::= '*' | '/' | '%'

⟨PTHREEOPERATOR⟩ ::= '+' | '-'

⟨PFOUROPERATOR⟩ ::= '=' | '!=' | '<' | '<=' | '>' | '>='

⟨PFIVEOPERATOR⟩ ::= 'NOT'

⟨PSIXOPERATOR⟩ ::= 'AND' | 'NAND'

⟨PSEVENOPERATOR⟩ ::= 'OR' | 'XOR' | 'NOR'
```

The precedence of the operators are created in the grammar. Here, the parse tree will be created, such that the $\langle PSEVENOPERATOR\rangle$ is placed highest in the tree and the $\langle PZEROOPERATOR\rangle$ is placed lowest as standard, meaning that the precedence goes from $\langle PZEROOPERATOR\rangle$ to $\langle PSEVENOPERATOR\rangle$ with zero having highest precedence. This precedence can be overwritten by parentheses, which resets the order so expressions inside parentheses are placed lowest in the tree. All operators have left associativity. An example can be seen in *fig. 5.2* this shows how the parse tree is created from the expression:

$$\text{NOT } a \text{ AND } b \text{ XOR } 2 < 3 * (2+2) + 4$$

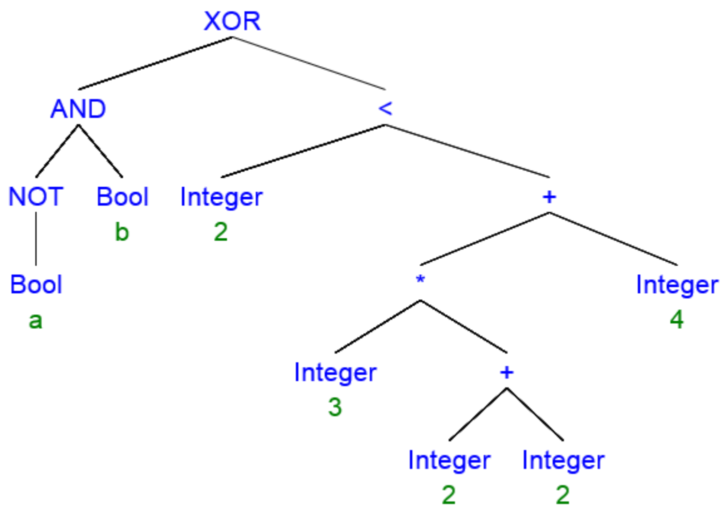


Figure 5.2: *The parse tree for* "NOT a AND b XOR $2 < 3 * (2+2) + 4$"

We can see that the operator placed lowest in the tree is "+" even though "*" has a higher precedence, and should therefore be lower in the tree. This is done because the parentheses overwrite the order, so that everything inside gets higher precedence, as intended. If we insert parentheses to illustrate the implicit precedence in the expression it would look like this:

$$((\text{NOT } a) \text{ AND } b) \text{ XOR } (2 < ((3 * (2+2)) + 4))$$

In *fig. 5.2* we can see that the "XOR" operator is placed highest in the tree meaning that it has the lowest precedence. We can also see from the tree that this operator takes the result from "AND" and "<" as arguments where "<" again takes the result from "+" and the integer 2 as arguments and so on. We can see that the arguments change type as we traverse the tree. We now look at the semantics and what impact this has on the formal type rules.

Arithmetic Expressions

First we look at arithmetic expressions. Arithmetic expressions are in TLDR defined as expressions that evaluate to a number, either a real or integer. The operators that create the arithmetic expressions are +, -, *, /, %, ^ and #. The semantics for these operators are as follows:

- "+" is a binary operator that adds two numbers of the same type

$$\text{ADD}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 + a_2 \Rightarrow_A a_1' + a_2} \quad \text{ADD}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 + a_1 \Rightarrow_A a_2 + a_1'}$$

$$\text{ADD}_\text{V}\frac{}{v_1 + v_2 \Rightarrow_A v}, v_1 + v_2 = v$$

- "-" is a binary operator that subtracts two numbers of the same type

$$\text{SUB}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 - a_2 \Rightarrow_A a_1' - a_2} \quad \text{SUB}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 - a_1 \Rightarrow_A a_2 - a_1'}$$

$$\text{SUB}_\text{V}\frac{}{v_1 - v_2 \Rightarrow_A v}, v_1 - v_2 = v$$

- "*" is a binary operator that multiplies two numbers of the same type

$$\text{MULT}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 * a_2 \Rightarrow_A a_1' * a_2} \quad \text{MULT}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 * a_1 \Rightarrow_A a_2 * a_1'}$$

$$\text{MULT}_\text{V}\frac{}{v_1 * v_2 \Rightarrow_A v}, v_1 * v_2 = v$$

- "/" is a binary operator that divides two numbers of the same type

$$\text{DIV}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 / a_2 \Rightarrow_A a_1' / a_2} \quad \text{DIV}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 / a_1 \Rightarrow_A a_2 / a_1'}$$

$$\text{DIV}_\text{V}\frac{}{v_1 / v_2 \Rightarrow_A v}, \frac{v_1}{v_2} = v$$

- "%" is a binary operator that returns the remainder of a floored division of two numbers of the same type

$$\text{MOD}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 \% a_2 \Rightarrow_A a_1' \% a_2} \qquad \text{MOD}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 \% a_1 \Rightarrow_A a_2 \% a_1'}$$

$$\text{MOD}_\text{V}\frac{}{v_1 \% v_2 \Rightarrow_A v}, v_1 \mod v_2 = v$$

- "^" is a binary operator that lifts the first number to the power of the second number

$$\text{POW}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 \wedge a_2 \Rightarrow_A a_1' \wedge a_2} \qquad \text{POW}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 \wedge a_1 \Rightarrow_A a_2 \wedge a_1'}$$

$$\text{POW}_\text{V}\frac{}{v_1 \wedge v_2 \Rightarrow_A v}, v_1^{v_2} = v$$

- "#" is a binary operator that roots the first operand to the second operand

$$\text{ROOT}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 \# a_2 \Rightarrow_A a_1' \# a_2} \qquad \text{ROOT}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 \# a_1 \Rightarrow_A a_2 \# a_1'}$$

$$\text{ROOT}_\text{V}\frac{}{v_1 \# v_2 \Rightarrow_A v}, \sqrt[v_1]{v_2} = v$$

- "( )" Parentheses gives what they surrounds the highest precedence.

$$\text{PARENS}_\text{A}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash (a_1) \Rightarrow_A (a_1')} \qquad \text{PARENS}_\text{V}\frac{}{(v) \Rightarrow_A v}$$

For simplicity we create the following set since all type rules for these are the same.

$$\text{AOP} = \{+, -, *, /, \%, \wedge\}$$

Due to the semantics of all "AOP" operators these cannot evaluate to a real, if the two inputs are both integers. Therefore all operators in this set can take either two integers and evaluate to an integer or two reals and evaluate to a real.

The “#” operator is a bit different. When taking the an integer root of another integer it can still evaluate to a real, for instance $\sqrt[2]{2}$ evaluates to 1.4142... Therefore both rules for “#” evaluate to a real.

No operator can take a combination of real and integer. This is done since all implicit type casts are avoided in TLDR. The reason for this is that the language is designed to give the programmer all errors as early as possible, preferably on compile-time, see *section 5.1.1*. With no implicit type casts the programmer is always aware when type casts are performed and will therefore not be as prone to make runtime type errors.

$$\text{EXPR}_{\text{int,int}}\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 \,\text{op}\, e_2 : \text{int}}, \text{op} \in \text{AOP}$$

$$\text{EXPR}_{\text{real,real}}\frac{E \vdash e_1 : \text{real} \qquad E \vdash e_2 : \text{real}}{E \vdash e_1 \,\text{op}\, e_2 : \text{real}}, \text{op} \in \text{AOP}$$

$$\text{ROOT}_{\text{int,int}}\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 \# e_2 : \text{real}}$$

$$\text{ROOT}_{\text{real,real}}\frac{E \vdash e_1 : \text{real} \qquad E \vdash e_2 : \text{real}}{E \vdash e_1 \# e_2 : \text{real}}$$

Boolean Expressions

Boolean expressions are in TLDR defined as expressions that takes boolean types as arguments and returns a boolean type. The boolean expressions can be constructed from the operators AND, NAND, OR, NOR, XOR, NOT.

The following are the semantics for all boolean operators and their truth table.

- "AND" is a binary operator that returns true if both values are true. False otherwise

$$\text{AND}_1 \frac{sEnv \vdash b_1 \Rightarrow_B \bot}{sEnv \vdash b_1 \ \text{AND} \ b_2 \Rightarrow_B \bot}$$

$$\text{AND}_2 \frac{\begin{array}{c} sEnv \vdash b_1 \Rightarrow_B \top \\ sEnv \vdash b_2 \Rightarrow_B \bot \end{array}}{sEnv \vdash b_1 \ \text{AND} \ b_2 \Rightarrow_B \bot}$$

$$\text{AND}_3 \frac{\begin{array}{c} sEnv \vdash b_1 \Rightarrow_B \top \\ sEnv \vdash b_2 \Rightarrow_B \top \end{array}}{sEnv \vdash b_1 \ \text{AND} \ b_2 \Rightarrow_B \top}$$

e_1	e_2	e_1 AND e_2
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

- "OR" is a binary operator that returns true if at least one value is true. False otherwise

$$\text{OR} \frac{}{sEnv \vdash b_1 \ \text{OR} \ b_2 \Rightarrow_B \text{NOT} \ (\text{NOT} \ b_1 \ \text{AND} \ \text{NOT} \ b_2)}$$

e_1	e_2	e_1 OR e_2
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$

- "XOR" is a binary operator that returns true if only one operand is true. False otherwise

$$\text{XOR} \frac{}{sEnv \vdash b_1 \ \text{XOR} \ b_2 \Rightarrow_B (\text{NOT} \ (b_1 \ \text{AND} \ b_2)) \ \text{AND} \ (b_1 \ \text{OR} \ b_2)}$$

e_1	e_2	e_1 XOR e_2
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$

- "NOR" is OR negated.

$$\text{NOR} \frac{}{sEnv \vdash b_1 \ \text{NOR} \ b_2 \Rightarrow_B \text{NOT} \ (b_1 \ \text{OR} \ b_2)}$$

e_1	e_2	e_1 NOR e_2
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$

- "NOT" is a unary operator that returns the opposite value of the operand.

$$\text{NOT}_\top \frac{sEnv \vdash b_1 \Rightarrow_B \top}{sEnv \vdash \text{NOT } b_1 \Rightarrow_B \bot}$$

$$\text{NOT}_\bot \frac{sEnv \vdash b_1 \Rightarrow_B \bot}{sEnv \vdash \text{NOT } b_1 \Rightarrow_B \top}$$

e_1	NOT e_1
$\bot$	$\top$
$\top$	$\bot$

- "NAND" is a binary operator that returns true if none or a single operand is true. False otherwise.

$$\text{NAND} \frac{}{sEnv \vdash b_1 \text{ NAND } b_2 \Rightarrow_B \text{NOT } (b_1 \text{ AND } b_2)}$$

e_1	e_2	e_1 NAND e_2
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$

- "()" Parentheses gives what they surround the highest precedence.

$$\text{PARENS}_\text{B} \frac{sEnv \vdash b_1 \Rightarrow_B b_1'}{sEnv \vdash (b_1) \Rightarrow_B (b_1')}$$

All boolean operators, except NOT, takes two booleans and returns a boolean. For the simplicity of the type rules we create the Boolean Operator (BOP):

$$\text{BOP} = \{\text{AND, NAND, OR, NOR, XOR}\}$$

Note that “NOT” is not included in the set since it only takes one boolean as input and return a boolean.

$$\text{BOOL}_{BOP}\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : \text{bool}}{E \vdash e_1 \,\text{op}\, e_2 : \text{bool}}, \text{op} \in \text{BOP}$$

$$\text{BOOL}_{NOT}\frac{E \vdash e : \text{bool}}{E \vdash \text{NOT}\, e : \text{bool}}$$

Logical Operations

Logical operators are defined as operators that take numbers, i.e. integers and reals, and evaluate to boolean types.

For logical comparisons we chose “=”. This was done in accordance with the goal of keeping a natural mathematical language. In mathematics “=” is read as “is equal to” or simply “equals”, and is used for stating that two parts are equivalent to each other. Sometimes mathematicians use this statement in a contradicting manner, where they expect to prove the statement to be false. It is from this perspective of being a statement, either true or false, that we chose “=” to be a logical comparison. The same arguments exist for other types of logical operations.

- "=" is a binary operator that compares the two operands for equality. Returns true if equal. False otherwise.

$$\text{EQUALS}_{\text{L}}\frac{sEnv \vdash a_1 \Rightarrow_B a_1'}{sEnv \vdash a_1 = a_2 \Rightarrow_B a_1' = a_2} \qquad \text{EQUALS}_{\text{R}}\frac{sEnv \vdash a_1 \Rightarrow_B a_1'}{sEnv \vdash a_2 = a_1 \Rightarrow_B a_2 = a_1'}$$

$$\text{EQUALS}_{\text{V1}}\frac{}{v_1 = v_2 \Rightarrow_B \top}, v_1 = v_2 \qquad \text{EQUALS}_{\text{V2}}\frac{}{v_1 = v_2 \Rightarrow_B \bot}, v_1 \neq v_2$$

- "!=" is a binary operator that compares the two operands for equality. Returns true if not equal. False otherwise.

$$NEQUALS\frac{}{sEnv \vdash a_1 != a_2 \Rightarrow_B \text{NOT}\ \ (a_1 = a_2)}$$

- "<" is a binary operator that compares the two operands. Returns true if the first operand is strictly less than the second operand. False otherwise.

$$\text{LT}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 < a_2 \Rightarrow_A a_1' < a_2} \qquad \text{LT}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 < a_1 \Rightarrow_A a_2 < a_1'}$$

$$\text{LT}_{\text{V1}}\frac{}{v_1 < v_2 \Rightarrow_B \top}, v_1 < v_2 \qquad \text{LT}_{\text{V2}}\frac{}{v_1 < v_2 \Rightarrow_B \bot}, v_1 \geq v_2$$

- "<=" is a binary operator that compares the two operands. Returns true if the first operand is less than or equal to the second operand. False otherwise.

$$LTEQ\frac{}{sEnv \vdash a_1 <= a_2 \Rightarrow_A (a_1 < a_2)\ \text{ OR }\ (a_1 = a_2)}$$

- ">" is a binary operator that compares the two operands. Returns true if the first operand is strictly greater than the second operand. False otherwise.

$$\text{GT}_\text{L}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_1 > a_2 \Rightarrow_A a_1' > a_2} \qquad \text{GT}_\text{R}\frac{sEnv \vdash a_1 \Rightarrow_A a_1'}{sEnv \vdash a_2 > a_1 \Rightarrow_A a_2 > a_1'}$$

$$\text{GT}_{\text{V1}}\frac{}{v_1 > v_2 \Rightarrow_B \top}, v_1 > v_2 \qquad \text{GT}_{\text{V2}}\frac{}{v_1 > v_2 \Rightarrow_B \bot}, v_1 \leq v_2$$

- ">=" is a binary operator that compares the two operands. Returns true if the first operand is greater than or equal to the second operand. False otherwise.

$$GTEQ\frac{}{sEnv \vdash a_1 >= a_2 \Rightarrow_A (a_1 > a_2)\ \text{ OR }\ (a_1 = a_2)}$$

Since all logical operators take a number and returns a boolean we create the set Logical Operators (LOP):

$$\text{LOP} = \{=, ! =, <, <=, >, >=\}$$

$$\text{BOOL}_{\text{int,int}}\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 \text{ op } e_2 : \text{bool}}, \text{op} \in \text{LOP}$$

$$\text{BOOL}_{\text{real,real}}\frac{E \vdash e_1 : \text{real} \qquad E \vdash e_2 : \text{real}}{E \vdash e_1 \text{ op } e_2 : \text{bool}}, \text{op} \in \text{LOP}$$

5.2.2 Operands

When creating expressions, one also needs operands to use in the operators. The operands are as follows in formal syntax:

⟨*Operand*⟩	::=	⟨*Block*⟩
	|	⟨*Integer*⟩
	|	⟨*Real*⟩
	|	⟨*Boolean*⟩
	|	⟨*Literals*⟩
	|	⟨*Invocation*⟩

Some of these operands can evaluate something that is neither an Integer, Real or Boolean; for instance lists, actors and structures. This means that, due to the type system seen in *section 5.2.1*, the operand cannot be an argument to any operand. This however does not mean that it cannot be an expression, they can because all operands by themselves also have value and are an expression.

Integer

The integer is a literal integer that is written directly in the code, for instance in the statement “let x := 2;”, “2” is an integer literal. All integers are interpreted as decimal numbers and can be any combination of the symbols 0-9 in any length. They can be either positive or negative, illustrated with the symbol “-”, but may not start with a 0 unless it is only the number “0”. This is done since readability is weighted higher than writeability, see *section 4.6*, and it was assessed for instance “00023” is not very readable compared to “23”.

Syntax

⟨*Integer*⟩	::=	’-’?[1-9][0-9]* | ’0’

Semantics

The num rule is mapping numerals to numbers.

$$\text{NUM}\frac{}{sEnv \vdash n \Rightarrow_A v}, \mathcal{N}(n) = v$$

Type Rules

The type of a integer literal is simply int

$$\text{NUM}\frac{E \vdash n : \text{int}}{E \vdash n : \text{int}}$$

Real

The real is a literal real that is written directly in the code, for instance in the statement “let x := 4.3;”, “4.3” is an real literal. All real are interpreted as decimal numbers and is an integer followed by a “.” and then any combination of the symbols 0-9. They can like integers be either positive or negative, indicated with the symbol “-”. There must be at least one number both before and after the “.”. This means that neither “.1” and “1.” are allowed. This is done since it was assessed that for instance “-0.1” is more readable than “-.1”.

Syntax

⟨*Real*⟩ ::= (’-’?[1-9][0-9]* | ’0’)’.’[0-9]+

Semantics

The real rule is mapping numerals to reals

$$\text{REAL}\,\frac{}{sEnv \vdash n \Rightarrow_A v}\,,\mathscr{R}(n) = v$$

Type Rules

$$\text{REAL}\,\frac{E \vdash n_1 : \text{int} \qquad E \vdash n_2 : \text{int}}{E \vdash n_1.n_2 : \text{real}}$$

Boolean

Booleans are symbols that hold the value of either true or false. For instance we can in mathematics write “2 > 3”, this is a contradiction and therefore false.

Syntax

⟨*Boolean*⟩ ::= ’true’ | ’false’

Semantics

The bool rules are mapping boolean literals to boolean values

$$\text{TRUE}\,\frac{}{\langle \text{true}, sEnv\rangle \Rightarrow_B \top} \qquad \text{FALSE}\,\frac{}{\langle \text{false}, sEnv\rangle \Rightarrow_B \bot}$$

Type Rules

The type of a bool literal is simply a bool type

$$\text{BOOL}\frac{E \vdash b : \text{bool}}{E \vdash b : \text{bool}}$$

Invocation

An invocation in TLDR is defined as a series of characters, from here on referred to as the symbol, which is associated with a value. The value can either depend on arguments, a function, or be independent from any input, a constant or variable. If the value depends on arguments the evaluation is done lazy, meaning that the output value is only associated with the symbol when the symbol is invoked. Otherwise the symbol is associated with the functionality, i.e. its statements and how to evaluate the function.

We differentiate between direct and indirect arguments. Direct arguments are put inside parentheses after the symbol. These arguments can be any value as long as the types follow the type rules. A symbol can be dependent on a value, that is not given as a direct input, but outside its body, as long as it is in scope. We call these indirect arguments. If a function is dependent on indirect arguments we call it impure. Note that this property will be important in *section 5.4*.

A symbols dependence on arguments, both direct and indirect, is illustrated with parentheses. Inside the parentheses are the direct arguments. Note that these parentheses can be empty, if the symbol is only dependent on indirect or no arguments. For instance if x is declared as a symbol that takes an integer and returns an integer, see *section 5.4* for declarations, writing “y := x” would mean that y also has to be a function that takes an integer and returns an integer, but writing “z := x(2)” means that z is an integer. Any symbol, both invoked and uninvoked, can be considered expressions, i.e. having a value.

This syntax was decided since it is very close to mathematics, where functions take arguments in parentheses.

⟨*Invocation*⟩	::=	⟨*Identifier*⟩ ('(' (⟨*Expression*⟩ (',' ⟨*Expression*⟩)*)? ')')?;
⟨*Identifier*⟩	::=	'me'
	|	⟨*Id*⟩
	|	⟨*Id*⟩ ⟨*Accessor*⟩
⟨*Id*⟩	::=	[a-zA-Z][a-zA-Z_0-9]*-('let' | 'var' | 'bool' | 'integer' | 'real' | 'char' | 'struct' | 'actor' | 'receive' | 'send' | 'spawn' | 'return' | 'for' | 'in' | 'if' | 'else' | 'while' | 'die' | 'me');

⟨*Accessor*⟩	::=	'.' ⟨*Identifier*⟩
	|	'.' '[' ⟨*Expression*⟩ ']'
⟨*Primitive*⟩	::=	'int' | 'real' | 'char' | 'bool'

A symbol, x, is evaluated to a value by finding the symbol in the symbol environment $sEnv$.

$$\text{INVOKE}_{A1}\frac{\langle S, sEnv\rangle \Rightarrow_A v}{\langle x, sEnv\rangle \Rightarrow_A v}, sEnv(x) = \langle S, \epsilon\rangle$$

A function, x_1 is invoked with the parameter x_2 and evaluated to a value by finding x_1 in the symbol environment $sEnv$. This lookup results in the statement S_1 which is the body of the function, and the formal parameter x_1'. The actual parameter x_2 is then looked up in the symbol environment $sEnv$ for the statement S_2 which is the body of the actual parameter, and x_2' which is the parameter for x_2. To evaluate the function, the formal parameter x_1' in the symbol environment is assigned to the result of the lookup of the formal parameter x_2, S_2 and x_2'.

$$\text{INVOKE}_{A2}\frac{\left\langle S_1, sEnv[x_1' \mapsto \left\langle S_2, x_2'\right\rangle]\right\rangle \Rightarrow_A v}{\langle x_1(x_2), sEnv\rangle \Rightarrow_A v}, sEnv(x_1) = \left\langle S_1, x_1'\right\rangle, sEnv(x_2) = \left\langle S_2, x_2'\right\rangle$$

A symbol x is evaluated to a boolean value of true, when the body S of the symbol is found in the symbol environment $sEnv$, evaluates to true.

$$\text{INVOKE}_{B1\top}\frac{\langle S, sEnv\rangle \Rightarrow_B \top}{\langle x, sEnv\rangle \Rightarrow_B \top}, sEnv(x) = \langle S, \epsilon\rangle$$

This rule is the same as the above, but in this case, the body of the symbol x evaluates to false, and therefore so does x too.

$$\text{INVOKE}_{B1\bot}\frac{\langle S, sEnv\rangle \Rightarrow_B \bot}{\langle x, sEnv\rangle \Rightarrow_B \bot}, sEnv(x) = \langle S, \epsilon\rangle$$

This rule is similar to the rule $INVOKE_{A2}$, but instead of the value being a arithmetic value, it is a boolean value.

$$\text{INVOKE}_{B2\top}\frac{\left\langle S_1, sEnv[x_1' \mapsto \left\langle S_2, x_2'\right\rangle]\right\rangle \Rightarrow_B \top}{\langle x_1(x_2), sEnv\rangle \Rightarrow_B \top}, sEnv(x_1) = \left\langle S_1, x_1'\right\rangle, sEnv(x_2) = \left\langle S_2, x_2'\right\rangle$$

This rule is the same as $INVOKE_{B2\top}$, but instead of the boolean value being true, it is false.

$$\text{INVOKE}_{B2\bot}\frac{\left\langle S_1, sEnv[x_1' \mapsto \left\langle S_2, x_2' \right\rangle] \right\rangle \Rightarrow_B \bot}{\langle x_1(x_2), sEnv\rangle \Rightarrow_B \bot}, sEnv(x_1) = \left\langle S_1, x_1' \right\rangle, sEnv(x_2) = \left\langle S_2, x_2' \right\rangle$$

The type of an invocation can be looked up in the type environment E.

$$\text{INVOKE}\frac{E \vdash x : \text{T}}{E \vdash x : \text{T}}$$

Literals

There are four types of literals other than the numerical types and booleans. These are *char*, *list*, *structLiteral* and *tuple*.
A *char* is a single alphanumerical character, written in the code with apostrophes on either side. An example of a char is *'a'*.
A *list* is a collection type which can contain an arbitrary number of elements of a single type. A *list* is written in the code inside square brackets, with the elements either separated by commas or written as a range from one number to another signified by .. Two examples of a *list* are *[1,2,3]* and *[1 .. 10]*.
A *structliteral* is an initialisation of a struct. If we have a struct *struct s := {x:int; y:real;};*, a literal of *s* could for example be *{x := 3; y:= 2.3;}*.
A *tuple* is much like a *struct* in the way it can contain an arbitrary amount of fields with no restrictions on types. In *tuples* however, the fields have no name. An example of a *tuple* is *(1, 2.5 , "hello world")*.

```
⟨Literals⟩      ::= ⟨String⟩
                  | ⟨Char⟩
                  | ⟨List⟩
                  | ⟨StructLiteral⟩
                  | ⟨Tuple⟩

⟨String⟩        ::= '"' (U+0020 .. U+007E)* '"'

⟨Char⟩          ::= ''' U+0020 .. U+007E '''
```

Block

In TLDR, blocks are a way to encapsulate statements, see *section 5.3*. This is done via curly brackets, "{" starting the block, and "}" ending it.

⟨*Block*⟩	::=	'{' ⟨*Body*⟩ '}'
⟨*Body*⟩	::=	⟨*Body*⟩ ';' ⟨*Statement*⟩
	|	⟨*Body*⟩ ';'
	|	⟨*Statement*⟩

Note the that the last semicolon in a block is optional.

Statements do not have a value in TLDR, but they update the state of the program or at least have the possibility to do so. When a block is evaluated the statements inside it are run.

$$\text{BLOCK}_{S1}\ \frac{\langle S, sEnv\rangle \Rightarrow_S \langle S', sEnv'\rangle}{\langle \{S\}, sEnv\rangle \Rightarrow_S \langle \{S'\}, sEnv'\rangle}$$

A block is bit different than most other constructs in TLDR in that it is a statement, but can at the same time be an expression too. If the last that is run inside the block is a statement the block itself is only a statement:

$$\text{BLOCK}_{S2}\ \frac{\langle S, sEnv\rangle \Rightarrow_S sEnv'}{\langle \{S\}, sEnv\rangle \Rightarrow_S sEnv'}$$

An expression by itself is in principle a statement in TLDR, for instance "2 + 2;". Note that this is not allowed in an expression, for instance "2 + 2; + 4". An expression which takes the form of a statement, will not have value, but can give a block value if it is the last thing that is evaluated in it. If it is, the block takes the value of this expression thereby making the block an expression. So for instance the block:

```
{
  let a:int := 5;
  a * 2
}
```

will have the value 10 when evaluated. This is formally, for aritmetic expressions, described as:

$$\text{BLOCK}_{A3}\ \frac{\langle x, sEnv\rangle \Rightarrow_A \langle x', sEnv\rangle}{\langle \{x\}, sEnv\rangle \Rightarrow_A \langle \{x'\}, sEnv\rangle} \qquad \text{BLOCK}_{A4}\ \frac{\langle x, sEnv\rangle \Rightarrow_A v}{\langle \{x\}, sEnv\rangle \Rightarrow_A v}$$

And for boolean and logical expressions:

$$\text{BLOCK}_{B1}\ \frac{\langle x, sEnv\rangle \Rightarrow_B \langle x', sEnv\rangle}{\langle \{x\}, sEnv\rangle \Rightarrow_B \langle \{x'\}, sEnv\rangle}$$

$$\text{BLOCK}_{B2}\ \frac{\langle x, sEnv\rangle \Rightarrow_B \top}{\langle \{x\}, sEnv\rangle \Rightarrow_B \top} \qquad \text{BLOCK}_{B3}\ \frac{\langle x, sEnv\rangle \Rightarrow_B \bot}{\langle \{x\}, sEnv\rangle \Rightarrow_B \bot}$$

The keyword "return" forces an exit from the block and gives the block the value of whatever that expression that is after the keyword. This is done simply by inserting the expression and removing any further statements.

$$\text{RETURN}_1 \frac{}{\langle\{\text{return x}\,; S\}, sEnv\rangle \Rightarrow_S \langle\{x\}, sEnv\rangle}$$

$$\text{RETURN}_2 \frac{}{\langle\{\text{return x}\,\}, sEnv\rangle \Rightarrow_S \langle\{x\}, sEnv\rangle}$$

$$\text{RETURN}_3 \frac{}{\langle\{\text{return}; S\}, sEnv\rangle \Rightarrow_S \langle sEnv\rangle}$$

$$\text{RETURN}_3 \frac{}{\langle\{\text{return}\}, sEnv\rangle \Rightarrow_S \langle sEnv\rangle}$$

If the block is not an expression we say it has the return type "unit". Any piece of code, that does not evaluate to a value, is said to be of type unit and cannot be used in an assignment. Note that all statements by themselves have the type unit, but that a block only has the type unit when evaluated.

Blocks have the type of the last statement run in the block.

$$\text{BLOCK} \frac{E \vdash s_1 : \text{T} \quad E \vdash s_2 : \text{T}'}{E \vdash \{s_1; s_2\} : \text{T}'}$$

5.3 Statements

Statements are defined as constructs that have the possibility to change the state of the program. The state of the program is defined as the symbol by which values are associated. We call this the environment and it is formally described as:

$$e = \text{Symbols} \rightharpoonup \text{Stm} \times \text{Symbols}$$

This is a partial function that maps symbols to statements and other statements. These statements are the statements that are run when the symbol is invoked. If these map to an expression, the invocation takes the value of that invocation in the current environment. This can be any expression and, as previously explained ,the symbol evaluates to whatever the expression maps to. Note that a block is also an expression, see *section 5.2.2*. If the statements evaluate only to a statement, the symbol is of type "unit" and it has no value. Note that the statements can still update the state of the program during the invocation.

If the symbol takes input arguments, these are placed inside the “Symbols” on the right hand side of the arrow. In an invocation these symbols will be mapped to the values given as input parameters.

5.3.1 Initialisations

A new symbol in TLDR can be created via an initialisation. A symbol always maps to at least one statement, i.e. a symbol cannot map to nothing. This means that the environment can never map a symbol to the empty set ϵ. For that reason, the initialisation always includes an assignment with statements and/or expressions. Since symbols can only map statements, a symbol can only have a value when evaluated.

In traditional mathematical notation, the “=” symbol is also sometimes used to let certain symbols represent a more complex meaning, in order to simplify something, such as an equation or a function. When used like this, often mathematicians put the word “let” in front of a statement to denote that it is a definition. We wanted to follow this construct as well letting immutable assignment be denoted in this fashion, since they are comparable to definitions.

But, since we wanted assignments, be it mutable or immutable, to have similarities, we chose “:=” for all assignments. This concept is less known in traditional mathematics, but is widely used in computational science. In the historically significant languages such as Fortran and C, the “=” symbol, was used for this. However, since we wish to keep that symbol closer to its original meaning, we needed something else. “:=” was chosen, since it is a known symbol from other languages. The asymmetry of the “:=” symbol also illustrates that it matters which side of the symbol a variable is on, as opposed to the “=” symbol.

When assigning statements, we differentiate between functions and constants/variables. Whether it is a constant or variable binding it is denoted with the keywords “let” for a constant binding and “var” for a variable binding. If the statements are bound as a constant they can never be changed. This is useful, especially in mathematics where many things both functions and constants never changes. But for things like results, state and generic behaviour a variable binding is useful.

Whether a symbol is a function or not is denoted via parentheses, parentheses meaning that it is a function and no parentheses meaning that it is not. Note that a symbol can have empty parentheses and still be a function, meaning that there is a difference between “let a():int := ... ” and “let a:int := ... ”.

$\langle$*Initialisation*$\rangle$ ::= ('let' | 'var') ($\langle$*FuncDecl*$\rangle$ | $\langle$*SymDecl*$\rangle$) ':=' $\langle$*Expression*$\rangle$

$\langle$*FuncDecl*$\rangle$ -> $\langle$*Identifier*$\rangle$ '(' $\langle$*Ids*$\rangle$? ')' ':' $\langle$*Types*$\rangle$

⟨*SymDecl*⟩ -> ⟨*Identifier*⟩ ':' ⟨*Types*⟩

If a symbol is a constant or variable, the right hand side of the ":=" is evaluated and the symbol is assigned the simplest statement that will always evaluate to the value of the evaluation. If the right hand side is evaluated to a number, it is formally described as follows:

$$\text{INIT}_{SYM-A1}\frac{\langle x, sEnv\rangle \Rightarrow_A \langle x', sEnv\rangle}{\langle \text{let a} := x, sEnv\rangle \Rightarrow_S \langle \text{let a} := x', sEnv\rangle}$$

$$\text{INIT}_{SYM-A2}\frac{\langle x, sEnv\rangle \Rightarrow_A v}{\langle \text{let a} := x, sEnv\rangle \Rightarrow_S sEnv'}$$
where $sEnv' = env_s[\text{ a} \mapsto \langle\{n\},\epsilon\rangle], \mathcal{N}(n) = v$

The value the expression evaluates to is converted, via the $\mathcal{N}$ function, to the numeral and the symbol is assigned this as within a block. In this way an invocation of the symbol will evaluate the block and the same value each time. If the right hand side is evaluated to a boolean the formal semantics are described as follows:

$$\text{INIT}_{SYM-BOOL}\frac{\langle b, sEnv\rangle \Rightarrow_B \langle b', sEnv\rangle}{\langle \text{let a} := b, sEnv\rangle \Rightarrow_S \langle \text{let a} := b', sEnv\rangle}$$

$$\text{INIT}_{SYM-\top}\frac{\langle b, sEnv\rangle \Rightarrow_B \top}{\langle \text{let a} := b, sEnv\rangle \Rightarrow_S sEnv'}$$
where $sEnv' = sEnv[\text{ a} \mapsto \langle\{\text{true}\},\epsilon\rangle]$

$$\text{INIT}_{SYM-\bot}\frac{\langle b, sEnv\rangle \Rightarrow_B \bot}{\langle \text{let a} := b, sEnv\rangle \Rightarrow_S sEnv'}$$
where $sEnv' = sEnv[\text{ a} \mapsto \langle\{\text{false}\},\epsilon\rangle]$

Again, the resulting value is placed as the statement "true" or "false" inside a block, thereby always evaluating to this value.

For the representation of functions in our language, we wanted to stay as close as possible to the mathematical functions, such as "f(x,y) = z". In order to achieve this, the type of the function arguments had to either be implicit or declared elsewhere. Since we chose to delimit ourselves from type inherence it must be declared. This is done in the same style as any type declaration, with a colon. Functions in TLDR are treated like values, being reassignable and potentially having a function take another function as a parameter or give it as a return value.

If a symbol is initialised as a function it is dynamically scoped. This means that the statements, that a symbol maps to, are evaluated when it is invoked, in the environment in which it is invoked, meaning that if the function is impure, see *section 5.2.2*, it will use the statements, that the symbols map to at the time of invocation, not at the time of initialisation. This means that a function when not invoked maps to the statements it was assigned and other functions can be assigned these or use them as inputs.

$$\text{INIT}_{FUNC1}\frac{}{\langle \text{let x () := } \{S\}, sEnv\rangle \Rightarrow_S sEnv'}, sEnv' = sEnv[\,\text{x} \mapsto \langle S, \epsilon\rangle]$$

If the symbol is a function and takes input parameters these are placed as formal parameters in the symbols ordered set the symbol maps to in the order they appear.

$$\text{INIT}_{FUNC2}\frac{}{\langle \text{let x } (x) := \{S\}, sEnv\rangle \Rightarrow_S sEnv'}, sEnv' = sEnv[\,\text{x} \mapsto \big\langle S, y\big\rangle]$$

Expanding the amount of input parameters is a trivial task. All symbols are placed in the set in the order they appear in the code.

$$\text{INIT}_{FUNC3}\frac{}{\big\langle \text{let x } (x, y) := \{S\};, sEnv\big\rangle \Rightarrow_S sEnv'}, sEnv' = sEnv[\,\text{x} \mapsto \big\langle S, \{x, y\}\big\rangle]$$

There was an initial push towards allowing signature declarations to occur completely seperated from the function declarations, in order to make it even further resemble mathematical notation. This was not included, since the long term goal is type inherency, by which this signature would become obsolete. Furthermore, the arguement of moving the signature away to resemble mathematics, becomes invalid since any move that actually achieves this, would greatly decrease readability, and if the signature is kept close to the function, it might as well have been tied directly to the function. In the case of the function having an intuitive signature, the inclusion would not decrease readability by any significance, since it is not intertwined with the function declaration.

Note that all rules are defined as constant bindings (“let”) but the exact same semantic holds for all variable bindings, since the allowance of reassignment is checked by the type system.

⟨*Identifier*⟩	::=	’me’
	|	⟨*Id*⟩
	|	⟨*Id*⟩ ⟨*Accessor*⟩

⟨*Id*⟩	::=	[a-zA-Z][a-zA-Z_0-9]*-('let' | 'var' | 'bool' | 'integer' | 'real' | 'char' | 'struct' | 'actor' | 'receive' | 'send' | 'spawn' | 'return' | 'for' | 'in' | 'if' | 'else' | 'while' | 'die' | 'me');
⟨*Ids*⟩	::=	⟨*Identifier*⟩ (',' ⟨*Identifier*⟩)*
⟨*Types*⟩	::=	⟨*Type*⟩ ('->' ⟨*Type*⟩)*
⟨*Type*⟩	::=	⟨*Primitive*⟩
	|	⟨*Identifier*⟩
	|	⟨*ListType*⟩
	|	⟨*TupleType*⟩
⟨*TupleType*⟩	::=	'(' ⟨*Types*⟩ ')'
⟨*ListType*⟩	::=	'[' ⟨*Types*⟩ ']'
⟨*Primitive*⟩	::=	'int' | 'real' | 'char' | 'bool'

As can be seen, values can either be an immutable constant, or a mutable variable.

A constant binding can never have its value changed. For example, if "a" is bound to "5", "a" can never refer to another value than "5" in the same lexical scope.

The syntax for constant value assignment is as follows.

```
  let <symbolName> : <type> := <value>
```

A concrete example:

```
  let x : int := 2
```

A variable binding can always change the value it refers to. For example, if "b" is bound to "2", it is perfectly possible to later in the source code refer to 10.

The syntax for variable value assignment is as follows.

```
  var <symbolName> : <type> := <value>

// a later reassignment
  <symbolName> := <value>
```

A concrete example:

```
  var a : int := 2

// a later reassignment
  a := 2
```

5.3.2 Assignment

As mentioned in *section 5.3.1,* variables can be assigned a value after their initialisation. This is only the case if the right hand side of the assignment is a value and of the same type as the variable. This is the grammar, specifying the syntax for assignment:

⟨*Reassignment*⟩ ::= ⟨*Identifier*⟩ ':=' ⟨*Expression*⟩

Assignment uses the ":=" operator. The left hand side of the operator is the symbol to be assigned the value. The right hand side is an expression which is the value to be assigned.

Since space is not allocated in semantics but the functions merely maps symbols to values the semantic rules for assignment are practically the same as initialisations:

$$\text{ASS}_{SYM-A1}\,\frac{\langle x, sEnv\rangle \Rightarrow_A \langle x', sEnv\rangle}{\langle\, \mathrm{a} := x, sEnv\rangle \Rightarrow_S \langle\, \mathrm{a} := x', sEnv\rangle}$$

$$\text{ASS}_{SYM-A2}\,\frac{\langle x, sEnv\rangle \Rightarrow_A v}{\langle\, \mathrm{a} := x, sEnv\rangle \Rightarrow_S sEnv'}$$
$$\text{where } sEnv' = env_s[\,\mathrm{a} \mapsto \langle\{n\},\epsilon\rangle],\mathcal{N}(n) = v$$

$$\text{ASS}_{SYM-BOOL}\,\frac{\langle b, sEnv\rangle \Rightarrow_B \langle b', sEnv\rangle}{\langle\, \mathrm{a} := b, sEnv\rangle \Rightarrow_S \langle\, \mathrm{a} := b', sEnv\rangle}$$

$$\text{ASS}_{SYM-\top}\,\frac{\langle b, sEnv\rangle \Rightarrow_B \top}{\langle\, \mathrm{a} := b, sEnv\rangle \Rightarrow_S sEnv'}$$
$$\text{where } sEnv' = sEnv[\,\mathrm{a} \mapsto \langle\{\text{true}\},\epsilon\rangle]$$

$$\text{ASS}_{SYM-\bot}\,\frac{\langle b, sEnv\rangle \Rightarrow_B \bot}{\langle\, \mathrm{a} := b, sEnv\rangle \Rightarrow_S sEnv'}$$
$$\text{where } sEnv' = sEnv[\,\mathrm{a} \mapsto \langle\{\text{false}\},\epsilon\rangle]$$

$$\text{INIT}_{FUNC1}\,\frac{}{\langle\, \mathrm{x}\,() := \{S\}, sEnv\rangle \Rightarrow_S sEnv'}, sEnv' = sEnv[\,\mathrm{x} \mapsto \langle S,\epsilon\rangle]$$

$$\text{INIT}_{FUNC2}\,\frac{}{\langle\, \mathrm{x}\,(x) := \{S\}, sEnv\rangle \Rightarrow_S sEnv'}, sEnv' = sEnv[\,\mathrm{x} \mapsto \langle S,y\rangle]$$

$$\text{INIT}_{FUNC3}\frac{}{\langle \text{x}\,(x,y) := \{S\};, sEnv\rangle \Rightarrow_S sEnv'}, sEnv' = sEnv[\text{x} \mapsto \langle S, \{x,y\}\rangle]$$

This is the type rule for assignment:

$$\text{ASS}\frac{E \vdash I : T \quad E \vdash e : T}{E \vdash \{\text{I} := \text{e}\} : ok}$$

5.3.3 Structures

In TLDR there are three ways to do encapsulation. Actors, Tuples and structs. Structs are unique by being fully accessible within the scope, and having named fields. Structs are especially useful in TLDR for creating messages.

Defining Structures

Structures are defined by using the “struct” keyword. The grammar for declaring structs are as follows:

⟨*Struct*⟩ ::= 'struct' ⟨*Identifier*⟩ ':= {' ⟨*TypeDecls*⟩ '}'

And a concrete example:

```
struct Person := {Name:[char]; Age:int}
```

In the case that we have multiple S of assignments, we can rewrite the T to now map to new s' that includes x, in the new st' environment and the rest of declaration statements

$$\text{STRUCT}\frac{}{\langle \text{struct } T := \{x : T'; S\}, sEnv, st\rangle \Rightarrow_S \langle \text{struct } T := \{S\}, sEnv, st'\rangle}$$
where $st' = st[T \mapsto s'], st(T) = s, s' = s \cup x]$

In the case that we have multiple S of assignments, we can rewrite the T to now map to new s' that includes x, in the new st' environment

$$\text{STRUCT}\frac{}{\langle \text{struct } T := \{x : T\}, sEnv, st\rangle \Rightarrow_S \langle sEnv, st'\rangle}$$
where $st' = st[T \mapsto s'], st(T) = s, s' = s \cup x]$

When initialising the struct a, a symbol $a.f$ is created that has the value of x', where x' is x evaluated, but not finished. Then, the struct a is assigned to the rest of the struct initialisation S.

$$\text{STRUCT}\frac{x \Rightarrow_a x' \qquad sEnv(a) = \langle (S_2), s\rangle}{\begin{array}{c}\left\langle \text{let a} := (f := x; S), sEnv\right\rangle \Rightarrow_S \\ \left\langle \text{let a} .f := x'; \text{let a} := (S), sEnv[a \mapsto \left\langle (S_2; f := a.f), s\right\rangle]\right\rangle\end{array}}$$

The rule is the same as above, but the value b is now evaluated to a boolean value.

$$\text{STRUCT}\frac{b \Rightarrow_b b' \qquad sEnv(a) = \langle (S_2), s\rangle}{\begin{array}{c}\left\langle \text{let a} := (f := x; S), sEnv\right\rangle \Rightarrow_S \\ \left\langle \text{let a} .f := b'; \text{let a} := (S), sEnv[a \mapsto \left\langle (S_2; f := a.f), s\right\rangle]\right\rangle\end{array}}$$

In this rule, the last field assignment in the struct initialisation is evaluated.

$$\text{STRUCT}\frac{sEnv(a) = \langle (S_2), s\rangle}{\begin{array}{c}\left\langle \text{let a} := (f := x), sEnv\right\rangle \Rightarrow_S \\ \left\langle \text{let a} .f := x, sEnv[a \mapsto \left\langle (S_2; f := a.f), s\right\rangle]\right\rangle\end{array}}$$

This rule is similar to the first struct rule, but the difference is that the symbol x is a Statement S_2 in the symbol environment.

$$\text{STRUCT}\frac{sEnv(x) = \langle (S_2), s\rangle \qquad sEnv(a) = \langle (S_3), s\rangle}{\begin{array}{c}\left\langle \text{let a} := (f := x; S_1), sEnv\right\rangle \Rightarrow_S \\ \left\langle \text{let a} .f := (S_2);\ \text{a} := (S_1), sEnv[a \mapsto \left\langle (S_3; f := a.f), s\right\rangle]\right\rangle\end{array}}$$

This rule is the same as the above, but here, the struct assignment is done evaluating.

$$\text{STRUCT}\frac{sEnv(x) = \langle (S_2), s\rangle \qquad sEnv(a) = \langle (S_3), s\rangle}{\begin{array}{c}\left\langle \text{let a} := (f := x), sEnv\right\rangle \Rightarrow_S \\ \left\langle \text{let a} .f := (S_2), sEnv[a \mapsto \left\langle (S_3; f := a.f), s\right\rangle]\right\rangle\end{array}}$$

In this rule, the field f being assigned in the struct, has the value of a struct. This allows for nested structs.

$$\text{STRUCT}\frac{sEnv(a) = \langle (S_2), s\rangle}{\begin{array}{c}\left\langle \text{let a} := (f := (S_2); S_1), sEnv\right\rangle \Rightarrow_S \\ \left\langle \text{let a} .f := (S_2);\ \text{a} := (S_1), sEnv[a \mapsto \left\langle (S_2; f := a.f), s\right\rangle]\right\rangle\end{array}}$$

This rule is similar to the rule above, but the difference is that this time, there are no more fields to initialise.

$$\text{STRUCT}\frac{sEnv(a) = \langle (S_2), s\rangle}{\left\langle \text{let a } := (f := (S_2)), sEnv\right\rangle \Rightarrow_S}$$
$$\left\langle \text{let a } .f := (S_2), sEnv[a \mapsto \left\langle (S_2; f := a.f), s\right\rangle]\right\rangle$$

The elements can be of any type defined in T

$$\text{STRUCT}\frac{E[s \mapsto (e_1 : \text{T}_1; e_2 : \text{T}_2; ...; e_n : \text{T}_n) \to ok] \vdash S : ok}{E \vdash \text{struct s} := \{e_1 : \text{T}_1; e_2 : \text{T}_2; ...; e_n : \text{T}_n\}; S : ok}$$

Initialising Structures

Structs can be initialised and assigned to symbols using either a constant assignment or a variable assignment. Structs initialised as a constant assignment cannot change any of the fields of the structs; the struct is immutable. Structs initialised as a variable assignment can change all of its fields at any time; the struct is mutable.

The syntax for initialising a struct is as follow.

⟨*StructLiteral*⟩ ::= '(' (⟨*Reassignment*⟩';')* ')' (':' ⟨*Identifier*⟩)?

With concrete examples for immutables:

```
let Alice:Person := (Name := "Alice"; Age := 20);
```

And for mutable struct is as follow.

```
var Alice:Person := (Name := "Alice"; Age := 20);
```

And for usage in lists.

```
[(Name := "Alice"; Age := 20):Person];
```

Access to Structure Fields

Fields can be access using the following syntax.

```
Alice.Name; // "Alice"
```

Structs declared as mutable can have fields reassigned using the following syntax.

```
Alice.Name; // "Alice"
Alice.Name := "Bob";
Alice.Name; // "Bob";
```

In the case that we have multiple S of assignments, we can rewrite the s to now having an accessor that maps to value of the first assignment and the rest of pending assignments in s

$$\text{STRUCT}\frac{}{\langle \text{let } s : T := (f := x; S), sEnv, st\rangle \Rightarrow_S \langle s.f := x; \text{let } s : T := (S), sEnv, st\rangle}$$

In the case that we have only one assignment, we can rewrite the s to now having an accessor that maps to value of the assignment

$$\text{STRUCT}\frac{}{\langle \text{let } s : T := (f := x), sEnv, st\rangle \Rightarrow_S \langle s.f := x, sEnv, st\rangle}$$

Each e of type t is matching the declared struct types, evaluates to the type of the declared struct.

$$\text{STRUCTLITERAL}\frac{E \vdash (e_1 : \mathrm{T}_1; e_2 : \mathrm{T}_2; ...; e_n : \mathrm{T}_n) : \mathrm{T}'}{E \vdash (e_1; e_2; ...; e_n) : \mathrm{T}' : \mathrm{T}'}$$

Comparison of Structs

$$\text{STRUCT}\frac{E \vdash e_1 : \mathrm{T} \quad E \vdash e_2 : \mathrm{T}}{E \vdash e_1 = e_2 : \text{bool}}$$

5.3.4 For-loop

A for-loop iterates through a list of elements.

The for loop statement follows this grammar:

⟨*ForIn*⟩ ::= 'for' ⟨*Identifier*⟩ 'in' ⟨*List* | *Identifier*⟩ ⟨*Block*⟩

A concrete example:

```
sum:int = 0;
for i:int in [0..10]:[int] { sum = sum + i }
```

Here follows the semantics for the for-loop statement:

In the case that the list contains more than one element, the rule can be rewritten as x now being the head of the list, the statements of the block and the for-loop statement again with rest of the list.

$$\text{FORIN}_1\,\frac{}{\langle \text{for x in}[n_1,\ldots,n_m]\{S\}, sEnv\rangle \Rightarrow_S \langle x := n_1; \{S\}; \text{for x in}[n_2,\ldots,n_m]\{S\}, sEnv\rangle}$$

$$\text{FORIN}_2\,\frac{}{\langle \text{for x in}[n_m]\{S\}, sEnv\rangle \Rightarrow_S \langle x := n_m; \{S\}, sEnv\rangle}$$

Initially a parallel for-loop was included in the language, and this is still potentially a desired feature. It was removed however, since the languages focuses on parallelism through actors, and so it was deemed too time consuming to focus on also parallelising statements and expressions.

The element i of the ForIn construct must be of the same type as L. The body of the ForIn-loop can be of any type defined in T$'$

$$\text{FORIN}\,\frac{E \vdash t : \text{T} \qquad E \vdash L : [\text{T}] \qquad E \vdash e : \text{T}'}{E \vdash \text{for } (t \text{ in} L)\ e : ok}$$

Loop Expressions

For loops can only work on lists. The counter variable is the type of a element in the list being iterated.

$$\text{FOR}\,\frac{E \vdash l : [\text{PT}]}{E \vdash \text{for x in } l : ok}, E \vdash x : \text{PT}$$

5.3.5 While-loop

The while-loop statement runs a block of statements until a boolean expression provided returns false.

Syntax

⟨*While*⟩ ::= 'while' ⟨*Expression*⟩ ⟨*Block*⟩

A concrete example:

```
var i:int := 0;
while (i < 10) {i := i + 1}
```

Semantics

In the case that the boolean expression b is true, the rule for composition of statements is used. First, the statement S is evaluated and then the while statement is run again. The statement S may have changed b.

$$\text{WHILE}_{\top}\frac{sEnv \vdash b \Rightarrow_B \top}{\langle \text{while}(b)\{S\}, sEnv\rangle \Rightarrow_S \langle \{S\}; \text{while}(b)\{S\}, sEnv\rangle}$$

In the case that the boolean expression b is false, the while statement has ended, and can simply be rewritten to the environment $sEnv$

$$\text{WHILE}_{\bot}\frac{sEnv \vdash b \Rightarrow_B \bot}{\langle \text{while}(b)\{S\}, sEnv\rangle \Rightarrow_S sEnv}$$

Type Rules

The conditional body of the while construct must be of type bool. The body of the while-loop can be of any type defined in T

$$\text{WHILE}\frac{E \vdash b : \text{bool} \qquad E \vdash e : \text{T}}{E \vdash \text{while } (b)\ e : ok}$$

5.3.6 If-statements

If-statements can be written either as a if-else statement or just as an if-statement. In an if-else statement, the body just after the condition is evaluated if the condition evaluates to the boolean value *true*. If the condition evaluates to *false*, the body after the else keyword is evaluated. In an if-statement, the body after the condition is run, if the condition has the value *true*. If the condition has value *false*, nothing is evaluated.

Syntax

⟨*If*⟩	::= 'if' ⟨*Expression*⟩ ⟨*Block*⟩
⟨*IfElse*⟩	::= 'if' ⟨*Expression*⟩ ⟨*Block*⟩ 'else' ⟨*Block*⟩

A concrete example:

```
// if-then-else statement
if (2 + 2 = 4) {"math works!"}
else {"something is wrong here!"}

// if-statement
if (remainingTime < 10) {initiateCountdown()}
```

Semantics

In the case that the condition b is true, the rule for the if-statement can simply be rewritten to the statement S.

$$\text{IF}_{\top}\frac{}{\left\langle if(b)\{S\}, sEnv\right\rangle \Rightarrow_S \langle\{S\}, sEnv\rangle}, b \Rightarrow_B \top$$

In the case that the condition b is false, nothing is evaluated, and the if-statement is rewritten to the symbol environment $sEnv$.

$$\text{IF}_{\bot}\frac{}{\left\langle if(b)\{S\}, sEnv\right\rangle \Rightarrow_S sEnv}, b \Rightarrow_B \bot$$

In the case that the condition b is true, the rule for the if-else statement is rewritten to S_1.

$$\text{IF-ELSE}_{\top}\frac{}{\langle if(b)\{S_1\}else\{S_2\}, e\rangle \Rightarrow_S \langle\{S_1\}, e\rangle}, b \Rightarrow_B \top$$

In the case that the condition b is false, the rule can be rewritten to S_2.

$$\text{IF-ELSE}_{\bot}\frac{}{\langle if(b)\{S_1\}else\{S_2\}, e\rangle \Rightarrow_S \langle\{S_2\}, e\rangle}, b \Rightarrow_B \bot$$

Type Rules

The conditional body of a if-statement must be of type bool. The body can be of any type defined in T. The type of the if-statement is well-typed.

$$\text{IF}\frac{E \vdash b : \text{bool} \qquad E \vdash e : \text{T}}{E \vdash \text{if}\ (b)\ \{e\} : ok}$$

The conditional body of a if-else-statement must be of type bool. The two bodies can be of any type defined in T. The type of the if-else-statement is well-typed.

$$\text{IF-ELSE}\frac{E \vdash b : \text{bool} \qquad E \vdash e_1 : \text{T} \qquad E \vdash e_2 : \text{T}}{E \vdash \text{if}\ (b)\ \{e_1\}\ \text{else}\ \{e_2\} : ok}$$

5.3.7 Match Statements

Match statements can be seen as syntactical sugar for multiple chained if-statements. The syntax is as follows.

```
match <whatToMatchOn> {
  <case1> -> <actionOnCase1>
  <case2> -> <actionOnCase2>
  ...
  <caseN> -> <actionOnCaseN>
}
```

A concrete example:

```
match (1, 2) {
  (0, n) -> // this case will never be reached
  (1, n) -> print("Case reached!");
  _ -> // default case that matches everything
}
```

Delimitation

Due to other work being deemed more importantly, match-statements will not be further developed. A future improvement to TLDR could possibly include match statements.

5.3.8 Statement Composition

Statements can be sequenced by semicolons. First the statement S_1 is run, but is not finished evaluating.

$$\text{STATEMENTS}_1 \frac{\langle S_1, e\rangle \Rightarrow_S \langle S_1', e'\rangle}{\langle S_1; S_2, e\rangle \Rightarrow_S \langle S_1'; S_2, e'\rangle}$$

In this rule, the statement S_1 is run, and finishes, rewriting the rule to simply be S_2.

$$\text{STATEMENTS}_2 \frac{\langle S_1, e\rangle \Rightarrow_S e'}{\langle S_1; S_2, e\rangle \Rightarrow_S \langle S_2, e'\rangle}$$

5.4 Actors

Here follows descriptions of the usage of actors in TLDR. This includes different principles, functionalities, syntax and semantics. But before we can discuss the use of actors in TLDR, we must first cover the semantics of the parallelism used.

Semantics of Parallelism

Parallelism is achieved through non-determinism in TLDR. The following four rules all have the same left side of the arrow in the conclusion meaning that for the statement

The left side of a parallel statement is executed, but not finished. The environment and actor model is updated when executing S_1.

$$\text{PAR}_1 \frac{\langle S_1, sEnv_1, aEnv\rangle \Rightarrow_S \langle S_1', sEnv_1', aEnv'\rangle}{\langle S_1, sEnv_1, aEnv\rangle \mid \langle S_2, sEnv_2, aEnv\rangle \Rightarrow_S \langle S_1', sEnv_1', aEnv'\rangle \mid \langle S_2, sEnv_2, aEnv'\rangle}$$

The left side of a parallel statement is executed, and finishes. The environment and actor model is updated when executing S_1.

$$\text{PAR}_2\;\frac{\langle S_1, sEnv_1, aEnv\rangle \Rightarrow_S \left\langle sEnv_1', aEnv'\right\rangle}{\langle S_1, sEnv_1, aEnv\rangle \mid \langle S_2, sEnv_2, aEnv\rangle \Rightarrow_S \left\langle S_2, sEnv_2, aEnv'\right\rangle}$$

The right side of a parallel statement is executed, but not finished. The environment and actor model is updated when executing S_2.

$$\text{PAR}_3\;\frac{\langle S_2, sEnv_2, aEnv\rangle \Rightarrow_S \left\langle S_2', sEnv_2', aEnv'\right\rangle}{\langle S_1, sEnv_1, aEnv\rangle \mid \langle S_2, sEnv_2, aEnv\rangle \Rightarrow_S \left\langle S_1, sEnv_1, aEnv'\right\rangle \mid \left\langle S_2', sEnv_2', aEnv'\right\rangle}$$

The right side of a parallel statement is executed, and finishes. The environment and actor model is updated when executing S_2.

$$\text{PAR}_4\;\frac{\langle S_2, sEnv_2, aEnv\rangle \Rightarrow_S \left\langle sEnv_2', aEnv'\right\rangle}{\langle S_1, sEnv_1, aEnv\rangle \mid \langle S_2, sEnv_2, aEnv\rangle \Rightarrow_S \left\langle S_1, sEnv_1, aEnv'\right\rangle}$$

Isolation and Independence

A central principle in TLDR is the use of actors, based on the actor model. Actors are to be seen as entities with interaction. In other words, in order for a construct to qualify as an actor, it must define a way to behave when other actors interact with it. Actors should function independently, and in that regard, not be open to direct manipulation and only able to be changed through the messages it receives. This requirement is due to the wish of separation of processes, which will allow for greater concurrency by letting processes operate on local data instead of global, shared data. Therefore, TLDR tries to encourage natural isolation of functionality through actors, which in turn will also give a greater control of race conditions as no data is ever accessed by more than one process.

The Main Actor

Main is always the first actor, and any program written in TLDR is started with main receiving the arguments message. This is done to force the programmer to start and end the program with an actor, which will better support the actor model perspective. This means that the main actor is started with a message for the arguments, and when the main actor is killed the program will stop executing, whether or not there are still working actors.

The argument message called args, is a struct which consists of an argument counter called argc, and a lists of char lists called argv.

Another way that this affects the programmer, is the idea of spawning actors and having them send messages to main, instead of calling functions and having them return. This also means that there must be a way to reference main, since it is not spawned by another actor. This is solved by the introduction of the “me” keyword, which will evaluate to a reference to the current actor. This is very useful, especially if an actor wishes to delegate work to other actors. The message that is sent with the work, must simply contain a reference back to the delegating actor.

Here are the specific semantics for the main actor.

$$\text{MAIN}\ \frac{input \mapsto \langle S, \epsilon\rangle}{\big\langle \text{receive r} : args := \{S\}, e\big\rangle \Rightarrow_S \big\langle \text{ r} := input; S, e]\big\rangle}$$

Declaration of an Actor

The syntactical declaration of an actor is as follows:

```
actor <identifier> := {
  <functionality>
}
```

And a concrete example could be:

```
actor earth := {
  var temperature:real := 0;
  receive sunlight:light := {
    temperature := temperature + 0.1;
  }
}
```

As shown, the actor keyword precedes the definition, denoting the meaning of said definition. After the keyword, an identifier of the declaration is needed, which will serve as the specific actor type. It is suggested that this identifier reflects the role of the actor in a context of use. Noticeably there is no “let” or “var” keyword in front of the definition, as there usually is when assigning. This is a deliberate choice since “let” and “var” implies interchangeability, which is not an option is this case. If variable declaration of actor definitions were possible, it would effectively be the equivalent of changing the definition of a type on run-time, which would make little sense, and completely undermine the type safety in the language.

Messages sent to an actor will be evaluated based on the type of the message.

$$\text{TYPEOF}\frac{}{sEnv \vdash m \Rightarrow_T t}, \mathbb{T}(m) = t$$

When an actor is declared in a given environment, then the declaration of that actor can be reached by all in that environment. Since the type rules only allow actors to be declared in the global environment, any actor declaration can be reached from anywhere.

$$\text{ACTOR}\frac{}{\langle \text{actor act} := \{S\}, at\rangle \Rightarrow_S \langle at[\text{ act} \mapsto S]\rangle}$$

$$\text{ACTOR}\frac{}{E \vdash \text{actor T} : ok}$$

Basic Actor Functionality

There are four basic functionalities for actors: “spawn”, “die”, “send”, and “receive”, which are all used through keywords. It was desired to keep the syntax of these functionalities different from the syntax of functions. Even though they behave much like functions, taking input and giving output, they are more powerful. For example, regular functions cannot contain a type as a parameter, but the “spawn” functionality does this. Due to this and more differences, it was decided to separate them syntactically.

The “spawn” functionality is used to create new instances of actors. And example could be:

```
actor <identifier> := {
 <functionality>
}

let MyActor:<identifier> := spawn <identifier> <message>;

or alternatively:

var MyActor:<identifier> := spawn <identifier> <message>;
```

As can be seen above, there are four parts of the spawn functionality: an identifier, the keyword, the type of the actor, and optionally an initial message. Firstly, the identifier is preceded by a keyword for mutability, such as “let” or “var”. This allows for the substitution of handles, which provides possibilities of dynamic changes. This however opens up the possibility of “losing contact” with an actor, if the handle is replaced. This could potentially lead to memory leaks if not handled properly.

When spawning a new actor, you can also choose to add a message. The reason for this is to give the programmer a way of initialising the new actor with a certain message. In object-oriented languages this is usually done with a constructor, however doing it via a constructor would conflict with a central principle, since it would mean manipulating an actors state directly.

The semantics of "spawn" is as follows:

The rule for spawning an actor can be rewritten as the send rule. Doing this, the actor environment *aEnv* is changed. The actor handle *act* is added to the actor environment *aEnv*. In addition, the *next* actor handle is updated to be a new unique actor handle.

$$\text{SPAWN}\frac{}{\langle \text{let act} : T := \text{spawn } T \text{ m}, at, aEnv, sEnv\rangle \Rightarrow_S \langle \text{send act m}, at, aEnv', sEnv\rangle}$$

$, aEnv(next) = e, aEnv' = aEnv[act \mapsto e, next \mapsto new(e)]$

Type rules for "spawn":

$$\text{SPAWN}\frac{}{E \vdash \text{spawn T} : \text{T}}$$

After an actor has been spawned, it will be possible to send messages to it. This is done with the "send"-keyword. This can be done as follows:

```
MyMsg:int := 42;

Send MyActor MyMsg;
```

This will send *MyMsg* to the actor *MyActor*. It is also possible for actors to send messages to themselves by using the "me"-keyword. Such messages will be treated the same as any other message.

The semantics of "send" is as follows:

$$\text{SEND}\frac{e_2 = a(act), m \Rightarrow_T t}{\langle \text{send act m}; S, \text{a}, e_1\rangle \Rightarrow_S \langle S, e_1, a\rangle \mid \langle \_t(m), e_2, a\rangle}$$

and the type rules for "send":

$$\text{SEND}\frac{E \vdash m : \text{T} \quad E \vdash a : \text{T}'}{E \vdash \text{send m a} : ok}$$

When an actor is sent a message, it must act according to a defined way of handling that type of message. This is declared with the "receive"-keyword, which creates a method within the actor that is called when the actor receives a corresponding message. The syntax can be seen below:

```
actor <nameOfActor> := {
 receive <nameOfMessage>:<typeOfMessage> := {
  <functionality>
 }
}
```

In this example the receive-method defines the way messages of the type "<typeOfMessage>" are handled. Within the functionality "<nameOfMessage>" is the reference to the message. This message is immutable no matter if it was mutable where it was sent from. This is done to discourage mutating messages.

It is also the intention to include a "wait on <typeOfMessage>" keyword, which will cause the actor to not evaluate the next message in the message queue, but instead traverse the queue, until a message matching "<typeOfMessage" is found. Then that message is de-queued and evaluated. This has not been implemented in the current version of the TLDR compiler, but it will be a central part of supporting discrete simulations.

The semantics of "receive" is as follows:

$$\text{RECEIVE}\frac{}{\langle \text{receive } r : t := \{S\};, e\rangle \Rightarrow_S \langle e[\_t \mapsto \langle S, r\rangle]\rangle}$$

The type rules for "receive" are:

$$\text{RECEIVE}\frac{E \vdash m : \mathrm{T}}{E \vdash \text{recieve m} : \mathrm{T} : ok}$$

When an actor is no longer needed, it is possible to discard it with the "die"-keyword. In other languages, using the actor model, "die", or similar functionality, is usually called by the parent of the actor, that is, the actor that spawned the actor. In TLDR however, "die" can only be called by the actor itself. This is done in order to keep the principle of only interacting with actors through messages, which is a simpler way of handling actors in TLDR, since the language does not have a built-in supervisor functionality as described in *section 4.5.1*.

When an actor dies it stops immediately and does not compute further, and whatever messages might have been in the actors message-queue, will be lost. Here is the semantic rule for *die*:

$$\text{DIE}\frac{}{\langle die; S, aEnv, sEnv, at\rangle \Rightarrow_S at}$$

Actor References

In TLDR declarations must be accomodated by an assignment of a value, since no primitives can have a null value. However, since actors handles have the special position in the language of being the only value which is passed by reference, it should encompass a null value. The reason for this becomes apparent if one considers the following example:

```
actor main := {
  receive arguments:args := {
    var jack:man := spawn man;
  }
}

actor man := {
  var bestMan:man := spawn man;
  }
}
```

In this example, the bestMan variable would result in an endless recursive spawn chain of actors. This is the most obvious problem, but there are multiple senarioes where forced value assignment for reference types become problematic. Due to this, the language allows for null references.

Comparison

When comparing actors, they are considered equal, if they are reference equals.

Here are the type rules for comparison:

$$\text{ACTOR}\frac{E \vdash e_1 : \mathrm{T} \qquad E \vdash e_2 : \mathrm{T}}{E \vdash e_1 = e_2 : \mathrm{bool}}$$

6 Compiler

In this chapter, the implementation of the TLDR compiler will be described. Firstly, a general overview of the phases of the compiler will be depicted. Next, the method for traversing the abstract syntax tree will be described and the choice of target language is presented. Lastly, specifics and inner workings of each phase of the compiler will be specified.

6.1 Phases

The compiler described in this report, is divided into three phases; parsing, analysis and code generation. This section aims to give an outline of how the different phases interact. Each of these phases will be described in greater detail in the following sections.

If a phase fails, the compilation stops and the error is reported to the user. If a phase succeeds, the next phase is started.

6.1.1 Parsing

Upon start of compilation, the source file of a given program written in TLDR is loaded into memory. Then, the source code is parsed by the parser as will be described in *section 6.4*. The parser constructs an abstract syntax tree [14], containing the language constructions used in the TLDR source program, by forming tokens based on the grammatical rules described in *chapter 5*.

The parsing phase is now done, and the abstract syntax tree can be processed by the analysis phase.

6.1.2 Analysis

The analysis phase checks for categories of semantic errors that are defined as being invalid in TLDR. These errors cannot be checked by the syntactic rules of the grammar enforced by the parser, and therefore must be checked in the analysis phase. The analysis phase uses the abstract syntax tree given from the parser to do these checks. This includes decorating the abstract syntax tree to include additional information, or by simply traversing the tree for information.

Once the analysis phase accepts the TLDR program, the code generation phase is run.

6.1.3 Code Generation

The code generation phase transforms the abstract syntax tree, produced by the parser and later checked for invalid operations, into LLVM IR. The phase traverses the abstract syntax tree, and for each language construction, the code generator constructs LLVM IR code that reflects the semantics of the language as described in *chapter 5*.

Once the code generation has succeeded, the Clang compiler is invoked to transform the LLVM IR code into an executable file.

6.2 AST Traversal

This section describes how the AST is operated on during the different phases of the compiler. As an example, let us look at the code generation phase. The phase is essentially a function that takes an AST and returns a string of characters representing the generated target code. The function must traverse all the nodes in the AST. As seen in *listing 6.5*, the AST is recursively defined. The root of the AST is “Program”, containing a list of ASTs. The traversal is done by using pattern matching on the AST. In the case of the code generation function, if the AST is a “Program” node, the code generation function is simply applied to all subtrees contained in the data type, and the resulting generated target code is merged together. A simplified example of the pattern matching can be seen in *listing 6.1*.

Listing 6.1: Example of traversing AST

```
let rec codeGen (ast:AST) : string =
  match ast with
  | Program stms -> // apply codeGen to all stms and merge
      result
  | Constant (type, value) -> // generate the code for the
      constant based on the type and value of the constant.
```

6.3 Target Language

The target language is the language that the compiler translates the source language into. In this section the choice of target language will be described.

6.3.1 Choice of Target Language

Different language types were considered as the target language of choice: assembly languages such as Java bytecode, CLI and LLVM IR, and languages just a level higher than assembly languages, such as C.

Languages with higher levels of abstractions were not an opportunity because the authors of this report did not want to create a “search-and-replace” compiler. Such a compiler could for example translate a *until* construct in a language to the *while* loop of C. Such a compilation was deemed too trivial for the academic context of this project.

The following target languages were considered:

C

C is the only high level language considered, because of its complexity, due to the the lack of memory management. C has a lot of libraries, is a well known language and is more readable by humans than an assembly language. C was not chosen because C compilers interpret data types in different ways on different platforms, meaning program correctness is dependent on the platform it is run on.

Java Bytecode

Java bytecode runs on any Java Virtual Machine (JVM), which makes the language highly cross-platform. Java bytecode is a instruction and stack based language. Because of its maturity lots of documentation is available. Java bytecode has large set of framework and libraries, because many languages run on the JVM. Many popular implementations exist for the actor model; for example Akka. Scala and Clojure are both examples of languages built upon the JVM. Many implementations of JVMs support garbage collection. Java bytecode was not chosen as the language of choice because its strong connection to the object oriented paradigm. A more paradigm neutral language is preferred, in order to not taint the design of TLDR.

Common Intermediate Language

Common Intermediate Language (CIL) is an intermediate language, C#, F# and many more languages target. Common Intermediate Language and Common Language Infrastructure (CLI) have many things in common with Java bytecode and the JVM, including support for garbage collection. The specification for CLI is open, so implementations can be made for many platforms. Microsoft has implemented its own CLI implementation called .NET Framework. Support for Linux is given by the Mono implementation. CIL was not chosen for the same reasons as Java bytecode.

LLVM

LLVM is a register and instruction based language very similar to other assembly languages, such as X86. Register allocation is handled automatically, so arbitrarily many registers can be used in LLVM IR source language. LLVM has support for both explicitly freeing memory and in recent versions gained support for garbage collection hooks. The language reference is very comprehensive and easily understood. LLVM has a large set of libraries which are compatible, because many languages can be compiled to LLVM IR. Frontend compilers to LLVM exist for C, C++, Java and many others. LLVM IR is compiled to machine code for architectures that have a LLVM backend written for it.

Final Choice

LLVM was chosen because of its many libraries and very good documentation. It also has a lot in common with x86 assembly which the group has had prior experience with.

6.4 Lexing and Parsing

Lexing is the process of transforming a string of characters to a set of tokens. A token is a character or a sequence of characters. The tokens are in this report defined as a context free grammar (CFG). This grammar defines how a valid program is allowed to be written syntactically. For example in the program "if(true)", the tokens are *if*, *(*, *true* and *)*.

Parsing is the process of transforming the sequence of tokens, created in the lexing process, into a data structure to be used in later stages of the compiler.

6.4.1 Compiler-Compiler

A compiler-compiler tool is used in this project instead of a hand-written lexer and parser, because of the conveniences these kind of tools give. By using a compiler-compiler, code that generates a syntax tree from a input source file, is generated automatically from a separate grammar specification file. This separation of grammar specification and the actual code that parses the input source file, is deemed beneficial in the implementation of a compiler, because of the belief that it is easier to maintain the grammar specification rather than the code for a hand written parser. The focus of the project being able to experiment with syntax for expressing various constructs in a parallel manner, choosing a tool that supports the largest domain of languages and not having to rewrite the parser if a prefered way of writing grammar is deemed needed.

The choice of a compiler-compiler is based on a series of criteria, listed below with the corresponding reasons for the criteria.

1. **Output language - C# or F#** It was chosen to construct the compiler in a language compatible with the common language infrastructure (CLI), since

the language developers had previous experience with languages running on this. This makes C# and F# preferred target languages for the compiler-compiler.

2. **Type of parser - LR or derivatives** The compiler-compiler should support LR parsing since the authors of this project value left recursion in a grammar specification.

Over the years, many compiler-compiler tools have been developed that satisfy the criteria listed above. Although no tool stood out as being superior, a choice had to be made anyway. In the end, Hime was chosen based on its documentation, frequent rate of updates and its spirit of being developed with the features of C# in mind. Hime also support ENBF, which allows for more constructs in the grammar.

Hime [15] implements multiple *LR* parsing methods, for this project the LALR(1) algorithm is chosen, this is a simplification of a LR(1) parser, but yet almost as expressive. The LARL(1) is less powerful in the language domain than the LR(1), in that it can produce a *reduce/reduce conflict* due to not knowing the left context as well as LR(1). The parse table of LR(1) includes duplicates of the same "core" states (same productions and same positions) in its parse table, where LALR(1) does not, but this can potentialy greatly increase the number of entries of states in the parse table, and thereby the usage of memory. This was not a requirement but merely a convenience.

Using Hime, the amount of work required in developing the lexing and parsing phase is greatly reduced, as the lexing and parsing phase is done by Hime.

6.4.2 Tree Transformations

The output of running Hime on a grammar is a syntax tree. By default, the output syntax tree matches the grammar rules one-to-one. That is, the syntax tree is a concrete syntax tree (CST). In the later stages of the compiler, many nodes of the CST are unneeded. *Listing 6.2* lists two types of nodes, not needed in the final abstract syntax tree (AST), adding an extra layer of indirection and altogether superfluous nodes.

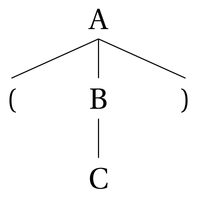
Listing 6.2: Starting example grammar

```
A -> ( B )
B -> C
```

Listing 6.2 corresponds to the CST seen in *fig. 6.1*.

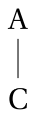
In *fig. 6.1*, the node *B* is not needed as it is simply an alias for *C*. It would be preferable to just have *A* as the root, with *C* as a child of *A*, in the final syntax tree

Figure 6.1: *CST without tree transformations.*



to be used in later compiler phases. In addition, the parentheses do not add any information to the final syntax tree. The desired AST can be seen in *fig. 6.2*.

Figure 6.2: *The desired AST.*



To solve these two problems, the promote and drop actions can be utilised.

Promote Action

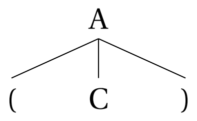
The promote action, when applied to a node, replaces its parent node with itself.

The promote action can be applied to *C* in *fig. 6.1* as seen in *listing 6.3* to achieve the desired AST.

Listing 6.3: Applying a promote action to the grammar

```
A -> ( B )
B -> C^
```

Figure 6.3: *The result of applying the promote action to* C.



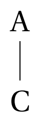
Drop Action

The drop action removes the node it is applied to, and all of its children, entirely from the syntax tree. This can be utilised in *fig. 6.4* to remove the two parentheses that have no meaning to the compiler. Continuing from the result of applying the promote action, the drop action can be applied to the parentheses as seen in *listing 6.4*.

Listing 6.4: Applying a drop action to grammar

```
A -> (! B )!
B -> C
```

Figure 6.4: *The result of applying the drop action to* (*and*).



Using both the promote and drop actions, the CST has been transformed into an AST with only the relevant information in it.

6.4.3 Improving AST Representation

The feature of tree transformations allow for stripping a CST to a AST, as described in *section 6.4.2*. The Hime AST generated by applying tree transformations contains all the information needed for later phases of the compiler. However, every node in the AST is "stringly typed", meaning that every information is encoded as strings. While containing the information, a more optimal and safe encoding can be made using the discriminated union feature of F#. It would be ideal to convert the stringly typed Hime AST to an AST with different types, encoding different language constructs. By doing this conversion, the AST can be described more precise by using types that the F# compiler can statically verify as being used correctly, hereby avoiding a large range of potential implementation bugs in the compiler described in this report.

To do this conversion from the Hime AST to the more optimal typed representation, a function, "toAST", converts every known node string representation to the stronger typed desired representation. The AST data type used throughout the compiler can be seen in *listing 6.5*.

The AST data type includes all the data needed to perform all the phases of the compiler. Every language construct in TLDR is a case of the union type of the AST data type. The AST is, as the name implies, a tree. It is therefore defined recursively. For example, the "if" case has the value of a tuple containing two subtrees: the AST for the condition and the AST for the body.

Listing 6.5: Data type of AST used throughout the compiler

```
data AST =
    | Program of AST list
    | Block of AST list
    | Body of AST list
    | Reassignment of Identifier * AST // varId, rhs
    | Initialisation of LValue * AST // lvalue, rhs
    | Constant of PrimitiveType * PrimitiveValue // type, value
    | Actor of string * AST // name, body FIXME: Add more fields?
    | Struct of string * (TypeDeclaration list) // name, fields
    | If of AST * AST // conditional, body
    | IfElse of AST * AST * AST // conditional, trueBody, falseBody
    | Send of string * string * AST // actorHandle, actorToSendTo, msg
    | Spawn of LValue * (string * AST option) option // lvalue, (actorName, initMsg) or uninit spawn
    | Receive of string * PrimitiveType * AST // msgName, msgType, body
    | ForIn of string * AST * AST // counterName, list, body
    | While of AST * AST // condition, body
    | List of AST list * PrimitiveType // content, type
    | BinOperation of AST * BinOperator * AST // lhs, op, rhs
    | UnaryOperation of UnaryOperator * AST // op, rhs
    | Identifier of Identifier * PrimitiveType // id, typeOfId
    | Function of string * string list * PrimitiveType * AST // funcName, arguments, types, body
    | StructLiteral of AST * (string * AST) list // struct, (fieldName, fieldValue) list
    | Invocation of string * string list * PrimitiveType // functionName, parameters, functionSignature
    | Tuple of AST list * PrimitiveType // Entries
    | Return of AST option // body
    | Die
```

6.5 Static Analysis

This section will describe what the purpose of the different parts of the static analysis is and how it is performed in the TLDR compiler. The static analysis part of the compiler has the following phases:

- Build symbol table
- Checking illegal reassignment
- Checking use before declaration
- Checking hiding
- Checking types

Each of these will be explained in the following sections.

6.5.1 Symbol Table

An important aspect of the static analysis in the TLDR compiler is the construction of a symbol table, a table which contains relevant information about the symbols, that are used in the program that is analysed. In *listing 6.6*, the definition of an entry in the symbol table can be seen.

Listing 6.6: The definition of an entry in the symbol table.

```
[<ReferenceEquality>]
type SymbolTableEntry = {
  symbol: LValue
  statementType:StatementType
  scope:Scope
  value:AST
}
```

In the entry, “symbol” is a type which contains the relevant information regarding the identity of the symbol, that is, the identifier, the declared type and whether or not the symbol is mutable.

“statementType” in the entry defines whether the specific use of the symbol is a usage, an initialisation or a reassignment.

“scope” specifies the unique scope of the symbol usage.

“value” is the actual value of the symbol usage. In an assignment, this would be set to the right hand side of the assignment operator.

6.5.2 Checking Illegal Reassignment

Checking illegal reassignment, that is, reassignment of an immutable symbol, is fairly trivial. As can be seen in *listing 6.7*, the function simply takes the symbol table as an argument and reports a failure whenever an entry in the table is both a reassignment and immutable. The pattern matching at the end of the function is performed to summarise any failures that were reported during execution.

Listing 6.7: The function responsible for checking illegal reassignments.

```
let checkReass symTable =
  let res = symTable |>
    List.filter (fun entry -> entry.statementType = Reass &&
        entry.symbol.isMutable = false) |>
      List.map (fun e -> Failure [sprintf "Invalid
          reassignment, %A is not mutable" e.symbol.identity
          ])
  match res with
  | [] -> Success symTable
  | xs -> sumResults xs
```

6.5.3 Checking Use Before Declaration

Checking usage of a symbol before its declaration is similarly trivial. This function takes advantage of the fact that in the function that builds the symbol table, only symbol usages, without prior initialisation or declaration, and definitions are given the type "HasNoType". As seen in *listing 6.8* this function simply sorts out entries which are definitions and reports a failure for every entry with the type "HasNoType".

Listing 6.8: The function responsible for checking usage before declaration.

```
let checkUsedBeforeDecl symTable =
  let res = symTable |>
    List.filter (fun e -> e.statementType <> Def && e.symbol.
       primitiveType = PrimitiveType.HasNoType ) |>
      List.map (fun e -> Failure [sprintf "Symbol \"%A\" is
         used before its declaration." e.symbol.identity])
  match res with
  | [] -> Success symTable
  | xs -> sumResults xs
```

6.5.4 Checking Hiding

In order to check if any symbol hides another, the function seen in *listing 6.9* is used. Hiding can never be done by a reassignment or usage of a symbol, so these are filtered out at first. We then iterate through the remaining entries in the symbol table and for each entry, report a failure if we can find another symbol with the same identifier in its scope.

Listing 6.9: The function responsible for checking hiding.

```
let checkHiding symTable =
  let mutable res:Result<SymbolTable> list = []
  let NoReass = symTable |> List.filter (fun e -> e.
     statementType <> Reass && e.statementType <> Use)
  for i in NoReass do
    let seq = List.filter (fun e -> e <> i && isVisible i e
       && i.symbol.identity = e.symbol.identity && i <> e)
       NoReass
    if seq.Length > 0 then
      res <- Failure [sprintf "Element %A hides %A" i.symbol.
         identity seq] :: res

  match res with
  | [] -> Success symTable
  | xs -> sumResults xs
```

6.5.5 Checking Types

In order to check that the type of the value of every symbol matches the declared types, the function shown in *listing 6.10* is used. This function calls another function, "checkTypesAST" which takes an AST node and returns the type of that node. As seen in *listing 6.6* the value of a symbol table entry is an AST node. The type checker can therefore get the type of every symbol inferred from their value and see if that matches the declared type, which is also stored in the symbol table, and report a failure if the two types do not match.

Listing 6.10: The main function responsible for checking types.

```
let checkTypesSymTable symTable =
  let results = List.map (fun entry ->
                    match checkTypesAST entry.value with
                    | Success pType ->
                      if entry.symbol.primitiveType = pType
                         then
                        Success ()
                      else
                        Failure [sprintf "symbol %A expected
                            to have type %A, but has type %A"
                            entry.symbol.identity entry.symbol
                            .primitiveType pType]
                    | Failure err -> Failure err) symTable
  if results.Length = 0 then
    Success ()
  else
    sumResults results
```

6.6 Code Generation

This section describes the code generation phase of the compiler described in this report. The focus is on the overall aspect of transforming an abstract syntax tree to the various language constructs in TLDR, however not every language construct will have its code generation described.

The purpose of the phase is to output code that can later be run on the computer as an executable. LLVM was chosen as the target language for this compiler in *section 6.3*, so this is what the code generation phase will output. The generated LLVM IR is then compiled into native code by using the Clang compiler.

6.6.1 Data Sizes

Even though the language design of TLDR requires integers and reals to be of arbitrary precision as described in *chapter 5*, the code generation phase generates integers and reals of 64-bit size. However, one way of adding arbitrary precision arithmetic to the compiler would be to use a library such as GMP [16].

6.6.2 Actors

Actors are implemented using the C library “libactor” [17]. Using this library can will reduce the development time required to implement actors. However, as the library is third party, some control over the functionality is lost. This also means that there might be differences in the semantics between “libactor” and the semantics for actors as described in *section 5.4*, since the project group does not have in-depth knowledge about the implementation of the library.

Every actor in TLDR is generated into an LLVM function with the following behaviour outlined in pseudo code.

```
active = true
msg = null
<initialisation of fields in actor>
start:
while (active) do
  msg = actor_receive()
  switch (msg.type) do
  case 101:
    goto recv_kill
  case 102:
    goto recv_102
  case 103:
    goto recv_103

return

recv_102:
<body of first receive>
goto start
recv_103:
<body of second receive>
goto start
recv_kill:
active = false
goto start
```

An actor simply checks for new messages, and performs the corresponding function, based on the type of the message, by jumping to a label. This while loops ends when the actor receives a kill message.

By having receive constructs simply be labels, all data of an actor can be contained in a single function.

6.6.3 Lists

Lists in TLDR are simply generated as static LLVM arrays. This mean that their size is known at compile time, and that no elements can be added at run-time. This is a limitation in the current compiler.

One way of improving the current implementation is to allow for arrays with sizes determined at run-time. This can be done by allocating a pointer to the elements with the size of the array, known at runtime.

6.6.4 Optimisations For Free

Even though speed has not been a primary goal of this implementation of TLDR, many optimisations come for free by generating LLVM IR. The code generation implemented in this project is very naïve and thus does not do any optimisations in its generation. However, many optimisation passes are developed in the LLVM project. These optimisations can be applied very easily to our naïve code generation, simply by running an LLVM optimiser on the LLVM IR generated by the compiler described in this report.

As an example, let us try and compile the following TLDR code to LLVM IR.

```
actor main := {
  receive arguments:args := {
    let res:int := (2 + 3) * 4 / 2;
    printint(res);
    die;
  };
};
```

The code generation generates the following LLVM IR.

```
declare double @llvm.pow.f64(double , double)
declare double @llvm.powi.f64(double , i64)
declare i32 @puts(i8*)
declare i32 @printf(i8*, ...)
declare void @actor_init(...)
declare void @actor_wait_finish(...)
declare void @actor_destroy_all(...)
%struct.actor_message_struct = type { %struct.
    actor_message_struct*, i64, i64, i64, i8*, i64}
%struct.actor_main = type { i32, i8** }
declare void @exit(i32)
declare i64 @spawn_actor(i8* (i8*)*, i8*)
declare void @actor_send_msg(i64, i64, i8*, i64)
declare %struct.actor_message_struct* @actor_receive(...)
```

```
@g1 = constant [4 x i8] c"%d
\00"
define i32 @main(i32 %argc, i8** %argv) {
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i8**
  store i32 0, i32* %1
  store i32 %argc, i32* %2
  store i8** %argv, i8*** %3
  call void (...)* @actor_init()
  %4 = call i64 @spawn_actor(i8* (i8*)* bitcast (i8* ()*
     @_actor_main to i8* (i8*)*), i8* null)
  call void (...)* @actor_wait_finish()
  call void (...)* @actor_destroy_all()
  call void @exit(i32 0)
  unreachable
  %6 = load i32* %1
  ret i32 %6
}
define i8* @_actor_main() {
%_active = alloca i1
store i1 true, i1* %_active
%_msg = alloca %struct.actor_message_struct*
%res = alloca i64
%_1 = add i64 2, 3
%_2 = mul i64 %_1, 4
%_3 = sdiv i64 %_2, 2
store i64 %_3, i64* %res
%_5 = load i64* %res
%_4 = getelementptr [4 x i8]* @g1, i64 0, i64 0
call i32 (i8*, ...)* @printf(i8* %_4, i64 %_5)
store i1 false, i1* %_active
br label %start
br label %start
start:
br label %switch_6
switch_6:
%_6 = load i1* %_active
br i1 %_6, label %body_6, label %cont_6
body_6:
%_7 = call %struct.actor_message_struct* (...)* @actor_receive
    ()
store %struct.actor_message_struct* %_7, %struct.
    actor_message_struct** %_msg
%_8 = load %struct.actor_message_struct** %_msg
%_9 = getelementptr %struct.actor_message_struct* %_8, i32 0,
    i32 3
%_10 = load i64* %_9
switch i64 %_10, label %start [  ]
br label %switch_6
cont_6:
ret i8* null
ret i8* null
}
```

After running a LLVM IR optimiser on the above LLVM IR, the result is as follows.

```
@g1 = constant [4 x i8] c"%d\0A\00"
; Function Attrs: nounwind
declare i32 @printf(i8* nocapture readonly, ...) #0
declare void @actor_init(...)
declare void @actor_wait_finish(...)
declare void @actor_destroy_all(...)
declare void @exit(i32)
declare i64 @spawn_actor(i8* (i8*)*, i8*)
; Function Attrs: noreturn
define i32 @main(i32 %argc, i8** nocapture readnone %argv) #1 {
  tail call void (...)* @actor_init()
  %1 = tail call i64 @spawn_actor(i8* (i8*)* bitcast (i8* ()*
     @_actor_main to i8* (i8*)*), i8* null)
  tail call void (...)* @actor_wait_finish()
  tail call void (...)* @actor_destroy_all()
  tail call void @exit(i32 0)
  unreachable
}
define noalias i8* @_actor_main() {
cont_6.critedge:
  %0 = tail call i32 (i8*, ...)* @printf(i8* getelementptr
     inbounds ([4 x i8]* @g1, i64 0, i64 0), i64 10)
  ret i8* null
}
attributes #0 = { nounwind }
attributes #1 = { noreturn }
```

The optimiser (run with “opt -O3 -S”) applies constant folding on $(2+3)*4/2$ such that this expression is evaluated on compile time and 10 is stored. It also removes all unnecessary operations. For example, all the actor specific code, such as the active flag, is not present in the optimised LLVM IR code. While this seems simple, no extra work had to be done on the development of the TLDR compiler, and significant optimisations were achieved.

Running Clang (run with “clang -O3 -lactor -S”) on this optimised LLVM IR, results in the following X86-64 assembly.

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 10
    .globl  _main
    .align  4, 0x90
```

```
_main:                                  ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
Ltmp0:
    .cfi_def_cfa_offset 16
Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Ltmp2:
    .cfi_def_cfa_register %rbp
    xorl    %eax, %eax
    callq   _actor_init
    leaq    __actor_main(%rip), %rdi
    xorl    %esi, %esi
    callq   _spawn_actor
    xorl    %eax, %eax
    callq   _actor_wait_finish
    xorl    %eax, %eax
    callq   _actor_destroy_all
    xorl    %edi, %edi
    callq   _exit
    .cfi_endproc

    .globl  __actor_main
    .align  4, 0x90
__actor_main:                           ## @_actor_main
## BB#0:                                ## %switch_6
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    _g1(%rip), %rdi
    movl    $10, %esi
    xorl    %eax, %eax
    callq   _printf
    xorl    %eax, %eax
    popq    %rbp
    retq

    .section    __TEXT,__const
    .globl  _g1                     ## @g1
_g1:
    .asciz  "%d\n"

.subsections_via_symbols
```

6.7 Compiler Status

The compiler described in this report does not fully implement TLDR. This section lists which parts of TLDR is implemented in the compiler, and which parts are not. In addition to that, it presents a sample program written in TLDR.

Implemented:

- Arithmetic operations
- And, or, equality operations on bools
- For-in statement
- Lists as arrays. See *section 6.6.3*
- If and if-else statements
- While loops
- print, printint, printreal functions
- Actors
- Spawn, send, receive and die operations
- Integer, boolean and real number constants
- Struct definition, struct field access and initialisation
- Initialisation and reassignment constructs

Not implemented:

- Functions - Code generation is not implemented
- Dynamically allocated lists. The lists currently implemented are simply arrays. See *section 6.6.3*
- Tuples - Code generation is not implemented
- List operations - Currently lists can only be manipulated by indexing into the list and can be iterated with a for-in-loop
- Data casts - Only supported in grammar
- NAND, NOR and XOR - Code generation not implemented
- Char constant - Code generation not implemented
- Comparison of tuples, structs - Type checking and code generation not implemented
- Send/receive of structs/tuples
- Arguments from command line

6.7.1 Sample Program

The following program demonstrates addition of ints and reals, output of results, and how interaction between actors occur. The output of the program is:

```
5.00000000000000000000
150
```

Listing 6.11: Example program, able to be compiled by the compiler described in this report

```
actor main := {
  receive arguments:args := {
    let msg1:realInputMsg := (x := 2.5; y := 2.5;);
    let msg2:intInputMsg := (x := 50; y := 100;);
    let plusser:plusActor := spawn plusActor;
    let list:[int] := [0];
    send plusser msg1;
    send plusser msg2;
    die;
  };
};

actor plusActor := {
  var messagesReceived:int := 0;
  let messagesToReceive:int := 2;

  receive rMsg:realInputMsg := {
    messagesReceived := messagesReceived + 1;
    let x1:real := rMsg.x;
    let y1:real := rMsg.y;
    let realRes:real := x1 + y1;
    let printer1:printActor := spawn printActor realRes;
    if(messagesReceived = messagesToReceive) {
      die;
    };
  };

  receive iMsg:intInputMsg := {
    messagesReceived := messagesReceived + 1;
    let x2:int := iMsg.x;
    let y2:int := iMsg.y;
    let intRes:int := x2 + y2;
    let printer2:printActor := spawn printActor intRes;
    if(messagesReceived = messagesToReceive) {
      die;
    };
  };
};

actor printActor := {
  receive intMsg:int := {
    printint(intMsg);
    die;
  };

  receive realMsg:real := {
    printreal(realMsg);
    die;
  };
};

struct realInputMsg := {
  x:real;
  y:real;
};

struct intInputMsg := {
  x:int;
  y:int;
};
```

7 Reflection

During the development of TLDR there has been several crossroads that had major impacts on the direction of the language. This chapter will entail some reflections on what could have been done differently, and how it might have changed the development of the language.

7.1 Decisions Without Experience

This is the first language the group members have developed, so a lot of decisions were made without the wisdom of experience. Many times this resulted in decisions which seemed right at first, but turned out to have ripple effects, which were not foreseen. One such choice was the omittance of a null value. This was wanted, since we wanted TLDR to not require checks for the null value every time a variable was being used. This decision caused TLDR to require the programmer to assign a value whenever something was declared. Later this proved problematic since certain common usages of actors would require you to spawn an actor, only to kill it immediately, if all you wanted was a handle, generating a massive overhead. Even when doing so, the null problem was still around, since an actor could still send messages to actors whom where killed, which essentially is a null reference. This choice and thereby the reversion could probably have been avoided, if the group had had more experience with language design.

7.2 Language Design Process

The language design was done democratically, with voting where an agreement could not be reached. Sometimes this process resulted in a compromise between two ideas, since this seemed most fair. However, as was learned, language design should not be about fairness for the developers of the language, but what is best for the language. And even though all the ideas might have worked in their own regard, a mix of different ideas may very well turn out to be two halfway working functionalities, and a loss of the principles behind the idea in the first place. In any future language design project the group members might end up in, we will better understand the importance of having a truly united direction for the language.

During the design process it also became obvious to us, just how subjective the quality of a language can be. Two group members might have discussed the inclusion of a feature in the language, arguing for and against, only to find out that they wanted to improve readability, but had different ideas on what makes a language readable. If we were to develop further on TLDR, we would probably try and gather a focus group, matching the target audience of the language, since it would ultimately be their opinion that counts.

TLDR was developed with the waterfall design method. This method, or at least a sequential design method, was prefered due to the progression of the semester courses. Even so, a couple of aspects of the development process is worth some reflection as to what would have been different, had we followed an iterative design method. But first, a few consequences of the chosen model.

The waterfall method works well with language design on the point of language principles. If the design of the language is done without much thought as to the implementation, the language will be more true to the principles, and cut less corners due to hardship during implementation. This is great, since it allows for pure design, as long as the design ideas are not directly impossible to implement. The method can, due to this freedom, potentially increase the difficulty of implementing a compiler which will compile the language. In addition, some features might seem like a good idea during language design, but might be less desirable by the programmer than expected.

So what could have happened with an iterative design method? Likely an iterative process would have been better for discovering design flaws earlier, allowing programs to be written in the language earlier, albeit with a more limited set of features. Writing more in the language could also have given a better idea of which features were important from a programmers perspective. An iterative method would also allow for testing which constructs were possible to implement, before the decision is set in stone. In hindsight, an iterative design method, would probably be preferred.

8 Discussion

This chapter discusses TLDR as a language, the features not implemented in the compiler, and how the language can be further improved in the future.

8.1 Language

The Language Described in this report is a domain specific programming language, borrowing ideas and constructs from other languages. This section will discuss some of these ideas and constructs, and describe TLDRs rationale and usefulness in the domain of programming languages.

8.1.1 Distinguishing Features

One of the main features, and the pièce de résistance of TLDR, is the central role of actors. Actors allow the programmer to model concurrent entities using a fairly simple construct. As opposed to other languages which incorporate actors, actors in TLDR is an independent construct, where in Erlang an actor is a function and Scala (Akka) where actors are just objects. Having actors as a unique construct in TLDR gives the programmer a clue as to the power which actors bring to the language.

Another difference between TLDR and other languages with actors is that TLDR enforces the use of actors. Every program has at least one actor, main, where the execution of the program starts. This enforces the principle of a central controller in a network of concurrent processes. This means that any TLDR program can be considered, and reasoned about, as a graph-like structure rooted in the main actor, with vertices representing actors and edges representing messages between them.

8.1.2 Paradigms

TLDR focuses a lot on the “swarm of bees”-style of programming, that is, having large amounts of simple logic rather than having a unified complex algorithm for all the logic in the program. This can be thought of as a characteristic of modular programming languages.

Another focus of TLDR is that of functions as first class citizens. This means that the language does not distinguish between functions and other symbols declared by the programmer. Any place a symbol can be used in a program, a function can be used as well, e.g. as a parameter for a function, a return value of a function or a message to an actor. This, combined with the use of expressions and the avoidance of state and mutable data gives the language a very functional, or declarative, feel, similar to that of F# or Scala.

The Actor Model does however give the language a modelling form which resembles an object oriented approach. And as will be explained in *section 8.2.1*, creative use of actors can make functions obsolete, which would seem strange for a functional language.

Considering these things, TLDR might not belong strictly to a specific paradigm, but can be adaptive based on programmers approach.

8.1.3 Abstraction

TLDR is a rather high level language. TLDR provides a high level of abstraction over data, with arbitrary precision integer and floating point data types, control flow, with mathematical operations built into the language as well as message passing and loop-structures, and concurrency, with actors and message passing.

8.1.4 Representing fractions

When writing fractions on a single line of text, a common way is to express them as divisions: 1/3, 3/5, 3/10 etc. With this type of notation 1/3 + 2/3 would equal 1. In TLDR, however, the usage of euclidean division would cause the calculation to equal 0, since the two division are evaluated first and both are 0. If TLDR is to consider the divisions as fractions and get the result 1, the fractions must explicitly be noted as real numbers. This could be done with 1.0/3.0 + 2.0/3.0 which would equal 1.0. The reason for this is the principle of not having implicit casting, and also since it would be impossible for a compiler to figure out if euclidean division or fractions is the desired interpretation of the division operator. This limits the languages ability to fulfill the criteria of having traditional mathematical representations of numbers, but it was considered the best alternative.

8.2 Compiler

This section will discuss some language constructs not implemented in the compiler. The full list can be seen in *section 6.7*.

8.2.1 Functions

As the group members were writing programs in TLDR, the absence of functions in the implementation of the compiler, caused some creative ways of modelling a program without the use of functions. As actors were implemented in the compiler, these could be used as a stand-in for functions. A simple function taking a int and returning the int squared, can be modelled as a actor with a receive-method, taking a struct containing an int and a sender actor.

Listing 8.1: Simple example using a function.

```
let f(x) : int -> int := {
  x + x;
};

actor main := {
  receive arguments:args := {
    let res:int := f(2);
    printint(res);
  };
};
```

Listing 8.2: Example of modeling a function taking a int and returning the int squared, as an actor.

```
actor main := {
  receive arguments:args := {
    let msg:st := {n := 2; sender := me;};
    let handle:a := spawn a msg;
  };

  receive res:int := {
    printint(res);
  };
};

actor a := {
  receive msg:st := {
    send msg.sender (msg.n + msg.n);
  };
};

struct st := {
  n:int;
  sender:main;
};
```

Theoretically, any function can be modelled as actors taking structures, however there is some mental overhead involved. The function example is shorter to write and arguably also easier to understand.

On the other side, the fact that every function can be modelled as an actor, shows how powerful actors can be in modeling a program into small logical sections.

8.2.2 Lists

The lists implemented in the compiler cannot be reallocated at runtime, and there is no way of concatenating lists. This limits the usefulness of lists, as observed when writing programs in TLDR. Having concatenation of lists implemented in the compiler would allow for several other operations to be implemented based on concatenation. For example, appending an element to a list, can simply be implemented as a concatenation of the list to append to and a list containing 1 element: the element to append. Assuming the concatenation function is implemented, the append function can be implemented as follows.

Listing 8.3: Implementation of the append function based on the concatenation function.

```
let append(xs, x) : [int] -> int -> [int] := {
  concat(xs, [x]);
};
```

8.3 Future Language Improvements

TLDR, being a language focused on allowing easy modeling of real world system, shares a lot of principles with object oriented programming languages, and so the inclusion of common object oriented features is a likely idea. Inheritance would be a good way to allow for faster and more precise programming. Other alternatives could be traits, interfaces or abstract actors. Interfaces could be very useful for making sure that messages are only sent to actors which actually implements a receive for that message, while still allowing multiple types of actors to be candidates. The usefulness of inheritance, or other ways of sharing functionalities across different actors, would allow for more concise code and faster writing.

Another useful language structure which could be implemented in future editions, could be a reply feature. This feature would make it possible to send a message, while evaluating a receive, without explicitly writing which actor will receive said message. Instead the sender of the message being evaluated, will be the target. Besides the possibility for more concise code with such a construct, it would also allow actors to send messages, without having a local reference to their target. This latter point would make it possible for actors to send messages to targets of which they do not know the type of. This makes actors further independent, but at the cost of a more complex compile time check for illegal messages.

8.3.1 Support for Discrete Simulations

Another matter is the support for discrete simulations, as described in *section 4.5.3*. The solutions suggested are based around grouping actors into environments, which work isolated from the rest of the actors. This idea of grouping actors, could potentially bring advantages to the language, such as highly modular programs and functionalities, and promoting purity when writing code in TLDR. But what is more interesting, is the possibility of a change of focus, from having the programmer think in terms of how actors mimic real things, to focusing on having environments mimic the conditions reality sets for things. Examples could be chemistry or physics, where one could want to find the optimal conditions for a chain reaction to occur. This problem could be approached by creating a single version of the actor, and then have the environments implement the changes in values, temperatures, pressure or what it might be. Such an approach might be useful for problems where certain objects behavior can be described through known mathematical structures, but where the conditions can change immensely and by extension, the results of a given simulation.

9 Conclusion

This chapter of the report will conclude on the problem statement presented in *chapter 2*. Based on the problem statement an analysis of the problem domain was made in *chapter 2*. Furthermore, research of concurrency and an analysis of the solution domain resulted in a basis for the design of the functionality of the language in form of criterias for the language described in *section 4.6*.

The following entail an examination of the way TLDR tries to accomodate the criteria set for the language, as well as the constructed compilers ability to implement the language.

The first criteria was for the language to be simple. This was achieved through staying true to a few select principles, and not allowing them to be broken. The most prominent one being the principle of only letting actors be manipulated through messages. The language also incorporates simplicity in the way that it allows for modeling concurrent programs. The way which the actor model abstracts over the handling of threads in the program removes a lot of technical considerations and thereby simplifies the way the programmer approaches the modeling.

On the subject of orthogonality, the language performs as expected. The focus on simplicity interferes with the orthogonality of the language, but not quite as expected, where the way in which other criteria have interfered. One such criteria could be the lack of implicit casts between number primitives, which was done to uphold readability and the strongly typed language criterias. These criteria have also influenced the language through the explicit difference between the notation for integers and real numbers.

The focus in the syntax design on modeling has also been satisfied. This can be seen in the keywords used for actors.

On the subject of data types and mathematical notation the language strides towards it, but as explained in *section 8.1.4* sometimes other criterias were prioritized.

Based upon the problem statement, the purpose of this project was to describe how a language for simulation could be designed for concurrent computing. This question has been answered in *chapter 5*, by specifying the syntax and semantics for the language. To ensure concurrency, the actor model is the recommended way to do concurrency modeling in TLDR, which helps the programmer to abstract over the specifics of concurrency and parallelism. The problem statement also asks how the language can be implemented, to answer this, a delimited set of the language has been implemented. *section 6.7* describes which features were implemented and which ones were not. The section also discusses why some features were prioritized higher than others.

To conclude on the project in general, the compiler and the language documentation answers the problem statement by giving an example of how such a language could be implemented. Although the compiler lacks some features, due to priorities, the compiler is in a working state, where it can compile some code containing fundamental features such as actors. This is assessed to satisfy the criteria for the language.

Bibliography

[1] S. Hartmann, “The world as a process: simulation in the natural and social sciences”.

[2] R. Hegselmann, U. Mueller, and K. Troitzsch, *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View*, ser. Mathematics and Its Applications. Springer, 1996, ISBN: 9780792341253. [Online]. Available: `https://books.google.dk/books?id=y1dFny4vWt0C`.

[3] B. Barney. (2015). Introduction to parallel computing, [Online]. Available: `https://computing.llnl.gov/tutorials/parallel_comp/` (visited on 05/06/2015).

[4] H. S. Erich Strohmaier Jack Dongarra and M. Meuer. (2015). Top500, [Online]. Available: `http://top500.org/` (visited on 05/04/2015).

[5] (2015). Amdahl’s law, [Online]. Available: `https://en.wikipedia.org/wiki/Amdahl's_law` (visited on 05/25/2015).

[6] (2015). Parallel programming in c, [Online]. Available: `http://gribblelab.org/CBootcamp/A2_Parallel_Programming_in_C.html#sec-2-1` (visited on 05/25/2015).

[7] E. G. Coffman, M. Elphick, and A. Shoshani, “System deadlocks”, *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971, ISSN: 0360-0300. DOI: `10.1145/356586.356588`. [Online]. Available: `http://doi.acm.org/10.1145/356586.356588`.

[8] (1994). Mpi - a message passing interface standard, [Online]. Available: `http://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps` (visited on 05/27/2015).

[9] B. Barney. (2015). Message passing interface (mpi), [Online]. Available: `https://computing.llnl.gov/tutorials/mpi/` (visited on 05/27/2015).

[10] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence”, in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI’73, Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

[11] C. A. R. Hoare *et al.*, *Communicating sequential processes.* Prentice-hall Englewood Cliffs, 1985, vol. 178.

[12] R. Sebesta, *Concepts of Programming Languages*. Pearson Education, Limited, 2015, ISBN: 9780133943023. [Online]. Available: `https://books.google.dk/books?id=Wv6toQEACAAJ`.

[13] H. Hüttel, *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010, ISBN: 9781139788595. [Online]. Available: `https://books.google.com.pe/books?id=f9zmrShQj3YC`.

[14] R. J. L. J. Charles N. Fischer Ron K. Cytron, *Crafting a Compiler*. Pearson Education, Limited, 2009, ISBN: 978-0-13-606705-4.

[15] (2015). Hime, [Online]. Available: `https://bitbucket.org/laurentw/hime/overview` (visited on 05/27/2015).

[16] F. S. Foundation. (2015). The gnu multiple precisio arithmetic library, [Online]. Available: `https://gmplib.org/` (visited on 05/20/2015).

[17] C. Moos. (2015). Libactor, [Online]. Available: `https://code.google.com/p/libactor/` (visited on 05/18/2015).

Grammar

```
/*——— Identifiers ———*/

⟨Id⟩              ::=  [a-zA-Z][a-zA-Z_0-9]*-('let' | 'var' | 'bool' | 'integer' | 'real' |
                       'char' | 'struct' | 'actor' | 'receive' | 'send' | 'spawn' | 'return' |
                       'for' | 'in' | 'if' | 'else' | 'while' | 'die' | 'me');

/*——— Operators ———*/

⟨Pzerooperator⟩   ::=  '(' primitive ')'

⟨Poneoperator⟩    ::=  '^' | '#'

⟨Ptwooperator⟩    ::=  '*' | '/' | '%'

⟨Pthreeoperator⟩ ::=  '+' | '-'

⟨Pfouroperator⟩   ::=  '=' | '!=' | '⟨' | '<=' | '⟩' | '>='

⟨Pfiveoperator⟩   ::=  'NOT'

⟨Psixoperator⟩    ::=  'AND' | 'NAND'

⟨Psevenoperator⟩ ::=  'OR' | 'XOR' | 'NOR'

/*——— Primitives ———*/

⟨Primitive⟩       ::=  ⟨Boolean⟩
                   |   ⟨Integer⟩
                   |   ⟨Real⟩
                   |   ⟨Char⟩

⟨Boolean⟩         ::=  'true' | 'false'

⟨Integer⟩         ::=  '-'?[1-9][0-9]* | '0'

⟨Real⟩            ::=  ([0-9]+'.'[0-9]+)|([0-9]+'.')|('.'([0-9])+)

⟨Char⟩            ::=  ''' U+0020 .. U+007E '''

⟨String⟩          ::=  '"' (U+0020 .. U+007E)* '"'

/*——— Program ———*/

⟨Program⟩         ::=  ⟨Body⟩
```

```
⟨Body⟩          ::= ⟨Body⟩ ';' ⟨Statement⟩
                  | ⟨Body⟩ ';'
                  | ⟨Statement⟩

/*——— Statement ———*/

⟨Statement⟩     ::= ⟨Declaration⟩
                  | ⟨Reassignment⟩
                  | ⟨Expression⟩
                  | ⟨Receive⟩
                  | ⟨Spawn⟩
                  | ⟨Return⟩
                  | ⟨Die⟩
                  | ⟨Send⟩
                  | ⟨If⟩
                  | ⟨IfElse⟩
                  | ⟨While⟩
                  | ⟨ForIn⟩

/*——— Spawn ———*/

⟨Spawn⟩         ::= ('let' | 'var') ⟨SymDecl⟩ := 'spawn' ⟨Identifier⟩ ⟨Expression⟩

/*——— Send ———*/

⟨Send⟩          ::= 'send' ⟨Identifier⟩ ⟨Expression⟩

/*——— Return ———*/

⟨Return⟩        ::= 'return' ⟨Expression⟩

/*——— Receive ———*/

⟨Receive⟩       ::= 'receive' ⟨SymDecl⟩ := ⟨Block⟩

/*——— Die ———*/

⟨Die⟩           ::= 'die'

/*——— For ———*/

⟨ForIn⟩         ::= 'for' ⟨Identifier⟩ 'in' (⟨List⟩ | ⟨Identifier⟩) ⟨Block⟩

/*——— While ———*/

⟨While⟩         ::= 'while' ⟨Expression⟩ ⟨Block⟩

/*——— If ———*/
```

```
⟨If⟩            ::= 'if' ⟨Expression⟩ ⟨Block⟩

⟨IfElse⟩        ::= 'if' ⟨Expression⟩ ⟨Block⟩ 'else' ⟨Block⟩

/*——— Block ———*/

⟨Block⟩         ::= '{' ⟨Body⟩ '}'

/*——— Expression ———*/

⟨Expression⟩    ::= ⟨Expression⟩ ⟨Psevenoperator⟩ ⟨OP6⟩
                 |  ⟨OP7⟩

⟨OP7⟩           ::= ⟨OP7⟩ ⟨Psixoperator⟩ ⟨OP6⟩
                 |  ⟨OP6⟩

⟨OP6⟩           ::= ⟨Pfiveoperator⟩ ⟨OP6⟩
                 |  ⟨OP5⟩

⟨OP5⟩           ::= ⟨OP5⟩ ⟨Pfouroperator⟩ ⟨OP4⟩
                 |  ⟨OP4⟩

⟨OP4⟩           ::= ⟨OP4⟩ ⟨Pthreeoperator⟩ ⟨OP3⟩
                 |  ⟨OP3⟩

⟨OP3⟩           ::= ⟨OP3⟩ ⟨Ptwooperator⟩ ⟨OP2⟩
                 |  ⟨OP2⟩

⟨OP2⟩           ::= ⟨OP2⟩ ⟨Poneoperator⟩ ⟨OP1⟩
                 |  ⟨OP1⟩

⟨OP1⟩           ::= ⟨Pzerooperator⟩ ⟨OP1⟩
                 |  ⟨OP0⟩

⟨OP0⟩           ::= ⟨Operand⟩
                 |  '(' ⟨Expression⟩ ')'

⟨Operand⟩       ::= ⟨Block⟩
                 |  ⟨Integer⟩
                 |  ⟨Real⟩
                 |  ⟨Boolean⟩
                 |  ⟨Literals⟩
                 |  ⟨Invocation⟩

⟨Accessor⟩      ::= '.' ⟨Identifier⟩
                 |  '.' '[' ⟨Expression⟩ ']'
```

```
⟨Literals⟩        ::=  ⟨String⟩
                    |  ⟨Char⟩
                    |  ⟨List⟩
                    |  ⟨StructLiteral⟩
                    |  ⟨Tuple⟩

⟨StructLiteral⟩   ::=  ’(’ ((⟨Reassignment⟩’;’)* ’)’ (’:’ ⟨Identifier⟩)?

⟨Invocation⟩      ::=  ⟨Identifier⟩ ’(’ ((⟨Expression⟩ (’,’ ⟨Expression⟩)*)? ’)’

⟨Identifier⟩      ::=  ’me’
                    |  ⟨Id⟩
                    |  ⟨Id⟩ ⟨Accessor⟩

/*——— Declaration ———*/

⟨Declaration⟩     ::=  ⟨Struct⟩
                    |  ⟨Actor⟩
                    |  ⟨Initialisation⟩

⟨Struct⟩          ::=  ’struct’ ⟨Identifier⟩ := ’{’ ⟨TypeDecls⟩? ’}’

⟨TypeDecls⟩       ::=  ((⟨FuncDecl⟩ | ⟨SymDecl⟩) ’;’ ⟨TypeDecls⟩)+

⟨Actor⟩           ::=  ’actor’ ⟨Identifier⟩ := ⟨Block⟩

⟨Initialisation⟩  ::=  (’let’ | ’var’) (⟨FuncDecl⟩ | ⟨SymDecl⟩) := ⟨Expression⟩

⟨FuncDecl⟩         ->  ⟨Identifier⟩ ’(’ ⟨Ids⟩? ’)’ ’;’ ⟨Types⟩ ⟨SymDecl⟩ -> ⟨Identifier⟩ ’;’
                       ⟨Types⟩

⟨Ids⟩             ::=  ⟨Identifier⟩ (’,’ ⟨Identifier⟩)*

/*——— Reassignment ———*/

⟨Reassignment⟩    ::=  ⟨Identifier⟩ ’:=’ ⟨Expression⟩

/*——— Types ———*/

⟨Types⟩           ::=  ⟨Type⟩ (’->’ ⟨Type⟩)*

⟨Type⟩            ::=  ⟨Primitive⟩
                    |  ⟨Identifier⟩
                    |  ⟨ListType⟩
                    |  ⟨TupleType⟩
```

⟨*TupleType*⟩ ::= '(' ⟨*Types*⟩ ')'

⟨*ListType*⟩ ::= '[' ⟨*Types*⟩ ']'

/*——— List ———*/

⟨*List*⟩ ::= '[' ⟨*ListElem*⟩ (',' ⟨*ListElem*⟩)* ']'

⟨*ListElem*⟩ ::= ⟨*Expression*⟩ '..' ⟨*Expression*⟩
| ⟨*Expression*⟩

/*——— Tuple ———*/

⟨*Tuple*⟩ ::= '(' ⟨*Expression*⟩ (',' ⟨*Expression*⟩)+ ')'

/*——— Comments ———*/

⟨*COMMENT_LINE*⟩ ::= '//' (.* - (.* ⟨*NEW_LINE*⟩ .*)) ⟨*NEW_LINE*⟩?

⟨*COMMENT_BLOCK*⟩ ::= '/*' (.* - (.* '*/' .*)) '*/'