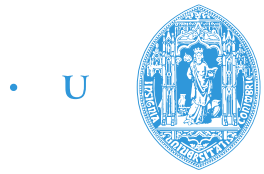


University of Coimbra

Department of Informatics Engineering
Faculty of Sciences and Technology

Master thesis

A programming language for parallel event-driven development

Author

João Pedro Maia Rafael

jprafael@student.dei.uc.pt

Supervisor

Bruno Miguel Bras Cabral

bcabral@dei.uc.pt

September 5, 2013

Abstract

Recently, event-oriented programming frameworks have surfaced as a solution to highly scalable network applications. This model as been adopted under many languages resulting in frameworks such as Node.js, Gevent and EventMachine. These frameworks are capable of handling many concurrent requests by using asynchronous IO. However, in order to make use all available cores, parallelism is exploited by creating multiple instances of the same application. Under this solution instances don't share memory making synchronization mechanisms required. The same problem applies when using the actor model for concurrency.

The EVE framework provides support for event-oriented programming under a shared-memory model. It encompasses the EVE language definition, its compiler and a runtime system capable of executing the resulting applications. Using our model, the programmer divides the application logic into tasks and each task indicates what variables it can access. The runtime schedules compatible tasks to multiple cores using a work-stealing algorithm for load balancing. In this work, we present a formal description of the language and it's runtime, including their operational semantics. Our benchmarks indicate that our solution delivers the best performance on IO heavy problems when compared to existing of-the-shelf solutions and performance comparable to the state-of-the-art architectures for CPU-bounded applications.

Keywords: parallel languages, event-driven programming, shared memory, runtime systems

Acknowledgments

I would like to express my gratitude to my supervisor at DEI, Bruno Cabral, for introducing me to the AEminium research group. He also made this work possible by encouraging me to follow my own research topics with much appreciated independence.

I would also like to thank my friends and colleagues with whom I endured many hard working nights and shared many treasured moments. They provided me with much needed laughs, trust and support.

Finally, I would also like to thank my beloved family, specially my parents and my sister who provided me with time and means to proceed with my studies but who also taught me to always strive for the best. Their support was truly invaluable.

This work was partially funded by the Foundation for Science and Technology grant `Lugus 84769` under the *AEminium - Freeing Programmers from the Shackles of Sequentiality* project.

Contents

List of Tables

List of Programs

List of Figures

Chapter 1

Introduction

This chapter is organized into six sections. The first section explains the motivation and relevance of this work. In the second section we introduce the AEminium project. The third section defines the goals of this work. The fourth section states the contributions. The fifth section describes the methodology used. Finally, the last section describes the structure of the remaining document.

1.1 Motivation

According to Moore's law, CPU clock speed should double approximately every two years [22]. This however, has become generally accepted as obsolete due to existing limitations. Although transistor counts continue to rise, clock speed has reached a plateu and has remained stable in the last years. This is can be observed in figure 1.1.

Hardware developers such as Intel and AMD have shifted their research efforts towards other ways to increase computing performance. Instead of increasing clock speed, the focus is now on new means to execute more operations per clock. **Parallelism** is the capability of the computing system to execute multiple operations at the same time. However, to tap the full performance of architectures that make use of parallelism, the programmer must explicitly indicate what operations can be executed simultaneously. This task becomes harder and error-prone as the size of applications increase, largely due to the exponential growth of interactions between functions. In order to achieve the best performance in these platforms, fine-tuning is required to adjust the application to the machine, taking into account the number of processing cores, cache size, instruction set, etc. However, due to the multitude of hardware with different characteristics this becomes a cumbersome task.

To overcome this problem, new programming languages such as **Cilk** allow the programmer to indicate available parallelism [10]. Whether or not this parallelism is used is determined by the runtime system, depending on the available resources and their usage. Notably, these

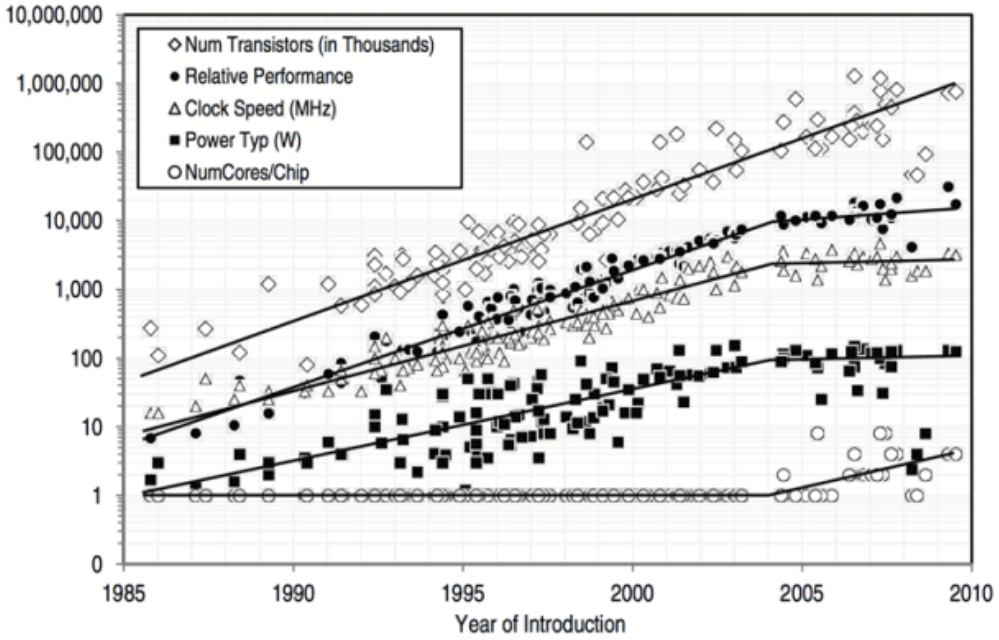


Figure 1.1: Clock frequency, performance, clock speed, power consumption and number of cores per chip on consumer level processors according to [27].

frameworks excel at solving embarrassingly parallel problems where mapping each sub-problem to a thread would otherwise lead to a large overhead[1]. Furthermore, they also solve with ease problems where the size of each sub-problem is not known *apriori*.

Alongside these developments, the interest for asynchronous programming as also increased. One notable example of this is the adoption of JavaScript for the development of server-side applications with **Node.js**. These applications often have to deal with large amounts of IO operations. Using the procedural approach, a single thread would handle a conversation between the server and a client, and would block on operations such as `read()` until data is available. This solution scales poorly due to the large number of threads created under heavy loads. With the asynchronous paradigm, instead of blocking on pending data the application registers a **callback** function and proceeds to execute other work. When the data becomes available the callback is invoked to handle the response. This solution enables the application to handle multiple clients concurrently with a single thread. However, because only one thread is employed the performance does not match the maximum throughput of multi-core platforms. Currently, to maximize performance with these languages, parallelism must be created manually, usually through the use of workers that don't share state. This simple solution suffices for most common cases but it presents two key disadvantages:

[1]Kernel Level Threads (KLT) require the creation of a dedicated stack space and are scheduled using expensive preemptive and fair algorithms.

- Instead of focusing on the application logic the programmer must deal with the creation of separate workers. Code refactoring is sometimes necessary to map application logic to a worker, keeping in mind that there is an optimal number of workers for each machine.
- Because workers do not share memory, a consistent view of the application must be created by using explicit messages or additional software that manages data synchronization (i.e: a database). The overhead of this extra communication might overcome the benefit of parallelization.

As such, it becomes desirable to conceive a language and runtime system that employs the asynchronous paradigm with shared state to achieve true parallelism.

1.2 Context

This work was partially funded by the grant `Lugus 84769` under the *AEminium - Freeing Programmers from the Shackles of Sequentiality* project. The team is currently composed by Alcides Fonseca, Bruno Cabral, João Lourenço and João Rafael.

The AEminium project was part of the CMU|Portugal program and results from a partnership between the universities of Carnegie Mellon, Coimbra, and Madeira and the R&D center of Novabase. The project aimed to provide a practical framework for development of massively concurrent applications.

1.3 Goals

The goal of this work is to ease the development of high-performance event-based applications. For this purpose, a new programming language **EVE** was designed taking into account event oriented programming as a first-order paradigm.

The language should provide an easy to use syntax, with minimal structure overhead, for the definition of common event-related operations such as registering a callback function and triggering an event.

EVE also aims to be a general purpose **high performance** language. Therefore, a strong type system becomes desirable to allow for better compile-time optimizations. Furthermore, the language definition should also contain statement types that concede easy parallelization without boilerplate code. This parallelization should be managed by a specially crafted runtime system. This runtime should provide three aspects when combined with event-handling capabilities: low latency, low overhead, and high throughput.

Finally, the language should be put to **practical use**, demonstrating its features on common showcase problems such as web servers, event stream processing, and soft real-time systems like chat servers.

1.4 Contributions

- Analysis language-level support for event-driven programming model with native parallelism.
- Definition of the **EVE programing language** and its semantics.
- Implementation of a **compiler** capable of translating the original program written in EVE into machine code or some other intermediary language such as C. Development of a **runtime platform** capable of executing the final binary. This runtime must use a scalable approach to share work across all available CPU cores. It must also focus on high event handling throughput, with low latency desirable. Creation of **native libraries** for low-level OS operations regarding IO, file systems, sockets, processes, etc; and also user level utilities such as timers and loggers. Whenever possible, reuse of existing libraries such as the C++'s STL containers should be used as a back-end.
- Extensive performance analysis of the proposed framework and comparison with existing solutions.

1.5 Methodology

Due to the unpredictable nature of the project's evolution and work rate, a development process based agile methodologies was adopted. The process consisted of a long term plan which underwent revisions during weekly meetings of the entire team. In these meetings, the work executed throughout the week was discussed and based on the obtained results a work plan for the upcoming sprint was outlined. This iterative approach allows for timely deliverables while allowing the developer to work on unplanned work items. The *Redmine* project management platform and the *GIT* versioning control system were employed to validate our approach. These application help producing deliverables such as activity logs, status overviews and Gantt diagrams.

The work was conducted over the course of two semesters. The student only exerted for half-time during the first semester and full-time for the second semester. Figure 1.2 depicts the Gantt diagram of the long-term plan executed during the first semester. Initially, work was dedicated to research of the state of the art. This was followed by the formulation of scientific contributions and architecture planning. A partial implementation was executed and some experiments conducted. In conclusion, time was allocated for the execution of the preliminary report presented at the end of the semester.

Figure 1.3 shows the long-term plan for the second semester. During the first weeks the runtime system was completed. The implementation of the compiller scheduled for the following two months followed. However due to unexpected difficulties indicated in 4.2 the conclusion of

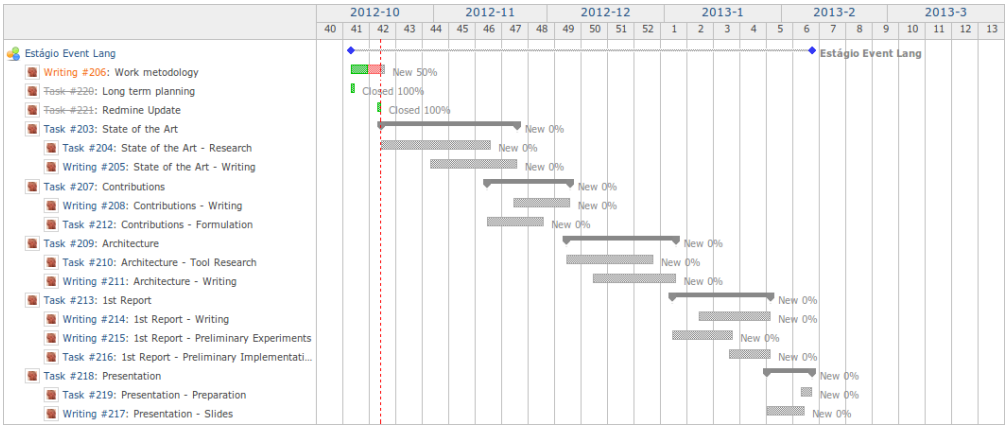


Figure 1.2: Long-term development plan for the first semester.

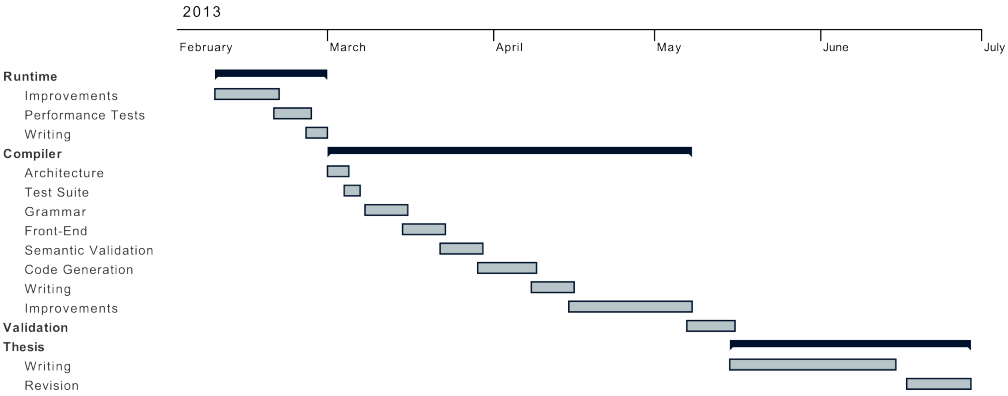


Figure 1.3: Long-term development plan for the second semester.

this softwared took two aditional months. The next week was reserved for experimental analysis and validation. Finally, the last month was allocated for the execution of this document.

1.6 Structure

The remaining document is structured as follows. Chapter two introduces the state of the art in both parallel software methodologies and event-based processing. In chapter three the specification of the EVE language is defined. Chapter four presents implementation details information about the EVE runtime, and compiler. Chapter five presents the evaluation conducted. Finally, chapter six indicates directions where improvement is possible for future work and concludes.

Chapter 2

State of the Art

This chapter presents the benefits and caveats of two distinct areas of research, the first being the use of parallelism in applications and the second being event-oriented programming. It is divided into four sections. The first section gives an introduction to parallelism and the multiple forms in which it appears. Section two gives a brief overview of language constructs used for parallelism found in both academic literature and production ready languages. Section three enumerates some examples of event-driven programing both with, and without language level support. Section four presents an overview of the current solutions to the problem and their limitations.

2.1 Exploiting Parallelism

In [9], Flynn proposed the classification of computer architectures based on the number of parallel instructions and the number of data streams processed by each instruction. This classification is visible in table 2.1 and is still in use today.

Single Instruction Single Data Architectures that do not exploit parallelism at instruction or data level.

Multiple Instruction Single Data Architectures capable of processing multiple operations over the same data. This capability can be used to achieve some level of redundancy but

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Flynn's taxonomy of computer architectures.

is inappropriate for most parallelizable problems.

Single Instruction Multiple Data Architectures that can execute the same instruction over different operands. This solution is often employed to achieve speedups on image manipulation applications where each pixel must be processed the same way.

Multiple Instruction Multiple Data Architectures that can execute separate instructions on separate operands. This is the most flexible architecture.

2.1.1 Vectorization

Vector operations became available on mainstream processors with the addition of the Streaming SIMD Extension[1] (SSE) to the x86 architecture. This extension added instructions that would execute arithmetic operations simultaneously on 4 values, essentially becoming a Single Instruction, Multiple Data (SIMD) architecture. SSE2, SSE3, SSE4 and more recently Advanced Vector Extensions (AVX) are additions that increase the number of possible simultaneous operations, the size of each operand, and the available instruction set.

Efficient implementation of algorithms using vector operations requires data to be contiguously aligned. This not only maximizes cache hits but also allows the SIMD registers to be loaded with a single operation. To achieve this memory configuration, some refactoring might be necessary to use arrays of native types instead of arrays of objects. One other problem arises from the multitude of hardware specifications. In order to ensure retro-compatibility the programmer is forced to implement one version for each target platform. In order to mitigate this problem, libraries that support these operations are used to abstract the programmer from the actual implementation. Intel's The Math Kernel Library [15] is one example. Its interface provides several common used math features such as BLAS 1 (vector-vector operations), BLAS 2 (vector-matrix operations), BLAS 3 (matrix-matrix operations), FFTs, Statistics and Regression.

Auto-vectorization Auto-vectorization is a compiler technique used to transform groups of scalar operations into vector operations. This technique uses loop dependency analysis to identify sections where vectorization is advantageous while maintaining the same operational semantics. In practice, this analysis searches for the following characteristics:

1. Loops with pre-determined number of iterations, either at compile time or at runtime;
2. No data-dependent jumps or branches;
3. No nested loops;
4. Without backward loop-carried dependencies within the stride of the vector size;
5. Data boundaries aligned to vector size.

[1] A successor of MMX

2.1.2 Multiprocessing

The second solution is to employ multiple processing units for the same system. This can be achieved at various levels: multiple cores *per* die, multiple dies *per* package, multiple packages *per* device and multiple devices *per* system. Unlike vectorization this approach uses a Multiple Instruction, Multiple Data (MIMD) architecture allowing divergent instruction paths to be executed in parallel.

However, efficient parallelization is also a difficult task. Careful synchronization is required to ensure the applications' correctness and wrong implementations often lead to problems such as race conditions, deadlocks, priority inversions and false sharing. Automatic detection of these problems is possible, however the complexity grows exponentially large with the number of concurrent instructions []. Furthermore, synchronization operations are particularly expensive (performance wise). In order to minimize the use of these operations work is divided into units that can be executed in parallel with minimal communication. Several patterns can be employed for this task such as divide-and-conquer, map-reduce, and pipelining. The granularity of division is also an issue programmers have to deal with. A coarse division may lead to under-usage of the available resources. For instance, in cases where some operations take more time than others or when the execution time is not known *apriori*, simple homogeneous division of work is not enough. On the other hand, a fine grained approach may lose its benefits as the overhead of resource creation (e.g.: threads, messages) and communication increases. For these reasons, a wide range of approaches that flourish parallelization have emerged. They can be grouped into three classifications: *parallel libraries*, *parallelizing compilers* and *parallel languages*.

Parallel Libraries

Perhaps the most widely deployed parallel library is JAVA's `java.util.concurrent` package. It was introduced in JAVA SE 5.0 and provides many components including concurrent data structures such queues and hash maps, and utility abstractions such as semaphores, futures and executors. Another example is Intel Threading Building Blocks (TBB), a C++ library that implements almost the same set of components included in `java.util.concurrent`. It also provides parallel algorithms such as `parallel_sort` and `parallel_pipeline`. Furthermore, a task scheduler using work-stealing (as described in section 2.2.3) is also available.

Parallelizing Compilers

Parallelizing compilers analyze and transform sequential programs into parallel versions that produce the same results. Focus is usually on parallelizing loop operations since in the majority of applications this is where the most work is found. Initially, the program goes through a step called data dependency analysis which produces a set of order constraints between statements. The compiler makes use of these constraints to check whether or not its safe to parallelize the

code. An example of a loop without constraints between iterations can be found in program 1. Because each iteration is independent the same result will be obtained regardless of the order of their execution. This type of loop can be parallelized by computing different sets of iterations on each available thread.

```
for (int i = 0; i < N; ++i)
    a[i] = b[i] * b[i];
```

Program 1: Example of a loop that computes the point-wise square of a vector `b` and stores the result in `a`. This loop can be parallelized efficiently using static allocation.

Multiple scheduling mechanisms are available to allocate iterations [19]. The scheduling can be determined either staticly at compile time or dynamically at runtime. The static allocation assigns to each processor a stride of iterations. Considering P processors, the first executes iterations $i = 0, P, 2P, ...$. Alternatively, a chunk of contiguous operations can be assigned to the same processor. In this case, the first executes iterations $i = 0, 1, ..., c-1, cP, cP+1, ..., cP+c-1, ...$ where c is the chunk size. If $c = \lceil N/P \rceil$ the scheduler assigns a single chunk for each processor.

```
for (int i = 0; i < N; ++i)
    a[i] = pow(b[i], i);
```

Program 2: Loop with uneven work per iteration. Assuming the complexity of the `pow()` operation increases with i, equally dividing the loop into P blocks would lead to the last processor taking longer to complete while the remaining stay idle.

These scheduling algorithms work well when all iterations are equally complex. Consider however program 2. In this example, the last iterations, where i is larger, are more complex than the first ones. Using a single-chunk scheduling algorithm would produce unbalanced work loads many processors would become idle. Instead, the compiler should opt for a dynamic scheduling algorithm dispatching work as processors become available. A simple implementation maintains a shared counter to indicate the next available iteration. Because this requires synchronization between processors some overhead is unavoidable. However, it can be mitigated by increasing the counter by c each time. A small c leads to good balancing at the cost of a larger overhead, while a large c incurs in low overhead but may become unbalanced.

Guided self-scheduling uses a dynamic value for c. Using $c = \lceil (N-i)/P \rceil$ the algorithm starts with a large value for c which decreases over time to accommodate unbalanced work loads.

Loop-carried dependencies make this parallelization impossible as they impose an order between executions. In some cases however, they can be safely removed by use of loop transformations. Loop splitting is one of such transformations. Given a loop with multiple independent statements, loop splitting creates two loops separating the statements with loop-carried dependencies from those without. This allows for parallelization of the second loop. An example of

this transformation put in practice can be observed in program 3. Other available transformations include loop peeling, skewing, fission and tilling [13]. In addition, the polytope model is a mathematical framework for analyzing loop-carried dependencies and generating the necessary transformations to remove them [18].

```
for (int i = 0; i < N; ++i)
{
    if (i > 0)
        a[i] += a[i-1];
    b[i] *= a[i];
}
```

```
for (int i = 1; i < N; ++i)
    a[i] += a[i-1];

for (int i = 0; i < N; ++i)
    b[i] *= a[i];
```

Program 3: Application of loop splitting. On the left, iteration `i` must be preceded by iteration `i-1`. On the right, iterations of the second loop can execute in any order.

Although compilers are able to analyze and correctly parallelize some loops automatically, the obtained performance is, in some cases, worse than the one obtained by manual parallelization. The OpenMP standard presents a solution [5]. Through the use of `pragma` directives, the programmer can indicate to the compiler what sections to parallelize. Thread creation, task allocation, scheduling and synchronization are all handled by the compiler and a runtime environment. Program 4 shows use of OpenMP to parallelize a loop that would not be parallelized otherwise since `work` could be thread unsafe.

```
#pragma omp for schedule(static)
for (int i = 0; i < n; ++i)
    a[i] = work(i);
```

Program 4: This example makes use of the `parallel for` directive. Without this additional information the compiler will not parallelize the loop. The `schedule(static)` clause is used to indicate static scheduling but the implementation is free to choose any chunk size.

```
cilk int fib(int n)
{
     if (n < 2)
          return n;

     int x, y;

     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;

    return x + y;
}
```

Program 5: The recursive Fibonacci algorithm implemented in Cilk. Task spawning is much faster than creating a thread and only happens when not all processors are working. For this reason the programmers can use the `spawn` keyword to indicate available parallelism with minimal overhead.

Parallel Languages

Parallel languages are oriented distinctively at allowing the programmer easy development of parallelized applications. They allow the programmer to express this parallelization either explicitly, through the use of well defined statements, or implicitly as a result of the structure the language enforces.

Cilk is an extension to the popular ANSI C language that only defines a few extra keywords [10]. Using the `cilk` modifier to a function declaration the programmer indicates that it is a Cilk procedure. When the invocation of these procedures is preceded by the `spawn` keyword, the runtime *may* execute it in parallel. The runtime system decides whether or not to execute both functions simultaneously. To allow for synchronization the programmer can use `sync` to define a local barrier. When this statement is encountered the execution waits until the spawned children are completed. Program 5 is an implementation written in cilk of the recursive Fibonacci algorithm.

Because the two spawned tasks don't access the same variables no race condition is present. However, this restriction is limiting on more complex problems. Consider the task of searching a binary tree for the first element that matches a key. Ideally, when one element is found, all parallel tasks can be stopped. This complex behavior can be achieved with the use of an `inlet` function and the `abort` statement. Inlets are inner functions to a Cilk procedure, and are executed atomically once a child task terminates. The `abort` statement is only allowed inside an `inlet` function and indicates that already spawned sub-tasks can be safely aborted. Example usage both features is visible in program 6.

X10 is language for parallel and distributed computing [3] It implements the Partitioned Global

```
cilk node_t* search(node_t* node, int key)
{
    if (!node) return NULL;
    if (node->key == key) return node;

    node_t* found = NULL;

    inlet void match(node_t* n)
    {
        if (n)
        {
            found = n;
            abort;
        }
    }

    match(spawn search(node->left, key));
    if (!found)
        match(spawn search(node->right, key));

    sync;
    return found;
}
```

Program 6: Recursive search over a binary tree implemented with Cilk. When match is found in one of the sub-trees, the search over the other branch is aborted if being processed in parallel. In the sequential case, the second branch is only visited if the first did not return any match.

Address Space (PGAS) model where the computation is divided into a set of *places* (single OS processes). Each place is capable of creating, aborting and waiting for *activities* either locally or at a specified remote place. To create an activity the `async` keyword can be perpended to any statement. After creation, the parent activity may resume execution prior to the child being completed. Unless specified, the activities may execute in any available place. The `at(place)` clause can be used to control the place where it executes. One advantage over Cilk tasks is that they can be created from any statement instead of just procedure invocations. Synchronization is handled by the `finish` block[2]. This statement will block the execution until all children activities defined inside this block are completed. Finally, the `atomic` statement provides mutual exclusion between all activities in the same place. Program 7 shows usage of this features.

Erlang is a general purpose, strong typed, functional language built for scalable systems with high availability requirements. An Erlang program follows the actor model, incorporating several *processes* that communicate through message passing. Each process executes se-

[2]Technically a generic statement like `async`

```
public static def compute()
{
    var local_count: Long = 0;

    /* multiple threads */
    finish for (var t: Int = 0; t < THREADS; t++)
    {
        val tt = t;
        async {
            val r = new Random(tt);
            var thread_count: Long = 0;

            for (var j: Long = 0; j < N; j++)
            {
                val x = r.nextDouble();
                val y = r.nextDouble();
                if (x * x + y * y <= 1.0)
                    thread_count++;
            }

            atomic local_count += thread_count;
        }
    }

    return (4.0 * local_count) / (N * THREADS);
}
```

Program 7: X10 implementation of the Monte Carlo method for approximating π. This example uses resources from a single place.

quentially, handling received messages one at a time. New processes can be created using the `spawn` built-in function and execute in parallel. Because processes don't share memory, synchronization with locks is not required. Furthermore, message sending and receiving is decoupled through the use of message queues. This enables one process to resume computation without needing to wait for the receiver end. Program 8 shows three processes communicating with messages to form two ping-pong patterns in parallel.

Go is a garbage collected, compiled language for concurrent programming. Concurrency is obtained by launching multiple *goroutines*: function or statement that executes in parallel. Like actors, goroutines do not shared memory. However, communication and synchronization is managed with indirect message passing, by placing an additional entity called *channel* between the goroutines. Although formaly equivalent, this allows multiple goroutines to read and write to the same channel.

```erlang
-module(pong_example).
-export([start/0, pong/0, ping/3]).

pong() ->
    receive
        {ping, Name, Ping_PID} ->
            io:format("~s pong~n", [Name]),
            Ping_PID ! pong,
            pong()
    end.

ping(0, Name, Pong_PID) ->
    done;
ping(N, Name, Pong_PID) ->
    io:format("~s ping~n", [Name]),
    Pong_PID ! { ping, Name, self() },
    receive
        pong -> ping(N - 1, Name, Pong_PID)
    end.

start() ->
    Pong_PID = spawn(pong_example, pong, []),
    spawn(pong_example, ping, [5, 'A', Pong_PID]),
    spawn(pong_example, ping, [5, 'B', Pong_PID]).
```

Program 8: An example in Erlang of a ping-pong server and client. The pong server replies with `pong` each time a `ping` message is received. Ping clients, named A and B, send five `ping` messages each time waiting for the reply.

```go
func pi(n int) float64 {
    ch := make(chan float64)
    for k := 0; k <= n; k++ {
        go term(ch, float64(k))
    }
    f := 0.0
    for k := 0; k <= n; k++ {
        f += <-ch
    }
    return f
}

func term(ch chan float64, k float64) {
    ch <- 4 * math.Pow(-1, k) / (2*k + 1)
}
```

Program 9: An example in Go of goroutines and channels to compute an approximation of π using n terms of the Gregory-Leibniz series.

2.2 Parallel Constructs

This section describes language constructs from which parallelism can be extracted and discusses the advantages and disadvantages of each approach.

2.2.1 Threads

Threads are a low level approach to parallelism. When using threads to develop an application, the programmer defines exactly how the computation should be distributed by mapping to each thread the work it must execute. For this purpose, the programmer can use directly the OS thread interface (`pthreads` or otherwise). When doing so, the execution of the applications incurs in a low overhead. For this reason threads are widely used on performance critical systems and as a back-end of many other parallel constructs.

Threads are not implicitly synchronized. As such, instructions between two threads may occur in any order, originating race conditions. To avoid this undesired behavior the programmer must make use of additional mechanisms such as semaphores, barriers or atomic operations. This primitives make it hard for the programmer to express his intents [20] often leading to software bugs.

When using threads the programmer is responsible of sizing and partitioning of the problem. A solution with poor quality will lead to bad runtime performance. On one hand, when using a coarse sub-division one thread may become idle even though work is still available. On the other hand, a fine-grained sub-division will lead to a higher scheduling and synchronization overhead and a larger memory footprint. This process becomes more difficult when dealing with problems with irregular structure or input dependent behavior.

2.2.2 Futures

A *future*, *promise* or *delay* is an object whose value is not yet known because its computation has not completed yet. When a future is created, resolving its value is handled by another thread (either created explicitly or obtained from a worker pool). It is assumed that this computation does not modify values accessed by the creator (and vice-versa) therefore being race-free. When the value is required by the caller the future is *resolved*. This operation returns the value of the object immediately if it has been computed already or blocks until the value is available. Program 10 shows the usage of futures in C++11.

2.2.3 Tasks

A *task* is similar concept to a future. Unlike futures however, tasks can access and modify shared variables. For this reason a mechanism for synchronizing tasks is required. Programs written with tasks form a Direct Acyclic Graph (DAG). Nodes identify tasks and edges their

```
int fib(int n)
{
    if (n < 2)
        return n;

    future<int> a = async([n] () { return fib(n - 1); });
    future<int> b = async([n] () { return fib(n - 2); });

    return a.get() + b.get();
}
```

Program 10: Implementation of the recursive Fibonacci method using C++11's futures. Using the default options, `async` may create a new worker thread or lazy evaluate the value only when the `get()` method is invoked.

dependencies. An edge from node A to node B indicates that task A precedes B. A task is allowed to execute only when its in-degree (number of dependencies) is zero. Once it completes, the corresponding node and all exit edges are removed from the DAG, allowing other tasks to be executed. Because the graph is acyclic, deadlocks are impossible under the assumption that every task eventually finishes.

Task creation is very efficient when compared against threads. For this reason the programmer is capable of partitioning the problems into smaller work units. This allows a runtime task scheduler to perform a better load-balancing.

Work-Stealing is a policy for efficient load-balancing. Each worker thread maintains a deque of its ready-to-execute tasks. When new tasks become available they are inserted in the deque's head. Threads obtain work by removing items also from the head. When the deque is empty, workers can can *steal* work from other deques. This operation attempts to remove one item from another worker's deque at its tail.

The original proposal of the work-stealing policy describes the THE protocol used in the implementation of Cilk-5 [11]. This protocol assumes the high contention scenario is unlikely and uses mutexes as primitives for synchronization. Improvements to this algorithm have since been published. In [2], Arora, Blumofe and Plaxton present an algorithm for conducting work-stealing using Compare-And-Swap primitives. This is commonly referred to as ABP work-stealing. In [4], an extension to the ABP algorithm is described allowing for dynamic sized deques. In [26], the termination detection problem is solved by introducing lifeline graphs. This solution is used for global load balancing in X10. Finally, [16] introduces a new work-stealing model with lower overhead based on thread suspension.

2.2.4 Actors

An actor is an isolated entity that communicates with the system (other actors) through message passing. Each actor acts in response to messages received, being able to change its own behavior, create new actors and asynchronously send new messages. Because actors don't share memory (isolation) the model is inherently concurrent and doesn't require synchronization mechanisms like locks. However, because no shared state is allowed messages must be sent back and forth between actors. As such, message creation, transferring, storing and parsing increases the pattern's overhead which might incur in parallel slowdown.

2.3 Event-Driven Languages and Libraries

In event-driven programming the flow of execution of an application is determined by the manifestation of events and its handling. This pattern has been widely used to develop graphical user interfaces. Recently, it has also been adopted as a model to develop highly scalable web applications. In this section we present a list of some languages and libraries that follow this model.

Qt is a cross-platform C++ framework used mostly for developing GUIs. Qt offers the programmer two mechanisms called *signals* and *slots* that are used together as communication bridges between objects. Signal are special special method declarations with no return type (e.g. `void`) and with arbitrary arguments. The method implementation must not be defined by the programmer as it is created automatically by the Qt meta object compiler. To trigger a signal, the programmer just needs to invoke the declared method with the `emit` keyword. Slots are user defined functions containing the event-handling code. Slots can be invoked manually but are most useful when *connected* to a signal. Signals and slots offer an flexible interface that allows M-by-N relationships: triggering a signal can invoke multiple slots and a slot can be invoked by multiple signals. A small example of Qt is shown in program 11.

JavaScript is a single-threaded, multi-paradigm language with first-class functions. It is the *de facto* client-side application language for web browsers. JavaScript is based around an event-loop which handles events one by one. For this reason, event handler can not block, otherwise the application would become unresponsive. Instead, a non-blocking IO API is used. Using this API, the programmer request some action (e.g.: read data from a socket) by passing a callback function as an argument. When the action completes the callback is pushed into the event-loop and is eventually executed. **Node.js** is a framework for building scalable web servers with JavasSript. It uses Google's V8 engine as the JavaScript Virtual Machine and libUV to provide asynchronous IO over multiple platforms. Program 12 shows a simple HTTP server replying with the *Hello World* message to every request.

2.4 Overview

This chapter described a transversal overview of the state of the art in three fields of study. The first introduced parallelism and approaches to exploit it. The second analyzed language constructs used to expose parallelism. Finally the third section addresses the event-oriented programming model. Some elements have been purposefully kept out. For instance, GPU programming provides alternative to achieve large speedups on arithmetic heavy problems. However, event-driven programming is mainly used in other contexts where IO and flow-control operations

```
class Counter : public QObject
{
    Q_OBJECT
    int m_value;

public:
    Counter() { m_value = 0; }

public slots:
    void set(int value)
    {
        if (value == this->m_value)
            return;

        this->m_value = value;
        emit changed(value);
    }

signals:
    void changed(int new_value);
};
```

```
void main()
{
    Counter a, b;
    QObject::connect(
        &a, SIGNAL(changed(int)),
        &b, SLOT(set(int))
    );

    a.set(42);
}
```

Program 11: Usage of Qt's signal and slots mechanism. Setting a counter to a new value triggers the `changed` signal. This signal can be connected to any slot using the `connect` function. This example creates two Counter objects the `changed` signal of `a` to the `set` slot of `b`. In practice, changing the value of `a` also changes `b` but not the other way around.

are pervasive. Ports of Node.js such as python's Twisted or ruby's EventMachine present the same basic functionality and add little theoretical value.

Current approaches to event-oriented frameworks rely on isolated memory. When developing with Node.js the programmer spawns multiple processes to handle requests in parallel. This is trivial for stateless applications. In the other case, the programmer must use some mechanism to transfer data between processes or offload the synchronization to another application (e.g. external database). Erlang's actor model suffers the same limitation at a lower level. The programmer is responsible for partitioning the application into actors. On one hand, using a single actor for a centralized resource (e.g. a lookup table) might introduce a computation bottleneck and sequentialize the execution. On the other hand, using many actors will lead to a large communication overhead. Additionally, this model implies that ownership of data belongs to a single actor while on the real world it can be shared across many entities (like a blackboard can be used simultaneously by multiple persons).

```
var http = require('http');

http.createServer(function (request, response)
{
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello World\n');
}).listen(8000);

console.log('Server running at http://localhost:8000/');
```

Program 12: Node.js' `http` libraries allow the simple creation of HTTP servers. By hiding all the protocol implementation the programmer only needs to write the application logic inside the request handler callback.

Chapter 3

EVE Language

This chapter provides a description of EVE language. Section 1 contains an overall description of language and its concepts. Section 2 starts with a brief introduction to the analysis of operational semantics and introduces the syntax used. This section also provides a detailed analysis of each of the language's components, delivering documentation of its syntax and operational semantics.

3.1 Introduction

EVE is an event-based language targeted for scalable high performance under heavy IO loads. For this purpose, EVE makes transparent use of multiple task-loops managed by worker-stealing threads. Under EVE, tasks are procedures ready to execute. They can be created manually by the programmer or automatically by the runtime system when a event is triggered. EVE tasks do not form a graph with dependencies. Instead, synchronization is maintained at the data level: each task defines the set of shared objects it reads and writes, and the runtime is responsible for scheduling compatible tasks simultaneously. Program 13 shows the Hello World program written in eve.

```
import io.stdout

main: [+stdout] () void:
    stdout.write("Hello World\n")
```

Program 13: Hello World example with EVE. Line 1 imports the `out` object to the global name-space, line 3 defines the `main` function which returns no value and receives no arguments. Additionally, write permissions are indicated for the object `out`. Line 4 writes the message to the output stream.

Correct programs cannot contain infinite loops inside a task as this could lead to live-locks. Additionally, no blocking operation is allowed in EVE. Instead, an event-oriented approach is used: any type enumerates the set of events it can trigger at its declaration. Events are named

types and can contain objects of any type. To trigger an event, the binary `!` operator is used. This operator takes the emitter object as the first operand and the event object as the second. Subscriptions to events can be added to any object accessible in the current scope using the `on` statement.

```
import io.socket.*
import util.tiemout

main: () void:
    clients: set<socket@>@

    tcp_socket.listen(8080):
        connection = [+clients] (c: connection&) bool:
            client: socket@ = c()
            clients.insert(client)

            on client data [clients]:
                message : vector<char>@ = client.read_buffer()
                @ for (c: socket@ in clients) [message, +c]:
                    c.write_buffer(message)

            on client close [+clients]:
                clients.remove(client)

            return true

        error = [+stderr] (e: error&) bool:
            stderr.write("Failed to start server: %s", error)
            return true
```

Program 14: A TCP broadcast server that accepts connections on port 8080.

There are no guarantees that tasks are executed in a strict order. To maximize performance worker threads consume tasks in a First-In Last-Out (FILO) fashion. This scheduling algorithm improves locality and is space-bounded by the recursion depth of the program. Nevertheless, older tasks can be executed first if they are stolen by another worker. Task starvation could only occur if new tasks were added in front of existing tasks. For this reason, event-originated tasks are only executed after all scheduled tasks have completed. This solution gives priority to existing work.

Shared objects are heap allocated and garbage collected. Each of these objects is paired with a control object which monitors concurrent accesses. Pointers to these objects can be copied, however the underlying control structure remains the same. This solution, guarantees that access synchronization, even conducted through different pointers, is managed by the same control object, avoiding the aliasing problem.

When a worker obtains a task it tries to acquire the set of shared objects referred one at a time. The order of acquisition is determined at runtime and is guaranteed to be the same for all workers. If any acquisition fails, the worker backs out, releasing all objects in the inverse order, and re-inserting the task at the bottom of its deque for later execution. This algorithm grows in complexity when the set of objects is large. For this reason tasks with small variable sets should be used. To avoid active starvation of tasks with large variable sets, tasks can only be re-inserted a fixed number of times. When this number is reached, the worker acquires the set of shared objects, waiting if necessary for other workers to release them. Although this solution may incur in a large performance penalty, no deadlock is created because the task currently owning the desired objects will eventually complete.

```
import io.stdout

data: vector<int>@ = [1, 2, 3]

print: [+stdout] (list: vector<int>&) void:
    for (i: int in list):
        stdout.write(i)

main: [+stdout, data] () void:
    print(data)
```

Program 15: Another simple example of the eve language. Notice how the `main` function still defines write access to the `stdout` object even though it is only used inside the `print` function. Additionally, read permissions to `data` object are requested, allowing print function access it as if it by reference.

When defining a function the programmer indicates the set of permissions required over shared objects. This set can reference any object belonging to the scope where the function is defined. *Read permissions* allow a function to obtain a constant view of an object. The programmer can request read permissions explicitly by adding the variable identifier to the permission set. *Full permissions* allow the variable to be both read and modified. To request this permission type the variable must be prepended with the plus sign (`+`). Attempting to modify a variable without full permissions will yield a compile-time error. Example definition of access permissions can be observed in program 15. Additionally, the programmer can also request *Null permissions* by prepending the variable with a minus sign (`-`). This allows a function to receive bind a reference to the variable without needing to have permissions to view or modify the object.

Parallel programs with EVE can be created by spawning new tasks. Tasks are created using the `@` statement. This statement is defined by an optional context definition and statement containing the task body. A context definition is list of permission requests (like a function's permission set) but can also contain additional local variable captures. Using the same syntax as `C++` a variable can be captured by copy or by reference using the `=` and `&` prefixes respectively. Program 16 shows how to use the `@` statement to create new tasks.

```
import io.out

prefix_sum: (n: int) void:
    sum: int = 0

    for (i: int = 1; i <= n; i++):
        sum += i

    @ [+out, =n, =sum]: out.write("prefix sum of %d = %d", n, sum)

main: () void:
    @ prefix_sum(100)
    @ prefix_sum(200)
```

Program 16: An example that computes two prefix sums in parallel and outputs the result to standart output.

Creating a task immediately schedules it for execution. However, it will only start after the termination of tasks referring to the same shared objects. Additionally, creating a task does *not* release the objects defined in the context definition. For this reason, if the child task accesses at least one object referred by its parent, it will only execute *after* the parent has completed. This can be observed in example 17.

```
print: [+out] () void:
    out.write("This comes first")

    @[+out]:
        out.write("This last")

    out.write("This one second")
```

```
This comes first
This one second
This last
```

Program 17: On the left, a program in EVE that writes three messages to standard output. On the right, the only possible result obtained by executing this code.

A `finish` block can be used to suspend a parent task until sub-task are terminated. Because sub-tasks commonly require access to the same objects as the parent task, upon entering a `finish` block, all shared objects are released. However, acquisition of these objects is only allowed by the children tasks. This property allows the programmer to assume the views over shared objects before and after the `finish` construct are consistent. Inside a `finish` block tasks can only require a subset of the parent's permission set. This guarantees that if a task enters a finish block all sub-task are able to execute without blocking for access to additional shared objects (although they might need synchronization between siblings). For this reason the inclusion of `finish` statements does not introduce deadlocks. Program 18 shows an implementation of the recursive Fibonacci method making use of `finish` for synchronization.

The `finish` statement allows the compiler many optimizations:

```
fib: (n: int) int:
    if n < 2:
        return n

    a, b : int@
    finish:
        @ [+a, =n]: a = fib(n - 1)
        @ [+b, =n]: b = fib(n - 2)

    return a + b
```

Program 18: Parallel computation of the n^{th} Fibonacci number using the `finish` block for synchronization. `int@` defines a shared object of base type `int`.

- Because the life-span of children tasks is contained in the life-span of its parent, task objects can be allocated directly on the stack instead of the heap;
- For the same reason, shared objects accessed only by the parent task and tasks spawned inside `finish` statements can be stack allocated;
- If there is no intersection between the permission sets of each sub-task, object allocation and deallocation can by omitted completely. On the other case, the allocation of the first task can assume it will always succeed;
- The `finish` statement freezes execution of the parent task. For this reason local variables can be accessed directly by reference. In this case, synchronization of tasks can be omitted if read-only access is required by all tasks.

The EVE language also include parallel `for` loops using the `@for` statement. The behavior of this construct is the same as if one task were created for each iteration, however the runtime might choose to group iterations to improve performance. Example 19 shows usage of this feature.

```
map: <type X, type Y> (x: vector<X>&, f: <(X&) Y>& f) vector<Y>@:
    y: vector<Y>@ (x.length())

    finish:
        @ for (i: int = 0; i < x.length(); i++)[&x, &f]:
            yi: Y = f(x[i])

            @ [+y, =i, &yi]:
                y[i] = yi

    return y
```

Program 19: The `map` pattern implemented using a parallel for. Note that `y` is a shared object but the function returns a normal `vector<int>`. This normally requires a copy operation but it can be bypassed using the same semantics as Return Value Optimizations.

3.2 Operational Semantics

Operational semantic analysis is a mathematical tool used to describe the expected behaviour of an application. This analysis is based on a precise description of each operation, focusing on *what* it means instead of *how* it executes. This allows a consistent view of a language despite the existence of distinct implementations, each with its unique sets of rules.

3.2.1 Syntax

Structured operational semantic rules define each operation as a transformation between an initial environment, the state of the application before the execution, and a final environment where its side-effects are visible [23]. In this document, each rule has the following format:

$$\frac{\textit{pre-conditions}}{\textit{environment} \vdash \textit{operation} \Rightarrow \textit{result}, \textit{new environment}}$$

Rule 1: An example of operational semantic rules.

For this section consider the following notation: $\Sigma = \{a, b, ...\}$ is a set of known ids, $\Pi = \{A, B, ..., \Phi\}$ is a set containing all available types, either built-in, user defined (Φ is the void type), $\Gamma = (\{\alpha, \beta, ...\}, +)$ is a group of locations where values can be stored, such as memory addresses or registers, $\Psi = \{\pi, \psi, ..., \phi\} \cup \Gamma$ is the set of all possible values a variable can have (ϕ is the value of void).

In EVE, a local environment is defined as $\mathbb{L} : \langle \mathbb{T}, \mathbb{V} \rangle$, where $\mathbb{T} : \Sigma \to \Pi$ is the *type context*, a function that maps a variable id to its static type, $\mathbb{V} : \Sigma \to \Gamma$ is the *variable context*, a function that maps a variable id to the location where it is stored. New contexts can be created using the notation $\mathbb{Y} = \mathbb{X}[a/X]$ which indicates $\mathbb{Y}(a) = X$ and $\mathbb{Y}(x) = \mathbb{X}(x), \forall x \neq a$. For convenience $\mathbb{L}_i$ is implicitly defined as $\mathbb{L}_i : \langle \mathbb{T}_i, \mathbb{V}_i \rangle$ where $\mathbb{X}_i = \mathbb{X}_{i-1}$ unless otherwise specified.

An additional global environment is defined as $\mathbb{G} : \langle \mathbb{S}, \mathbb{C}, \mathbb{R} \rangle$, where $\mathbb{S} : \Gamma \to \Psi$ is the *storage context*, a function that describes the current value stored at each location, $\mathbb{C} : \Gamma \times \Pi \to \Gamma^*$ is the *callback context*, a function that identifies for each object-event pair the set of callback functions and $\mathbb{R}$ is the set of existing *runnable* tasks. Keep in mind that, while multiple local environments can co-exist, only one global environment is present in the system. For this reason, global contexts are transformed (instead of created) using the notation $\mathbb{X} \leftarrow \mathbb{X}[a/X]$.

The *result* is a value-type pair identified as $\langle \pi : A \rangle$. Type inspection opens for observation the attributes of a type using the notation $A := \langle \langle \beta, B \rangle, \langle \zeta, C \rangle, ...; x : \langle \alpha, X \rangle, y : \langle \gamma, Y \rangle, ... \rangle$, indicating that type A is a subtype of B and C, has attributes x, y of types X, Y with location offsets α and γ. Using a similar notation, task inspection allows observation of the attributes of a task $r \in \mathbb{R} := \langle \alpha, \{\beta : \pi_\beta, \zeta : \pi_\zeta, ...\} \rangle$, where α is the location of the task, β, ζ, the locations of bound variables, and π_β, π_ζ the permissions requested . If multiple semantic rules are available

for one operation (e.g. all pre-conditions are met) the first is selected. If no rule is available, then such operation is invalid and an error will be raised during static analysis.

3.2.2 Utility Functions

This section contains the description of three functions, which will aid the definition of EVE semantics in the following parts of the document. The $callable(\Theta, \Omega, stmt)$ function creates an object which can be invoked using the n-ary parenthesis operator. This function receives three arguments: Θ is the set of formal arguments, including identifiers, types and optional default values, Ω is the permission set which identifies access permissions on shared variables, and *stmt* is an executable operation. The $call(A, f, [\pi_1, \pi_2, ...])$ function is used to execute objects created with $callable()$, where A identifies the type on which the f function will be invoked with $\pi_1, \pi_2, ...$ as arguments. If A is a function object, $call()$ executes the *stmt* operation after ensuring that the required access permissions are met. If A is not a function object, $call()$ will lookup the f object inside type A, and proceed to invoke $call(f, (), [\pi_1, \pi_2, ...])$. The third utility function is $offset(A, x)$. This function is hardware and implementation specific, and returns the memory offset where the x object is located inside an object of type A.

3.2.3 IDs

```
id
    : SIMPLE_ID
    | qualified_id
    ;
```

```
qualified_id
    : SIMPLE_ID '.' SIMPLE_ID
    | qualified_id '.' SIMPLE_ID
    ;
```

Grammar 1: Rules for identifiers.

In EVE simple identifiers are strings containing letters, numbers or the `_` (underscore) character, provided they do not start with a number. Qualified identifiers are dot separated lists of simple identifiers. Additionally the following strings are reserved as keywords: `import`, `as`, `buitin`, `type`, `def`, `var`, `self`, `operator`, `constructor`, `if`, `else`, `for`, `while`, `break`, `continue`, `return`, `pass`, `in`, `on`, `switch`, `case`, `default`, `true`, `false`. Rules for IDs can be observed in grammar 1.

3.2.4 Types

```
type
    : id ('<' type_params '>')?
    | type '@'
    | type '&'
    | '<' '(' type_list? ')' type '>'
    ;

type_param
    : constant
    | type
    | id
    ;

type_params
    : type_param
    | type_params ',' type_param
    ;

type_list
    : type
    | type_list ',' type
    ;
```

Grammar 2: Rules for types specifiers.

EVE accepts four type formats. The first format uses an identifier of a built-in or already declared type with an optional argument list. Acceptable arguments, are constant literals or other type specifiers. Additionally identifiers can also be used as type arguments if they can be determined at compile time to be either a constant or another type specifier. The second format identifies shared objects and use the normal specification with the `@` (at) suffix. Likewise, using the `&` (ampersand) suffix identifies the object reference type. Finally, the last format is used to identify function types. Rules for type specifiers can be observed in grammar 2.

Built-ins and Literals

This section contains the specification for each built-in types in EVE.

Boolean values are of type `bool` and can be either `true` or `false`. Rules 2 identify the operational semantics of these two literals:

$$\frac{}{\mathbb{L}, \mathbb{G} \vdash \texttt{true} \Rightarrow \langle \texttt{true} : \texttt{bool} \rangle, \mathbb{L}, \mathbb{G}}$$

$$\frac{}{\mathbb{L}, \mathbb{G} \vdash \texttt{false} \Rightarrow \langle \texttt{false} : \texttt{bool} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 2: Semantic rules for boolean literals.

Integer values are of type `int<N>`, where `N` identifies the number of bits used to store the value. If omitted, `N` is assumed to be of the CPU word size of the target platform. By default integer literals can be written in base 10. Literals that start with `0` (zero) are considered to be in base 8. Literals that start with `0x` are parsed as in base 16. Rule 3 identifies the operational semantics of integer literals:

$$\frac{\pi \textit{ is a int literal}}{\mathbb{L}, \mathbb{G} \vdash \pi \Rightarrow \langle \pi : \texttt{int<}\lceil \log_2(\pi) \rceil\texttt{>} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 3: Semantic rules for integer literals.

Floating Point values are of type `float<N>`, where `N` identifies the number of bits used to store the value. Float literals are stored as specified by the IEEE 754 standard. If `N` is omitted, it is assumed to be the maximum provided by the target platform such that built-in operations such as *sin* and *cos* are available. Rule 4 identifies the semantics for floating point literals.

$$\frac{\begin{array}{c}\pi \textit{ is a float literal} \\ \psi \textit{ is the maximum float size}\end{array}}{\mathbb{L}, \mathbb{G} \vdash f \Rightarrow \langle \pi : \texttt{float<}\psi\texttt{>} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 4: Semantic rules for floating point literals.

Character values are of type `char`. Character literals are enclosed in `''` (single-quotes), and can be normal characters such as letters `'a'`, or special, backslash escaped characters such as `'\n'` or `'\011'`. Rule 5 displays the semantics for character literals.

$$\frac{c \textit{ is a char literal}}{\mathbb{L}, \mathbb{G} \vdash c \Rightarrow \langle c : \texttt{char} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 5: Semantic rules for character literals.

String values are of type `string`. String literals are enclosed in `""` (double-quotes), and can contain any character used in a `char` type. Rule 6 displays the semantics for string literals.

$$\frac{\begin{array}{c}c \textit{ is a string literal} \\ n \textit{ is the length of } c\end{array}}{\mathbb{L}, \mathbb{G} \vdash c \Rightarrow \langle (c, n) : \texttt{string} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 6: Semantic rules for string literals.

Function values are of type `<(...) R>` where `R` is the type of the returned value. Function definitions and function literals will be described in sections 3.2.5 and 3.2.6.

Type conversion

EVE allows for implicit cast and conversion of variables to different types. Rules 7 and 8 describe the semantics of these operation, where *convert* identifies conversion where temporary variables may be allocated and *cast* means emplace conversion where they are not required. The identity operation, *id*, requires no transformation, $deref_1$ retrieves the value from an object reference, $deref_2$ obtains an object reference from a shared object, the *subclass* rule transforms a reference to a sub-class to one of its parent classes, and *construct* creates a new object of the desired type. Additionally, *transitive* rules allow for successive casts and conversions between types.

$$id\ \frac{}{\mathbb{L}, \mathbb{G} \vdash cast(\pi, A, A) \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G}}$$

$$deref_1\ \frac{}{\mathbb{L}, \mathbb{G} \vdash cast(\pi, A\texttt{\&}, A) \Rightarrow \langle \mathbb{S}(\pi) : A \rangle, \mathbb{L}, \mathbb{G}}$$

$$deref_2\ \frac{\begin{array}{c}\mathbb{T} \vdash A\texttt{@} := \langle ..., object : \langle \delta : A\texttt{\&} \rangle, ... \rangle \\ \mathbb{G} \vdash \mathbb{S}(\pi + \delta) = \psi\end{array}}{\mathbb{L}, \mathbb{G} \vdash cast(\pi, A\texttt{@}, A\texttt{\&}) \Rightarrow \langle \psi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

$$subclass\ \frac{\begin{array}{c}\mathbb{T} \vdash A < B \\ \mathbb{T} \vdash A := \langle ..., \langle \beta, B \rangle, ...; ... \rangle \\ \psi = \pi + \beta\end{array}}{\mathbb{L}, \mathbb{G} \vdash cast(\pi, A\texttt{\&}, B\texttt{\&}) \Rightarrow \langle \psi : B\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

$$transitive\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash cast(\pi, A, B) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash cast(\psi, B, C) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash cast(\pi, A, C) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 7: Semantic rules for type casting.

$$cast\ \frac{\mathbb{L}, \mathbb{G} \vdash cast(\pi, A, B) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash convert(\pi, A, B) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G}}$$

$$construct\ \frac{\begin{array}{c}\mathbb{T} \vdash B := \langle ...; ..., constructor : \langle \lambda : \texttt{<(}B\texttt{\&}, A'\texttt{)}\,B\texttt{\&>} \rangle, ... \rangle \\ \mathbb{L}, \mathbb{G} \vdash new(\mathbb{V}, B) \Rightarrow \langle \psi, B\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash convert(\pi, A, A') \Rightarrow \langle \pi' : A' \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash \lambda(\psi, \pi') \Rightarrow \langle \tau : B\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash convert(\pi, A, B) \Rightarrow \langle \mathbb{S}(\tau) : B \rangle, \mathbb{L}, \mathbb{G}}$$

$$transitive\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash convert(\pi, A, B) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash convert(\psi, B, C) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash convert(\pi, A, C) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 8: Semantic rules for type conversion.

3.2.5 Expressions

```
expr
    : expr_inline
    | expr_block
    ;

expr_block
    : expr_block_unary
    | expr_block_binary
    | expr_block_n_ary
    | expr_block_lambda
    ;
```

```
expr_inline
    : CONSTANT
    | SELF
    | SIMPLE_ID
    | expr_inline '.' SIMPLE_ID
    | '(' expr_inline ')'
    | expr_inline_unary
    | expr_inline_binary
    | expr_inline_n_ary
    | expr_inline_lambda
    ;
```

Grammar 3: Grammar rules for expressions.

Expressions are operations that return a meaningful values. They do not modify the local environment, but may modify the global environment. In EVE, nine types of expressions are covered: constants, self, variable access, field access, unary, binary, n-ary, parentheses and lambda. The syntax for expressions is described in grammar 3. Expressions are syntactically divided into two groups: `inline` and `block`. While inline expressions do not use any delimiter, block expressions must be terminated with a new line. The syntax and semantics for each expression type are detailed in the following section.

Constant

Constant expressions take the value from a constant literal, without modifying the context. Semantics for these expressions are described in rule 9.

$$\frac{\begin{array}{c}\textit{lit is a constant literal} \\ \textit{lit} = \langle \pi : A \rangle\end{array}}{\mathbb{L}, \mathbb{G} \vdash \textit{lit} \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 9: Semantic rules for constant expressions.

Self

The `self` expression returns a reference to a variable of the enclosing type. Semantics for this expression is described in rule 10.

$$\frac{\begin{array}{c}\mathbb{V} \vdash \mathbb{V}(\texttt{self}) = \alpha \\ \mathbb{T} \vdash \mathbb{T}(\texttt{self}) = A@\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{self} \Rightarrow \langle \alpha : A@ \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 10: Semantic rules for the `self` expression.

Variable

The variable expression returns a reference to a previously declared variable. Semantics for this expression is described in rule 11.

$$\frac{\begin{array}{c}\mathbb{V} \vdash \mathbb{V}(v) = \alpha \\ \mathbb{T} \vdash \mathbb{T}(v) = A@\end{array}}{\mathbb{L}, \mathbb{G} \vdash v \Rightarrow \langle \alpha : A@ \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 11: Semantic rules for variable expression.

Field Access

The field access expressions enable access to inner declarations of a type or variable reference. The semantic description of these expressions is available in rule 12.

$$\text{type-type}\ \frac{\mathbb{T} \vdash \mathbb{T}(a) = A \quad \mathbb{T} \vdash A := \langle \dots ; \dots, x : \langle \psi, X \rangle , \dots \rangle}{\mathbb{L}, \mathbb{G} \vdash a \,.\, x \Rightarrow \langle \psi : X \rangle, \mathbb{L}, \mathbb{G}}$$

$$\text{type-ref}\ \frac{\mathbb{T} \vdash \mathbb{T}(a) = A \quad \mathbb{T} \vdash A := \langle \dots ; \dots, x : \langle \pi, X\,\&\rangle , \dots \rangle}{\mathbb{L}, \mathbb{G} \vdash a \,.\, x \Rightarrow \langle \pi : X\,\&\rangle, \mathbb{L}, \mathbb{G}}$$

$$\text{ref-ref}\ \frac{\mathbb{T} \vdash A := \langle \dots ; \dots, x : \langle \delta, X\,\&\rangle , \dots \rangle \quad \mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A\,\&\rangle, \mathbb{L}, \mathbb{G} \quad \psi = \pi + \delta}{\mathbb{L}, \mathbb{G} \vdash expr \,.\, x \Rightarrow \langle \psi : X\,\&\rangle, \mathbb{L}, \mathbb{G}}$$

$$\text{ref-cast-ref}\ \frac{\begin{array}{c}\mathbb{T} \vdash A < B \quad \mathbb{T} \vdash B := \langle \dots ; \dots, x : \langle \delta, X\,\&\rangle , \dots \rangle \\ \mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A\,\&\rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash cast(\pi, A\,\&, B\,\&) \Rightarrow \langle \psi : B\,\&\rangle, \mathbb{L}, \mathbb{G} \quad \tau = \psi + \delta\end{array}}{\mathbb{L}, \mathbb{G} \vdash expr \,.\, x \Rightarrow \langle \tau : X\,\&\rangle, \mathbb{L}, \mathbb{G}}$$

Rule 12: Semantic rules for field access expressions.

Parentheses

Parentheses expressions are syntactic only elements. For this reason the semantic value of these expressions is deferred to the wrapped expression, as indicated in rule 13.

$$\frac{\mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash (\ expr\ ) \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 13: Semantic rules for parentheses expressions.

Unary

```
expr_inline_unary
    : prefix_op expr_inline
    | incr_op  expr_inline
    | expr_inline incr_op
    ;

expr_block_unary
    : prefix_op expr_block
    : incr_op expr_block
    ;
```

```
incr_op
    : "++"
    | "--"
    ;

prefix_op
    : '+'
    | '-'
    | '~'
    | '!'
    ;
```

Grammar 4: Grammar rules for unary expressions.

Unary expressions apply transformations to a single operand. Prefix operations are one of the two types of unary expressions. These operations modify the returned value of the argument expression. Increment and decrement operations are the second type of unary expressions. These operations receive an object reference and modify the stored value. Syntax rules for these expressions are available in grammar 4 and their semantic descriptions in rules 14.

$$\text{prefix}\ \frac{\begin{array}{c} op \in \{\texttt{+}, \texttt{-}, \texttt{\~{}}, \texttt{!}\} \\ \mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A \rangle, op) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \end{array}}{\mathbb{L}, \mathbb{G} \vdash op\ \ expr \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G}}$$

$$\text{pre-incr}\ \frac{\begin{array}{c} op \in \{\texttt{++}, \texttt{--}\} \\ \mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A \rangle, op) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \end{array}}{\mathbb{L}, \mathbb{G} \vdash op\ \ expr \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G}}$$

$$\text{post-incr}\ \frac{\begin{array}{c} op \in \{\texttt{++}, \texttt{--}\} \\ \mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A \rangle, op, [\langle 1 : \texttt{int} \rangle]) \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \end{array}}{\mathbb{L}, \mathbb{G} \vdash expr\ \ op \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 14: Semantic rules for unary expressions.

Binary

```
expr_inline_binary
    : expr_inline binary_op '=' expr_inline
    | expr_inline binary_op expr_inline
    | expr_inline comp_op expr_inline
    ;

expr_block_binary
    : expr_inline binary_op '=' expr_block
    | expr_inline binary_op expr_block
    | expr_inline comp_op expr_block
    ;
```

```
binary_op:
    : '+'
    | '-'
    | '*'
    | '/'
    | '^'
    | '&'
    | '|'
    | '%'
    | "<<"
    | ">>"
    | "**"
    ;
```

```
comp_op
    : '<'
    | '>'
    | "<="
    | ">="
    | "=="
    | "!="
    | "==="
    | "!=="
    ;
```

Grammar 5: Grammar rules for binary expressions.

Binary expressions apply transformations to two operands. Three types of operators are available, including arithmetic (`+`, `-`, `*`, ...), boolean (`<`, `>`, `<=`, `>=`, ...), and assignment (`=`). Assignment expressions are a subtype of binary expressions, and can be combined with any arithmetic operator into a single operation-assignment expression. Syntax rules for binary expressions are

available in grammar 5 and their semantic descriptions in rules 15.

$$op \frac{op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{!}, \texttt{/}, \texttt{\^{}}, \texttt{\&}, \texttt{|}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{**}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}, \texttt{===}, \texttt{!==}\} \quad \mathbb{L}, \mathbb{G} \vdash expr_1 \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash expr_2 \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A \rangle, op, [\langle \psi : B \rangle]) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash expr_1 \ \ op \ \ expr_2 \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}}$$

$$assign_{op} \frac{\mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A\texttt{\&} \rangle, \texttt{=}, [\langle \psi : B \rangle]) \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash assign(\langle \pi : A\texttt{\&} \rangle, \langle \psi : B \rangle) \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

$$assign_{def} \frac{\mathbb{L}, \mathbb{G} \vdash convert(\psi, B, A) \Rightarrow \langle \tau : A \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{G} \vdash \mathbb{S} \leftarrow \mathbb{S}\,[\tau/\pi]}{\mathbb{L}, \mathbb{G} \vdash assign(\langle \pi : A\texttt{\&} \rangle, \langle \psi : B \rangle) \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

$$assign \frac{\mathbb{L}, \mathbb{G} \vdash expr_1 \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash expr_2 \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash assign(\langle \pi : A\texttt{\&} \rangle, \langle \psi : B \rangle) \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash expr_1 \ \texttt{=} \ expr_2 \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

$$op\text{-}assign_{op} \frac{op \in \{\texttt{+=}, \texttt{-=}, \texttt{*=}, \texttt{!=}, \texttt{/=}, \texttt{\^{}=}, \texttt{\&=}, \texttt{|=}, \texttt{\%=}, \texttt{<<=}, \texttt{>>=}, \texttt{**=}\} \quad \mathbb{L}, \mathbb{G} \vdash expr_1 \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash expr_2 \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A\texttt{\&} \rangle, op, [\langle \psi : B \rangle]) \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash expr_1 \ \ op \ \ expr_2 \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

$$op\text{-}assign_{def} \frac{op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{!}, \texttt{/}, \texttt{\^{}}, \texttt{\&}, \texttt{|}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{**}\} \quad \mathbb{L}, \mathbb{G} \vdash expr_1 \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash expr_2 \Rightarrow \langle \psi : B \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash call(\langle \pi : A \rangle, op, [\langle \psi : B \rangle]) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G} \quad \mathbb{L}, \mathbb{G} \vdash assign(\langle \pi : A\texttt{\&} \rangle, \langle \tau : C \rangle) \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash expr_1 \ \ op\texttt{=} \ \ expr_2 \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 15: Semantic rules for binary expressions.

N-ary

```
expr_inline_n_ary
    : expr_inline '[' expr_inline_list? ']'
    | expr_inline '(' expr_inline_list? ')'
    ;

expr_block_n_ary
    : expr_inline '[' expr_inline_list? ']' ':' EOL INDENT argument_named+ DEDENT
    | expr_inline '(' expr_inline_list? ')' ':' EOL INDENT argument_named+ DEDENT
    ;

argument_named
    : SIMPLE_ID '=' expr_inline EOL
    | SIMPLE_ID '=' expr_block
    ;
```

Grammar 6: Grammar rules for n-ary expressions.

N-ary expressions apply transformations to any positive number of operands. These expressions include the invoke operator `()`, and the indexing operator `[]`. Syntax rules for n-ary expressions are available in grammar 6 and their semantic descriptions in rules 16.

$$op\frac{\begin{array}{c}\mathbb{L},\mathbb{G}\vdash expr_b \Rightarrow \langle \pi : B\& \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G}\vdash expr_1 \Rightarrow \langle \psi_1 : A_1 \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{L},\mathbb{G}\vdash expr_n \Rightarrow \langle \psi_n : A_n \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G}\vdash call(\langle \pi : B\& \rangle, (), [\langle \psi_1 : A_1 \rangle, \langle \psi_2 : A_2 \rangle, ..., \langle \psi_n : A_n \rangle]) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L},\mathbb{G}\vdash expr_b(expr_1, expr_2, ...) \Rightarrow \langle \tau : C \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 16: Semantic rules for n-ary expressions.

Lambda

```
expr_inline_lambda
    : perms '(' parameters? ')' type? ':' expr_inline
    ;

expr_block_lambda
    : perms '(' parameters? ')' type? ':' EOL stmt_block
    ;
```

Grammar 7: Grammar rules for lambda expressions.

Lambda expressions are used to create function literals. The syntax rules for these expressions are available in grammar 7 and their semantics in rules 17.

$$\frac{\begin{array}{c}\mathbb{L},\mathbb{G} \vdash expr_1 \to \langle \pi_1, : X_1' \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash expr_2 \to \langle \pi_2, : X_2' \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{L},\mathbb{G} \vdash convert(\pi_1, X_1', X_1) \to \langle \psi_1, : X_1 \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash convert(\pi_2, X_2', X_2) \to \langle \psi_2, : X_2 \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{T} \vdash \mathbb{T}' = \{a : \mathbb{T}(a), b : \mathbb{T}(b), ...\} \\ \mathbb{V} \vdash \mathbb{V}' = \{a \to \mathbb{V}(a), b \to \mathbb{V}(b), ...\} \\ \Theta = \{\langle x_1 : X_1 = \psi_1 \rangle, \langle x_2 : X_2 = \psi_2 \rangle, ...\} \\ \Omega = \{\eta_a : \mathbb{V}(a), \eta_b : \mathbb{V}(b), ...\} \\ \mathbb{L},\mathbb{G} \vdash callable(\Theta, \Omega, stmt) \Rightarrow \langle \alpha : \texttt{<(}X_1, X_2, ...\texttt{)}\, R\texttt{>\&} \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L},\mathbb{G} \vdash \texttt{[}\eta_a a, \eta_b b, ...\texttt{]}\ \texttt{(}x_1 : X_1 = expr_1,\ x_2 : X_2 = expr_2,\ ...\texttt{)}\, R\texttt{:}\ stmt \Rightarrow \langle \alpha : \texttt{<(}X_1, X_2, ...\texttt{)}\, R\texttt{>\&} \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 17: Semantic rules for lambda expressions.

3.2.6 Statements

```
stmt
    : PASS EOL
    | stmt_expr
    | stmt_while
    | stmt_for
    | stmt_for_in
    | stmt_return
    | stmt_par
    | stmt_on
    | stmt_finish
    | definition
    ;
```

Grammar 8: Grammar rules for statements.

Statements are operations that do not meaningful values. Instead, their execution modifies either the local environment, the global environment, or both. In EVE, twelve types of statements are covered: block, pass, expression, while, for, for-in, return, par, finish, on, function definition, variable definition and type definition. The syntax for statements is described in grammar 8. The syntax and semantics for each statement type is covered in the following section.

Block Statement

The block statement is a list of sub-statements that are executed in order. The semantics of block statements are described in rule 18.

$$\frac{\begin{array}{c}\mathbb{L}_1, \mathbb{G} \vdash stmt_1 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}_2, \mathbb{G} \\ \mathbb{L}_2, \mathbb{G} \vdash stmt_2 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}_3, \mathbb{G} \\ \vdots \\ \mathbb{L}_{n-1}, \mathbb{G} \vdash stmt_n \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}_n, \mathbb{G}\end{array}}{\mathbb{L}_1, \mathbb{G} \vdash stmt_1\ stmt_2\ ...\ stmt_n \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}_n, \mathbb{G}}$$

Rule 18: Semantic rule for block statements.

Pass Statement

The `pass` statement is used when a statement is syntacticly necessary but no operation is desired. This statement does not modify any environment and can be ignored during execution, as can be observed in rule 19.

$$\frac{}{\mathbb{L}, \mathbb{G} \vdash \texttt{pass EOL} \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 19: Semantic rule for the `pass` statement.

Expression Statement

```
stmt_expr
    : expr_inline EOL
    | expr_block
    ;
```

Grammar 9: Grammar rules for expression statements.

The expression statement executes a single expression and ignores its return value. The semantics of this operation can be observed in rule 20.

$$\frac{\mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G}}{\mathbb{L}, \mathbb{G} \vdash expr\ \texttt{EOL} \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 20: Semantic rule for the expression statement.

If Statement

```
stmt_if
    : IF expr_inline ':' stmt_or_block
    | IF expr_inline ':' EOL stmt_block ELSE ':' EOL stmt_block
    ;
```

Grammar 10: Grammar rules for `if` statements.

The `if` statement evaluates the boolean value of *expr* and if equals to `true` executes $stmt_1$ otherwise executes $stmt_2$. The syntax rules for the `if` statement are available in grammar 10 and their semantics in rules 21.

$$true\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash convert(\pi, A, \texttt{bool}) \Rightarrow \langle \texttt{true} : \texttt{bool} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash stmt_1 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{if}\ \ expr\texttt{:}\ stmt_1\ \ \texttt{else:}\ stmt_2 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

$$false\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash convert(\pi, A, \texttt{bool}) \Rightarrow \langle \texttt{false} : \texttt{bool} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash stmt_2 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{if}\ \ expr\texttt{:}\ stmt_1\ \ \texttt{else:}\ stmt_2 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 21: Semantic rules for the `if` statement.

While Statement

```
stmt_while
    : WHILE expr_inline ':' stmt_or_block
    ;
```

Grammar 11: Grammar rules for `while` statements.

The `while` statement evaluates the boolean value of *expr* and if it equals to `true` executes *stmt* and repeats. The syntax rules for the `while` statement are available in grammar 11 and their semantics in rules 22.

$$true\frac{\begin{array}{c}\mathbb{L},\mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash convert(\pi, A, \texttt{bool}) \Rightarrow \langle \texttt{true} : \texttt{bool} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L},\mathbb{G} \vdash \texttt{while}\ expr\texttt{:}\ stmt \Rightarrow \mathbb{L},\mathbb{G} \vdash \texttt{while}\ expr\texttt{:}\ stmt}$$

$$false\frac{\begin{array}{c}\mathbb{L},\mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash convert(\pi, A, \texttt{bool}) \Rightarrow \langle \texttt{false} : \texttt{bool} \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L},\mathbb{G} \vdash \texttt{while}\ expr\texttt{:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 22: Semantic rules for the `while` statement.

For Statement

```
stmt_for
    : FOR '(' for_init ';' for_cond ';' for_incr ')' ':' stmt_or_block
    | '@' FOR '(' for_init ';' for_cond ';' for_incr ')' perms? ':' stmt_or_block
    ;

stmt_for_in
    : FOR '(' SIMPLE_ID IN expr_inline ')' ':' stmt_or_block
    | '@' FOR '(' SIMPLE_ID IN expr_inline ')' perms? ':' stmt_or_block
    ;


for_init
    : /* EMPTY */
    | expr_inline
    | definition_variable
    ;

for_cond
    : /* EMPTY */
    | expr_inline
    ;

for_incr
    : /* EMPTY */
    | expr_inline
    ;
```

Grammar 12: Grammar rules for `for` statements.

The `for` statement creates a new local environment where *init* is executed. After this first step, the boolean value of *cond* is evaluated. If it is equal to `true`, *stmt* is executed proceeded by

incr. This process repeats until *cond* evaluates to `false`. The parallel version of this statement is semantically equivalent to executing all *stmt* instances in parallel. However, a specific implementation may choose different parallelization mechanisms or even a sequential implementation. The syntax rules for the `for` statement are available in grammar 12 and their semantics in rules 23.

$$sequential\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash init \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{L}', \mathbb{G} \vdash loop(cond, incr, stmt) \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{for (}init\texttt{;}\ cond\texttt{;}\ incr\texttt{)}\texttt{:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

$$parallel\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash init \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{L}', \mathbb{G} \vdash loop(cond, incr, \texttt{@[}\eta_a a, \eta_b b, ...\texttt{]}\texttt{:}\ stmt) \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L} \vdash \texttt{@ for (}init\texttt{;}\ cond\texttt{;}\ incr\texttt{)[}\eta_a a, \eta_b b, ...\texttt{]}\texttt{:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

$$true\ \frac{\begin{array}{c}\mathbb{L}', \mathbb{G} \vdash cond \Rightarrow \langle \pi : A \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{L}', \mathbb{G} \vdash convert(\pi, A, \texttt{bool}) \Rightarrow \langle \texttt{true} : \texttt{bool} \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{L}', \mathbb{G} \vdash stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}'', \mathbb{G} \\ \mathbb{L}', \mathbb{G} \vdash incr \Rightarrow \langle \psi : B \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L}', \mathbb{G} \vdash loop(cond, incr, stmt) \Rightarrow \mathbb{L}', \mathbb{G} \vdash loop(cond, incr, stmt)}$$

$$false\ \frac{\begin{array}{c}\mathbb{L}', \mathbb{G} \vdash cond \Rightarrow \langle \pi : A \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{L}', \mathbb{G} \vdash convert(\pi, A, \texttt{bool}) \Rightarrow \langle \texttt{false} : \texttt{bool} \rangle, \mathbb{L}', \mathbb{G}\end{array}}{\mathbb{L}', \mathbb{G} \vdash loop(cond, incr, stmt) \Rightarrow \mathbb{L}', \mathbb{G} \vdash \langle \phi : \Phi \rangle, \mathbb{L}', \mathbb{G}}$$

Rule 23: Semantic rules for the `for` statement.

For-in Statement

The `for-in` statement executes *stmt* once for each element inside an iterable object. The semantics for this statement are described using the regular `for` in rule 24.

$$sequential\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash \texttt{e :=}expr \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{T} \vdash A < \texttt{iterable<}X\texttt{>} \\ \mathbb{L}, \mathbb{G} \vdash \texttt{for (it := e.begin(); it != e.end(); ++it):} \\ a\ \texttt{:= *it},\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{for (}a\ \texttt{in}\ expr\texttt{)}\texttt{:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

$$parallel\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash \texttt{e :=}expr \Rightarrow \langle \pi : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{T} \vdash A < \texttt{iterable<}X\texttt{>} \\ \mathbb{L}, \mathbb{G} \vdash \texttt{@ for (it := e.begin(); it != e.end(); ++it):} \\ a\ \texttt{:= *it},\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{@ for (}a\ \texttt{in}\ expr\texttt{)}\texttt{:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 24: Semantic rules for the `for-in` statement.

Return Statement

```
stmt_return
    : RETURN expr_inline EOL
    | RETURN expr_block
    ;
```

Grammar 13: Grammar rules for `return` statements.

The `return` statement evaluates the result of *expr* and sets that value as return value of the executing function. This operation is executed by assigning to reference variable created by *call*() indicating the location where to store the result.

$$\frac{\begin{array}{c}\mathbb{V} \vdash \mathbb{V}(\texttt{return}) = \alpha \\ \mathbb{T} \vdash \mathbb{T}(\texttt{return}) = R\& \\ \mathbb{L}, \mathbb{G} \vdash expr \Rightarrow \langle \pi : A \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash assign(\langle \alpha : R\& \rangle, \langle \pi : A \rangle) \Rightarrow \langle \alpha, R\& \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{return}\, expr \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 25: Semantic rule for the `return` statement.

Par Statement

```
stmt_par
    : '@' perms? ':' stmt_or_block
    ;

perm_list
    : perm
    | perm_list ',' perm
    ;
```

```
perms
    : '[' perm_list? ']'
    ;

perm
    : id
    | '&' id
    | '=' id
    | '-' id
    | '+' id
    ;
```

Grammar 14: Grammar rules for `par` statements.

The `par` statement schedules for parallel execution another statement by creating a *callable* object and adding it to the set of runnable tasks. The local execution environment for this new statement is a subset of the current environment, described using the permission set $\eta_a a, \eta_b b, ...$ where a, b are variable identifiers and η_a, η_b are the required access permissions which can be the default *read-only* permission, the *full* permission (`+`) or *none* (`-`). Additionally, variables can be passed by *value* (`=`) or by *reference* (`&`). The syntax rules for the `par` statement are available in grammar 14 and their semantics in rule 26.

$$\frac{\begin{array}{c}\mathbb{L} \vdash \mathbb{V}(\texttt{finish}) = \pi \\ \mathbb{T} \vdash \mathbb{T}' = \{a : \mathbb{T}(a), b : \mathbb{T}(b), ...\} \\ \mathbb{V} \vdash \mathbb{V}' = \{a \to \mathbb{V}(a), b \to \mathbb{V}(b), ...\} \\ \Omega = \{\eta_a : \mathbb{V}(a), \eta_b : \mathbb{V}(b), ...\} \\ \mathbb{L}', \mathbb{G} \vdash \mathit{callable}(\varnothing, \Omega, \mathit{stmt}) \Rightarrow \langle \alpha : \texttt{<() void>\&} \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{G} \vdash \mathbb{R} \leftarrow \mathbb{R} \cup \{\alpha\} \\ \mathbb{G} \vdash \mathbb{S} \leftarrow \mathbb{S}[\pi / \mathbb{S}(\pi) + 1]\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{@[}\eta_a a, \eta_b b, ...\texttt{]} : \mathit{stmt} \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 26: Semantic rule for the `par` statement.

Finish Statement

```
stmt_finish
    : FINISH ':' stmt_or_block
    ;
```

Grammar 15: Grammar rules for `finish` statements.

The `finish` statement enables synchronization with parallel tasks by executing other work until their are complete. This statement creates a local variable which is incremented each time a task is created (with the `par` statement or the parallel versions of `for` / `for-in`). When leaving a finish block, the execution of the remaining statements is postponed until this variable reaches zero. The syntax rules for the `finish` statement are available in grammar 15 and their semantics in rule 26.

$$wait\ \frac{\begin{array}{c}\mathbb{G} \vdash \mathbb{S}(\pi) > 0 \\ \mathbb{G} \vdash \textit{execute other runnable task or sleep}\end{array}}{\mathbb{L}, \mathbb{G} \vdash wait(\pi) \Rightarrow \mathbb{L}, \mathbb{G} \vdash wait(\pi)}$$

$$no\text{-}wait\ \frac{\mathbb{G} \vdash \mathbb{S}(\pi) = 0}{\mathbb{L}, \mathbb{G} \vdash wait(\pi) \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

$$\frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash new(\mathbb{V}, \texttt{int}) \Rightarrow \langle \pi, \texttt{int\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{G} \vdash \mathbb{S} \leftarrow \mathbb{S}[\pi/0] \\ \mathbb{V} \vdash \mathbb{V}' = \mathbb{V}[\texttt{finish}/\pi] \\ \mathbb{L}', \mathbb{G} \vdash stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}'', \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash wait(\pi) \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash \texttt{finish:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 27: Semantic rule for the `finish` statement.

On Statement

```
stmt_on
    : ON expr_inline SIMPLE_ID '=' expr_inline EOL
    | ON expr_inline SIMPLE_ID '=' expr_block
    | ON expr_inline SIMPLE_ID SIMPLE_ID? perms? ':' stmt_or_block
    ;
```

Grammar 16: Grammar rules for `on` statements.

The `on` statement enables the programmer to set up callbacks for events. This statement can be used with either a function literal or a statement, in which case a function literal will be created implicitly receiving a single argument (the event). The syntax rules for the `on` statement are available in grammar 16 and their semantics in rule 28.

$$\frac{\begin{gathered}\mathbb{T} \vdash X < \texttt{emitter<}A\texttt{>} \\ \mathbb{L}, \mathbb{G} \vdash expr_1 \Rightarrow \langle \alpha : X\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash expr_2 \Rightarrow \langle \beta : \texttt{<(A\&) bool>\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{G} \vdash \mathbb{R}[\alpha \times A / \mathbb{R}(\alpha \times A) \cup \beta]\end{gathered}}{\mathbb{L}, \mathbb{G} \vdash \texttt{on}\, expr_1\, A \texttt{=} expr_2 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

$$\frac{\begin{gathered}\mathbb{T} \vdash X < \texttt{emitter<}A\texttt{>} \\ \mathbb{L}, \mathbb{G} \vdash expr_1 \Rightarrow \langle \alpha : X\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{T} \vdash \mathbb{T}' = \{a : \mathbb{T}(a), b : \mathbb{T}(b), ...\} \\ \mathbb{V} \vdash \mathbb{V}' = \{a \to \mathbb{V}(a), b \to \mathbb{V}(b), ...\} \\ \Theta = \{\langle x : A\texttt{\&} \rangle\} \\ \Omega = \{\eta_a : \mathbb{V}(a), \eta_b : \mathbb{V}(b), ...\} \\ \mathbb{L}', \mathbb{G} \vdash callable(\Theta, \Omega, stmt) \Rightarrow \langle \beta : \texttt{<(}A\texttt{\&) bool>\&} \rangle, \mathbb{L}', \mathbb{G} \\ \mathbb{G} \vdash \mathbb{R}[\alpha \times A / \mathbb{R}(\alpha \times A) \cup \beta]\end{gathered}}{\mathbb{L}, \mathbb{G} \vdash \texttt{on}\, expr_1\, A\, x\, \texttt{[}\eta_a a, \eta_b b, ...\texttt{]} : stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}, \mathbb{G}}$$

Rule 28: Semantic rule for the `on` statement.

Function Definition

```
definition_function
    : func_name ':' templates? perms? '(' parameters? ')' type? ':' stmt_or_block
    ;

templates
    : '<' template_list '>'
    ;

template_list
    : template
    | template_list ',' template
    ;

template
    : (VAR|TYPE) SIMPLE_ID ELIPSIS?
    | VAR SIMPLE_ID '=' CONSTANT
    | TYPE SIMPLE_ID '=' type
    ;

func_name
    : CONSTRUCTOR
    | SIMPLE_ID
    | operator
    ;

operator
    : incr_op
    | prefix_op
    | comp_op
    | binary_op
    | binary_op '='
    | '[' ']'
    | '(' ')'
    ;

parameters
    : parameter
    | parameters ',' parameter
    ;

parameter
    : SIMPLE_ID ':' type ('=' expr_inline)?
    | SELF
    ;
```

Grammar 17: Grammar rules for function definitions.

The function definition statement is the second method of creating callable objects. The syntax rules for these statements are available in grammar 17 and their semantics in rules 29.

$$\frac{\begin{array}{c}\mathbb{L},\mathbb{G} \vdash expr_1 \rightarrow \langle \pi_1, : X_1' \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash expr_2 \rightarrow \langle \pi_2, : X_2' \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{L},\mathbb{G} \vdash convert(\pi_1, X_1', X_1) \rightarrow \langle \psi_1, : X_1 \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L},\mathbb{G} \vdash convert(\pi_2, X_2', X_2) \rightarrow \langle \psi_2, : X_2 \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{T} \vdash \mathbb{T}' = \{a : \mathbb{T}(a), b : \mathbb{T}(b), ...\} \\ \mathbb{V} \vdash \mathbb{V}' = \{a \rightarrow \mathbb{V}(a), b \rightarrow \mathbb{V}(b), ...\} \\ \Theta = \{\langle x_1 : X_1 = \psi_1 \rangle, \langle x_2 : X_2 = \psi_2 \rangle, ...\} \\ \Omega = \{\eta_a : \mathbb{V}(a), \eta_b : \mathbb{V}(b), ...\} \\ \mathbb{L},\mathbb{G} \vdash callable(\Theta, \Omega, stmt) \Rightarrow \langle \alpha : \texttt{<(}X_1, X_2, ...\texttt{)}\, R\texttt{>\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{V}^* = \mathbb{V}[f/\alpha] \\ \mathbb{T}^* = \mathbb{T}[f/\texttt{<(}X_1, X_2, ...\texttt{)}\, R\texttt{>\&}] \end{array}}{\mathbb{L},\mathbb{G} \vdash f\texttt{:[}\eta_a a, \eta_b b, ...\texttt{]}\ \texttt{(}x_1 : X_1 = expr_1,\ x_2 : X_2 = expr_2,\ ...\texttt{)}\, R\texttt{:}\ stmt \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G}}$$

Rule 29: Semantic rules for function definitions.

Type Definition

```
definition_type
    : TYPE SIMPLE_ID templates? parents? ':' EOL INDENT definition+ DEDENT
    | TYPE SIMPLE_ID templates? parents? ':' EOL INDENT PASS EOL DEDENT
    ;

parents
    : '(' type_list ')'
    ;
```

Grammar 18: Grammar rules for type definitions.

Type definitions enable the creation of new types either by extending existing types using multiple inheritance, by composite creation with other previously defined types or both. The syntax rules for variable definitions are available in grammar 18 and their semantics in rules 30.

$$\frac{\begin{array}{c}\mathbb{T} \vdash T(p_1) = P_1,\ T(p_2) = P_2, ... \\ \mathbb{T} \vdash \alpha_1 = offset(A, p_1),\ \alpha_1 = offset(A, p_2), ... \\ A = \langle \langle \alpha_1, P_1 \rangle, \langle \alpha_2, P_2 \rangle, ...; \varnothing \rangle \\ \mathbb{L} \vdash \mathbb{T}_1' = \mathbb{T}[a/A,\ \texttt{self}/A] \\ \mathbb{L}_1', \mathbb{G} \vdash def_1 \Rightarrow \langle \pi_1 : X_1 \rangle, \mathbb{L}_2', \mathbb{G} \\ \mathbb{L}_2', \mathbb{G} \vdash def_2 \Rightarrow \langle \pi_2 : X_1 \rangle, \mathbb{L}_3', \mathbb{G} \\ \vdots \\ A' = \langle \langle \alpha_1, P_1 \rangle, \langle \alpha_2, P_2 \rangle, ...; label(def_1) : \langle \pi_1 : X_1 \rangle, label(def_2) : \langle \pi_2 : X_2 \rangle, ... \rangle \\ \mathbb{L} \vdash \mathbb{T}^* = \mathbb{T}[a/A'] \end{array}}{\mathbb{L},\mathbb{G} \vdash \texttt{type}\ \ a\ \texttt{(}p_1, p_2, ...\texttt{)}\texttt{:}def_1\ def_2\ def_3 \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G}}$$

Rule 30: Semantic rules for type definitions.

Variable Definition

```
definition_variable
    : SIMPLE_ID ':' type EOL
    | SIMPLE_ID ':' type '(' expr_inline_list? ')' EOL
    | SIMPLE_ID ':' type '(' expr_inline_list? ')' ':' EOL INDENT argument_named+
    | SIMPLE_ID ':' type '=' expr_inline EOL
    | SIMPLE_ID ':' type '=' expr_block
    ;
```

Grammar 19: Grammar rules for variable definitions.

Finally, The variable definition statement a new object of the specified type, and adds it to the local environment. The object initialization method is the default constructor if no arguments are supplied, otherwise the appropriate constructed is invoked. When using the assignment variant, the object is first constructed using the default operator, then the assign operator is invoked. Additionally, variable definitions can be used to define variable fields inside type definitions. In this case, the initialization of the variables is conducted by *call*() when executing the appropriate constructor. The syntax rules for variable definitions are available in grammar 19 and their semantics in rules 31.

$$\textit{field}\ \frac{\begin{array}{c}\mathbb{T} \vdash \mathbb{T}(\texttt{self}) = X \\ \mathbb{T} \vdash \mathbb{T}(\textit{type}) = A \\ \mathbb{T} \vdash A := \langle \ldots ; \ldots, \textit{constructor} : \langle \lambda : \texttt{<()}\ \ A\texttt{\&>} \rangle, \ldots \rangle \\ \mathbb{L}, \mathbb{G} \vdash \alpha = offset(X, a) \\ \mathbb{L} \vdash \mathbb{V}^* \leftarrow \mathbb{V}[a/\alpha] \\ \mathbb{L} \vdash \mathbb{T}^* \leftarrow \mathbb{T}[a/A\texttt{\&}]\end{array}}{\mathbb{L}, \mathbb{G} \vdash a\texttt{:}\textit{type} \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G}}$$

$$\textit{default}\ \frac{\begin{array}{c}\mathbb{T} \vdash \mathbb{T}(\textit{type}) = A \\ \mathbb{T} \vdash A := \langle \ldots ; \ldots, \textit{constructor} : \langle \lambda : \texttt{<()}\ \ A\texttt{\&>} \rangle, \ldots \rangle \\ \mathbb{L}, \mathbb{G} \vdash \textit{new}(\mathbb{V}, A) \Rightarrow \langle \alpha : A\texttt{\&}, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash \textit{call}(\langle \alpha : A\texttt{\&} \rangle, \textit{constructor}) \Rightarrow \langle \alpha : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L} \vdash \mathbb{V}^* \leftarrow \mathbb{V}[a/\alpha] \\ \mathbb{L} \vdash \mathbb{T}^* \leftarrow \mathbb{T}[a/A\texttt{\&}]\end{array}}{\mathbb{L}, \mathbb{G} \vdash a\texttt{:}\textit{type} \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G}}$$

$$\textit{arguments}\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash \textit{new}(\mathbb{V}, A) \Rightarrow \langle \alpha : A\texttt{\&}, \mathbb{L}, \mathbb{G} \\ \mathbb{T} \vdash A := \langle \ldots ; \ldots, \textit{constructor} : \langle \lambda : \texttt{<(}X_1, X_2, \ldots\texttt{)}\ \ A\texttt{\&>} \rangle, \ldots \rangle \\ \mathbb{L}, \mathbb{G} \vdash \textit{expr}_1 \Rightarrow \langle \pi_1, X_1' \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash \textit{expr}_1 \Rightarrow \langle \pi_1, X_1' \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{L}, \mathbb{G} \vdash \textit{convert}(\pi_1, X_1', X_1) \rightarrow \langle \psi_1, : X_1 \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash \textit{convert}(\pi_2, X_2', X_2) \rightarrow \langle \psi_2, : X_2 \rangle, \mathbb{L}, \mathbb{G} \\ \vdots \\ \mathbb{L}, \mathbb{G} \vdash \textit{call}(\langle \alpha : A\texttt{\&} \rangle, \textit{constructor}, [\langle \psi_1, : X_1 \rangle, \langle \psi_2, : X_2 \rangle, \ldots]) \Rightarrow \langle \alpha : A\texttt{\&} \rangle, \mathbb{L}, \mathbb{G} \\ \mathbb{L} \vdash \mathbb{V}^* \leftarrow \mathbb{V}[a/\alpha] \\ \mathbb{L} \vdash \mathbb{T}^* \leftarrow \mathbb{T}[a/A\texttt{\&}]\end{array}}{\mathbb{L}, \mathbb{G} \vdash a\texttt{:}A(\textit{expr}_1, \textit{expr}_2, \ldots) \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G}}$$

$$\textit{assign}\ \frac{\begin{array}{c}\mathbb{L}, \mathbb{G} \vdash a\texttt{:}A \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G} \\ \mathbb{L}, \mathbb{G} \vdash a\texttt{=}\textit{expr} \Rightarrow \langle \alpha : A\texttt{\&} \rangle, \mathbb{L}^*, \mathbb{G}\end{array}}{\mathbb{L}, \mathbb{G} \vdash a\texttt{:}A\texttt{=}\textit{expr} \Rightarrow \langle \phi : \Phi \rangle, \mathbb{L}^*, \mathbb{G}}$$

Rule 31: Semantic rules for variable definitions.

Chapter 4

Implementation

This chapter contains information concerning the design and implementation of the EVE environment, including choices, trade-offs and difficulties that came to be during both semesters. Section 4.1 describes the details related to the implementation of the EVE runtime, including a global description of the architecture, and details of several components such as workers, spin-locks, monitors and event handlers. Section 4.2, embodies information related to the EVE compiler, such as the choice of lexer and parser tools, type-checking and code-generation.

4.1 Runtime

The layout of the EVE runtime is depicted in figure 4.1. It composed by two major packages. The *core* package is responsible for the creation, manipulation and execution of tasks, by worker instances. It is also responsible for the synchronization of workers, ensuring proper load balancing and preventing conflicting access to shared variables. Additionally, it also provides a simple wrapper library for the `epoll()` system calls, enabling the transition between kernel triggered events and the EVE event API. The *libraries* package makes use of this interface to provide an API for common system tasks such as socket operations and execution termination. It also contains the event handling

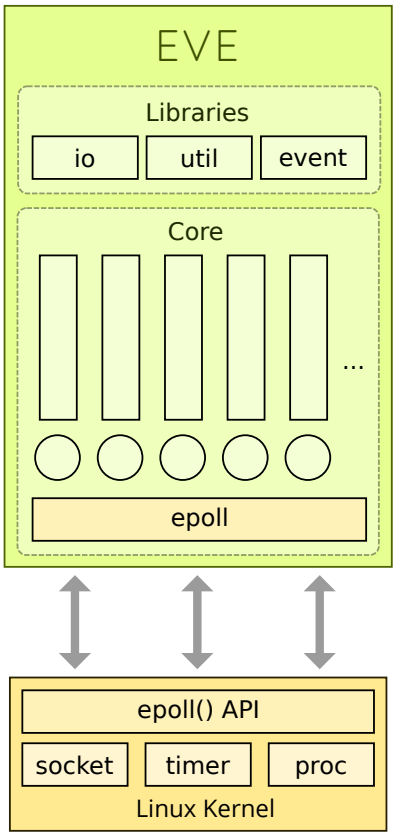


Figure 4.1: Architecture of the EVE runtime, and its connection with the Linux kernel using the `epoll` interface.

routines for callback registration and event triggers. For
the implementation of this system, the C++11 language was selected. This choice was primarily due to the control of low-level details such as atomic operations and memory layouts. Additionally, the author was proficient with this language, having already developed similar concurrent software.

4.1.1 Workers

A worker is an entity capable of executing tasks in its own thread. By default a worker is created for each processor thread available. Each worker contains a local double-ended queue of tasks. The initial implementation was heavily based on the THE work-stealing algorithm presented in [10]. In this algorithm, each worker can insert and remove tasks from the deque from one of the sides. When a worker completes all local tasks, it can steal tasks from the queue of other workers, using the other end. This solution minimizes contention, under the assumption that the steal operation is uncommon. However, when the queue only contains one element, the victim worker must be careful not to attempt to remove the same task that was already stolen by another worker. This verification adds a large overhead to the remove operation, even in the uncontended case. To overcome this bottleneck, the suspend-steal method proposed in [16] was implemented. With this method, each worker assumes it has complete access to its own queue, thus not requiring the verification step. The thief thread must ensure that it causes no conflict by performing the steal operation. To achieve this, the victim thread is first pooled to check if it has tasks to be stolen. This is a read-only operation and requires no synchronization. If this test is positive, the victim thread is briefly suspended, allowing the stealer to observe the state of the queue. This observation is only partially-consistent because the victim could be in the middle of performing remove operations. However, because the these operations only remove a single task from one end, if the queue has at least two tasks, it is safe to assume that the task at the other end can be stolen. After this process the victim thread can be resumed. Unfortunately, the current Linux implementation of POSIX threads does not implement the `pthread_suspend` function. To emulate this behavior we made use of signals and semaphores. The thief thread first sends a signal to the victim and waits on a semaphore A. Upon receiving the signal, the victim starts executing signal handler, which immediately posts on lock A, resuming the thief thread, and waits on lock B. In this state the victim thread is blocked and the stealer thread can continue the steal operation. After finishing this procedure the thief thread posts on lock B, resuming the victim thread.

During the execution of an EVE program, the number of tasks inside worker queues can vary greatly. A conservative approach can reserve a large enough queue so that it never reaches its size limit. A dynamic solution, that grows and shrinks the queues is preferable because it avoids having a large memory footprint unless necessary, while still being capable of handling an

arbitrary number of tasks. The implementation of these queues was based on [4].

Robust implementations of work-stealing schedulers must be able to handle a large number of tasks. However, it is also necessary to set processors to idle, when no work is available. To achieve this, an sleep function is used to deschedule the worker thread from a processor. The thread is later rescheduled after a timeout occurs. An alternate, and more efficient solution, uses an active approach, where workers with surplus tasks can wake-up sleeping threads [26]. In EVE however, new tasks can originate from outside the work-stealing environment with kernel triggered events. For this reason, instead of using the sleep function, the blocking `epoll_wait()` function is employed. The function is only called when a thread has no available tasks. This conceives the runtime a desirable property: tasks already scheduled for execution have higher priority than incoming tasks, preventing the system to overwhelm itself and become increasingly slower due to cache misses and page faults. This operation is also invoked once before attempting to steal tasks, minimizing conflicts between workers. The algorithm for the worker loop, implementing the already mentioned modifications, is represented in program 20.

```
procedure loop
begin
    repeat
        while !queue.empty()
            t = queue.pop()
            execute t
        end while
        if epoll.process()
            failed = 0
            continue
        end if
        steal()
    until shutting down
end procedure
```

```
procedure steal
    blocking = true
begin
    victim = next_victim
    next_victim = (victim + 1) % N

    lock thief.l

    if blocking && failed > N
        unlock lock

        if shutting down
            return
        end if

        if (epoll.process(timeout) > 0)
            timeout = MIN_T
        else
            timeout = min(timeout * 2, MAX_T)
        end if

        failed = 0
        return
    end if

    if victim.queue.size() < 2 or
      !trylock victim.l
        failed = failed + 1
        unlock thief.l
    end if

    if shutting down
        failed = failed +1
        unlock victim.l
        unlock thief.l
        return
    end if

    suspend victim

    if victim.queue.size() < 2
        resume victim
        unlock victim.local
        unlock thief.local
        return
    end if

    t = victim.tasks.pop_bottom()
    failed = 0

    resume victim
    unlock victim.l
    unlock thief.l

    execute t
end procedure
```

Program 20: Pseudo-code for the worker loop and steal operations in the EVE runtime.

4.1.2 Spin-locks

Spin-locks are used in performance critical sections where kernel (blocking) locks provide an unnecessary overhead. One downside of this usage, is that in the contended case the blocked processor keeps consuming CPU cycles. The second downside, is the lack of order guarantee between two or more contending processors. Our implementation implements the spin-on-read lock [1], and makes use of the available GCC lock built-in operations. Pseudo-code for the `lock`, `trylock` and `unlock` operations is available in program 21.

```
procedure lock
begin
    while lock_test_and_set(l, true)
        while l
            yield
        end while
    end while
end procedure

procedure trylock
begin
    return !lock_test_and_set(l, true)
end procedure
```

```
procedure unlock
begin
    lock_release(l)
end procedure
```

Program 21: Pseudo-code for the three spin-lock operations.

4.1.3 Monitors

Monitors are synchronization primitives that enable safe concurrent access to one resource. On top of allowing only one processor to execute with *unique* access to the underlying resource (equivalent to locks), it can also provide *shared* access to multiple processors if they only execute read operations. Our implementation makes use of atomic instructions to avoid expensive system calls. For this implementation a single variable `bitmap` is used with the size of the processor word (usually 32 or 64 bits). The left significant bit indicates that a processor has locked, or is attempting to lock the resource with unique permissions. The remaining bits form an unsigned integer representing the number of processors currently owning shared access. Pseudo-code for monitor operations is available in program 22. The `lock_unique` operation attempts to set the unique bit to true. If it was already set, then another processor has previously requested unique permissions. In this case, the processor is yielded until this bit is unset, and the operation is repeated. If the bit was previously false, then unique permissions were correctly locked to this process. All that remains, is to wait for all shared accesses currently executing to terminate. The `try_lock_unique` operation implements the same logic, but does not repeat the operation in case of failure. The `lock_shared` is constructed using the `try_lock_shared` operation. This

operation first begins with an heuristic check to avoid atomic operations if possible. If this heuristic is accepted, the number of shared processors is increased by one, keeping in mind that incrementing this number is the same as adding two to the entire bitmap variable. If the atomic operation returns an odd number, indicating that the unique bit was set, the operation fails and immediately decreases the number of shared processors. Using this approach, write operations (requiring unique access) have higher priority: once a unique permission is required, all incoming shared operations are suspended. Nevertheless, it is not assured that two scheduled unique accesses are carried out consecutively (timing circumstances may allow shared accesses in-between). Additionally, there is no execution order guarantee among two or more unique tasks once the unique bit has been cleared.

```
procedure lock_unique
begin
    loop
        if fetch_and_or(bitmap, 1) & 1
            while bitmap & 1
                yield
            end while

            continue
        end if

        while bitmap != 1
            yield
        end while

        return
    end loop
end procedure

procedure unlock_unique
begin
    fetch_and_and(bitmap, ~1)
end procedure

procedure try_lock_unique
begin
    if fetch_and_or(bitmap, 1) & 1
        return false
    end if

    while bitmap != 1
        yield
    end while

    return true
end procedure
```

```
procedure lock_shared
begin
    while !try_lock_shared()
        yield
    end while
end procedure

procedure try_lock_shared
begin
    if bitmap & 1
        return false
    end if

    if fetch_and_add(bitmap, 2) & 1
        fetch_and_sub(bitmap, 2)
        return false
    end if

    return true
end procedure

procedure unlock_shared
begin
    fetch_and_sub(bitmap, 2)
end procedure
```

Program 22: Pseudo-code for the six monitor operations.

4.1.4 Events

Events in EVE are instances of any class. Event emitters on the other hand, extend the `emitter<T>` class, indicating that it is an origin of events of type `T`. Each instance of the class stores callbacks for this event on this object. This allows for a distributed callback table, effectively avoiding unnecessary contention with global table locks.

For kernel originated events, the `epoll()` interface was selected among the alternatives. The `select()` and `poll()` interfaces have poor scalability [12]. Kqueue is not currently implemented in Linux. Additionally, frameworks such as libev, libevent and Boost.ASIO can be used in multi-threaded environments, but the underlying event loop is executed in only one thread. This is a limitation of the platforms, `epoll()` system call is, by itself, thread safe. However, using `epoll()` to develop a race-condition-free runtime is not trivial. Special care must be taken into account to ensure that 1) each event is processed by only one thread, 2) that they are processed in the correct order and 3) that there is no conflicts between different types of operations.

For problem 1, consider the situation where a read operation on a socket is requested. Internally, a read callback is created and registered with `epoll_ctl()`. Later, when such operation is feasible, incoming `epoll_wait()` calls return this event. With a naïve solution, multiple threads would receive this event and wrongfully process it. The `EPOLLONESHOT` mode is used to prevent this error. In this mode, when one event is triggered, the associated file descriptor is disabled from the epoll set. Until it has been reactivated, other invocations to `epoll_wait()` will not return events referring to this file descriptor.

Problem 2 emerges when multiple operations of the same type are scheduled (i.e.: sending two objects over a socket). Order between the operations is important, and it is necessary to avoid interleaving operations where a partial write is interrupted by the other. Our implementation solves this problem by using an ordered list of callbacks. When an event is triggered, the first callback is executed, writing all possible data to the socket. If only a partial write is possible at the time, the event is not propagated to other callbacks. Later, when the event is re-triggered, the operation resumes from the new position. If the operation completes, the callback is removed from the list, and the event is propagated to the next callback in the list.

Problems 3 surfaces when operations of multiple types are scheduled. Consider a read and two write operations. After each event, the file descriptor must be reactivated by indicating the new set of events to be monitored. After a partial read on the first instruction, the monitored set must include both the read and write events. Likewise, after completing the first write operation, the write event must still be present. For these reasons, the `rearm()` function, computes the monitored set, taking into account the presence of callbacks for each event. This verification is thread-safe since it is always invoked when unique access permissions to the emitter object are given.

4.1.5 Shared Objects

Shared objects are heap allocated, automatically managed objects whose access is controlled by a monitor. The first attempt at implementing these object used the available implementation of the Boehm-Demers-Weiser Garbage Collector [8]. This GC implements the mark-and-sweep algorithm which transverses memory from root nodes (i.e.: thread stacks and registers) to find pointers to all currently used memory. Unreachable nodes are identified and freed automatically. This solution makes the assumption that at any time, a pointer to the memory page must exist for the object to remain active. This is not true in EVE. By registering callback functions using `epoll()` the pointer is stored inside kernel memory (not reachable by the GC). When garbage collection is triggered, the object is marked as inactive and destroyed prematurely. Although it could be possible to mark these objects as root nodes, the Boehm-GC API does not allow this operation to be performed on already allocated objects. Additionally, the C++ layer is reportedly unstable and very platform dependent. Furthermore, the mark-and-sweep step requires all threads to be paused which creates undesirable for jitter in response times.

The second attempt made use of the STL smart pointers. In particular, `std::shared_ptr<T>` which is a pointer to an object of type `T`. This structure contains not only a pointer to the memory location where the object is stored but also to a control block which is used for reference counting: each time a pointer is created the reference is increased and when pointers are destroyed, the reference is decreased. When it reaches zero, the object is destructed. Because the emitter objects store the pointers to callbacks and vice-versa, they would not be destructed (notice that this requirement is easier than a pointer being reachable from a root note). However, the implementation of `std::shared_ptr` added other unpredicted restrictions. Our shared object implementation is `shared<T>`, containing the underlying resource `T` and a `monitor` object. Assume two classes A and B, where B is a subclass of A (e.g.: A is a file descriptor, and B is a socket). Even though A and B are related, and a pointer of B can be converted to a pointer of A, `shared<A>` and `shared<B>` are not related. This makes the use of `std::shared_ptr<shared<T>>` impossible, since type conversion would lead to compile type errors. Making `shared<A>` and `shared<B>` related is also impossible due to the nature of template instantiation[1]. The use of two pointers, one for the object, and one for the monitor, was an option. However, this would lead to double the overhead of construction, copying and moving of pointers (which is already significant).

Our solution was to develop a replacement for `std::shared_ptr<T>` available inside the package `eve::rt`. This enabled us to move the `monitor` object to the control block. Therefore, `eve::rt::shared_control` contains a monitor object, an integer containing the number of active references and a pointer to the heap allocated object. Initially this pointer, was of `void*` type.

[1] It would require a function that expands to a list of subclasses of a type, which is inexistent in the C++11 specification,

Accessing it through a `std::shared_ptr<A>` pointer would make use of a `std::static_cast` to `A*`. However, this solution is also not correct: when used with a class hierarchy with multiple inheritance, the `std::static_cast` fails to recognize the necessary memory offsets and creates a corrupted pointer. In this case a `std::dynamic_cast` is required. However, the C++11 specification forbids the use of `std::dynamic_cast` with `void*` simply because it does not store runtime type information. To overcome this inconvenience, an virtual class `shareable` was created, and all objects that were used inside `eve::rt::shared_ptr` would have it as a parent class. This in turn, allows the pointer to be of type `shareable*` and `std::dynamic_cast` to work correctly. However, native objects (e.g.: `int`, `string`, ...) can not be modified to extend this class. For this reason, they are automatically wrapped inside a `shared_wrapper` which contains the native object and extends `shareable`.

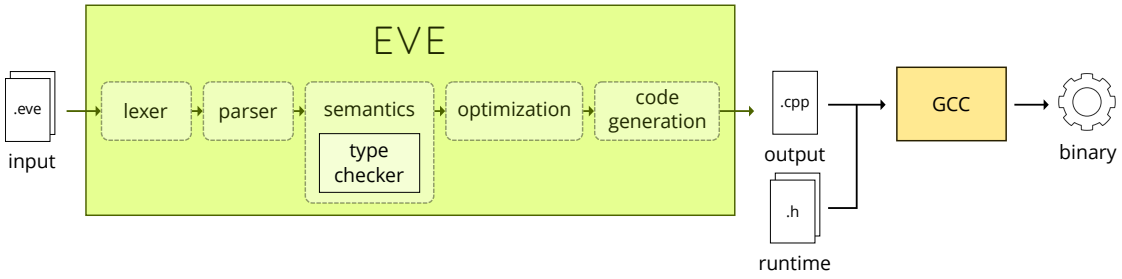


Figure 4.2: Architecture of the EVE compiler, and the compilation proccess of EVE programs.

4.2 Compiler

The EVE compiler, is a compiler implementation for the EVE language. It translate `.eve` source files into a single output file containing a `c++` equivalent of the application. This file, in conjunction with the EVE runtime, can be parsed using a standard C++11 compliant compiler such as GCC or Clang to deliver a executable binary. The architecture of the EVE compiler forms a pipeline of five stages, as can be observed in figure 4.2. The initial step is the lexical analysis, where a stream of characters (the input file) is transformed into a sequence of tokens (e.g. a reserved keyword, an operator, an id, etc.). A parser then analyses the token sequence using the rules defined by a deterministic context-free grammar and creates an Abstract Syntax Tree (AST) representation of the program. This tree is processed by a semantic pass, where additional information is deduced and stored. The optimization step modifies the AST without changing its semantics in a way that allows for the resulting application to perform better. Finally, the code generation step uses the AST and all associated information to create a representation of the program in the target format.

4.2.1 Lexer and Parser

For the implementation of the parser phase, several alternative were studied. The first being GNU Bison. This software analyses a LR(1) grammar using a LALR(1) parser and produces a fast, recursive implementation written in C++. This solution has the downside of encountering reduce/reduce conflicts which require the grammar to be modified[2]. It can also be instructed to produce a slower GLR parser that does not have this limitation. The second alternative was ANTLR (ANother Tool for Language Recognition). This solution includes a GUI development environment called ANTLRWorks, which enables easy development and analysis of grammars. ANTLR grammars are LL(*), which unlike LR(1) do not include all deterministic context-free languages [25]. By default, ANTLRv3 creates a target implementation in Java. To generate a C++ parser, the older ANTLRv2 version needed to be used, which is not compatible with

[2]LALR(1) grammars have additional restrictions that are not present in LR(1) grammars.

ANTLRWorks[3]. The third alternative was to use the Elkhound GLR parser generator [21]. This parser analyses context-free grammars even if they are non-deterministic. However, this project looks neglected since 2005-08-22 (release date of the last version). Ultimately, the Bison parser was selected due to previous experience with the software and the target C++ language. The Flex++ software was used to create the Lexer to be used with Bison.

4.2.2 Type checking

The implementation of a type checker is arguably, the hardest part of developing a compiler. In particular, languages with multiple-inheritance, template types, variadic functions (and templates) and overloaded operators offer significant complications. Although the EVE compiler supports all of these, the implementation is still in alpha stage and lacks in robustness. We found that our algorithm for evaluating templated code fails under some conditions. In particular, functions which receive arguments with arbitrary types fail to validate, simply because the operations on those variables are unknown until the actual types are fixed. This is a problem inherent to the implemented bottom-up validation. Section 6.3 includes a brief description of what must be rectified do correcly execute this validation.

4.2.3 Code generation

The primary goal of a compiler is the generation of a binary executable. Ideally, this is carried out by a compiler back-end which processes an intermediate representation and creates the binary. This architecture enables the front-end compiler to reuse existing back-ends which are thoroughly tested and already implement many optimizations. The first attempted solution was the GCC back-end. This choice was selected with the intention of creating a monolithic compiler: the GCC front-end would parse the runtime code, the EVE front-end would parse the program code and the GCC back-end would merge the two and create the binary in a single pass. However, GCC has a very complex architecture (the result of its long development time and existing legacy code). For this reason, the option of using the LLVM back-end was explored. LLVM is a modern compiler infrastructure written in C++. The major selling point of LLVM is its simple, human readable intermediate representation which promotes its use as a back-end compiler. Using the LLVM system, the compiler would make use of the Clang front-end to parse the runtime code and create its LLVM IR. This IR would be merged with the IR generated by the EVE compiler and used to create the final binary. However, the runtime implementation makes extensive use of C++ templates which are only translated by Clang to IR if an instantiation is found. Because this instantiations are only expected in the EVE program, the IR for the runtime would contain missing parts. Attempts were made to manually create

[3]The C++ target was latter ported to ANTLRv3.4 which was not available during the development of the compiler.

the template instantiations using the Clang API. Nevertheless, this API is poorly documented (unlike the remaining LLVM infrastructure) and the desired result was not achieved. To overcome this problems, the monolithic solution was abandoned and a two step compilation process was selected. With this solution the EVE compiler translates EVE programs into a single C++ source file. This file makes use of preprocessor directives to include the runtime code when it is parsed by a C++11 compatible compiler (e.g.: GCC or Clang). This approach enables the implementation of the EVE compiler to be easier while still making use of the optimizations available in production ready back-ends. To avoid compile errors originated by redefinitions of existing C++ symbols (e.g. the program defines a function that part of the C++ language) all generated symbols are prefixed with `eve_`.

4.2.4 Optimizations

Two optimizations related to the `finish` statement where developed. The first optimization attempts to minimize parallel slowdown, by creating fewer tasks depending if there is enough parallelism available. If this option is enabled, the generated code first invokes an heuristic operation `parallelize()`. When the returned value is `true`, indicating more tasks are necessary, the proceeding `par` statements are handled as usual. However, if the return is `false`, the `par` statements are executed immediately avoiding the creation of new tasks. This optimization is only possible in the presence of the `finish` statement where it is guaranteed that `par` statements are readily executable. The second optimization was made to the `finish` synchronization routines. Under default conditions, the `finish` block contains an atomic integer counter which is incremented each time a task is created, and decremented each time its execution is completed. However, there are instances where the number of created tasks can be determined during static analysis. If this is the case, the increment operations can be removed in by initializing the counter to the correct value. Additionally, if the finish block contains a single parallel task, the integer counter is replaced by a faster boolean counter.

Chapter 5

Evaluation

In this section we compare the performance of the EVE platform with other off-the-shelf solutions. Each experiment was repeated 30 times and we report the average values and associated standard deviations. Additionally, a each experiment was executed once before the mesurements to avoid interference of JIT compilation and caching mechanisms. Single host benchmarks where executed in the *Astrid* machine. For communication tests, the same machine was used for the server implementation and *Ingrid* was used for the client. The two hosts where connected directly with a crossover to prevent external interference. The specifications for both machines are described in table 5.1. Additionally, information regarding external software is present in table 5.2.

	Ingrid	Astrid
Motherboard	Dell Inc. 0CRH6C	SuperMicro X9DAi
Processor	2x Intel(R) Xeon(R) X5660 2.80GHz, 24 hardware threads	2x Intel(R) Xeon(R) E5-2650 2.00GHz, 32 hardware threads
Memory	24 GB DDR3 1333 MHz	32 GB DDR3 1600 MHz
Connectivity	Broadcom Corporation NetXtreme BCM5761 Gigabit Ethernet PCIe	Intel Corporation I350 Gigabit Network Connection

Table 5.1: Hardware specification of benchmark hosts.

	Version	Flags
GCC	4.7.2	`-std=c++11 -pthread -ffast-math -march=native -O4 -DNDEBUG -lrt`
Erlang	R15B01	`erlc -smp-enabled -S16` `erl -noshell -s -smp-enabled -S16`
Go	1.0.2	`export GOMAXPROCS=16`
GHC	7.4.2	`ghc -O3 -threaded -make`
node.js	0.6.19	
Ruby	1.9.3p194	
Python	2.7.3	
Java	1.7.0-25 OpenJDK 23.7-b01	
libev	1.4-2	`-lev`
Intel TBB	4.0+r233-1	`-ltbb`
gevent	0.13.7	
REV	0.3.2	

Table 5.2: Software versions and usage flags used for the benchmarks.

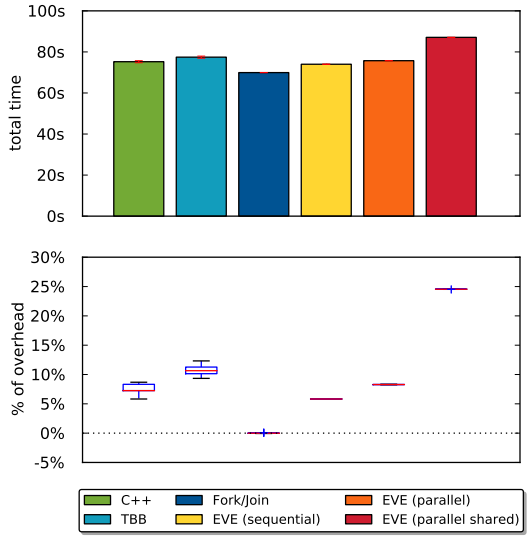


Figure 5.1: Overhead of each implementation of the Fibonacci program relative to the fastest single-core implementation.

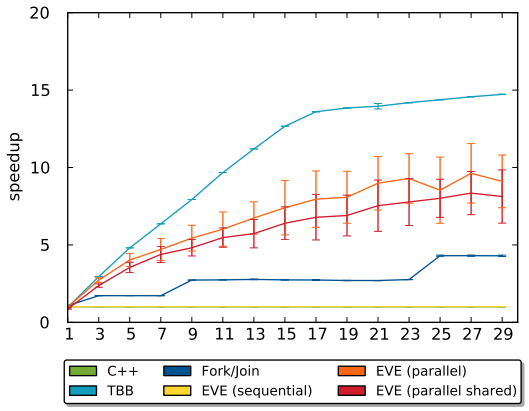


Figure 5.2: Scalability test of the Fibonacci implementations using heuristic parallelization.

5.1 Fibonacci

The Fibonacci test is benchmark on the capabilities of the EVE as a platform for parallel programming. The test consists of computing the 50^{th} Fibonacci number using the recursive definition:

$$F_n = \begin{cases} n & n < 2 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

For this benchmark EVE was compared against JAVA Fork/Join framework and Intel's TBB implementation of the work-stealing model. Additionally, a sequential version using C++ was used since it serves as a base line for all C++ supported implementations. Figure 5.1 shows the measured times of each implementation, and its relative overhead when compared to the fastest single-core solution (Fork/Join). In this graph, three implementations with EVE are compared. *EVE (sequential)* is a direct translation of the recursive formula with no `par` statements. *EVE (parallel)* makes use of one `par` statement and one `finish` statement. In this solution local variables are passed by reference. In *EVE (parallel shared)* the implementation of shared objects and access permissions is used.

The Fork/Join solution is the fastest implementation using a single-core. This is a result of JAVA's Just In Time Compilation technique that enable runtime information such as code branching to be analyzed and used for optimizing the execution. Achieving similar performance with the other statically compiled solutions would require profile-guided compilation, which is outside the scope of this work.

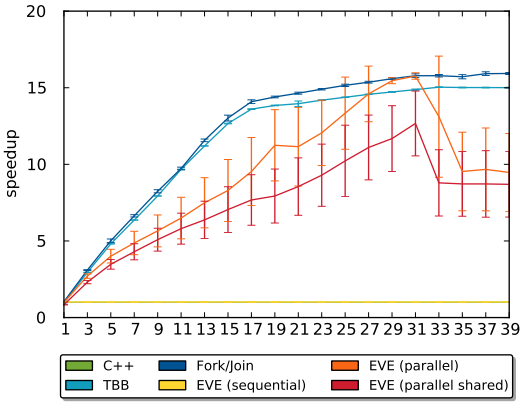


Figure 5.3: Scalability test of the Fibonacci implementations without heuristic parallelization.

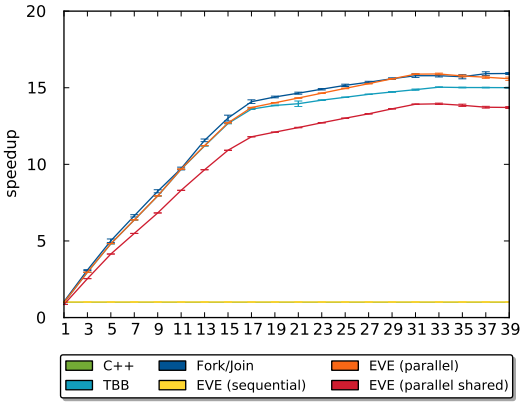


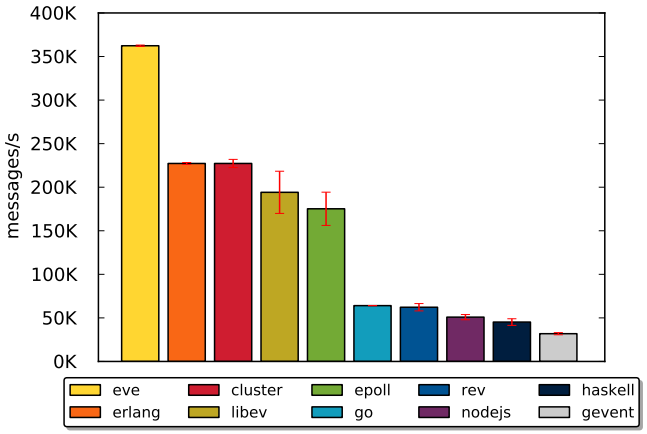
Figure 5.4: Scalability test of the Fibonacci implementations without waiting operations.

It can also be observed that two EVE implementations are also faster than simple C++. Although this is unexpected, our hypothesis is that the additional complexity present in the EVE runtime forces the GCC optimizer to be executed more often, resulting in better performing code for the Fibonacci function. The performance difference of the sequential version of EVE and its parallel implementation is only 2.33%, which indicates a very low overhead in execution time. However, difference between the use of shared and native objects is rather noticeable (15.01%), even in the presence of no parallelism. In this test, the EVE runtime also outperforms the Intel TBB implementation by a very small amount (2.28%).

Figure 5.2 indicates how each implementation performed when additional parallelism is available. Not surprisingly, the performance of both sequential solutions was constant. In this plot, the EVE implementation shows sever limitations, only achieving half the performance of that obtained with TBB. However, the performance of the Fork/Join implementation was even more unexpected. In particular, the staircase-like curve, with minimal standard deviation, lead us to believe there was a problem with our implementation. After analysis, we discovered that the culprit was the use of Fork/Join's `getSurplusQueuedTaskCount` function as an heuristic to reduce the number of parallel tasks. To analyze the influence of the heuristics, the benchmark was re-executed without this functions (both in Fork/Join and in EVE). Figure 5.3 shows the obtained result.

Without the problem of under parallelization created by bad heuristics, the Fork/Join solution now outperforms TBB by a small amount. Nevertheless, the performance of EVE with this implementation is still lacking. As it can be observed, the speedup obtained with EVE grows continuously with the number of threads until it reaches a breaking point at 32, where it falls significantly. The growth, until this point, does not show the expected fall-off around 16 threads (the number of processor cores). Additionally, the observed standard deviation is one to three

orders of magnitude higher than the remaining solutions. These observations lead us to believe that worker threads could be inadvertently sleeping, even in the presence of tasks. Timing circumstances would then explain the large standard deviation observed. In fact, the runtime uses a blocking call to `epoll_wait`() to avoid busy waiting for new tasks. This is only used when each worker has repeatedly failed to steal tasks from the others (because their deque also contains no tasks). This occurs frequently, between the successive invocations to `fibonacci()` (one for each of the thirty runs). Even though the timeout used is dynamically updated it creates a large decrease in performance. For this reason we decided to test the application with a maximum timeout of 0, effectively creating a non blocking call. The results obtained are depicted in figure 5.4. Under this conditions the EVE runtime shows very good performance, reaching peak speedups of 14.99 with reference passing and 13.11 with shared objects. This indicates that the current solution for power saving is suboptimal and should be improved. Details on this task are described in section 6.3. Nevertheless, the EVE runtime shows very good scalability and performance, comparable to the selected solutions.



	$\overline{x}$ (req/s)	σ (req/s)
eve	362.368K	775
erlang	227.249K	1099
cluster	227.264K	4670
libev	194.133K	24208
epoll	175.217K	19081
go	64.075K	143
rev	62.225K	4210
node	50.866K	2972
haskell	45.210K	3799
gevent	31.864K	1243

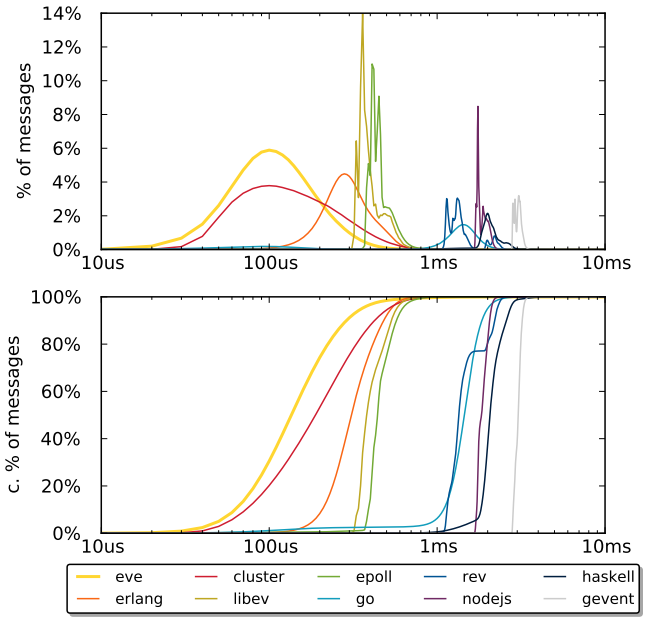
Figure 5.5: Request throughput of the echo servers (localhost).

5.2 Echo Server

Facilitating the developing high-performance web applications is one of the goals of EVE. This benchmark compares EVE to other languages and frameworks used for this purpose. The test consists of creating a server that accepts TCP connections and re-emits the received data until the socket is closed. Although very simple, this test enables the comparison of key features of web servers. The first measured attribute is the request throughput. This indicates the number of requests per second the server can handle. The second measured attribute is latency. Low latency times are critical for soft real-time applications. Additionally, even for other applications, latency higher than 100ms is noticeable and has been linked to lower user dissatisfaction, higher bounce rates and overall lower revenue [14].

For this benchmark, the following solutions were tested: `eve`, `erlang`, `haskell`, `go` and `nodejs` are implementations of an echo server using the respective languages, `rev` is an implementation using the Ruby Event Machine platform, `gevent` and `libev` make use of the homonymous libraries (for python and C++ respectively), and finally `cluster` is a nodejs application that uses the cluster library for parallelism. The source code for each application was selected from an existing benchmark, publicly available at `https://github.com/methane/echoserver`. However, this benchmark suite does contain the cluster implementation. Additionally, the client software used the thread-per-connection model which delivered low performance. A new implementation based on this code was created using the EVE runtime. For each test, 150 concurrent connections were created, each sending 10000 sixteen byte messages.

Figures 5.5 and 5.6 indicate the performance obtained using these solutions when the server and client applications are executed in the same host. The `gevent` implementation is the slowest of all alternatives. This is most likely because the entire framework is executed by the python



	Q_1 (µs)	Q_2 (µs)	Q_3 (µs)
eve	100	140	210
erlang	260	310	390
cluster	120	190	300
libev	360	390	460
epoll	410	450	510
go	1280	1460	1650
rev	1250	1350	1560
node	1750	1840	1970
haskell	1940	2070	2270
gevent	2910	3050	3130

Figure 5.6: Reply latency of the echo servers (localhost).

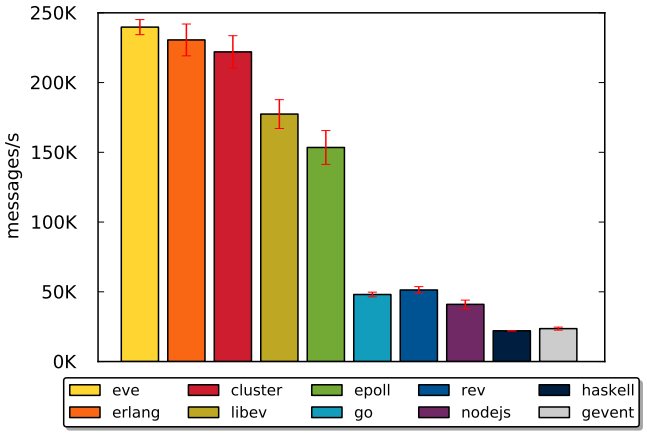
interpreter which is inherently slower than a native implementation. This is also true for `nodejs` and `rev`, although not to the same extent (the event loop is implemented in native code). Additionally, `gevent` uses the *libevent* library while `nodejs` uses *libuv*[1] and `rev` uses a custom implementation. According to [17], *libev* outperforms *libevent*/*libevent2* which also reinforces the poor performance of `gevent`. The `epoll` implementation makes direct access to the `epoll()` system call using C++ while `libev` uses the wrapper library which uses `epoll()` on UNIX systems. As expected, their performance is much better than the already mentioned solutions. In fact, `libev` alone is 3.88 times faster than `nodejs`. However, in our case, the `epoll` implementation is slightly worse than `libev`. This is because the benchmarked `epoll` code is poorly optimized, making use of unnecessary memory allocations that are not present in `libev`.

All the remaining solutions make use of multi-threaded runtime environments and were expected to outperform the single-threaded implementations. This is not true for `haskell` and `go`. Regarding the first case, the `haskell` runtime has known IO scalability issues. According to [29] this has been fixed in GHC version 7.8.1, which has not yet been released. The reason behind `go`'s poor performance is more obscure since documentation of its runtime architecture is not available. Both the `erlang` runtime and `cluster` implementation show good performance. Nonetheless, the `eve` framework surpasses both with a 35.5% increase in throughput. One interpretation of this value is that the EVE runtime is more optimized and/or requires less operations. In fact, the `erlang` language was designed for real-time systems and each actor is scheduled using

[1]The *libuv* implementation was initially a wrapper arround *libev*.

a preemptive fair algorithm. Even if no preemption occurs, this algorithm is more expensive than the execution of eve tasks. Regardless, even though the EVE runtime provides less guarantees on the response time, the obtained latency and jitter are comparable if not better than `erlang`'s. The second interpretation is that the Erlang runtime architecture does not scale well with IO. In fact, the current version of this runtime only allows one scheduler to check for IO at a time [6]. The EVE runtime does not have this limitation. Regarding the remaining systems a pattern can be observed: the latency curve for parallel architectures resemble Gaussian distributions while with single-threaded architectures the curve is noisy. This seems to be an implication of the Central Limit Theorem[2] considering that the response times of one thread can be modeled as an independent random variable such that the global performance is constructed from the set of all variables.

[2]Given certain conditions, the arithmetic mean of a large number of iterations of independent random variables, each with a well-defined expected value and well-defined variance, will be approximately normally distributed [24].



	$\overline{x}$ (req/s)	σ (req/s)
eve	239.753K	5393
erlang	230.569K	11407
cluster	222.000K	11604
libev	177.409K	10332
epoll	153.470K	12119
go	48.063K	1680
rev	51.304K	2406
node	40.996K	3043
haskell	22.029K	139
gevent	23.603K	1076

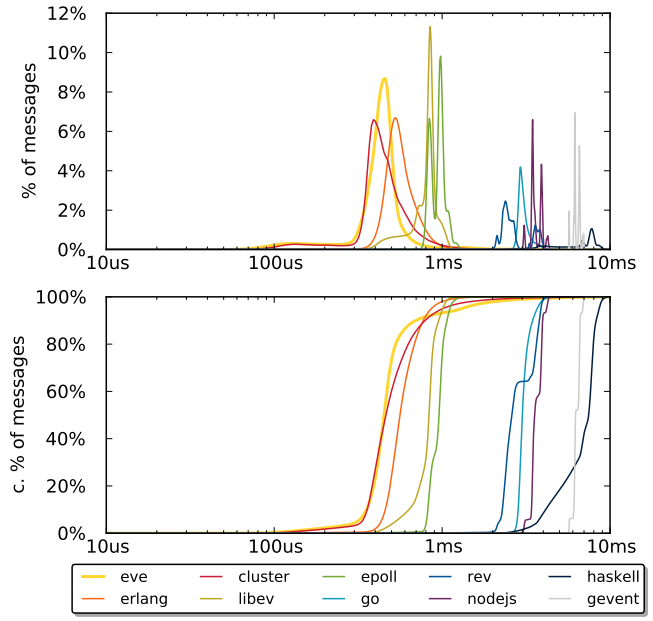
Figure 5.7: Throughput of the echo servers (remote).

Considering the `libev` implementation as a baseline for a single-threaded runtime, one would expect the performance of parallel implementations to achieve better speedups. Two reasons were found that can explain this fact. The first is the very nature of the problem. Unlike the Fibonacci test, which is CPU intensive, the echo server test is IO intensive. In particular, `read/write()` operations require large memory bandwidth, which unfortunately does not scale with added worker threads. To mitigate this bottleneck zero-copy operations could be implemented [28]. The second has to deal with normal parallel slowdown causes. Problems such as cache misses aggravate the memory bandwidth bottleneck and are more common in parallel architectures due to inter-process invalidation [7]. Additionally, synchronization is required to maintain a coherent application state. This synchronization is employed by the EVE runtime (using spinlocks, monitors and atomic operations), but also by the Linux kernel[3]. Even in the absence of concurrent accesses, these primitives incur in additional overhead that is not present in single threaded architectures. Additionally, this overhead may increase when used simultaneously by more threads. Possible solutions for these problems are described in section 6.3.

Figures 5.7 and 5.8 indicate the results obtained for the same benchmark, except that the client and the server applications where deployed to distinct hosts. With this setup, interference between the client and server applications is no longer possible. However, the presence of network interference introduces new variables that must be accounted for. Our measurements reported an average round trip time of 189±9.5 microseconds. This indicates that mean reply latency should increase by at least this amount. In fact, the latency increment is higher than this value for all servers. This is because additional work is conducted by the Linux kernel and is transparent for the programmer[4]. Nonetheless, the performance of each server deteriorates even further. This

[3]Spinlocks and mutexes are used in epoll functions to prevent race-conditions

[4]The network device driver must be used instead of the faster loopback device pseudo-driver.

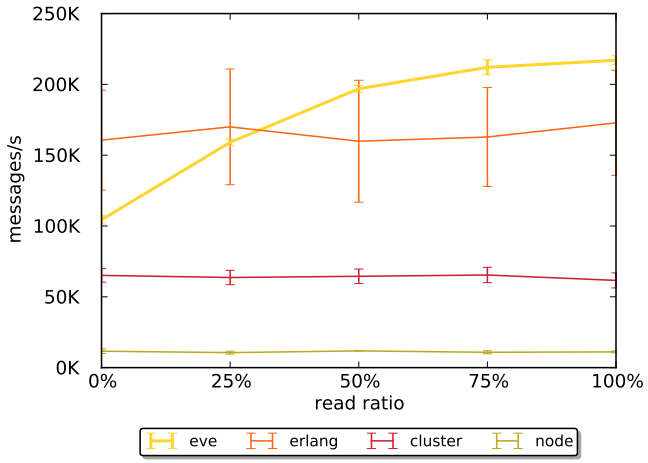


	Q_1 (µs)	Q_2 (µs)	Q_3 (µs)
eve	410	460	510
erlang	510	570	660
cluster	400	470	590
libev	770	840	880
epoll	860	960	1000
go	2910	3010	3170
rev	2390	2600	3580
node	3460	3560	3900
haskell	5770	7460	7870
gevent	6160	6230	6570

Figure 5.8: Reply latency of the echo servers (remote).

suggests that the network latency influences the application processing and increases latency at this level. The request throughput also suffers from decreased performance. Whether or not this is a result of increased latency or the reverse was not verified, even though both hypothesis are possible. On one hand, if the network latency is high it is more likely that the application stalls with no work to be processed, resulting in decreased throughput. On the other hand, if the bottleneck is the application processing then incoming requests will be stored longer in queues, which increases latency.

Implementations that had low performance in the first benchmark suffer a small downgrade from the change of environment (17% to 24%). This is most likely because the execution is already CPU bounded. The `haskell` implementation is an exception achieving even worse results with a 48.8% decrease in throughput. Faster implementations such as `erlang` and `cluster` do not encounter significant performance loss. EVE however, suffers a drop in 122.61K requests per second (33.84%). Our hypothesis is that using networked communications the IO bound was tightened to a level that is superior to `erlang`'s processing capabilities but lower than `eve`'s. Despite this performance loss, EVE still remains the best solution, both in request throughput and in reply latency.



	0%	σ	25%	σ	50%	σ	75%	σ	100%	σ
eve	104.362	2.927	159.060	2.086	196.891	2.370	212.036	5.340	217.039	3.168
erlang	160.622	35.288	169.999	40.847	159.863	43.042	162.829	34.917	172.819	37.139
cluster	65.123	4.800	63.631	5.071	64.482	5.132	65.402	5.409	61.589	5.308
node	11.602	1.595	10.600	1.033	11.830	0.142	10.834	1.088	11.115	0.694

Figure 5.9: Throughput of the atomic counter servers subject to the percentage of read operations.

5.3 Atomic Counter

The echo test described in the previous section exemplifies an embarrassingly parallel problem. There is no shared state between clients, which allows them to be handled separately without synchronization. The atomic counter test is a modification to this example, where shared state is maintained. In particular, a single variable `counter` is accessed by all clients. Two types of operations are permitted: *read* which allows each client to retrieve the value stored in `counter` and *increment* which atomically reads and increases stored value by one and returns the new value. These operations are transmitted through the network using a single byte `00` and `01` respectively. Because both operations require a response packet, this application has similar IO patterns to the echo server, allowing the previous results to estimate an upper limit on performance.

Ideally, `read` operations can execute in parallel but each `increment` operation must be executed in mutual exclusion. The EVE implementation makes use of the language's access permissions to achieve this semantics. Erlang however, does not have this feature. For this reason, the counter is maintained by a single actor, using message passing for synchronization. This solution sequentializes all accesses, including `read` operations. The remaining actions are still

executed in parallel (e.g.: IO, parsing). With nodejs `cluster` library, each worker executes in a new process. For this reason, shared memory solutions are not possible. For testing purposes we decided to approach this limitation with a commonly used alternative: in-memory databases. In particular we selected mongodb which suports the required atomic operations. Figure 5.9 shows the throughput obtained for each server. A sequential `node` implementation was also included for observation. It's performance is inferior than the other solutions. This is a consequence of its single threaded runtime, but also because the native support for binary packets is limited[5]. Using `cluster` to parallelize IO yields better results. However, the throughput obtained with this solution is still low, most likely due to the extra overhead off communication with the mongodb database. In fact, only 28.8% of the previously achieved throughput is reached. The `erlang` implementation also suffers from performance loss, averaging at 74%. The large standard deviation observed for this test is very high, ranging from 20.99% to 27.04%.

The proportion of `read` operations are key to the performance of the EVE runtime. On one end, with 100% `read` operations the `counter` value is constant and complete parallelization is possible. The performance obtained for this case is around 90% of the expected value, indicating that the overhead of additional synchronization is low. For this ratio, EVE outperforms the erlang by 25%. On the other end, with 0% `read` operations, each action must wait for its predecessor to relinquish access to the shared variable. In this case, the performance drops to 43.5%, being slower than `erlang` by 35%. The other implementations do not suffer significantly from this ratio: `erlang`'s implementation sequentializes every operation and mongodb uses atomic operations instead.

[5]The `binary` library used for this purpose is not implemented using native code.

	Program	LOC
Fibonacci	C++	30
	TBB	73
	Fork/Join	41
	EVE (sequential)	10
	EVE (parallel)	18
	EVE (parallel shared)	18
Echo Server	EVE	16
	Cluster	12
	Erlang	26
	Libev	99
	Epoll	195
	Go	15
	REV	14
	Node.js	3
	Haskell	27
	Gevent	17
Counter	EVE	27
	Erlang	42
	Cluster	32
	Node	13

Table 5.3: Lines of code of each implementation.

5.4 Overview

Previous tests shown that the EVE runtime achieves a good performance both on CPU and IO bounded applications. However, the success of a framework also is determined by other metrics. In particular, the easiness of use is one of the features that programmers look at when selecting a framework for development. Lines of Code (LOC) are hardly a precise description of a program's complexity. Nevertheless, this measurement can be used as an indicator of the expressiveness of a language. Table 5.3 contains the LOC of every implementation used in our benchmarks. In this metric, EVE bests every low-level framework (e.g.: libev, TBB, Fork/Join). It also performs fairly well against higher-level frameworks such as gevent and REV despite achieving much better performance. For this reason we believe that EVE facilitates the development of high-performance applications, without relinquishing the easiness of use found in other frameworks.

Chapter 6

Conclusion

This chapter draws the conclusions of the document by summarizing the work that has been done, reviewing its importance and indicating how it can be improved.

6.1 Overview

EVE is a framework for the development of high-performance parallel applications. It encompasses the EVE language definition, complete with its operational semantics, the EVE compiler and a runtime system. The framework is heavily based on the event-oriented programming paradigm. However, unlike other event-oriented solutions it enables the use of multi-core hardware using a shared-memory model.

The runtime system uses a task based approach and a fast work-stealing algorithm for load-balancing. Our benchmarks indicate that it achieves not only good single-core performance but also presents good scalability. Using the `epoll()` system call right in the work loop, our system achieves an architecture that handles IO in parallel. This solution also performs better than the alternatives presented by existing frameworks.

We also created the EVE compiler that closes the gap between the theoretical language definition and the runtime implementation, allowing practitioners immediate access to these new tools.

6.2 Relevance

Existing programming languages either provide good solutions for CPU-bounded applications (by using parallelism) or provide good IO handling (using event-oriented programming). The few solutions that attempt to do both (e.g.: erlang and cluster for node.js) convey additional restrictions to the programming task. To circumvent this restrictions the programmer is faced

with the trade-off of sequentializing accesses using the actor-model or to offload the complexity to another application that synchronizes data accesses.

To the best of our knowledge, EVE is the first framework that couples event-oriented programming with a shared-memory model for concurrency. This framework is novel in this sense and proves that the two models are compatible. In fact, our benchmarks suggest that they work well together, achieving a good stand point between CPU-bounded and IO-bounded performance, as well as development complexity.

EVE is targeted mainly at the development of web applications where it excels. Nonetheless, due to its characteristics (e.g.: event-oriented architecture, typed variables, managed shared memory and automatic parallelism) it has practical applicability to several fields such as system tools and high-performance computing. The introduction of new user-level libraries could also introduce new possibilities to the realm of EVE like graphical user interfaces, machine learning, signal processing, and others.

6.3 Future Work

Throughout this document, items that expressed sub-par performance or that could otherwise be improved were elicited. This section contains a synopsis of the problems that were identified and possible solutions that could be implemented in future work.

The benchmarks on runtime system demonstrated its good performance and scalability. However a problem was identified where workers could inadvertently sleep even in the presence of executable tasks. This was associated with the algorithm used to reduce power consumption when the available parallelism is insufficient for all workers. The current solution makes use of dynamically updated timeouts that increase exponentially up to a maximum of five seconds. This timeout is local to each worker, and is reset each time new work is encountered (i.e.: stolen from other workers or received from the IO system). The problem with this solution is encountered when applications without significant IO have considerable fluctuations in parallelism level. When the parallelism is low, workers begin to wait with the specified timeout for IO. In this time, new tasks generated by other workers are not observed. A solution is to explicitly wake up workers when such tasks arrive. To do so, a new algorithm for global balancing should be implemented, such as the one described in [26].

The runtime system is able to scale well on the available 16-core hardware due to its architecture. However, in the current system, a single `epoll` set is shared by all workers. Even though workers are capable of executing simultaneous operations on this set, we believe that unnecessary contention is managed at kernel level (where various locks are implemented to maintain coherency). Using an `epoll` set for each worker could reduce the overhead of synchronization. Additionally, having an extra global set that contains all `epoll` instances would enable correct distribution of IO across the many available workers.

The compiler implementation is still in alpha stage. All major features have been implemented. Nevertheless, some syntactic sugar components are missing (e.g.: optional types in variable declarations). Additionally, the type checker implementation is still lacking in robustness and fails under some conditions. This is an inherent consequence of the chosen architecture. In particular, the type checker attempts to validate all functions using a bottom-up approach, starting at the expression levels. With this approach, functions that receive template type parameters fail passing the semantic analysis since operations on such types are undefined. A correct solution is to delay the parsing of these functions to a posterior phase. In particular, when an invocation of such functions is encountered, the appropriate implementation can be parsed because, at that location, type parameters are known.

The value of a development framework is tightly bound to the amount of users it influences. The dissemination of EVE is a task that we expect to achieve in the near future. For this purpose, a paper on the architecture of EVE, the implementation of its runtime and the performance obtained will be submitted for acceptance to the 35^{th} conference on Programming Language Design and Implementation (June 2014). PLDI is one of the top peer-reviewed conferences in its field, and a publication in this event will surely attract interest in both researchers and practitioners. The deadline for the submission of abstracts ends 8th November 2013. Due to the immediate usability of our framework, on-line availability is also recommended. For this reason, all deliverables were made accessible to the public at `http://githib.com/jprafael/eve`. We hope this encourages the hacker and tinkerer community to explore and improve on our research.

6.4 Final Remarks

Taking into account the objectives that were defined at the beginning of this work (see section 1.3) we can conclude that the goals set for this work were all successfully accomplished. The EVE framework was designed and implemented, evidencing good results. Efforts for dissemination of our work are in progress.

Bibliography

[1] T.E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta*, pages 119–129, 1998.

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[4] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[5] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[6] Erisa Dervishi. *Evaluate the benefits of SMP support for IO-intensive Erlang applications.* PhD thesis, KTH, 2012.

[7] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. *SIGARCH Comput. Archit. News*, 17(2):257–270, April 1989.

[8] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 48–48. IEEE, 1997.

[9] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972.

[10] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[12] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *In Proceedings of 6th Annual Linux Symposium*, 2004.

[13] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[14] James Hamilton. The cost of latency. *URL: http://perspectives. mvdirona. com/2009/10/31/TheCostOfLatency. asp*, 2009.

[15] Intel. Intel Math Kernel Library. `http://software.intel.com/en-us/intel-mkl`, 2003. [Online; accessed 4-Jan-2013].

[16] Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. Work-stealing without the baggage. *SIGPLAN Not.*, 47(10):297–314, October 2012.

[17] Marc Alexander Lehmann. Benchmarking libevent against libev. `http://libev.schmorp.de/bench.html/`, 2011. [Online; accessed 31-Aug-2013].

[18] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer Berlin Heidelberg, 1993.

[19] David Lilja. The impact of parallel loop scheduling strategies on prefetching in a shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 5:573–584, 1994.

[20] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, March 2008.

[21] Scott McPeak and George C Necula. Elkhound: A fast, practical glr parser generator. In *Compiler Construction*, pages 73–88. Springer, 2004.

[22] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[23] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[24] John A Rice. *Mathematical statistics and data analysis*. Cengage Learning, 2007.

[25] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proceedings of the first annual ACM symposium on Theory of computing*, STOC '69, pages 165–180, New York, NY, USA, 1969. ACM.

[26] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

[27] Stephen Shankland. What would happen if Moore's Law did fizzle? `http://news.cnet.com/8301-11386\_3-57526583-76/what-would-happen-if-moores-law-did-fizzle/`, 2012. [Online; accessed 4-Jan-2013].

[28] Moti N Thadani and Yousef A Khalidi. *An efficient zero-copy I/O framework for UNIX*. Citeseer, 1995.

[29] Andreas Voellmy, Junchang Wang, Paul Hudak, and Kazuhiko Yamamoto. Mio: A high-performance multicore io manager for ghc.