VRIJE UNIVERSITEIT AMSTERDAM

Faculty of Science

MSc Parallel and Distributed Computer Systems

Master Thesis

Verification of GPU Program Optimizations in Lean

---

by

Björn Fischer

2613401

September 26, 2019

36 ECTS

January 2019 – September 2019

Supervisor:
Pieter Hijma
Jasmin Blanchette

Assessor:
Atze van der Ploeg

Abstract

Graphics processing units (GPUs) have become of major importance for high-performance computing due to their high throughput. To get the best possible performance, GPU programs are frequently optimized. However, every optimization carries the risk of introducing bugs. In this thesis, we present a framework for the theorem prover Lean to formally verify transformations of GPU programs. Our formalization generalizes the concepts of GPU programming and follows a layered approach. We define an abstract programming language that captures the essential primitives of GPU programming and construct a logic to relate two implementations of a program. The abstract language forms the basis to formalize Many-Core Levels (MCL), a GPU programming language with a focus on optimizations.

I dedicate this thesis to my parents Sonja and Martin

Ich widme diese These meinen Eltern Sonja und Martin

Contents

1 Introduction **1**

1.1 Structure of This Thesis . . . 3

1.2 Reproducibility . . . 3

2 Background **4**

2.1 GPU Programming . . . 4

2.1.1 Architecture of GPUs . . . 4

2.1.2 Challenges of GPU Programming . . . 8

2.1.3 Stepwise Refinement with MCL . . . 9

2.2 Theorem Proving with Lean . . . 12

2.2.1 Inductive Types . . . 15

2.2.2 Propositions and Proofs . . . 18

2.2.3 Tactics . . . 19

2.3 Program Verification using Hoare Logic . . . 21

2.3.1 `while`-program Syntax and Semantics . . . 21

2.3.2 Hoare Logic . . . 22

2.3.3 Relational Hoare Logic . . . 24

3 Design **27**

3.1 Layered Architecture . . . 27

3.2 Memory Model . . . 28

3.3 Parlang . . . 29

3.3.1 State . . . 30

3.3.2 Synchronization on State . . . 31

3.3.3 Active Map 32
3.3.4 Kernels 34
3.3.5 Semantics of Kernels and States 35
3.3.6 Programs and Programs Semantics 38
3.3.7 Properties of Parlang 38
3.3.8 Equivalence to Non-Monotonic Semantics 39
3.4 Relational Hoare Logic for Parlang 42
3.4.1 Chaining Relational Proofs 44
3.4.2 Inference Rules 45
3.4.3 Verification of Multiple Thread Groups 48
3.5 MCL 49
3.5.1 Primitive Types 49
3.5.2 Arrays 50
3.5.3 Signature 50
3.5.4 State 51
3.5.5 Expressions 52
3.5.6 Kernels and Programs 55
3.5.7 Translation to Parlang 56
3.6 Relational Hoare Logic for MCL 60

4 Use Cases 64
4.1 Example 1: Swapping Assignments 64
4.2 Example 2: Known Branch 67

5 Related Work 70
5.1 Semantics and Logics of GPU Programs 70
5.2 Verification Tools 71
5.3 Relational Hoare Logic 72
5.4 Optimization Tools and Automation 72

6 Conclusion and Future Work 73

A Lean Variables 78

Chapter 1

Introduction

In recent years, there has been increased interest in using graphics processing units (GPUs) to perform computations instead of using the central processor unit (CPU). GPUs offer a high computational power at a lower cost. The increased performance, compared to CPUs, comes from a high level of parallelism at the hardware level. With the change in hardware also comes a change in the programming paradigm: a developer is exposed to the parallelism and must make use of it to gain performance. Programs written for CPUs must be rewritten to run on GPUs. To this day, GPU programming remains a complex challenge.

One challenge of GPU programming is to write error-free programs. Parallelism in GPUs is achieved by running multiple threads at once. The parallel nature makes it harder to check programs for errors. Threads progress independently and must coordinate to access shared resources. An uncoordinated access can cause undefined behavior, which is generally undesirable. The GPU cannot find such faults at runtime, because checking for it would decrease the performance.

Another challenge of GPU programming is to get the highest performance. The most naive implementation of a program is often not the best performing one. Among others, the architecture of the hardware and the type of workload can influence how the best performance can be achieved.

One approach to gain performance is to stepwise refine a program with the aid of the compiler, e.g. using Many-Core Level (MCL) [1]. MCL is an academic language to program many-core processors for optimal GPU utilization. It captures the architecture and behavior of a GPU in a *hardware description*. The compiler uses the hardware description and program to suggest optimizations to the developer. The developer can react to it by manually rewriting the program. This process can be repeated iteratively. The benefit of this approach is that the developer is in charge of the program transformation and can use domain-specific knowledge during the optimization.

The problem with program transformations is that every rewrite can introduce bugs or change the behavior of the program against the intention of the developer. Even trivial transformation might lead to subtle differences during the execution. One approach to test the program is to run it on a test set and compare its outputs. However, this is likely not to cover all cases. Furthermore, the nondeterministic nature of GPUs does not even guarantee conclusive evidence about the cases covered by the test set.

One approach to increase confidence in programs is to use formal methods to systemically check them for flaws. There already exists research on the formal verification of GPU programs. Tools such as GPUVerify [2] and PUG [3] can be used to identify common programming mistakes; however, they have limitations:

- They focus on the verification of a single program and do not make use of previous program versions.
- They are automated and cannot verify all programs.
- The tools are often not verified themselves. Hence, a flaw in the tool would invalidate its results.

In this thesis, we address the issue of verifying GPU program transformations. We develop a prototype of a framework to formally check that a change in the program does not introduce bugs and preserves the existing behavior. In particular, our work pursues the following objectives:

- Our framework should produce formal proofs for the correctness of program transformations that are checked by a theorem prover.
- The framework must be flexible enough to support arbitrary program transformations. That is, the framework must not restrict itself to particular transformations in favor of automation.
- The focus lies on proving program transformations interactively by hand, with the help of theorems. This forms the basis to eventually perform fully automated verifications. However, the latter is out of reach of this thesis.
- The input language for our framework is MCL. We aim to model a reasonable subset of the language specification.
- We investigate how to generalize the concepts of GPU programs based on MCL, such that our framework can be adapted to other programming languages.

Our work is a useful addition to the current research because we provide a complementing method of verification, which reasons about transformations. By proving a few use cases we show that the approach is feasible.

1.1 Structure of This Thesis

In Section 2.1, we explain how GPUs work and what the challenges of GPU programming are. Furthermore, we explain the programming paradigm *stepwise refinement* that is embraced by MCL.

We use the interactive theorem prover Lean (Section 2.2) as the basis for our framework. It provides us with the environment to formalize the programming languages and what program transformations are. Furthermore, it can be used as a proof system to carry out the proofs interactively.

Section 2.3 discusses the technical background of program verification. For the verification of program transformations, we rely on Hoare logic, which is a formal system to reason about program correctness. We use a relational version of the Hoare logic that reasons about two programs instead of one.

In Section 3.3, we define our own abstract programming language called Parlang that captures the essential primitives of GPU programs. Having this abstract programming allows us to reason about the behavior of GPU programs, without having to deal with the complexity of a real programming language. We model the syntax and operational semantics in Lean. Section 3.4 defines a relational Hoare logic for Parlang and investigates how to reason about GPU program transformations.

In Section 3.5, we formalize the programming language MCL. We define its semantic in terms of Parlang to make use of theorems that have been defined on the latter. Section 3.6 adapts the relational Hoare for Parlang to MCL. Furthermore, we define additional theorems that are specific to MCL.

In Chapter 4 we show how applicable our framework is by proving the transformation of programs. Chapter 5 discusses related work and Chapter 6 draws a conclusion and provides an outlook.

1.2 Reproducibility

The practical part of this work has been checked by the Lean theorem prover. The formalization including some use cases is available at
`github.com/fischerman/GPU-transformation-verifier`.

Chapter 2

Background

2.1 GPU Programming

Nowadays, GPUs play an important in high-performance computing due to their design for compute intensive applications [4]. They contain many processing units working in parallel connected to a large memory. A GPU is a throughput oriented machine that achieves its high performance by executing the same instruction on multiple data elements. Therefore, typically, each unit only works on part of the data. To that end, arrays are partitioned and distributed among the threads by the developer. To increase performance there are typically no runtime checks to catch concurrency issues.

2.1.1 Architecture of GPUs

A GPU is a coprocessor that is driven by the CPU, called the host. The latter initializes the input data and transfers it to the GPU. It then starts the workload on the GPU. The piece of code that runs on the GPU is called the kernel. The host program can start multiple kernels in parallel and in sequence (which may work on the same data). As such, the host program serves as a coordinator for dependent GPU workload.

GPU architectures evolve rapidly with new advances in technology and developers needs. To maintain a stable environment for developers, programs are typically written in high-level languages, such as CUDA [5] and OpenCL [6]. We distinguish between the device layer and the programming abstraction layer. All program statements are translated to hardware instructions by the compiler, the runtime environment or a hardware scheduler. However, this additional layer comes at a price. To get the best performance of a GPU, aspects of the device layer should be considered when writing programs. Depending on the language the device layer is exposed to different degrees.

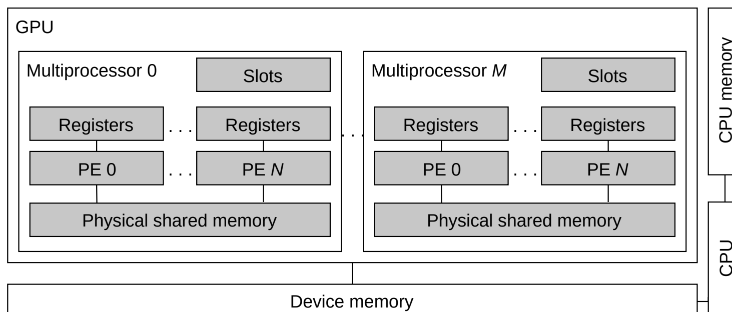

Figure 2.1: A typical GPU device architecture (adapted from [2]).

The core paradigm of a GPU is high-degree data-parallelism. When workload is scheduled on the GPU, the host defines how many threads run the kernel in parallel. Threads run independently of each other and have local memory that is only accessible to them. They are assigned a unique thread identifier (abbreviated *tid*), which allows threads to take different control paths. A common use case is to divide and distribute an array among the threads. The thread identifier is used to compute the respective index ranges of the array.

Figure 2.1 depicts a generic GPU device architecture. A GPU consists of many multiprocessors (abbreviated MP). Each multiprocessor accommodates a set of processing elements (abbreviated PE), which perform arithmetic operations. Each PE has dedicated registers and they all share access to the physical shared memory (also called on-chip scratchpad). The slots contain meta-data of the scheduled threads, such as the instruction counter. Additionally, all multiprocessors have access to the large off-chip device memory.

On the programming abstraction layer, threads are organized in groups as depicted in Figure 2.2. On invocation, the host program determines the kernel code, the kernel arguments and the number of threads in the group. This group configuration influences how threads are scheduled on the device layer, what memory they can access and how they can communicate.

The global memory is shared between all groups and resides in the device memory. Memory space is allocated and initialized by the host program. For example, the host program may transfer two input arrays to device memory and allocate one output array to perform an array multiplication. To make the arrays accessible to the kernel they can be provided as arguments. If one group writes to a global memory location while others read from it, the behavior is undefined. It is the programmer's responsibility to avoid such conflicting accesses.

Furthermore, each thread has thread-local memory, which is only accessible to the respective thread. The thread-local memory resides in the registers and the device memory. How many registers a kernel has is automatically decided at compile time. It can be manually overwritten but is limited by the physical

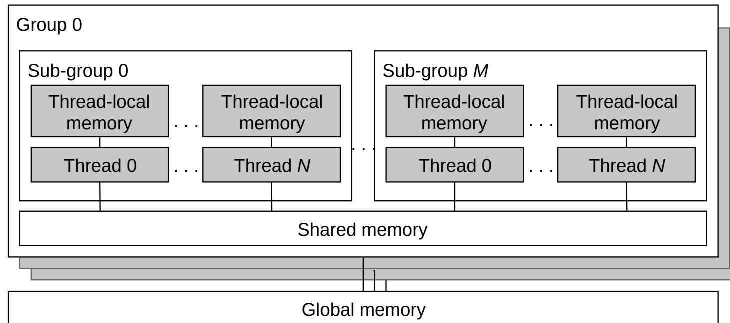

Figure 2.2: Programming abstraction layer

number of registers. The remaining thread-local memory, which does not fit in registers, is allocated in the slower device memory (but is not part of the global memory). This is referred to as spilling.

Additionally, all threads in a group have access to the shared virtual memory, or just called shared memory. This can be used to exchange data within a group. Without further restrictions, a developer has to assume that threads progress independently. To allow coordinated access to shared memory, threads within a group can synchronize using a barrier instruction. When a thread reaches a barrier it stalls until *all* threads in the group reach *this* instruction (execution barrier). Furthermore, accesses to the memory that happen before the barrier are visible to other threads in the group after the barrier. Threads in different groups have no ability to synchronize.

Groups are scheduled on multiprocessors by the runtime environment, following a few rules. Threads within a group are always scheduled on the same multiprocessor. This ensures that they have a shared memory and can synchronize. If there are more threads than processing elements (which by itself is valid) the scheduler decides which threads run at which time. However, a multiprocessor may accommodate multiple groups at the same time. Whether this is possible is determined by the following factors:

Slots: The MP must have enough slots to fit all threads.

Physical shared memory: The physical shared memory must be large enough to accommodate the shared virtual memory of all groups that are scheduled on the MP.

Registers: The requested number of registers of all groups combined must not exceed the available number of registers in the MP. How many registers a group needs depends on the number of threads relative to the number of PEs. If the number of threads is 10 times as high as the number of PEs, each PE must have the capacity to accommodate registers for 10 threads.

As such, for performance reasons, it may make sense to reduce the number of requested registers per thread to fit more groups on an MP.

Whether a group is co-scheduled on the same multiprocessor does not affect the (in)ability to synchronize between groups. Communication between groups is only possible by means of sequential kernel execution or special atomic instructions.

To reduce circuit complexity all processing elements of a multiprocessor run the same instruction at the same time (on the device layer). This is called lockstep execution or Single-Instruction-Multiple-Threads (SIMT) architecture. Groups are divided into sub-groups (also called warp [7]) and threads in the same sub-group run in lockstep, while threads in different sub-group progress independently. Because threads in the same group run the same kernel, only one stream of instructions must be fetched and only one instruction counter is required per sub-group. Special care must be taken when threads perform conditional statements or loops. Consider this simple program where `a` is a thread-local variable:

```
if (a) {
    doThen();
} else {
    doElse();
}
```

The truth value of the condition is dependent on `a` and therefore specific to each thread. Hence, a few threads may go to the if-branch while the others go to the else-branch. To do this in lockstep, all threads execute the same instruction; however, threads get marked as active or inactive. If a thread is inactive an operation has no effect, most notably the memory does not change. Suppose the condition is true for some thread `t`. Then `t` will be active in the if-branch and inactive in the else-branch. Consider the example of a loop:

```
int i = 0;  -- thread-local variable
while (i < tid) {
    i++;
}
```

Before every iteration, the condition is evaluated per thread and threads get marked as active and inactive accordingly. A thread stays inactive if it is inactive before entering the loop (cf. Figure 2.3). The loop terminates if, after reevaluating the condition, all threads are inactive. The activeness is reset to the state before the loop was entered. According to specification, a thread may be inactive in iteration `i` and active again in iteration `i+1`. In our formalization, we show that reactivation of threads *to perform another loop-iteration* always coincides with faulty behavior.

Threads in the same group do not all run in lockstep if they do not fit into a single sub-group. Some languages do not expose the concept of sub-groups,

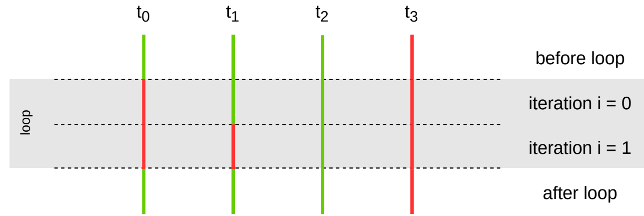

Figure 2.3: Illustrating lockstep execution with four threads. Red lines denote inactive threads, green lines denote active threads. Thread 3 is inactive before the loop.

whereas, for example, CUDA allows assignments from threads to sub-group. In general, the SIMT architecture is transparent, but information about the hardware might be used for optimal resource utilization [8]. We will use the concept of lockstep in our semantics.

2.1.2 Challenges of GPU Programming

Highly concurrent programs are error-prone [9, 10]. We identify three common problems with GPU programs: non-termination, barrier divergence, and interference.

Non-termination is the problem that a program may not terminate on certain inputs. As opposed to interactive programs, the result of a GPU program is obtained after termination. The problem is not specific to GPU programs, but the concurrent nature of GPUs makes it harder to check for termination. Non termination can arise from infinite loops.

The problem of barrier divergence arises when not all threads reach a barrier instruction. A barrier is resolved when *all* threads in a group reach it. Otherwise, the behavior is undefined [5]. That means that if a barrier is placed inside an if branch, all threads must take the same branch. In the case of a barrier inside a loop, all threads must either perform an iteration or exit the loop. This reasoning becomes more complicated in the case of nested conditional statements.

The problem of interference comes from uncoordinated accesses to shared memory. We consider an example with a shared variable `a`:

```
if (tid == 0) {
    a = a + 1;
}
if (tid == 1) {
    b = a;
}
```

Thread 0 stores into a and thread 1 reads from the same variable. Without assuming that these two particular threads run in lockstep, the order of the load and store is undetermined. Hence, the program shows nondeterministic behavior. The content of variable b either reflects thread 0's changes or it does not. This makes the programs result irreproducible which is generally unwanted. However, syntactically this program is accepted by the compiler and it runs properly on a GPU. The program can be changed to a deterministic version by placing a barrier between the two **if** statements. The absence of interference is called non-interference.

All of the problems above should be avoided when writing GPU programs. Our formalization will consider all these problems as faulty behavior.

2.1.3 Stepwise Refinement with MCL

Many-Core Levels (MCL) [1], an academic language to program many-core processors, has been designed to get optimal GPU utilization while giving the developer the choice to retain portability. To that end, it makes the trade-off between high-level and low-level abstraction explicit to the programmer. High-level programs have good portability while low-level programs can yield better performance. In MCL the developer has the choice to move to a lower level if required.

An abstraction level is expressed as a hardware description. The levels form a tree with the level `perfect` on top. The level `perfect` has no restrictions on the number of threads in a group or the amount of shared memory. Under these (unrealistic) conditions, a naive version of a kernel can be constructed. As an example, we consider a kernel for matrix multiplication:[1]

```
perfect void matmul(int n, int m, int p,
        float[n, m] c,
        float[n, p] a, float[p, m] b) {
    foreach (int i in n threads) {
        foreach (int j in m threads) {
            float sum = 0.0;
            for (int k = 0; k < p; k++) {
                sum = sum + a[i, k] * b[k, j];
            }
            c[i, j] += sum;
        }
    }
}
```

The matrices a, b and c are global with the respective dimensions n, m, and p. The `foreach` instruction denotes parallelism. In this version, every thread is

[1] MCL examples are adapted from `https://github.com/JungleComputing/mcl/tree/add-matmul-versions/input/mcpl/matrixmultiplication`.

responsible for computing a single value in the output matrix. `i` and `j` together make up the thread identifier, that is used to access the respective parts of `a`, `b` and `c`. This implementation ignores much of the concepts of a GPU, such as groups and shared memory. In fact, they are not accessible on the level `perfect`. We would have to move down one level (to `gpu`) where the same program looks as follows:

```
gpu void matmul(...) {
const int nrThreadsM =
↪  gpu.hierarchy.blocks.block.threads.max_nr_units;
const int nrBlocksM = m / nrThreadsM;
foreach (const int i in n blocks) {
    foreach (const int bj in nrBlocksM blocks) {
        foreach (const int tj in nrThreadsM threads) {
            const int j = bj * nrThreadsM + tj;
            float sum = 0.0;
            for (int k = 0; k < p; k++) {

                sum += a[i,k] * b[k,j];
            }
            c[i,j] += sum;
        }
    }
}
}
```

A block in MCL refers to a group. The transformation to a lower level can be performed automatically using the compiler, but by itself does not yield better results. However, it has more potential for optimization. For example, the level `gpu` introduces the concept of shared memory. In the above version of matrix multiplication, threads in a group partially require the same data. They can share that data through the shared memory, by distributing the store-instructions among the threads.

The methodology that MCL embraces to optimize programs is called stepwise refinement for performance [11]. The idea is to start with a naive implementation, as shown above with the matrix multiplication. In multiple steps, a part of the program is rewritten by the programmer while preserving the input/output behavior. Unlike compiler optimizations, the programmer is in control of the rewrite. They can use domain knowledge, such as common patterns in the input data, that cannot be inferred by a compiler. In the case of matrix multiplication, copying values to shared memory can reduce the number of accesses to the slow global memory drastically.

The process of optimization is integrated with the transition to a lower abstraction-level, as depicted in Figure 2.4. The programmer optimizes the program with the aid of compiler feedback (inner loop on the right). For instance, the compiler might report an inefficient memory access pattern. By moving to a lower abstraction level (outer loop) the programmer is given a more accurate descrip-

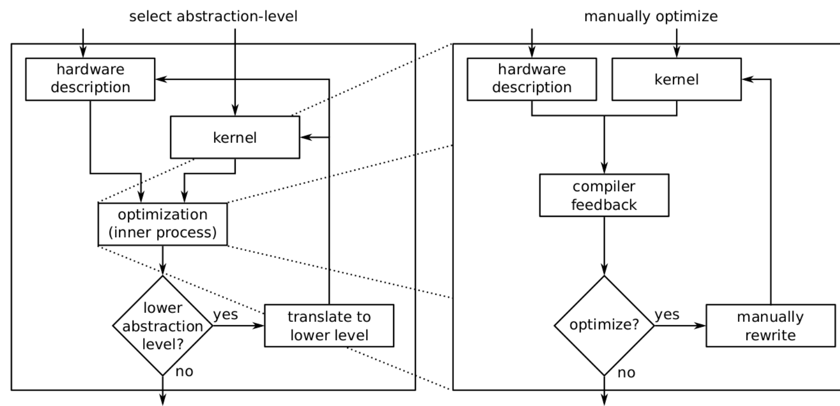

Figure 2.4: Stepwise refinement in MCL (from [1]). The left box depicts the outer process *selection abstraction-level*. The inner process *manually optimize* is depicted by the right box.

tion of the hardware, and hence more potential for optimization. The rewrite to a lower level can be performed by the compiler. However, this transformation is not formally verified.

In MCL, the kernel is expressed as a function. At the top level, there are multiple foreach-blocks that express group- and thread-parallelism. Executable code must only exist in the innermost foreach-body. This code is executed on the processing elements. The syntax follows a C-like imperative language. The for-loop is used for non-parallel looping. Kernel code may call regular functions, but there is no support for recursion[2].

MCL supports multi-dimensional arrays. The size of each dimension is fixed but can be dependent on a constant argument. Ergo, multiple arrays can be syntactically fixed to the same size. There is no dynamic memory allocation (inside of the kernel).

[2] Recursion is not supported on most GPU architectures.

2.2 Theorem Proving with Lean

Lean [12] is the theorem prover that we use to build our environment and perform the proofs. It ensures that the purported proofs are correct and, in turn, that the claim we make is true. Our main theorem will be the equivalence of two MCL programs.

Lean is an *interactive* theorem prover (also called a proof assistant). It is not designed to perform proofs efficiently and entirely automatically. Instead, the user is equipped with a framework to engage automated tools and methods in the proof work. However, unlike automated theorem provers, those tools are not trusted and their output is verified.

The part of Lean that is responsible for verification is called the Lean kernel.[3] The verification is based on dependent type theory, more precisely Calculus of Constructions [13]. The idea of type theory is that every object we study has a type. Examples of types are *booleans* and *natural numbers* (`ℕ`). A type can also be constructed from other types, most notably to define functions. A definition of addition of 1 on natural numbers can be formalized in one of those three equivalent ways:

```
def add1 : ℕ → ℕ := λ n, n + 1
def add1' : ∀ (m : ℕ), ℕ := λ n, n + 1
def add1'' (n : ℕ) : ℕ := n + 1
```

A definition encapsulates a term under a name. Unfolding a definition is called *delta reduction*. The colon is used (also in other places) to denote the type of the defined object. The type of `add1` is constructed from the type `ℕ` and the function type constructor `→` (one of the foundations of Lean). While most type constructors are defined using Lean (we see a few later), the meaning of `→` is predefined by the kernel. A term with the type `α → β`, where `α` and `β` are arbitrary types, is a (side-effect free) function from `α` to `β`. A function is defined as follows: `λ a : α, (b : β)`, where `b` may depend on `a`. This is called abstraction, i.e. we abstract `a` from the term `b`. To eliminate a lambda expression the argument can be applied by appending it to the term. This is the complement to abstraction and is called substitution or *beta reduction*. Each occurrence of `a` in `b` is substituted by the applied argument.

Using function application the term `add1 a` can be reduced to `a + 1`, where `a` is of type `ℕ`. The kernel ensures that the argument is of the correct type. We consider a few examples. Type checking can be explicitly performed on a term using the `#check` command (the reported results are depicted in the comments). We define a few variables with their types using the `variables` keyword:

```
variables (a : ℕ) (b : bool)
```

[3] The Lean kernel and GPU kernel are unrelated.

```
#check a       -- ℕ
#check add1    -- ℕ → ℕ
#check add1 a -- ℕ
#check add1 b -- fails
```

The type of `a` is as expected. Also, the type of the function is according to its definition. The third example is a type-correct function application. The resulting type is the type of the function without the left-most argument. The fourth example shows a failed type check, in which case Lean reports an error. We provide a constant of type `bool`, but type ℕ is expected. This sort of type-checking will also be used to verify proofs.

Function application is left associative. If the terms to be applied are compound, they can be encapsulated in parentheses or preceded by a `$`. The following terms are equivalent:

```
#check add1 7 (4 * (3 + 1))
#check add1 7 $ 4 * (3 + 1)
#check add1 7 $ 4 $ 3 + 1
```

Types, such as ℕ and `bool`, are terms themselves, hence they must also have a type.

```
#check ℕ        -- Type
#check Type     -- Type 1
#check Type 1   -- Type 2
```

Following the types of types, we have an infinite but countable chain of types, which are called type universes. **Type** is short for **Type** `0`. Furthermore, **Type** `n` is a synonym for `Sort (n + 1)`, for every natural number `n`. We learn later about the lowest type universe `Sort 0` (or **Prop**).

List is a common structure to combine multiple elements of a type. We ignore how they work for the most part here, but the type of the containing elements is fixed by the type of list. Given a variable `l`, the type can, for example, be `list nat` or `list bool`. `list` itself is not a type but a type constructor, as can be observed by checking its type.

```
#check list -- Type → Type
```

`list` requires a type to construct a type. Suppose we want to define a function that returns a list containing the same element twice. A definition on natural numbers could look like this:

```
def list2 (n : ℕ) : list ℕ := [n, n]
```

The square brackets are constructor syntax for lists. If we need the same for bool we could write another definition replacing ℕ by `bool`. However, nothing

in `list2` is specific to natural numbers. We can write a definition that works for all types.

```
def list2' (α : Type) (n : α) : list α := [n, n]
```

Note that only the type has changed but not the term. We have introduced an argument `α` which holds the type. The type of `n` is now dependent on `α`. Also, the return type depends on `α`.

Lean can also be seen as a programming language. A term can be reduced using the command **`#reduce`**, which implies that the term is type correct.

```
#reduce (λn, 4 + n) 7 -- 11
```

The reduction includes, but is not limited to, beta reduction and delta reduction. If two terms reduce to the same term they are called definitionally equal. Definitional equivalence is the basis for proving equality.

Arguments that are required by multiple definitions can be abstracted into variables. If a variable is referenced in a definition, the Lean parser adds it as an argument before any other argument. If multiple variables are referenced, the arguments are in the order in which the variables are defined.[4]

```
variables (a : ℕ) (b : ℕ)

def f (c : ℕ) : ℕ := a + b * c
#reduce f 3 7 2 -- 17
```

The Lean parser has a powerful inference engine. Whenever a type or term can be inferred we can replace it by an underscore. The parser tries to fill in those blanks before the code is type-checked by the kernel, or fails with an error. The type (including the colon) can also be dropped entirely if it can be inferred. An underscore can also be used instead of a name when it's not needed.

```
def add1 : _ := λn, n + 1
def add1' := λn, n + 1
def const7 : ℕ → ℕ := λ_, 7
```

When arguments of a definition should always be inferred they can be marked as implicit. Implicit arguments use braces instead of parentheses. They are skipped during function application and must be inferred from other arguments. To demonstrate, in the definition of `list2'` the type variable can be made implicit because it can be inferred from the argument `n`.

```
def list2'' {α : Type} (n : α) : list α := [n, n]
#reduce list2'' 5
```

[4] In this thesis, we make intensive use of variables to reduce the size of listings. A list of all variables, their types and their order are provided in the appendix.

2.2.1 Inductive Types

The type system of Lean can be extended using inductive types [14]. An inductive type extends the axiomatic system of the kernel by construction and elimination of type instances. We start with the simplest inductive type – an enumerative type:

```
inductive weekday
| monday
...
| sunday
```

Every pipe denotes a constructor for the type `weekday`, followed by its name. An instance of this type can only be constructed from those constructors, i.e. `weekday` has exactly seven instances. This property is called *no junk*. By default, the constructor names are only accessible under the name of the inductive type, e.g. `weekday.monday`. If no conflict arises, we will expose the constructor names directly.

We can define inductive types more flexible by providing arguments to the constructors.

```
inductive weekday_in_month
| jan (day : weekday)
| feb (day : weekday)
...
```

We can create an instance of `weekday_in_month` by providing all arguments to one of the constructors, e.g. `jan sunday` is an instance. If two instances are built from different constructors (also w.r.t. their arguments) they are not equal. For example, `jan _ ≠ feb _` (for any weekday), as well as `jan sunday ≠ jan saturday`. This property is called *no confusion*.

A constructor can also reference the type that it defines (hence the name inductive type). We consider a definition of the natural numbers:

```
inductive nat
| zero
| succ (n : nat)
```

To construct a successor we already need an instance of the natural numbers. Consequently, we can nest the constructors to create larger numbers.

A single definition can yield an entire inductive type family. That is, the type itself carries arguments. A good example is list, which by itself is not a type but a type constructor.

```
inductive list (T : Type) : Type
| nil {} : list
| cons (hd : T) (tl : list) : list
```

We have to provide the type for the elements of the list to make it a type, e.g. `list nat`. The empty curly braces in the constructor `nil` instruct Lean to infer `T` from the context, rather than from its non-existent arguments. Type arguments that are placed before the colon are fixed for all constructors and recursive arguments. For example, observe that the argument `tl` does not require `T` in its type. However, constructors may also take arguments from other types *within* the type family. We consider the example `vector` that is similar to `list`, but the length is part of the type.

```
inductive vector (T : Type) : ℕ → Type
| nil {} : vector 0
| cons {n} (hd : T) (tl : vector n) : vector (nat.succ n)
```

To create a vector of length 2, a vector of length 1 is required as an argument. The first constructor can be used to create the initial empty vector. We adapt the list notation (e.g. `[a, b]`), to define a notation for vectors: `v[a, b]`.

Given an instance of an inductively defined type, the recursor of that type (`rec_on`) can be used to eliminate the instance to some arbitrary type α. To illustrate, we consider the unary representation of a natural number `a` as a string (where we do recursion on `a`):

```
def unary (a : ℕ) : string := ℕ.rec_on a "" (λ _ r, "i" ++ r)
#eval unary 5 -- prints 'iiiii'
```

The recursor takes one argument per constructor (also known as minor premises). If the constructor has arguments, the minor premise is a function from the argument values to α (`unary` ignores this argument). The function from the inductive type to a (called major premise) is defined implicitly by the minor premises. The constructor `nat.succ` has an argument of type ℕ. We get the result of applying this argument to the major premise as a separate argument, ergo the definition of `unary` is recursive.

Instead of using the recursor directly, Lean has the pattern matching syntax. The same definition of `unary` looks as follows:

```
def unary : ℕ → string
| 0 := ""
| (n + 1) := "i" ++ unary n
```

Each pipe denotes a case. Cases must be exhaustive (no case missing) and are evaluated top to bottom. Inductive type variables can be matched by their constructors or as a whole in the form of a variable. The definition can be used recursively, which is only possible in the context of pattern matching. Lean tries its best to prove termination automatically. Pattern matching is more convenient to read but not as powerful. We use it where possible but must still work with the recursor on a few occasions.

Lean ships with many inductive types as part of the standard library and also proofs on those types. This includes ℕ (natural numbers), `list`, `vector` and `bool`. We use the standard library when possible. Furthermore we use the *mathlib* library for some additional proofs [15].

A function name can be hierarchical, where the names at each level are combined by dots. If the part proceeding the dot is a type, there is alternative syntax to apply the function on an instance of that type (which resembles methods in object-oriented langauges). The instance is applied to the first argument with the correct type.

```
def list.prepend (n : ℕ) (l : list ℕ) : list ℕ := n :: l
#reduce [1, 2].prepend 6 -- [6, 1, 2]
```

An inductive type with a single constructor (by default called `mk`) can alternatively be defined as a structure. It packs all constructor arguments into a new inductive type and additionally provides projections for all argument. If the type of a term is known, it can be constructed using the anonymous constructor (using angle brackets). For the latter, the arguments must be provided in the order of their definition.

```
structure color := (red : ℕ) (green : ℕ) (blue : ℕ)
def yellow : color := ⟨255, 255, 0⟩
#reduce yellow.red -- 255
```

Alternatively, an instance can be constructed by explicitly naming the arguments and optionally taking the unnamed arguments from an existing instance.

```
def yellow : color := { red := 255, green := 255, blue := 0 }
def white := { blue := 255, ..yellow }
```

Constructor arguments may depend on each other. A typical use-case is to construct a data pair where the type of the second argument depends on the value of the first argument. Lean has this type predefined including notation:

```
structure sigma {α : Type} (β : α → Type) :=
mk :: (fst : α) (snd : β fst)

variables (m : ℕ → Type) (e : m 1)
-- example usage: Σ refers to sigma
def a : (Σ n, m n) := ⟨1, e⟩
```

By defining our own inductive types and functions we can create the environment that is required to prove properties of GPU programs. Note that by extending the type system we are changing the axiomatic system. In other words, a flaw in one of the definitions invalidates all the proofs which use that definition.

2.2.2 Propositions and Proofs

The purpose of a theorem prover is to prove propositions. In this section, we will look at how propositions are expressed and proven. Curry, Howard and de Bruijn state that a dependent type system (such as Calculus of Constructions) is isomorphic to a proof system; an observation called *propositions as types* [16]. The idea behind this paradigm is that we already have a type system that can be repurposed as a proof system. Proofs in Lean follow this principle.

Propositions are expressed as types. An example proposition is `p → q` (where `p` and `q` itself are propositions), which reads *p implies q*. Note that the arrow is the same as for function definitions because it has the same meaning. Intuitively, a function that takes a proof for `p` and returns a proof for `q` is a proof for `p → q`. If the type has an instance, the proposition is true and the instance serves as the proof. If it can be shown that no instance can be constructed, the proposition is false. An instance can be explicitly constructed from a type-correct term. Stating a proof can be done using the theorem keyword.

```
theorem name {p q r : Prop} (hpq : p -> q) (hqr : q -> r) :
p -> r := λ p, hqr (hpq p)
```

Theorems can have assumptions, which are instances of propositions provided as arguments (as seen with `hpq` and `hqr`).

Propositions have their own type universe that has special properties. This universe is called **`Prop`**, which is syntactic sugar for `Sort 0` – the universe below **`Type`**. We can define inductive types (or rather families) in this universe to build up the logic. The following connectives are already defined in Lean:

```
structure and (a b : Prop) : Prop :=
intro :: (left : a) (right : b)

inductive or (a b : Prop) : Prop
| inl {} (h : a) : or
| inr {} (h : b) : or

inductive eq {α : Sort u} (a : α) : α → Prop
| refl : eq a
```

The three type constructors have the infix notations ∧, ∨, and = respectively. A conjunction can only be instantiated if there is a proof for both conjuncts. The proof for a disjunction can be written using either constructor.

```
theorem a {p q r : Prop} (hp : p) (hq : q) : p ∧ q := and.intro
↪  hp hq
theorem b {p q r : Prop} (hp : p) (hq : q) : p ∨ q := or.inl hp
theorem c {p q r : Prop} (hp : p) (hq : q) : p ∨ q := or.inr hq
```

The definition of equality directly relates to definitional equality (by using the same variable on both sides). To prove an equality (using the constructor `refl`) both sides must be rewritten to be definitionally equal.

Inductive types of type **`Prop`** are called inductive predicates. Inductive predicates can be defined, for example, to express properties on the type ℕ:

```
inductive even : ℕ → Prop
| zero : even 0
| two (n) : even n → even (n + 2)
```

2.2.3 Tactics

Proving theorems by writing out terms can be cumbersome and unreadable. Lean provides a more convenient approach through tactics. Tactics are programs that construct proof terms. Instead of writing the terms explicitly, one or multiple tactics build up the term dynamically [17].

To enter tactic mode the term is replaced by a begin-end block, in which we instruct tactics to solve the goal. A tactic is given a list of goals and assumptions plus optional user-defined parameters. It can transform the goal by constructing a subterm but must not close the goal. It can also leave multiple (sub-)goals for the user to be solved, e.g. splitting up a conjunctive. The goal `true` is solved automatically, hence no more tactic is needed.

Lean includes a set of general-purpose tactics. To provide some intuition on how to perform proofs we explain a few tactics.

`apply` applies the conclusion of a term (e.g. a theorem) to the goal and leaves a subgoal for every premise. **`exact`** is similar to **`apply`** but fails if it creates a new subgoal.

```
example (a b : ℕ) (h : a = b) : b = a := begin
    apply eq.symm,  -- goal is now a = b
    exact h,        -- no more goal
end
```

A goal can be rewritten using the tactic `rw` in combination with an equality term. The tactic tries to match the left-hand side of the equation with a subterm of the target and and replaces it with the right-hand side. By default, the target is the goal. It can be changed to a hypothesis using the `at` directive. The rewrite direction can be changed to right-to-left using ←. `rw` cannot rewrite terms that contain a lambda-abstracted variable.

```
example (a b c : ℕ) (h₁ : a = b) (h₂ : c = b) : a = c := begin
    rw h₁,
    rw ← h₂,
end
```

`simp` attempts to simplify the target using a set of rewrite rules. By default, this set contains of all theorems that contain the attribute `simp`. It can be extended with local hypothesis and other theorems. The tactic heuristically rewrites the target and must not leave additional proof obligations. As opposed to `rw`, `simp` can rewrite terms that are lambda-abstracted.

```
example (a : Prop) (h : a) : (a ∧ a) := begin
    simp [h],    -- rewrites a ^ a to a
                 -- and closes a with h
end
```

Many of the formal proofs in this thesis are omitted for readability but are available in Lean in the appendant repository.

2.3 Program Verification using Hoare Logic

Program verification is the act of systematically checking a program for given properties [18]. The properties depend on the type of program. For sequential programs, the properties are correctness of the result, termination, and absence of failures. We formalize programs and the logic for correctness in Lean, in order to perform the correctness proofs using the theorem prover.

2.3.1 `while`-program Syntax and Semantics

To illustrate program verification we use sequential programs instead of GPU programs. We mechanize `while`-programs, which are entirely standard (e.g. see [19]):

```
inductive program : Type
| skip {} : program
| compute : (state → state) → program
| seq : program → program → program
| ite : (state → Prop) → program → program → program
| while : (state → Prop) → program → program
```

`compute` is the only instruction that mutates state. The `sequence` constructor allows for composing programs, while `ite` (short for if-then-else) and `while` are used to divert control flow. A sequence of instructions can be written as `instr₁ ;; ... ;; instrN`. The expressions for the constructors `compute`, `ite` and `while` are captured by Lean functions. This is called shallow embedding; the details of the language are modeled using Lean as a programming language. The alternative is deep embedding, where the syntax is extended with definitions for expressions. Shallow embedding enable the reuse all definitions available in Lean (such as the inductive type ℕ and all its lemmas).

Programs operate on state. We also define correctness of a program as assertions on states. The state holds the values of all variables and is mechanized as a function from variable name to value.

```
def state := string → ℕ
```

For simplicity, we consider all fields to store values of type ℕ. A field can be updated using the following function:

```
def state.update (var : string) (val : ℕ) (s : state) :=
λ v, if v = var then val else s v
```

The operational semantics is a mapping from the syntax to its meaning. For `while`-programs this is the behavior on state. There are multiple ways to define the semantics. We use a *big-step semantics*, which relates a program, an initial state and a resulting state (after termination):

```
inductive exec : program → state → state → Prop
| skip {s} :
  exec (skip) s s
| compute {f s} :
  exec (compute f) s (f s)
| seq {p₁ p₂ s u} (t) (h₁ : exec p₁ s t)
  (h₂ : exec p₂ t u) :
  exec (seq p₁ p₂) s u
| ite_true {c : state → Prop} {p₁ p₀ s t} (hs : c s)
  (h : exec p₁ s t) :
  exec (ite c p₁ p₀) s t
| ite_false {c : state → Prop} {p₁ p₀ s t} (hs : ¬ c s)
  (h : exec p₀ s t) :
  exec (ite c p₁ p₀) s t
| while_true {c : state → Prop} {p s u} (t) (hs : c s)
  (hp : exec p s t)
  (hw : exec (while c p) t u) :
  exec (while c p) s u
| while_false {c : state → Prop} {p s} (hs : ¬ c s) :
  exec (while c p) s s
```

The cases of `exec` relate to the instructions of the `while`-program. Most instructions have a single constructor in the inductive predicate. To prove the valid execution of `skip`, the initial and resulting state must be equal. The change of a `compute`-instruction must be reflected in the resulting state. A sequence of instructions must be proven instruction by instruction, where the initial state of the second instruction is the resulting state of the first instruction. `ite` has two constructors: one for a true condition and one for a false condition on the initial state. Respectively, the execution of either the if branch or the else branch must be proven. Similarly, `while` has one constructor to perform an iteration and one constructor to exit the loop. A valid derivation implies that the program terminates.

2.3.2 Hoare Logic

The semantics can be used to prove the validity of a single initial and resulting state. To prove correctness of a program, we are interested in many possible states. We consider the example of a program that computes the quotient and remainder by dividing `x` by `y`:[5]

```
def div : program :=
    compute (λs, s.update "quo" 0) ;;
    compute (λs, s.update "rem" $ s "x") ;;
    while (λs, s "rem" ≥ s "y") (
        compute (λs, s.update "rem" $ s "rem" - s "y") ;;
```

[5] Example adapted from [18].

```
        compute (λs, s.update "quo" $ s "quo" + 1)
    )
```

On the resulting state, rem should be smaller than y for all initial states. Furthermore, the q * y + rem should be equal x. We can state this correctness property as an assertion on state.

```
def postcondition := λs, s "q" * s "y" + s "rem" = s "x" ∧ s
↪  "rem" < s "y"
```

Using Hoare logic, this property can be proven for all valid executions.

```
def hoare (P : state → Prop) (p : program) (Q : state → Prop) :
↪  Prop :=
∀s t, P s → exec p s t → Q t
```

A proposition in Hoare logic is referred to as a Hoare triple. The three components are a program and two assertions: precondition and postcondition. The postcondition must be proven for all valid executions, where the initial state fulfills the precondition. We introduce a notation for Hoare triples:

```
{* P *} p {* Q *}
```

The Hoare triple for div is defined as follows:

```
example : {* λ_,true *} div {* postcondition *}
```

The precondition is true. Hence, all (type-correct) initial states fulfill the precondition.

The standard Hoare logic above only proofs partial correctness. The postcondition must only hold for executions that terminate. Given that in the program div division by zero never terminates, the postcondition need not hold in this case. Suppose a program p is incorrectly constructed, such that it never terminates. In this case, a Hoare triple {* P *} p {* Q *} holds for all P and Q. Hence, partial correctness proofs are not useful to identify problems related to non-termination. A variation of the Hoare logic where termination has to be proven (on states where the precondition holds) is called total correctness.

```
def hoare_tc (P : state → Prop) (p : program) (Q : state →
↪  Prop) := ∀ s, P s → ∃ t, exec p s t ∧ Q t
```

We use square brackets to denote total correctness:

```
[* P *] p [* Q *]
```

To prove total correctness of div, y must not be zero on the initial state.

```
example : [* λs, s "y" ≠ 0 *] div [* postcondition *]
```

Proving a Hoare triple can be tedious. A fruitful approach is to break down the proof based on the structure of the program. To that end, we define a set of inference rules.

```
theorem seq_intro (P R Q p₁ p₂) : {* P *} p₁ {* Q *} →
{* Q *} p₂ {* R *} → {* P *} (p₁ ;; p₂) {* R *}

theorem skip_intro (P) : {* P *} skip {* P *}

theorem compute_intro (P f) :
{* λs, P (f s) *} compute f {* P *}
```

The sequence rule enables the introduction of intermediate assertions. Given a program consisting of (at least) two instructions, an assertion can be placed "between" the two instructions, serving as the precondition and postcondition for the resulting Hoare triples respectively. This way, a sequence can be broken down into the individual instructions. The intermediate assertions are usually determined by the surrounding instructions. An example to prove a single instruction is `skip_intro`. It requires that precondition and postcondition are equal.

2.3.3 Relational Hoare Logic

Relational Hoare logic (RHL) is a variation of Hoare logic that allows the comparison of the behavior of two programs. It extends Hoare triples to Hoare quadruples by adding an extra program.

```
{* P *} p₁ ~ p₂ {* Q *}
```

The programs may be completely independent but often fulfill the same purpose. The assertions work on the states of both programs and can relate them as well as constrain the individual states. Formally, we define RHL for `while`-programs:

```
def rhl (P Q p₁ p₂) : ∀ s₁ s₂, P s₁ s₂ →
(∀ t₁, exec p₁ s₁ t₁ → ∃ t₂, exec p₂ s₂ t₂ ∧ Q t₁ t₂) ∧
(∀ t₂, exec p₂ s₂ t₂ → ∃ t₁, exec p₁ s₁ t₁ ∧ Q t₁ t₂)
```

This definition is an adaption of Nick Benton's RHL semantics (based on big-step semantics instead of denotational semantics) [20]. If two programs can be related under the given assertion they co-terminate. That is, given two initial states are related by the precondition, if any of the two programs terminates, the other one has to terminate as well.

RHL is a convenient basis for reasoning about program transformations. We, therefore, relate a program with a refined version of the same program. Following the stepwise refinement of MCL programs, it is common to change a program for better performance. The intention is to do this in an input-output (I/O) preserving manner, i.e. given the same input, both programs should yield the same output (or both not terminate). The assertion, in this case, is equality for both precondition and postcondition. However, RHL is flexible enough to express more advanced relations. For example, a variable can be renamed from `a` in p_1 to `b` in p_2 (e.g. for readability) using the following condition:

```
def rename_eq : λ s₁ s₂, s₁ "a" = s₂ "b" ∧
∀ name, name ≠ "a" ∧ name ≠ "b" → s₁ name = s₂ name
```

We define inference rules for RHL. For an initial set of rules, the inference rules for non-relational Hoare logic can be adapted to two programs.

```
theorem compute_intro (f₁ f₂) :
{* λs₁ s₂, P (f₁ s₁) (f₂ s₂) *} compute f₁ ~ compute f₂ {* P *}

theorem skip_intro : {* P *} skip ~ skip {* P *}
```

We call them double-sided instruction rules. They match the constructor of `while`-programs in the most generic way. For simplicity we only show the rules for two constructors. `compute_intro` does not require that both programs perform the same computation, which already allows for some useful proofs. However, the rules require that the programs are structurally equal. This limitation can be overcome by single-sided instruction rules.

```
theorem compute_left (f) :
{* λ s₁ s₂, P (f s₁) s₂ *} compute f ~ skip {* P *}
```

We call this rule *single-sided*, because it has a skip on one side. It is used to reason about the left program without regarding the right program (or vice versa). If the right side is not a skip instruction, it can be injected without changing the assertions.

```
lemma skip_right : {* P *} p₁ ~ p₂ {* Q *} ↔ {* P *} p₁ ~ skip
 ↪  ;; p₂ {* Q *}
```

Moreover, there are transformation rules, where at least one side is a more complex program. These are a models of transformations in form of inference rules. For example, the known branch rule:

```
theorem known_branch :
{* P *} p₁ ~ p₂ {* Q *} →
{* λ s₁ s₂, P s₁ s₂ ∧ c s₁*} ite c p₁ p₁' ~ p₂ {* Q *}
```

We can change precondition and postcondition if one is the consequence of the other (in different directions).

```
lemma consequence (h : {* P *} p {* Q *}) (hp : ∀s, P' s → P s)
(hq : ∀s, Q s → Q' s) : {* P' *} p {* Q' *}
```

This is useful for rewriting an assertion to the structure required by the inference rules, e.g. the precondition of `compute_intro`.

There are multiple approaches to prove Hoare quadruples, which have different trade-offs. Using `consequence` and instruction rules, both programs can be decomposed individually (until no Hoare quadruple proof obligation remains). The relationship of the two programs, as stated in the assertions, has to be proven on the remaining proof obligations on states. Another approach is to use transformation rules to reshape the structure to simplify the remaining proof. We consider an example where p_2 is an arbitrary complex programs:

```
example : {* eq *}
    compute (λs, s.update "c" 1) ;;
    ite (λs, s "c" = 1) (
        compute (λs, s.update "a" 7)
    ) p₂ ~>
    compute (λs, s.update "a" 7) {* eq *}
```

Using instruction rules, the intermediate assertions grow with an increased complexity of p_2. With the help of the transformation rule `known_branch`, reasoning of p_2 is not required. We do not provide a thorough comparison of the approaches for `while`-programs here. However, in Chapter 4 we use different approaches to solve relational proofs of Parlang and MCL programs.

Chapter 3

Design

3.1 Layered Architecture

Hoare logic is commonly applied to `while`-programs that are purposely designed to be simple, with only a few constructors. Many of the modern language features can easily be transformed into `while`-programs without having to change the semantics. For example, while loops, for loops, and do-while loops can all be expressed as while loops without changing the I/O behavior of the program. Generally speaking, many syntactical elements are auxiliary definitions for convenience, which do not change the expressiveness of that language. This allows the study of programs written in complex languages while reasoning with the primitive building blocks. This type of abstraction helps to concentrate on the key properties of a model without the complexity that comes with a practical language.

Our first modeling approach was heavily driven by the syntax and semantics of MCL. The model covered structural elements (such as loops and branching) as well as expressions of different types (e.g. addition and multiplication on integers and floats). Furthermore, the model was designed to be type-safe by definition. That is, programs were only syntactically valid if all variables are type correct, e.g. an integer variable cannot be assigned a float value. The benefit is that the model is fairly comprehensible and a good approximation of MCL. However, we found that this model was too complex in the beginning and too much time went into considerations of small details. Therefore, we have broken down the model into two layers.

We first designed an abstract language called Parlang (Section 3.3), which is comparable to `while`-programs, however, targeted towards GPU programs. It resembles the characteristics of GPU architectures but it tries to be language independent. We constructed an operational semantics for Parlang on which we defined a relational Hoare logic. Although the language is abstract, we

can prove some inference rules for the RHL. On top of Parlang, we construct MCL (Section 3.5) by defining it in terms of Parlang. A benefit of the layered approach is that we can lift many lemmas and definitions including RHL.

Another benefit of the layered architecture is, that Parlang can be used as a basis for modeling other languages, such as CUDA. This saves overhead in the verification of a different language.

3.2 Memory Model

We are mostly interested in the input-output behavior of a program. Therefore, a crucial part of our design is how we store data. We generalize the storage model such that it can be used for different types of memory.

On the hardware level, memory is typically accessed using numeric addresses, i.e. pointers. Languages, such as MCL, allow access only through variable names. We do not restrict the memory model to a particular address type. Instead, we use a type variable (conventionally named ι) to abstract over the details of the address type. To perform equality checks on addresses, we have to assume that ι has decidable equality. If two addresses are not equal, they are distinct and operations on the memory cells do not interfere with each other. Hence, ι determines the granularity with which we access the memory – an aspect that will become important for shared memory. Typically, individual bits of an integer cannot be directly addressed while elements of an array can be addressed.

Every element (i.e. every address) in the memory can have an arbitrary but fixed Lean type. Therefore, the type depends on the address. We use the type variable τ to map from ι to that type. Finally, we define the memory:

```
def memory (τ : ι → Type) := ∀ (i : ι), τ i
```

We define two functions to get and update values respectively:

```
def memory.get (m : memory τ) (i : ι) : τ i := m i

def memory.update (m : memory τ) (i : ι) (v : τ i) : memory τ
:= function.update m i v
```

Furthermore, we define a few lemmas to eliminate `update` and `get` in terms.

```
lemma get_update_success : get (update m i val) i = val

lemma get_update_skip (h : i' ≠ i) :
get (update m i val) i' = get m i'
```

3.3 Parlang

Parlang generalizes the models of GPU programs and simplifies their construction. This goes beyond the capabilities of the `while`-programs, that are designed to model single-threaded programs. Although multiple threads can be modeled as multiple independent executions of `while`-programs, the language misses the primitive to synchronize threads and exchange data. Therefore, Parlang is fundamentally different from `while`-programs.

The Parlang semantics is limited to one thread group. Hence, synchronization occurs between all threads. In turn, the relational Hoare logic on Parlang is designed to relate a thread group to another thread group. The verification of splitting up or merging thread groups is out of reach of Parlang. Section 3.4 explains the verification of multiple independent thread groups.

The semantics of Parlang should primarily distinguish two behaviors:

- **Unique resulting state:** The semantics has exactly one deterministic result for an input state.
- **Faulty behavior:** The execution fails due to non-termination, barrier divergence or interference on an input state (cf. Section 2.1.2).

We consider all undefined behavior to be faulty. Interference does not necessarily cause a kernel to crash and might even give the intended output depending on the interleaving of the threads. However, a single successful execution on a particular GPU is no indication for a properly written program, i.e. the actual program behavior is nondeterministic. Because we consider all cases where thread interleaving causes different results invalid, Parlang is a deterministic language.

The Parlang semantics is inspired by the *synchronous, delayed visibility (SDV)* semantics [2]. It is based on lockstep execution, i.e. it advances instruction by instruction for all threads. To capture non-interference, all reads and writes between two barriers are recorded and analyzed for possible interference.

On the virtual layer, a kernel has access to three types of memory: global, shared and thread-local. For simplicity, we limit ourselves to shared and thread-local memory. If a kernel has allocated global memory, it is treated as shared memory. It follows that the kernel arguments also reside in shared memory. Because Parlang only works with a single thread group, the scope of global and shared memory is equal – all threads can access them.

Parlang programs are build from two types: `kernel` and `program`. The `kernel` is the code that is executed on the GPU. A `program` wraps the kernel and defines the number of threads.

3.3.1 State

Each thread is given a unique thread identifier. Conventionally, it is abbreviated as `tid`. Given that the number of threads is denoted by `n`, a thread identifier is of type `fin n`, i.e. a natural number that is smaller than `n`. If the upper boundary is irrelevant, we sometimes use the type ℕ instead. Lean provides an automatic coersion between `fin n` and ℕ. On multiple occasions, we use lists and vectors to maintain state per thread. We use the function `nth` of those datatypes to map from the thread identifier to the corresponding elements. Thread `i` refers to the thread with the thread identifier `i`.

Each thread has a state that changes during execution. We aggregate this information in a structure called `thread_state`.

```
structure thread_state {ι : Type} (σ : Type) (τ : ι → Type) :=
(tlocal : σ)
(shared : memory τ)
(loads  : set ι := ∅)
(stores : set ι := ∅)
```

The field `tlocal` is the thread-local memory, that is only accessible by the corresponding thread. We do not reason about the structure of `tlocal` in Parlang. This is dependent on the concrete language, and can, for example, be a map or an array-like structure. We abstract over the type of `tlocal`, which we conventionally name σ (not related to Σ). The second field is a copy of the shared memory, which is called shadow memory. It is the copy on which the thread performs all its loads and stores on. We encapsulate those operations in two functions, which are defined on `thread_state`.

```
def thread_state.store (f : σ → (Σi:ι, τ i))
(ts : thread_state σ τ) : thread_state σ τ :=
let ⟨i, v⟩ := f ts.tlocal in
{ shared := t.shared.update i v,
  stores := insert i t.stores,
  .. t}

def thread_state.load (f : σ → (Σi:ι, (τ i → σ)))
(ts : thread_state σ τ) : thread_state σ τ :=
let ⟨i, tr⟩ := f ts.tlocal in
{ tlocal := tr (t.shared.get i),
  loads := insert i t.loads,
  .. t }
```

To compute the address to store at, and the value to store, the thread-local memory is provided as an argument in `f`. The return value of `f` is a Σ-type where the type of the stored value is dependent on its address (based on the type-map τ). Similar to `store`, for `load`, the address may be based on the thread-local memory and the fetched value is provided as an argument to a

nested function to compute the new thread-local state. The **let** statement is used to decompose the Σ-type in f into its two fields. Apart from moving values between shared and thread-local memory, we keep track of the addresses that have been loaded and stored respectively between barriers. These are formalized as sets loads and stores over ι, that are empty sets by default. We define an access to be either a load or a store.

```
def thread_state.accesses (t : thread_state σ τ) : set ι
:= t.stores ∪ t.loads
```

Furthermore, we define compute to change the thread-local state.

```
def thread_state.compute (f : σ → σ)
(t : thread_state σ τ) : thread_state σ τ :=
{ tlocal := f t.tlocal, .. t}
```

We accumulate the thread states of all threads in state, abstracting over the number of threads.

```
structure state {ι : Type} (n : ℕ) (σ : Type) (τ : ι → Type) :=
(threads : vector (thread_state σ τ) n)
```

3.3.2 Synchronization on State

Non-interference is required to pass a barrier. We formalize this condition on state:

```
def syncable (s : state n σ τ) (m : memory τ) : Prop := ∀i:ι,
(∀ tid,
  i ∉ (s.threads.nth tid).stores ∧
  m i = (s.threads.nth tid).shared i)
∨
(∃ tid,
  i ∈ (s.threads.nth tid).stores ∧
  m i = (s.threads.nth tid).shared i ∧
  (∀ tid', tid ≠ tid' → i ∉ (s.threads.nth tid').accesses))
```

The definition serves two purposes. Having an instance of syncable guarantees non-interference on the state. It also guarantees that the memory argument reflects all and only the changes that have been applied to the shadow memories. We say that state s synchronizes to memory m. To prove an instance of syncable, we consider two possible cases for every addresses i. The first case is that no thread stored at i and the values at i are equal between all threads. We can choose any thread to get the value at i for the resulting memory. The second case is that exactly one thread stored at i and this thread determines the value at i for the resulting memory. All other threads must not have accessed i in this case. We consider benign interference to be interference [2]. If

two threads store the *same* value at the *same* address the state is not syncable. Although those programs show deterministic and valid behavior on most platforms, for simplicity, we do not consider them valid. Allowing benign interference can be implemented using a variation of `syncable`. However, we have not analyzed the implications on the dependent proofs.

Whenever the program passes a barrier, the thread state must reflect this fact. This is encapsulated in `sync`.

```
def sync (g : memory τ) (t : thread_state σ τ) :
thread_state σ τ :=
{ shared := g,
  loads := ∅,
  stores := ∅,
  .. t}
```

The update to the shadow memory makes the changes of all other threads visible to this thread. Furthermore, it removes all accesses that happened before the barrier. Note that sync should only be used in combination with `syncable` as defined in the semantics.

3.3.3 Active Map

During execution, we need to keep track of which threads are active. We conventionally use a vector of booleans over the number of threads `n`, named `ac` (abbreviated from active map). We define the boolean value `tt` to denote an active thread and `ff` to denote an inactive thread. We say some kernel `k` is executed on active map `ac`, meaning for every thread `i` the kernel `k` only changes the state if `ac.nth i` is `tt`. Because we keep track of the activeness of all threads we can decide whether all, none, or any threads are active based on the active map (without the state):

```
def no_thread_active (ac : vector bool n) : bool :=
 ↪  ¬ac.to_list.any id
def any_thread_active (ac : vector bool n) : bool :=
 ↪  ac.to_list.any id
def all_threads_active (ac : vector bool n) : bool :=
 ↪  ac.to_list.all id
```

Furthermore, we define two active maps to be distinct as

```
def ac_distinct (ac₁ ac₂ : vector bool n) : Prop :=
∀ i : fin n, ac₁.nth i = ff ∨ ac₂.nth i = ff
```

Threads can be deactivated based on a condition, which is evaluated on the thread-local state.

```
def deactivate_threads (f : σ → bool) (ac : vector bool n)
(s : state n σ τ) : vector bool n :=
ac.map₂
  (λ a (ts : thread_state σ τ), (bnot ∘ f) ts.tlocal && a)
  s.threads
```

The function `map₂` combines two vectors of different types element-wise and returns a vector of a third type (in this case boolean again). `a` is the current activeness of a thread and `ts` is the corresponding `thread_state`. Deactivation of threads is commutative:

```
lemma ac_deac_comm :
  deactivate_threads f (deactivate_threads f' ac s) t =
  deactivate_threads f' (deactivate_threads f ac t) s
```

We define a partial order on active maps of the same size.

```
def ac_ge (ac' : vector bool n) (ac : vector bool n) : Prop :=
∀ (t : fin n), ¬ (ac.nth t) → ¬ (ac'.nth t)
```

By deactivating threads we obtain a smaller (or equal) active map. The order is transitive.

Given nested applications of `deactivate_threads`, if we can prove an order on them, we can remove the smaller one.

```
lemma ac_deac_ge
(h : deactivate_threads f ac s ≥ deactivate_threads f' ac t) :
  deactivate_threads f' (deactivate_threads f ac s) t =
  deactivate_threads f' ac t
```

We keep the active map separate from the state. This makes the definition of the semantics more comprehensible. We consider a simple example, where `k₁`-`k₄` are arbitrary code blocks (cf. Figure 3.1). We first consider how the state behaves in an informal way (the formalization follows later). Without any assumptions, the state after executing k_n is named s_n and the initial state is denoted by s . The condition c is evaluated on state s and k_n is evaluated on state s_{n-1}. Observe that the state evolves as a chain where each state s_n (only) depends on state s_{n-1}. We compare this behavior to the active map. Suppose k is executed on ac , and k and k_3 are executed on ac and ac'. It follows that k_4 is executed on ac , which is completely independent of ac and ac'. In fact, ac' does not contain enough information to recover ac . If a thread is inactive in ac', it is either because the condition c did not hold or the thread was already inactive in ac . Hence, active maps do not follow a chain. Instead, the active map only ever changes when descending into subterms. Therefore, we keep the active map separate from the state.

Given a state and an active map, we can define a function that changes the thread states of active threads:

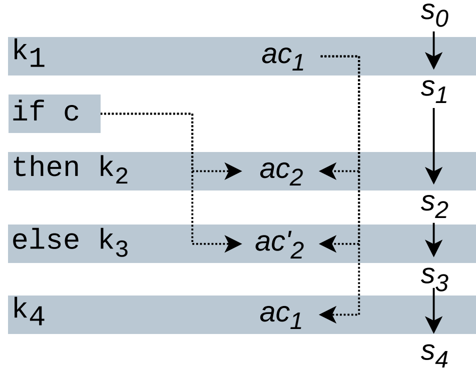

Figure 3.1: Advancement of active map and state during branching. The dotted lines denote the influence on active maps. Threads that are inactive in ac_1 remain inactive in ac_2 and ac_2'. Furthermore, threads are deactivated by evaluating the condition `c` on each thread.

```
def map_active_threads (ac : vector bool n)
(f : thread_state σ τ → thread_state σ τ) (s : state n σ τ) :
state n σ τ := { threads := (
    s.threads.map₂ (λ t (a : bool), if a then f t else t) ac
), ..s }
```

3.3.4 Kernels

We define the type `kernel` as follows:

```
inductive kernel {ι : Type} (σ : Type) (τ : ι → Type) : Type
| load       : (σ → (Σi:ι, (τ i → σ))) → kernel
| store      : (σ → (Σi:ι, τ i)) → kernel
| compute {} : (σ → σ) → kernel
| seq        : kernel → kernel → kernel
| ite        : (σ → bool) → kernel → kernel → kernel
| loop       : (σ → bool) → kernel → kernel
| sync {}    : kernel
```

The thread-local memory type σ, address type ι and type-map τ are fixed for the kernel. The empty curly braces in the compute and sync constructors instruct Lean to infer τ and ι from the context, rather than from their arguments. All non-recursive constructor arguments are Lean functions. This shallow-embedding gives Parlang the flexibility it was designed for. Concrete languages can later narrow down the type of those arguments. The meaning of the arguments is formalized by the operational semantics. However, we introduce them briefly and informally here – especially those that are different from

`while`-programs. The `compute` instruction modifies the thread-local memory, without any restriction. The `ite` instruction stands for if-then-else and is used for branching. The first argument is the condition. The function is used to decide whether an active thread executes the if or else branch, on a per-thread basis. The decision is based only on the thread-local memory. If shared variables are required in the condition, they must first be loaded. The `loop` constructor is similar. The instructions `store` and `load` are used to interact with the shared memory and have the same arguments as their respective functions `store` and `load` on `thread_state` (cf. Section 3.3.1).

For readability, we introduce the infix operator `;;` for sequential composition of kernels.

3.3.5 Semantics of Kernels and States

We define an operational big-step semantics for kernels over a number of threads, state, and the active map:

```
inductive exec_state {n : ℕ} :
kernel σ τ → vector bool n → state n σ τ → state n σ τ → Prop
| load (f) (s : state n σ τ) (ac : vector bool n) :
    exec_state (load f) ac s
    (s.map_active_threads ac $ thread_state.load f)
| store (f) (s : state n σ τ) (ac : vector bool n) :
    exec_state (store f) ac s
    (s.map_active_threads ac $ thread_state.store f)
| compute (f : σ → σ) (s : state n σ τ) (ac : vector bool n) :
    exec_state (compute f) ac s
    (s.map_active_threads ac $ thread_state.compute f)
| sync_all (s : state n σ τ)
    (ac : vector bool n) (m : memory τ)
    (hs : s.syncable m) (ha : all_threads_active ac) :
    exec_state sync ac s (s.map_threads $ thread_state.sync m)
| sync_none (s : state n σ τ) (ac : vector bool n)
    (h : no_thread_active ac) :
    exec_state sync ac s s
| seq (s t u : state n σ τ) (ac : vector bool n)
    (k₁ k₂ : kernel σ τ) :
    exec_state k₁ ac s t →
    exec_state k₂ ac t u →
    exec_state (seq k₁ k₂) ac s u
| ite (s t u : state n σ τ) (ac : vector bool n)
    (f : σ → bool) (k₁ k₂ : kernel σ τ) :
    exec_state k₁ (deactivate_threads (bnot ∘ f) ac s) s t →
    exec_state k₂ (deactivate_threads f ac s) t u →
    exec_state (ite f k₁ k₂) ac s u
| loop_stop (s : state n σ τ) (ac : vector bool n)
    (f : σ → bool) (k : kernel σ τ) :
    no_thread_active (deactivate_threads (bnot ∘ f) ac s) →
    exec_state (loop f k) ac s s
| loop_step (s t u : state n σ τ) (ac : vector bool n)
    (f : σ → bool) (k : kernel σ τ) :
    any_thread_active (deactivate_threads (bnot ∘ f) ac s) →
    exec_state k (deactivate_threads (bnot ∘ f) ac s) s t →
    exec_state (loop f k)
        (deactivate_threads (bnot ∘ f) ac s) t u →
    exec_state (loop f k) ac s u
```

Listing 1: Parlang kernel semantics

The definition is an inductive proposition, hence it is non-computable. Given that we have an instance of type `exec_state k ac` s_1 s_2, we have a proof that executing kernel `k` on the initial state s_1 and active map `ac` results in state s_2. We explain the cases by constructors of `kernel`. All `kernel` constructors, except `loop` and `sync`, match exactly one case. To derive a valid `load`-execution we have to put the resulting state in relation with the initial

state. More precisely, we have to show that a load is performed on all active threads. The cases for `store` and `compute` are similar. To derive a valid sequential composition execution from state `s` to `u`, we have to prove validity on the individual kernels. For that, kernel k_1 must execute from `s` to some state `t` and k_2 from state `t` to state `u`. In a sequential composition, both kernels are executed on the same threads.

The semantics for `ite` might appear counterintuitive at first, because there is only one case. Although every thread either executes the if- or the else-branch, in a set of threads some threads execute the if-branch while others execute the else-branch. To derive a valid execution, we have to prove the validity of the two branches similar to sequential composition. However, only a subset of active threads perform each branch. Therefore, we deactivate threads based on the condition evaluated over the initial state. Here, we see the benefit of keeping the active map separate from the state – there is no need to formalize reactivation of threads after branching. The if branch is arbitrarily executed before the else branch. We can show that we get an equivalent semantics if we change the order. On another note, graphics cards are optimized to skip branches if no thread is active. We ignore this optimization because it is hidden from programmer and I/O preserving. However, the semantics is designed such that, given $s_1 \neq s_2$ and `no_thread_active ac`, we cannot derive `exec_state k ac s1 s2` for any `k`. If a branch is executed on with all threads being inactive, we call this an inactive branch.

For the loop-constructor we require two cases: stop and step. Without any assumptions, we can only show validity if we perform all loop iterations. We define this inductively. A loop terminates on some active map if all threads are inactive after evaluating the loop condition (on each thread individually). This is the base case `loop_stop`, where 0 iterations are performed and we identify the initial and resulting state. The case `loop_step` adds an additional iteration "in front" of the existing iterations. We have to prove the validity of the loop body after deactivating threads on the condition. Furthermore, we have to show the validity of the remaining iterations. As expected, the resulting state of iteration `i` is the initial state of iteration `i + 1`. Note that with every iteration the active map is smaller (or equal) according to our definition. This is by definition and is called monotonic loop [21]. In principle, GPUs allow non-monotonic loops; however, they always cause a barrier divergence or interference according to our semantics. In Section 3.3.8 we present a variation of the semantics with non-monotonic loops and prove it to be equivalent to `exec_state`.

There are two valid cases for `sync`: either no or all threads are active.
`sync_none` is required to derive validity if a `sync` instruction is part of an inactive branch (in which case the initial state is equal to the resulting state). In case all threads are active, we have to show non-interference by proving `syncable`. The resulting state must reflect the synchronized memory. If at least one, but not all threads are active, a barrier divergence has been reached and there is no valid derivation.

3.3.6 Programs and Programs Semantics

The kernel is the composable part of a GPU program. What is missing is how many threads run this kernel. Therefore, we extend the type kernel to the type program.

```
inductive program {ι : Type} (σ : Type) (τ : ι → Type)
| intro (f : memory τ → ℕ) (k : kernel σ τ) : program
```

We have an additional field `f` that denotes the number of threads depending on the initial memory. Programs only operate on `memory` instead of `state`. This is in accordance with MCL. The host sets up the initial memory and starts the MCL program. Other languages follow a similar principle.

We extend the kernel semantics to programs:

```
def init_state (init : ℕ → σ) (f : memory τ → ℕ) (m : memory τ)
 ↪  :
state (f m) σ τ := { threads :=
    (vector.range (f m)).map
    (λ n, { tlocal := init n, shared := m })
}

inductive exec_prog : (ℕ → σ) → program σ τ → memory τ →
memory τ → Prop
| intro (k : kernel σ τ) (f : memory τ → ℕ) (a b : memory τ)
    (init : ℕ → σ) (s' : state (f a) σ τ)
    (hsync : s'.syncable b)
    (he : exec_state k (vector.repeat tt (f a))
        (init_state init f a) s') :
    exec_prog init (program.intro f k) a b
```

`init_state` is responsible for initializing the state. The initialization of the thread-local memory is abstracted in the function `init`, which takes a thread identifier and return a thread-local state. Initially, all threads are active. To show a valid execution of a program, a valid execution of its kernel is required. Additionally, we have to show that the resulting state is syncable to the resulting memory. This is to ensure that accesses after the last `sync` do not interfere and to retrieve the updated values from the shadow memories.

3.3.7 Properties of Parlang

No skip instruction A skip instruction is not explicitly defined. Instead, `(kernel.compute id)` can be used. We can show that the above has the behavior of skip:

```
example {n} {ac : vector bool n} {s} :
exec_state (kernel.compute id : kernel σ τ) ac s s
```

Not having an explicit skip instruction saves a proof obligation when doing induction on a kernel or the semantics.

`syncable` is deterministic if we consider at least one thread. That is, given any state, every valid derivation has the same resulting memory.

```
lemma syncable_unique {s : state n σ τ} {m m'}
(h₁ : syncable s m) (h₂ : syncable s m') (hl : 0 < n) : m = m'
```

`exec_state` is deterministic. That is, given any kernel, active map, and initial state, every valid derivation has the same resulting state.

```
lemma exec_state_unique {s u t : state n σ τ}
{ac : vector bool n} {k} (h₁ : exec_state k ac s u)
(h₂ : exec_state k ac s t) : t = u
```

3.3.8 Equivalence to Non-Monotonic Semantics

The Parlang semantics restricts itself to monotonic loops. That is, if a thread is inactive in some iteration `n` it must also be inactive in any iteration `n + m`, where `n` and `m` are natural numbers. In that regard, the semantics is not an accurate model of the GPU architecture. Given that a thread was active when the kernel enters the loop, its condition is checked in every iteration. If the condition depends on a shared variable, a thread has the ability to enable another thread. However, it can be shown that this causes a barrier divergence or interference – neither are valid according to Parlang.

To carry out the proof, we define a semantics that allows non-monotonic loops.

```
inductive exec_nonmono {n : ℕ} : kernel σ τ → vector bool n →
state n σ τ → state n σ τ → Prop
/-
...
same constructors as exec_state
-/
| loop_step (s t u : state n σ τ) (ac : vector bool n)
    (f : σ → bool) (k : kernel σ τ) :
    any_thread_active (deactivate_threads (bnot ∘ f) ac s) →
    exec_nonmono k (deactivate_threads (bnot ∘ f) ac s) s t →
    /- the only difference to parlang is the line below:
       here we don't deactivate threads -/
    exec_nonmono (loop f k) ac t u →
    exec_nonmono (loop f k) ac s u
```

The difference lies only in the `loop_step` case, where the condition is evaluated on the initial active map (before loop entry) for every iteration. The new semantics holds if and only if the original semantics holds.

```
lemma eq_parlang_nonmono :
exec_nonmono k ac s s' ↔ exec_state k ac s s'
```

The double implication is proven separately from left to right and right to left. Induction is performed on the respective assumptions resulting in a proof obligation for each constructor of the semantics. All constructors except for `loop_step` are trivial because they are equivalent. For the proof of `loop_step`, we provide some intuition. The full proof is available in Lean.

Given that the loop body is denoted by `k` and the condition by `f`, a valid loop iteration is expressed as
`<semantics> k (deactivate_threads (bnot o f) ac s) s t` in either semantics. Based on this statement, we relate `s` and `t` by their influence on `ac`.

```
example (k f) :
<semantics> k (deactivate_threads (bnot ∘ f) ac s) s t →
deactivate_threads (bnot ∘ f) ac s ≥
deactivate_threads (bnot ∘ f) ac t
```

To prove the above, we consider an arbitrary thread `tid` that is inactive at `deactivate_threads (bnot o f) ac s`. Because an inactive thread never changes its state, we show that the state of the thread `tid` is equivalent in `s` and `t`. We conclude that also the evaluation of the condition is the same on either state. Hence thread `tid` is inactive at
`deactivate_threads (bnot o f) ac t`.

Suppose the initial state is `s` and the resulting state is `u`. In both directions we have the assumption that at least one thread is active at
`deactivate_threads (bnot o f) ac s`, because we are working on the constructor `loop_step`. Furthermore, as induction hypothesis, we have valid executions of the first loop iteration from `s` to `t` and for the remaining iterations from `t` to `u`. We have to show that the active map after the first iteration corresponds with the semantics.

Left to right

```
lemma exec_ac_to_deac.aux {k}
(ha : any_thread_active (deactivate_threads (bnot ∘ f) ac s))
(hi : exec_state k (deactivate_threads (bnot ∘ f) ac s) s t)
(h : exec_state (loop f k) ac t u) :
exec_state (loop f k) (deactivate_threads (bnot ∘ f) ac s) t u
```

When we compare the goal with the assumption `h`, the active map `ac` of the goal undergoes a deactivation of threads based on state `s`. We do an induction on `h` and apply the corresponding constructors of `exec_state` on the goal. One can observe that all occurrences of `ac` undergo a deactivation of threads based on state `t`. For our goal this means, we have nested applications of

deactivate_threads. Using the lemmas defined in Section 3.3.3 and hi, we derive that the most recent deactivation is stronger and remove the weaker one. We can close the remaining goals with the hypotheses resulting from the induction on h.

Right to left

```
lemma exec_deac_to_ac.aux {k}
(ha : any_thread_active (deactivate_threads (bnot ∘ f) ac s))
(hi : exec_nonmono k (deactivate_threads (bnot ∘ f) ac s) s t)
(h : exec_nonmono (loop f k) (deactivate_threads (bnot ∘ f) ac
↪  s) t u) :
exec_nonmono (loop f k) ac t u
```

The proof follows the same principles as left to right, except that the nested deactivation of threads occurs in the hypotheses instead of the goal.

3.4 Relational Hoare Logic for Parlang

To prove the correctness of program transformations, we define a relational Hoare logic on Parlang, as well as some inference rules. As opposed to RHL on `while`-programs, the logic on Parlang is asymmetric.

There are multiple ways to treat faulty behavior of programs in relational Hoare logic. Most papers restrict the discussion to non-termination because it is the only option how a program may fail in their case (e.g. see [20]). We include interference and barrier divergence but only distinguish between faulty and non-faulty behavior. A fault is either non-termination, interference or a barrier divergence. Nick Benton's RHL requires that the two programs co-terminate [20]. That is, part of the proof obligation is to show that if one program terminates, the other one has to terminate as well. This can be observed by looking at the definition of RHL on `while`-programs in Section 2.3.3. The proof requires a witness for the resulting state of the execution of p_2 on s_2 if p_1 has a valid execution on s_1 (and vice versa). Termination has to be proven of both programs. A different approach is to abandon co-termination (e.g. see [22]). It only has to be shown that the postcondition holds if both programs terminate. However, this makes it possible to introduce data races or barriers divergence in the transformed program, which would reduce the expressiveness of our logic.

Both approaches that are described above are symmetric, i.e. swapping the arguments of the assertions and the programs preserve validity. In the context of stepwise refinement, we can drop this requirement and choose an asymmetric approach. We relate two versions of the same program: before and after the optimization. We always place the version before the optimization on the left side. Because the correctness is concerned with the optimization (i.e. the difference between the two programs), we assume that the left program is non-faulty for all states where the precondition holds. If the program makes assumptions about the input data, it can be expressed in the precondition. Termination has to be shown of the right (optimized) program (for all states that satisfy the precondition) by providing the resulting state. Compared to co-termination, only termination of the right program has to be shown as opposed to both sides. Hence, this approach comes with fewer proof obligations.

Parlang introduces two operational semantics, for kernels and programs respectively. It follows that we also define two relational Hoare logics, first on programs:

```
def rel_hoare_program (init₁ : ℕ → σ₁) (init₂ : ℕ → σ₂)
(P : memory τ₁ → memory τ₂ → Prop) (p₁ : program σ₁ τ₁)
(p₂ : program σ₂ τ₂) (Q : memory τ₁ → memory τ₂ → Prop) :=
∀ m₁ m₁' m₂, P m₁ m₂ → exec_prog init₁ p₁ m₁ m₁' →
∃ m₂', exec_prog init₂ p₂ m₂ m₂' ∧ Q m₁' m₂'
```

The definition uses an existential quantifier. Namely, to prove that p_2 is non-faulty, the resulting state is required as a witness. Because the semantics of

Parlang programs is deterministic, the witness is unique if it exists. The assertions are relations on the shared memory. The two programs may use different `init` functions, to initialize the thread-local memory.

The RHL on kernels uses a different semantics and other types of assertions:

```
def rhl_kernel_assertion := ∀ n₁:ℕ, state n₁ σ₁ τ₁ →
vector bool n₁ → ∀ n₂:ℕ, state n₂ σ₂ τ₂ → vector bool n₂ → Prop

def rel_hoare_state (P : rhl_kernel_assertion)
(k₁ : kernel σ₁ τ₁) (k₂ : kernel σ₂ τ₂)
(Q : rhl_kernel_assertion) : Prop :=
∀ (n₁ n₂ : ℕ) (s₁ s₁' : state n₁ σ₁ τ₁)
(s₂ : state n₂ σ₂ τ₂) ac₁ ac₂,
P n₁ s₁ ac₁ n₂ s₂ ac₂ → exec_state k₁ ac₁ s₁ s₁' →
∃ s₂', exec_state k₂ ac₂ s₂ s₂' ∧ Q n₁ s₁' ac₁ n₂ s₂' ac₂
```

The assertions can relate the states of all threads, as well as the number of threads and their activeness. To emphasize the asymmetry, we use an asymmetric syntax version:

```
{* P *} k₁ ~> k₂ {* Q *}
```

Following the relation of `exec_state` and `exec_program`, the two relational Hoare logics can be related:

```
def initial_kernel_assertion (init₁ : ℕ → σ₁) (init₂ : ℕ → σ₂)
(P : memory τ₁ → memory τ₂ → Prop) (f₁ : memory τ₁ → ℕ)
(f₂ : memory τ₂ → ℕ) (m₁ : memory τ₁) (m₂ : memory τ₂)
(n₁) (s₁ : state n₁ σ₁ τ₁) (ac₁ : vector bool n₁) (n₂)
(s₂ : state n₂ σ₂ τ₂) (ac₂ : vector bool n₂) :=
s₁.syncable m₁ ∧ s₂.syncable m₂ ∧ n₁ = f₁ m₁ ∧ n₂ = f₂ m₂ ∧
(∀ i : fin n₁,
    s₁.threads.nth i = { tlocal := init₁ i, shared := m₁ }) ∧
(∀ i : fin n₂,
    s₂.threads.nth i = { tlocal := init₂ i, shared := m₂ }) ∧
P m₁ m₂ ∧ all_threads_active ac₁ ∧ all_threads_active ac₂

lemma rel_kernel_to_program {k₁ : kernel σ₁ τ₁}
{k₂ : kernel σ₂ τ₂} {init₁ : ℕ → σ₁} {init₂ : ℕ → σ₂}
{P Q : memory τ₁ → memory τ₂ → Prop} {f₁ : memory τ₁ → ℕ}
{f₂ : memory τ₂ → ℕ}
(h : {* λn₁ s₁ ac₁ n₂ s₂ ac₂, ∃ m₁ m₂, initial_kernel_assertion
    init₁ init₂ P f₁ f₂ m₁ m₂ n₁ s₁ ac₁ n₂ s₂ ac₂ *} k₁ ~> k₂
    {* λ n₁ s₁ ac₁ n₂ s₂ ac₂, (∃ m₁, s₁.syncable m₁) →
    ∃ m₁ m₂, s₁.syncable m₁ ∧ s₂.syncable m₂ ∧ Q m₁ m₂ *} )
(hg : ∀ {m₁ m₂}, P m₁ m₂ → 0 < f₁ m₁) :
rel_hoare_program init₁ init₂ P (program.intro f₁ k₁)
(program.intro f₂ k₂) Q
```

This theorem can reduce a proof obligation on programs to a proof obligation on kernels. The latter can be proven using the inference rules presented in Section 3.4.2. As a benefit from the asymmetry of our Hoare logic, we have a proof that the resulting state of the left kernel is syncable (cf. postcondition of `h`). As a consequence, we only have to prove syncablity of the right kernel. We need the assumption `hg` due to how `syncable` is defined. Given that some state `s` has zero threads, `s.syncable m` holds for an arbitrary memory `m`. The hypothesis `hg` eliminates this case.

3.4.1 Chaining Relational Proofs

Stepwise refinement for optimization is an iterative process, where the developer rewrites the program in multiple steps. Suppose p_0 is the first version of a program and p_1 to `pN` are stepwise optimizations of p_0. Using relational Hoare logic, we can verify that the sum of all transformations is I/O preserving. We choose equality for the precondition and postcondition, i.e. no renaming of variables (which, strictly speaking, is not I/O preserving). p_0 and `pN` can be directly related using RHL:

```
example : rel_hoare_program  init init eq p₀ pN eq
```

This proof contains all transformations and, for that reason, grows with the number of optimization.

An alternative approach is to prove each transformation individually, e.g.

```
example : rel_hoare_program  init init eq p₂ p₃ eq
```

This gives the developer the flexibility to choose at what granularity to perform the proofs. Furthermore, automated tools can prove the transformations individually. To conclude that `pN` is I/O preserving w.r.t. p_0 from the individual proofs, we define a transitivity rule:

```
theorem trans (p₁ : program σ₁ τ₁) (p₂ : program σ₂ τ₁)
(p₃ : program σ₃ τ₁) (init₁ init₂ init₃)
(h₁ : rel_hoare_program init₁ init₂ eq p₁ p₂ eq)
(h₂ : rel_hoare_program init₂ init₃ eq p₂ p₃ eq) :
rel_hoare_program init₁ init₃ eq p₁ p₃ eq
```

Regardless of whether the optimizations are proven in one proof or using transitivity, the only program for which we assume non-faultiness is p_0. We consider this to be reasonable. For one thing, we already assume that the original program is functionally correct. Therefore, p_0 should be as simple as possible. For another thing, existing tools (e.g. [2]) may be used to prove non-faultiness and functional correctness of the original program.

3.4.2 Inference Rules

We define a set of inference rules to analyze Hoare quadruples and break them down into verification conditions. The rules are adaptions from Section 2.3.3.

Hoare quadruples can be broken down using

```
lemma seq (Q)
(h₁ : {* P *} k₁ ~> k₂ {* Q *})
(h₂ : {* Q *} k₁' ~> k₂' {* R *}) :
{* P *} (k₁ ;; k₁') ~> (k₂ ;; k₂') {* R *}
```

`seq` breaks down the kernels on both sides. To apply single-sided rules, we define variations of `seq` to relate a kernel with `skip`. For example, for a leading instruction of the left program:

```
lemma single_step_left :
{* P *} k ~> (kernel.compute id) {* R *} →
{* R *} k₁ ~> k₂ {* Q *} →
{* P *} (k ;; k₁) ~> k₂ {* Q *}
```

Using `consequence`, the precondition can be replaced by a stronger assertion and the postcondition can be replaced by a (not necessarily strictly) weaker assertion:

```
lemma consequence (h : {* P *} k₁ ~> k₂ {* Q *})
(hp : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂,
   P' n₁ s₁ ac₁ n₂ s₂ ac₂ → P n₁ s₁ ac₁ n₂ s₂ ac₂)
(hq : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂,
   Q n₁ s₁ ac₁ n₂ s₂ ac₂ → Q' n₁ s₁ ac₁ n₂ s₂ ac₂) :
{* P' *} k₁ ~> k₂ {* Q' *}
```

RHL on Parlang does not have a symmetry rule due to its asymmetry. However, the two kernels of a Hoare quadruple can be swapped under some circumstances:

```
def assertion_swap_side
(P : @rhl_kernel_assertion σ₁ σ₂ _ _ τ₁ τ₂ _ _) :=
λ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₂ s₂ ac₂ n₁ s₁ ac₁

lemma swap (h : {* P *} k₁ ~> k₂ {* Q *})
(he₁ : ∀ {n₁ s₁ ac₁ n₂ s₂ ac₂}, P n₁ s₁ ac₁ n₂ s₂ ac₂ →
  ∃ s₁', exec_state k₁ ac₁ s₁ s₁') :
{* assertion_swap_side P *} k₂ ~> k₁
{* assertion_swap_side Q *}
```

Non-faultiness of `k₁` has to be shown by providing a witness for the resulting state. `swap` can be used in combination with single-sided rules, which exists as left- and right-hand side. Using `swap`, a left-hand side rule can be derived

from the corresponding right-hand side rule. The proof obligation that `skip` is non-faulty is trivial.

We define single-sided instruction rules, with the operation on the right side. Rules that process the left program are created using `swap`.

```
theorem compute_right (f) :
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, Q n₁ s₁ ac₁ n₂
(s₂.map_active_threads ac₂ (thread_state.compute f)) ac₂ *}
kernel.compute id ~> kernel.compute f {* Q *}

theorem store_right (f : σ₂ → (Σ (i : ι₂), τ₂ i)) :
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, Q n₁ s₁ ac₁ n₂
(s₂.map_active_threads ac₂ (thread_state.store f)) ac₂ *}
kernel.compute id ~> kernel.store f {* Q *}

theorem ite_right (c : σ₂ → bool) (th) (el)
(AC : ∀ {n₂ : ℕ}, vector bool n₂ → Prop)
(h₁ : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₁ s₁ ac₁ n₂ s₂ ac₂ →
    P n₁ s₁ ac₁ n₂ s₂ (deactivate_threads (bnot ∘ c) ac₂ s₂))
(h₂ : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂ (s' : state n₂ σ₂ τ₂),
    Q n₁ s₁ ac₁ n₂ s₂ (deactivate_threads (bnot ∘ c) ac₂ s') →
    Q n₁ s₁ ac₁ n₂ s₂ ac₂)
(h₃ : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂ (s' : state n₂ σ₂ τ₂),
    Q n₁ s₁ ac₁ n₂ s₂ ac₂ →
    Q n₁ s₁ ac₁ n₂ s₂ (deactivate_threads c ac₂ s'))
(h₄ : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂ (s' : state n₂ σ₂ τ₂),
    R n₁ s₁ ac₁ n₂ s₂ (deactivate_threads c ac₂ s') →
    R n₁ s₁ ac₁ n₂ s₂ ac₂) :
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₁ s₁ ac₁ n₂ s₂ ac₂ ∧
    (s₂.active_threads ac₂).all (λts, c ts.tlocal) *}
    kernel.compute id ~> th {* Q *} →
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, Q n₁ s₁ ac₁ n₂ s₂ ac₂ ∧
    (s₂.active_threads ac₂).all (λts, bnot $ c ts.tlocal) *}
    kernel.compute id ~> el {* R *} →
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₁ s₁ ac₁ n₂ s₂ ac₂ ∧ AC ac₂ *}
    kernel.compute id ~> kernel.ite c th el
    {* λ n₁ s₁ ac₁ n₂ s₂ ac₂, R n₁ s₁ ac₁ n₂ s₂ ac₂ ∧ AC ac₂ *}

theorem while_right (c : σ₂ → bool)
(V : ∀ {n₂} (s₂ : state n₂ σ₂ τ₂) (ac₂ : vector bool n₂), ℕ)
(h₁ : ∀ {n₁ s₁ ac₁ n₂ s₂ ac₂},
    I n₁ s₁ ac₁ n₂ s₂ ac₂ →
    I n₁ s₁ ac₁ n₂ s₂ (deactivate_threads (bnot ∘ c) ac₂ s₂))
(h₂ : ∀ {n₁ s₁ ac₁ n₂ s₂ ac₂} {s : state n₂ σ₂ τ₂},
    I n₁ s₁ ac₁ n₂ s₂ (deactivate_threads (bnot ∘ c) ac₂ s) →
    I n₁ s₁ ac₁ n₂ s₂ ac₂)
(hb : ∀ n, {* λ n₁ s₁ ac₁ n₂ s₂ ac₂, I n₁ s₁ ac₁ n₂ s₂ ac₂ ∧
    (s₂.active_threads ac₂).all (λts, c ts.tlocal) ∧
    V s₂ ac₂ = n *} kernel.compute id ~> k
```

```
    {* λ n₁ s₁ ac₁ n₂ s₂ ac₂, I n₁ s₁ ac₁ n₂ s₂ ac₂ ∧
    V s₂ (deactivate_threads (bnot ∘ c) ac₂ s₂) < n *}) :
{* I *} kernel.compute id ~> kernel.loop c k {* I *}
```

The rules `compute_right` and `store_right` follow the same principle as `compute_right` for `while`-programs. However, we limit the change on the state to active threads.

Similar to `seq`, `ite_right` has an intermediary assertion between the if and else branch. This is because the threads do not all take the same branch. Assertion `P` reflects the behavior of the if branch and `R` reflects the behavior of both the if and else branch. The preconditions of the branches imply that the condition holds or does not hold on all active threads respectively. The converse does not hold: The validity of the condition does not necessarily yield information about the activeness of the thread. That is, if a thread was inactive before the branch, the condition may be either true or false.

`ite_right` requires additional assumptions h_1-h_4. Note that assertions are dependent on state and the active map. A change in either one of them can influence the validity of the assertion. In the context of a branch this means that deactivating threads based on the condition can invalidate the assertion. That is, the branches are executed on smaller (or equal) active maps than the `ite` instruction itself. For example, in order to show that `P` holds before executing `th`, `P` must be insensitive to the deactivation of threads (see h_1). Assertions on the active map can be carried over the branch using `AC`. For example, `AC` can capture that if all threads are active before branching, they are all active after branching.

The instruction rule for loops (`while_right`) requires an invariant and a variant. The invariant `I` must hold before and after every loop iteration. Similar to `ite_right`, the assertion must be insensitive to deactivation or activation of threads (h_1 and h_2). The variant ensures that the loop terminates. It is a function from the state and active map to ℕ. The variant must decrease with every loop iteration and is limited by the lower boundary 0 of type ℕ.

Transformations in larger kernels might only change a small fraction of the code. For example consider the following transformation:
`k₁ ;; k₂ ;; k₃ ~> k₁ ;; k₂' ;; k₃`. To verify the entire kernel without having to deconstruct k_1 and k_3, we define a rule to relate syntactically equal kernels.

```
lemma rhl_eq : {* λn₁ s₁ ac₁ n₂ s₂ ac₂, n₁ = n₂ ∧ ∀h : n₁ = n₂,
    s₁ = (by rw h; exact s₂) ∧ ac₁ = (by rw h; exact ac₂) *}
k₁ ~> k₁
{* λn₁ s₁ ac₁ n₂ s₂ ac₂, n₁ = n₂ ∧ ∀ h : n₁ = n₂,
    s₁ = (by rw h; exact s₂) ∧ ac₁ = (by rw h; exact ac₂) *}
```

The types of s_2 and ac_2 have to be rewritten because the number of threads, which is part of their type, is not definitionally equal.

3.4.3 Verification of Multiple Thread Groups

Due to the integration of shared and global memory into a single memory, Parlang is limited in the ability to verify multiple thread groups. Furthermore, it has no ability to verify host programs.

Multiple thread groups run in parallel and progress independently. They share a global memory. Note that in our semantics, a thread group has an observable behavior from the viewpoint of the global memory. This behavior can be analyzed in great detail. The aspect that we are interested in is how a thread group changes the memory by comparing the memory before execution and after termination, under the assumption that no other thread group manipulates the memory. Because there is no ability to synchronize on global memory, the order and timing of the updates are of no relevance in this case. Based on this view, we distinguish between two types of transformations. Either the observable behavior changes or it does not change. Our logic is only capable of verifying the latter.

We informally show that verifying multiple thread groups that *change the observable behavior* is not possible in Parlang. Consider two thread groups with one thread each running the same kernel in parallel. We call them thread 1 and thread 2. The thread group is provided as an argument to the kernel. Suppose that thread 1 writes to global memory at location `a` and thread 2 writes to global memory at location `b` (depending on the thread group id). We change the kernel, such that it now writes to location `a` regardless of the thread group id. Note that, according to Parlang, a single thread in a thread group cannot interfere with itself. However, the two thread groups interfere at location `a`.

We propose the following solution to verify multiple thread groups that *do not change the observable behavior*. Based on the RHL on Parlang, we assume that the left program is correct. Hence, we can also assume that the left program does not interfere on the global memory. If the observable behavior does not change after a transformation, the property of non-interference on the global memory persists. The RHL on Parlang verifies a program on (possibly infinitely) many initial memory configurations. It can cover multiple thread groups by providing a thread group identifier. The precondition can limit the range of the thread group identifier and hence the number of thread groups.

To ensure that the observable behavior does not change, the assertions for the relational proof on Parlang programs must imply that all global variables are equal in both program. The relation `eq` matches this requirement, but might be too restrictive for some cases.

3.5 MCL

Parlang is an abstract language that is designed to capture the essential primitives of GPU programs. However, it does not model a real language. Towards this end, we formalize the language MCL. This is the basis to provide developers with a framework to prove MCL program transformations, possibly using some automation. We will relate MCL and Parlang by defining the semantics of MCL in terms of Parlang.

Not all MCL features will be supported by our formalization. The limitations will be discussed in the individual subsections and future work (Chapter 6). The verification of the MCL compiler itself is out of reach of this work.

Because MCL is defined in terms of Parlang, the limitations of the latter carry over. This includes the simplistic memory structure with only shared and thread-local memory. Furthermore, multiple thread groups can only be verified as described in Section 3.4.3.

3.5.1 Primitive Types

MCL has the primitive types `bool`, `integers` and `float`. The types and their operations form the basis for building expressions, e.g. to calculate an array index or a new variable value. For simplicity, we use existing Lean types to approximate the primitive types. The benefit is that all existing lemmas of those types can be used in RHL proofs. We explicitly ignore particular aspects, such as overflows and leave them for future work.

Booleans are mapped to the Lean type `bool`, which is an accurate representation. The type has exactly two elements: `tt` and `ff` denoting true and false respectively. Floats in GPUs have a fixed size in memory. The precision depends on the value of the number, i.e. larger numbers have a smaller precision. We choose a type which does not lose precision; that is, rounding errors cannot be identified by our semantics. Furthermore, there is an upper and a lower boundary on the values, which we ignore. Floats are mapped to type `rat` (from the mathlib library) that models rational numbers. Integers are mapped to `ℕ`.

We define the mapping from MCL types to Lean types as a function.

```
def type_map : type → Type
| int := ℕ
| float := rat
| bool := _root_.bool
```

An instance of `type_map t` for any type `t` is definitionally equal to the underlying Lean type, which means we can directly reason with them using the equality predicate. For example, given some `x` of type `type_map int`, the term `x = (7 : nat)` is well typed.

In cases where it is not clear from the context, we use the term *MCL type* to distinguish from Lean types. MCL types are of type `type` (with a small `t`), while the Lean type universe is **`Type`** (with a capital `T`).

3.5.2 Arrays

Arrays are an essential part of MCL and GPU programs in general. Without them, it is very cumbersome for threads to read from and write to the shared state. The benefit of arrays is that we can compute array indices at runtime, possibly depending on the thread identifier. This allows for all sorts of operations, such as distributing data over threads or enabling adjacent threads (in a group) to communicate. It is therefore important that we formalize arrays and have convenient ways to reason about them in assertions.

We allow declarations of multi-dimensional arrays. The number of dimensions is fixed at compile time. However, we leave it for future work to formalize array sizes. Therefore, arrays are of infinite size and an access to any index is valid.

We have considered multiple options to formalize arrays:

- **Each primitive type has a dimensions argument.** This means we introduce arrays for each primitive type individually. However, all primitives behave the same as elements of arrays, so there is no good reason to reintroduce the concept for each type.
- **Make array a separate type.** Next to the primitives types, `array` is another type that (recursively) takes a `type` as an argument. However, this allows the nesting of arrays, which introduces unnecessary complexity.
- **Make every variable an array.** A variable is an array and has a fixed primitive type for its elements. MCL declarations for primitive types are treated as arrays with a single dimension and element.

We formalize the third option as a structure.

```
structure array := (dim : ℕ) (type : type)
```

3.5.3 Signature

The signature encapsulates the meta-information about all variables.

```
def signature_core := string → variable_def
```

A variable is addressed by string. By definition, there cannot be multiple variables with the same name. As described in section 3.5.2 a variable is always an array and the elements of those arrays are not considered variables.

Each declaration contains the following information:

```
inductive scope
| tlocal
| shared

structure variable_def := (type : array) (scope : scope)
```

A variable is declared *either* in shared *or* thread-local memory.

To make the thread identifier accessible to the threads, we impose some restrictions on the variable with the name `tid`. Its value is set in the initial state, i.e. not by the program itself. To do this in a type-safe manner, the variable must be declared as a thread-local scalar `int`. We achieve this by subtyping `signature_core`.

```
def signature := { sig : signature_core //
    type_of (sig "tid") = type.int ∧
    (sig "tid").type.dim = 1 ∧
    (sig "tid").scope = scope.tlocal }
```

The kernel can access any variable from every location. We do not allow declarations in loop bodies etc. This is a simplification that allows us to fix the signature for all definitions in Lean. That is, the signature never changes in any definition. Visibility can still be restricted by constraining MCL kernels using additional definitions. However, this is left for future work.

To extract information from the signature, we implement a couple of convenience functions:

```
def type_of (v : variable_def) : type := v.type.type
def lean_type_of (v : variable_def) := type_map (type_of v)
def signature.type_of (n : string) (sig :signature) :=
    type_of (sig.val n)
def signature.lean_type_of (n : string) (sig : signature) :=
    lean_type_of (sig.val n)
def is_tlocal (v : variable_def) := v.scope = scope.tlocal
def is_shared (v : variable_def) := v.scope = scope.shared
```

3.5.4 State

Parlang has a generic memory model that abstracts over types. In MCL we fix those types, but make them dependent on the signature.

MCL addresses memory using the variable name and the (computed) array indices.

```
def mcl_address (sig : signature) :=
(Σ n: string, vector ℕ (sig.val n).type.dim)
```

The number of dimensions is dependent on the particular variable. Given an instance `i` of type `mcl_address sig`, the variable name is accessible as `i.1`.

Given the address type and a signature, we can construct the type map for the shared memory.

```
def parlang_mcl_shared (sig : signature) :=
(λ i : mcl_address sig, sig.lean_type_of i.1)
```

Every thread has thread-local memory. We repurpose the same memory model, address type and type map from the shared memory.

```
def parlang_mcl_tlocal (sig : signature) :=
(λ i : mcl_address sig, sig.lean_type_of i.1)
```

Finally, we define the MCL kernel in terms of the Parlang kernel.

```
def parlang_mcl_kernel (sig : signature) := kernel
(memory $ parlang_mcl_tlocal sig) (parlang_mcl_shared sig)
```

3.5.5 Expressions

In Parlang, many parts of the program are defined using Lean functions. Namely, a compute statement is a function from a state to the resulting state (on all active threads). Other examples include the evaluation of conditions for branching and loops. This is called shallow embedding and is a flexible approach. The benefit is that we can use all existing definitions in Lean, for instance, to build up a complex arithmetic term. Moreover, we can use existing lemmas and tactics to reason about the functions. However, for MCL we move to a deep-embedding approach to more accurately capture the structure of MCL programs. The benefits are that inference rules can include expressions and automated tools have a limited inductive structure to cover (i.e. by the *no junk* property of inductive types). We define an inductive type for expressions that is indexed over MCL types.

```
inductive expression (sig : signature) : type → Type
| tlocal_var {t} {dim : ℕ} (n : string)
    (idx : fin dim → (expression int))
    (h₁ : type_of (sig.val n) = t)
    (h₂ : (sig.val n).type.dim = dim)
    (h₃ : is_tlocal (sig.val n)) : expression t
| shared_var {t} {dim : ℕ} (n : string)
    (idx : fin dim → (expression int))
    (h₁ : type_of (sig.val n) = t)
    (h₂ : (sig.val n).type.dim = dim)
    (h₃ : is_shared (sig.val n)) : expression t
| add {t} : expression t → expression t → expression t
```

```
| mult {t} : expression t → expression t → expression t
| literal_int {} {t} (n : ℕ) (h : t = type.int) : expression t
| lt {t} (h : t = type.bool) :
    expression int → expression int → expression t
```

We only formalize the subset of MCL's operators that are needed to create meaningful use cases. Constructors `tlocal_var` and `shared_var` are references to variables. `add` and `mult` are arithmetic operations. `lt` denotes *less than*. We use the common notations `+`, `*`, and `<` for the respective operators.

An expression is always well-typed. That is, all references to variables and nested expressions are type correct. Note that expressions of one type might contain expressions of another type. For example, an expression such as `(4 < 7)` nests two integer literals in a boolean expression.

The definition of expression is not only a recursive but a nested inductive type. A nested inductive type, nests the type to be defined in the type of a constructor argument. In `expression`, the arguments `idx` of constructors `tlocal_var` and `shared_var` require one expression per dimension of the array (which depends on the signature). The straight-forward approach would be to define the argument `idx` over the type
`(vector (expression int) (sig.val n).type.dim)`; however, this is not possible in Lean. A nested inductive type is resolved by the parser to a simple inductive type that is supported by the Lean kernel. However, not all types of nested inductive types are supported. To work around the limitations of the parser, we have to use a slightly different definition. Instead of using vectors directly, we accept a function that maps a natural number, bounded by the number of dimensions `dim`, to the corresponding expression. `vector` provides this function, namely `vector.nth`. We can transform the function back to a vector using `vector.of_fn`.

Because expression is a nested inductive type, pattern matching on it is limited. Suppose we define a function `f` on expression. If `f` needs to be recursively applied on the indices of `tlocal_var` or `shared_var`, pattern matching fails. Recursive definitions are not allowed by the Lean kernel because they generally do not terminate. In most situations, the parser can resolve recursive applications and proof termination automatically. But this is not the case for the recursive occurrences of the indices. Instead, the recursor must be used directly.

Most constructors of *expression* take implicit arguments `t` and `dim`. Furthermore, the constructors require proofs that `t` and `dim` are equal to a value originating from the signature. This is as opposed to substituting `t` and `dim` with the other side of the equality in all constructors. Although, this would increase the readability, our definition is more convenient to use in combination with the relational Hoare logic. We clarify this with an example. Assume that we have two expressions: a reference to variable `"a"` of type `int` and a literal integer. If we want to nest the two expressions using `lt`, we can choose `t` to be `int` for both expression to get a type-correct application. If we substitute `t` by

`type_of (sig.val n)`, the resulting type expression `(type_of (sig.val n))` would not fit as an argument of `lt`. As a consequence, we would have to wrap the expression to adjust its type. However, this would make it harder to use `rw` to rewrite expressions, because the wrapper changes the term.

To evaluate an expression on the thread-local memory we define

```
def eval {sig : signature}
(m : memory $ parlang_mcl_tlocal sig) {t : type}
(expr : expression sig t) : type_map t :=
expression.rec_on expr
    -- tlocal
    (λ t dim n idx h₁ h₂ h₃ ih, by rw ← h₁; rw ← h₂ at ih;
    ↪  exact m.get ⟨n, vector.of_fn ih⟩)
    -- shared
    (λ t dim n idx h₁ h₂ h₃ ih, by rw ← h₁; rw ← h₂ at ih;
    ↪  exact m.get ⟨n, vector.of_fn ih⟩)
    -- add
    (λ t a b ih_a ih_b, type_map_add ih_a ih_b)
    -- mult
    (λ t a b ih_a ih_b, type_map_mult ih_a ih_b)
    -- literal_int
    (λ t n h, (by rw [h]; exact n))
    -- lt
    (λ t h a b ih_a ih_b, (by rw h; exact (ih_a < ih_b)))
```

The comment preceding an argument denotes, to which constructor the small premise belongs to. For simplicity, the evaluation of expressions is computable and cannot fail.

We use the equality hypotheses from `expression` to rewrite the return type using the tactic `rw`. The tactic **`exact`** returns the type-correct term.

The function `eval` only has access to the thread-local memory. We assume that all shared variables that are referenced in the expression have been loaded into the thread-local memory, under the same address. Under this assumption, reading global variables is no different from reading thread-local variables. The loading of variables depends on the context of the expression and is explained in Section 3.5.7.

Addition and multiplication are evaluated in auxiliary definitions.

```
def type_map_add : ∀{t : type}, type_map t → type_map t →
 ↪  type_map t
| int a b := a + b
| float a b := a + b
| bool a b := a && b

def type_map_mult : ∀{t : type}, type_map t → type_map t →
 ↪  type_map t
```

```
| int a b := a * b
| float a b := a * b
| bool a b := a || b
```

For multiple purposes we need to know if an expression references a variable.

```
def expr_reads (n : string) {t : type}
(expr : expression sig t) : _root_.bool :=
expression.rec_on expr
    -- tlocal
    (λ t dim m idx h₁ h₂ h₃ ih, (m = n) ||
        (vector.of_fn ih).to_list.any id)
    -- shared
    (λ t dim m idx h₁ h₂ h₃ ih, (m = n) ||
        (vector.of_fn ih).to_list.any id)
    -- add
    (λ t a b ih_a ih_b, ih_a || ih_b)
    -- mult
    (λ t a b ih_a ih_b, ih_a || ih_b)
    -- literal_int
    (λ t n h, ff)
    -- lt
    (λ t h a b ih_a ih_b, ih_a || ih_b)
```

If a variable is not used in an expression, we can ignore corresponding updates.

```
lemma eval_update_ignore (h : expr_reads n expr = ff) :
eval (s.update ⟨n, idx₂⟩ v) expr = eval s expr
```

A reference to a variable in any part of an MCL program is often used to access memory. The index expressions of the reference must first be evaluated to numerical indices and rewritten to a type correct `vector`. We abstract this process in a function.

```
def mcl_addr_from_var {sig : signature} {n dim}
(h₂ : (sig.val n).type.dim = dim)
(idx : vector (expression sig type.int) dim)
(m : memory $ parlang_mcl_tlocal sig) : mcl_address sig :=
⟨n, by rw ← h₂ at idx; exact idx.map (eval m)⟩
```

3.5.6 Kernels and Programs

We define the syntax of MCL kernels, which look similar to Parlang kernels:

```
inductive mclk (sig : signature)
| tlocal_assign {t : type} {dim : ℕ} (n : string)
  (idx : vector (expression sig int) dim)
```

```
  (h₁ : type_of (sig.val n) = t)
  (h₂ : (sig.val n).type.dim = idx.length) :
  (expression sig t) → mclk
| shared_assign {t : type} {dim : ℕ} (n)
  (idx : vector (expression sig int) dim)
  (h₁ : type_of (sig.val n) = t)
  (h₂ : (sig.val n).type.dim = idx.length) :
  (expression sig t) → mclk
| seq : mclk → mclk → mclk
| for (n : string) (h : sig.type_of n = int)
  (h₂ : (sig.val n).type.dim = 1) :
  expression sig int → expression sig bool → mclk → mclk → mclk
| ite : expression sig bool → mclk → mclk → mclk
| skip {} : mclk
| sync {} : mclk
```

Similar to `expression`, we use arguments `t` and `dim` in combination with equality proofs on them (cf. Section 3.5.5). To model MCL accurately, we drop the `compute` constructor from Parlang and introduce thread-local and shared assignments. All assignments must be type-correct according to the signature. Furthermore, because there is no `compute` constructor, we introduce a `skip` instruction.

We formalize the commonly used for-loop. As opposed to while-loops, for-loops have an explicit loop variable (`n`). Using for-loops, we can define inference rules for common loop-patterns, e.g. a loop variable ranging over numbers.

We encapsulate the kernel in an MCL program:

```
inductive mclp (sig : signature)
| intro (f : memory (parlang_mcl_shared sig) → ℕ)
    (k : mclk sig) : mclp
```

Because we have few inference rules on programs, we chose a shallow embedding to compute the thread parallelism.

3.5.7 Translation to Parlang

To keep track of possible interference, Parlang requires that all accesses to shared memory are performed by `load` instructions. This makes it easier to reason about Parlang programs, because the `load` instructions show the read accesses to shared memory. If any part on the MCL program (including the conditions for `ite` and `for`) requires shared variables, their values must first be loaded into the thread-local memory. In this process, we have to ensure that the loaded values do not interfere with the integrity of the thread-local state, e.g. by overwriting a thread-local variable. Because shared and thread-local variables share the same signature, a variable name is exclusively scoped to either shared or thread-local.

Therefore, we can take the same address to keep the copy in the thread-local state without interfering with thread-local variables.

Additionally, we have to ensure that the `load` instructions are placed correctly in the code, such that the load happens before consumption and is not invalidated afterwards. Hence, there must not be a `store` (to the same address) or a `sync` between load and consumption. We have to place `load` instructions for every `mclk` constructor that uses expressions: `shared_assign`, `tlocal_assign`, `ite`, and `for`. For the first three constructors, we place the `load` instructions before any other Parlang instruction we generate for those cases. The `for` constructor is different, in that the loop condition is possibly evaluated many times. Therefore, we have to reload the values before every evaluation. The first evaluation happens before executing the loop body. To reload the local copies, we place the same set of `load` instructions inside the loop body after the regular `for` body and the incrementor function.

The list of `load` instructions is computed by recursively analyzing the expression and finding all occurrences of shared variables.

```
def load_shared_vars_for_expr {sig : signature} {t : type}
(expr : expression sig t) : list (parlang_mcl_kernel sig) :=
expression.rec_on expr
    -- tlocal
    (λ t dim n idx h₁ h₂ h₃ ih,
        (vector.of_fn ih).to_list.foldl list.append [])
    -- shared
    (λ t dim n idx h₁ h₂ h₃ ih,
        (vector.of_fn ih).to_list.foldl list.append [] ++
        [kernel.load (λ m,
        ⟨mcl_addr_from_var h₂ (vector.of_fn idx) m, λ v,
        m.update
        (mcl_addr_from_var h₂ (vector.of_fn idx) m) v⟩)])
    -- add
    (λ t a b ih_a ih_b, ih_a ++ ih_b)
    -- mult
    (λ t a b ih_a ih_b, ih_a ++ ih_b)
    -- literal_int
    (λ t n h, [])
    -- lt
    (λ t h a b ih_a ih_b, ih_a ++ ih_b)
```

To allow Lean to compute this function, we avoid the use of sets and use lists instead. This means we potentially load the same variable multiple times; however, this is idempotent. Moreover, the order of the `load` instructions is irrelevant. All expressions are fully evaluated, e.g. even if the left side of an `or` expression holds we still evaluate the right side, and therefore pessimistically load all values. The list of instructions is translated into a sequence of instructions and either prepended or appended to a given instruction.

```
def prepend_load_expr {sig : signature} {t : type}
(expr : expression sig t) (k : parlang_mcl_kernel sig) :=
(load_shared_vars_for_expr expr).foldr kernel.seq k

def append_load_expr  {sig : signature} {t : type}
(expr : expression sig t) (k : parlang_mcl_kernel sig) :=
(load_shared_vars_for_expr expr).foldl kernel.seq k
```

On a per-thread basis, the `load` instructions are only executed if the expression (containing the shared variables) is evaluated. This can be observed by the behavior of active maps on instruction sequences in Parlang's operational semantics. Given a sequence `a ;; b` (where `a` and `b` are kernels), then `a` and `b` are executed on the same active map. In other words, only nested kernels change the active map. Suppose `b` is an `ite`. Then the condition-expression is evaluated, regardless of what branch a thread takes.

Assignments in MCL are translated to `compute` instructions in Parlang. The given variable is updated with the new value, that is computed by evaluating the expression. Array index expressions are evaluated to calculate the location in memory. We encapsulate this process in a function, which also takes care of type correctness.

```
def memory.update_assign {sig : signature} {t : type} {dim : ℕ}
(n : string) (idx : vector (expression sig int) dim)
(h₁ : type_of (sig.val n) = t)
(h₂ : (sig.val n).type.dim = idx.length)
(expr : expression sig t)
(m : memory $ parlang_mcl_tlocal sig) :
memory $ parlang_mcl_tlocal sig :=
m.update (mcl_addr_from_var h₂ idx m)
(by rw ← h₁ at expr; exact (eval m expr))
```

The translation from `mclk` to a Parlang kernel is defined as a function.

```
def mclk_to_kernel {sig : signature} : mclk sig →
parlang_mcl_kernel sig
| (seq k₁ k₂) :=
    kernel.seq (mclk_to_kernel k₁) (mclk_to_kernel k₂)
| skip := kernel.compute id
| sync := kernel.sync
| (tlocal_assign n idx h₁ h₂ expr) := idx.to_list.foldr
    (λexpr' k, prepend_load_expr expr' k) $
    prepend_load_expr expr (kernel.compute $
        memory.update_assign n idx h₁ h₂ expr)
| (shared_assign n idx h₁ h₂ expr) := idx.to_list.foldr
    (λexpr' k, prepend_load_expr expr' k) $
    prepend_load_expr expr (kernel.compute $
        memory.update_assign n idx h₁ h₂ expr) ;;
    kernel.store (λ m, ⟨mcl_addr_from_var h₂ idx m, m.get $
```

```
        mcl_addr_from_var h₂ idx m⟩)
| (ite c th el) := prepend_load_expr c (
    kernel.ite (λm, eval m c)
        (mclk_to_kernel th) (mclk_to_kernel el))
| (for n h h₂ expr c k_inc k_body) := prepend_load_expr expr
    (kernel.compute $ memory.update_assign n v[0] h h₂ expr);;
    prepend_load_expr c (
        kernel.loop (λ s, eval s c) (mclk_to_kernel k_body ;;
            append_load_expr c (mclk_to_kernel k_inc))
    )
```

The translation of the constructors seq, skip and sync is trivial. The constructor tlocal_assign computes a new thread-local memory according to update_assign. The shared variables that are referenced in expr and all expressions in idx are prepended. shared_assign also computes the new value in thread-local memory first and stores it into shared memory in a dedicated instruction. Note that all occurrences of load and store instructions only copy values between thread-local and shared memory without applying any arithmetic operators. This was a design decision to simplify the proof of inference rules. for loops are translated to Parlang loops. The initial value of the loop variable is set before the loop. The for loop body and the incrementor together form the basis for the Parlang loop body. Because the loop condition c is evaluated after every iteration, the shared variables have to be reloaded with every iteration. For the first iteration, we load the shared variables of c before the loop. For every other iteration, we reload the variables in the loop body, after any other instruction.

The translation from mclp to Parlang programs is straight-forward.

```
def mclp_to_program {sig : signature} : mclp sig →
parlang.program (memory $ parlang_mcl_tlocal sig)
(parlang_mcl_shared sig)
| (mclp.intro f k) :=
    parlang.program.intro f (mclk_to_kernel k)
```

3.6 Relational Hoare Logic for MCL

To prove program transformations of MCL programs we define a relational Hoare logic for MCL. Due to the relation of MCL and Parlang, we can lift the existing RHL on Parlang to MCL. Because MCL has a fixed type map (however dependent on the signature), we create an alias (i.e. a Lean definition) for MCL assertions.

```
def state_assert (sig₁ sig₂ : signature) :=
∀ n₁:ℕ, parlang.state n₁ (memory (parlang_mcl_tlocal sig₁))
(parlang_mcl_shared sig₁) → vector bool n₁ →
∀ n₂:ℕ, parlang.state n₂ (memory (parlang_mcl_tlocal sig₂))
(parlang_mcl_shared sig₂) → vector bool n₂ → Prop
```

Programs and kernels are translated from MCL to Parlang according to the definitions from Section 3.5.

```
def mclk_rel {sig₁ sig₂ : signature}
    (P : state_assert sig₁ sig₂)
    (k₁ : mclk sig₁) (k₂ : mclk sig₂)
    (Q : state_assert sig₁ sig₂) :=
rel_hoare_state P (mclk_to_kernel k₁) (mclk_to_kernel k₂) Q
```

We use the same Hoare quadruple notation as for Parlang. Furthermore, we define RHL on `mclp`.

```
def mclp_rel {sig₁ sig₂ : signature} (P) (p₁ : mclp sig₁)
(p₂ : mclp sig₂) (Q) := rel_hoare_program mcl_init mcl_init
P (mclp_to_program p₁) (mclp_to_program p₂) Q
```

We can lift the proof to relate the logics of kernels and programs from Parlang to MCL.

```
lemma rel_mclk_to_mclp (h : mclk_rel (λ n₁ s₁ ac₁ n₂ s₂ ac₂,
    ∃ m₁ m₂, initial_kernel_assertion mcl_init mcl_init
    P f₁ f₂ m₁ m₂ n₁ s₁ ac₁ n₂ s₂ ac₂)
k₁ k₂
(λ n₁ s₁ ac₁ n₂ s₂ ac₂, (∃ m₁, s₁.syncable m₁) → ∃ m₁ m₂,
    s₁.syncable m₁ ∧ s₂.syncable m₂ ∧ Q m₁ m₂))
(hg : ∀ {m₁ m₂}, P m₁ m₂ → 0 < f₁ m₁) :
mclp_rel P (mclp.intro f₁ k₁) (mclp.intro f₂ k₂) Q :=
rel_kernel_to_program h @hg
```

Similarly, we can lift the rules `consequence`, `seq` and the single-step rules, e.g. `single_step_left`.

```
lemma consequence (h : {* P *} k₁ ~> k₂ {* Q *})
(hp : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂, P' n₁ s₁ ac₁ n₂ s₂ ac₂ →
```

```
    P n₁ s₁ ac₁ n₂ s₂ ac₂)
(hq : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂, Q n₁ s₁ ac₁ n₂ s₂ ac₂ →
    Q' n₁ s₁ ac₁ n₂ s₂ ac₂) :
{* P' *} k₁ ~> k₂ {* Q' *} := consequence h hp hq

lemma seq (Q)
(h₁ : {* P *} k₁ ~> k₂ {* Q *})
(h₂ : {* Q *} k₁' ~> k₂' {* R *}) :
{* P *} k₁ ;; k₁' ~> k₂ ;; k₂' {* R *} := parlang.seq Q h₁ h₂

lemma single_step_left :
{* P *} k₁ ~> skip {* Q *} →
{* Q *} k₁' ~> k₂ {* R *} →
{* P *} (k₁ ;; k₁') ~> k₂ {* R *} := parlang.single_step_left Q
```

The inference rule `seq` can be lifted by using the lemma from Parlang directly. Hence, MCL's rule is just an instance of Parlang's rule. This is because the sequence instruction in MCL directly relates to the sequence instruction in Parlang (cf. `mclk_to_kernel`). An inference rule where this line of reasoning does not apply is `shared_assign_right`.

```
lemma shared_assign_right {t dim n}
{idx : vector (expression sig₂ type.int) dim}
{h₁ : type_of (sig₂.val n) = t}
{h₂ : ((sig₂.val n).type).dim = dim}
{expr : expression sig₂ t} :
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₁ s₁ ac₁ n₂
    (s₂.map_active_threads ac₂ (
        thread_state.tlocal_to_shared n idx h₁ h₂ ∘
        thread_state.compute
            (memory.update_assign n idx h₁ h₂ expr) ∘
        thread_state.update_shared_vars_for_expr expr ∘
        thread_state.update_shared_vars_for_exprs idx
    )) ac₂ *}
(skip : mclk sig₁) ~> shared_assign n idx h₁ h₂ expr {* P *}
```

`shared_assign_right` is a single-sided instruction rule. It translates to at least one `compute` and one `store` instruction in Parlang. To avoid lambda-terms in the assertions, we define a function that copies a value from shared to thread-local memory.

```
def thread_state.tlocal_to_shared {sig : signature} {t} {dim}
(var : string) (idx : vector (expression sig type.int) dim)
(h₁ : type_of (sig.val var) = t)
(h₂ : ((sig.val var).type).dim = dim) :=
@thread_state.store _ _ (parlang_mcl_shared sig) _ (
    λ (m : memory $ parlang_mcl_tlocal sig),
    ⟨mcl_addr_from_var h₂ idx m,
        m.get $ mcl_addr_from_var h₂ idx m⟩)
```

Furthermore, there may be multiple load instructions. We abstract the changes on `thread_state` of all load instructions of an expression.

```
def thread_state.update_shared_vars_for_expr {t : type}
(expr : expression sig t) :
thread_state
  (memory $ parlang_mcl_tlocal sig) (parlang_mcl_shared sig) →
thread_state
  (memory $ parlang_mcl_tlocal sig) (parlang_mcl_shared sig) :=
expression.rec_on expr
    -- tlocal
    (λ t dim n idx h₁ h₂ h₃ ih, id)
    -- shared
    (λ t dim n idx h₁ h₂ h₃ ih, λ ts,
    ((list.range_fin dim).foldl (λ ts e, ih e ts) ts).load (
    λ m, ⟨mcl_addr_from_var h₂ (vector.of_fn idx) m, λ v,
    m.update (mcl_addr_from_var h₂ (vector.of_fn idx) m) v⟩))
    -- add
    (λ t a b ih_a ih_b, ih_b ∘ ih_a)
    -- mult
    (λ t a b ih_a ih_b, ih_b ∘ ih_a)
    -- literal_int
    (λ t n h, id)
    -- lt
    (λ t h a b ih_a ih_b, ih_b ∘ ih_a)
```

We also lift the definition to support a vector of expressions.

```
def thread_state.update_shared_vars_for_exprs {n} {t : type}
(exprs : vector (expression sig t) n) :
thread_state
  (memory $ parlang_mcl_tlocal sig) (parlang_mcl_shared sig) →
thread_state
  (memory $ parlang_mcl_tlocal sig) (parlang_mcl_shared sig) :=
λts, exprs.to_list.foldr (λexpr ts,
    thread_state.update_shared_vars_for_expr expr ts) ts
```

Using `swap` we can create `shared_assign_left`.

```
lemma shared_assign_left {t dim n expr}
{idx : vector (expression sig₁ type.int) dim}
{h₁ : type_of (sig₁.val n) = t}
{h₂ : ((sig₁.val n).type).dim = vector.length idx} :
{* λ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₁ (s₁.map_active_threads ac₁ (
    thread_state.tlocal_to_shared n idx h₁ h₂ ∘
    thread_state.compute
        (memory.update_assign n idx h₁ h₂ expr) ∘
    thread_state.update_shared_vars_for_expr expr ∘
    thread_state.update_shared_vars_for_exprs idx
```

```
)) ac₁ n₂ s₂ ac₂ *}
shared_assign n idx h₁ h₂ expr ~> skip {* P *} := begin
    apply swap_skip shared_assign_right,
end
```

Chapter 4

Use Cases

We evaluate the applicability of our approach by presenting two use cases. The first example is written in MCL and breaks down the two programs using instruction rules – a versatile but cumbersome approach. The second example is written in Parlang and shows the benefit of a transition rule.

4.1 Example 1: Swapping Assignments

We show that reordering of two assignment instructions preserves the input-output behavior. We do not have a transformation rule for this case. Hence, we use instruction rules to decompose the programs instruction by instruction.

First, we define a signature, where all variables except `tid` are shared one-dimensional arrays of type `int`.

```
def sigc : signature_core
| "tid" := { scope := scope.tlocal, type := ⟨1, type.int⟩ }
| _ := { scope := scope.shared, type := ⟨1, type.int⟩ }

def sig : signature := ⟨sigc, ⟨rfl, rfl, rfl⟩⟩
```

Because the signature is computable, all proofs regarding the signature can be proven by reflexivity. We define the initial implementation of the program with two assignments to different variable names.

```
def read_tid :=
(@expression.tlocal_var sig _ _ "tid" (λ_, 0) rfl rfl rfl)

def p₁ : mclp sig := mclp.intro (λ m, 100) (
    mclk.shared_assign "a" v[read_tid] rfl rfl read_tid ;;
    mclk.shared_assign "b" v[read_tid] rfl rfl
```

```
        (read_tid + (expression.literal_int 1 rfl))
)
```

A simple transformation is to swap the assignments.

```
def p₂ : mclp sig := mclp.intro (λ m, 100) (
    mclk.shared_assign "b" v[read_tid] rfl rfl
        (read_tid + (expression.literal_int 1 rfl)) ;;
    mclk.shared_assign "a" v[read_tid] rfl rfl read_tid
)
```

We want to show that the transformation from `p₁` to `p₂` is I/O preserving.

```
lemma assign_rel' : mclp_rel eq p₁ p₂ eq
```

For the pre- and postcondition we relate the memories using equality. This includes, that arrays `"a"` and `"b"` must be equal after execution. All other variables remain untouched by the programs.

By applying `rel_mclk_to_mclp` to the the goal, we can reduce the goal to a Hoare quadruple on the respective kernels of `p₁` and `p₂`. Using the single-step rules (e.g. `single_left_step_left`), we can decompose the Hoare quadruples, such that one kernel contains a single instruction and the other side is a `skip`. Using the single-sided instruction rules `shared_assign_right` and `shared_assign_left`, we can eliminate the Hoare quadruples. After this process, we have the following assumptions (among others).

```
∀ n₁ s₁ ac₁ n₂ s₂ ac₂, (∃ m₁ m₂),
initial_kernel_assertion mcl_init mcl_init eq
(λ _, 100) (λ _, 100) m₁ m₂ n₁ s₁ ac₁ n₂ s₂ ac₂
```

From the assumptions we can conclude that `s₁ = s₂`, `m₁ = m₂`, and `n₁ = n₂`. It remains to solve the following goal.

```
∃ m₁' m₂', syncable (map_active_threads ac₁ (
    tlocal_to_shared "b" v[read_tid] _ _ ∘
    compute (memory.update_assign "b" v[read_tid] _ _
        (read_tid + expression.literal_int 1 _)) ∘
    update_shared_vars_for_expr
        (read_tid + expression.literal_int 1 _) ∘
    update_shared_vars_for_exprs v[read_tid] ∘
    tlocal_to_shared "a" v[read_tid] _ _ ∘
    compute (memory.update_assign "a" v[read_tid] _ _
        read_tid) ∘
    update_shared_vars_for_expr read_tid ∘
    update_shared_vars_for_exprs v[read_tid]
) s₁) m₁' ∧ syncable (map_active_threads ac₂ (
    tlocal_to_shared "a" v[read_tid] _ _ ∘
```

```
    compute (memory.update_assign "a" v[read_tid] _ _
        read_tid) ∘
    update_shared_vars_for_expr read_tid ∘
    update_shared_vars_for_exprs v[read_tid] ∘
    tlocal_to_shared "b" v[read_tid] _ _ ∘
    compute (memory.update_assign "b" v[read_tid] _ _
        (read_tid + expression.literal_int 1 _) ∘
    update_shared_vars_for_expr
        (read_tid + expression.literal_int 1 _) ∘
    update_shared_vars_for_exprs v[read_tid]
) s₂) m₂' ∧ m₁' = m₂'
```

Among others, we have to prove `syncable` for the resulting state of both kernel executions. The structure of the resulting state is similar for both kernels. They map all active threads (hence all threads in our case) using the functors `tlocal_to_shared`, `compute`, `update_shared_vars_for_expr` and `update_shared_vars_for_exprs`.

In our goal, the arguments of `update_shared_vars_for_expr` and `update_shared_vars_for_exprs` that are of type `expression` do not contain any references to shared variables. Using computability, we can reduce the terms to `id` and remove the latter using the tactic `simp`. Only the functors `tlocal_to_shared` and `compute` remain.

In order to prove a proposition of type `syncable _ _`, we have to prove a proposition on every address `i` of type `mcl_address sig` (cf. the definition of `syncable`). We can do so by doing case distinct on `i`. If the variable name is neither `"a"` nor `"b"`, no thread stores at these addresses and this part of the resulting memory is equal to the initial memory. In the case that the variable name is `"a"` or `"b"`, we have to do case distinction once more; this time on the array index. If the array index is a below `0` or above `99`, again no thread stores at these addresses. In the case that the array index is between `0` and `99`, there exists a thread that stores at this address. For those cases it has to be proven that no other thread stores at that address. Furthermore, those cases influence the resulting memory. In all cases, we have to study the `state` argument of `syncable` in a repeating fashion, which is cumbersome.

Instead of doing case distinction on addresses directly, we proof `syncable` by the structure of the state. We define a variation of `syncable`, that allows us to process the functors of `map_active_threads` one by one.

```
def syncable' (shole : set ι) (lhole : set ι) (s : state n σ τ)
(m : memory τ) : Prop :=
state.syncable s m ∧ ∀ i tid,
(i ∈ shole → i ∉ (s.threads.nth tid).stores) ∧
(i ∈ lhole → i ∉ (s.threads.nth tid).loads)
```

The two new arguments `shole` and `lhole` denote addresses at which no thread is allowed to store to or load from, respectively. We can relate `syncable` and

syncable' and use this lemma to rewrite our goal.

```
lemma syncable_syncable' :
syncable' ∅ ∅ s m ↔ state.syncable s m
```

For brevity and due to the intensive need of auxiliary definitions, we only provide the intuition for the remaining part of the proof. The entire proof is available in Lean.

We have to provide the witnesses for the resulting memories m_1' and m_2'. Initially, we use the initial memories m_1 and m_2 respectively. With every tlocal_to_shared function that we process, we update cells in the memory (at most one per active thread). The new memory cell value is dependent on s_1 and all compute functions that come before tlocal_to_shared (in the order of execution according to the operational semantics). After we process a tlocal_to_shared, no more stores or loads are allowed to the related addresses. We use arguments shole and lhole of syncable' to keep track of these addresses. This way, we ensure that multiple tlocal_to_shared functions do not interfere with each other.

Using syncable' and the related lemmas, we obtain the resulting memories. It remains to show that the postcondition holds, i.e. that $m_1' = m_2'$. The memories are based on m_1 and m_2 respectively, which are equal. Therefore, we substitute m_2 with m_1. On each side of the equation, the same updates are applied to m_1, however, in a different order. By changing the order of the update instructions, we can make both sides structurally equal.

Non-termination cannot occur because the programs do not contain loops. Furthermore, there is no possibility of a barrier divergence because the programs do not contain sync instructions.

4.2 Example 2: Known Branch

We consider the following Parlang program, where k is an arbitrarily complex kernel:

```
def p₁ : program bool (λ (s : string), ℕ) :=
program.intro (λm, m.get "x") (
    compute (λ_, tt) ;;
    ite id (
        k
    ) (
        store (λ_, ⟨"a", 5⟩)
    )
)
```

The thread-local storage is of type bool. The condition of the ite instruction is true if the thread-local storage equals tt, which is the case due to the proceeding

compute instruction. Hence, the condition holds for all threads and the else branch is inactive in all executions. Therefore, we want to replace the ite instruction by its then branch.

```
def p₂ : program bool (λ (s : string), ℕ) :=
program.intro (λm, m.get "x") (
    compute (λ_, tt) ;;
    k
)
```

We want to show that the transformation from p_1 to p_2 is I/O preserving.

```
example : rel_hoare_program (λ_, ff) (λ_, ff)
(λ m₁ m₂, eq m₁ m₂ ∧ 0 < m₁.get "x") p₁ p₂ eq
```

We consider removing inactive branches to be a common transformation, therefore, we define a transformation rule for it, which is applicable here, but generic enough to be used for other kernels as well.

```
theorem known_branch_left (c : σ₁ → bool) (th) (el)
(h₁ : ∀ n₁ s₁ ac₁ n₂ s₂ ac₂, P n₁ s₁ ac₁ n₂ s₂ ac₂ →
    ∀ tid : fin n₁, c (s₁.threads.nth tid).tlocal)
(h₂ : {* P *} th ~> k₂ {* Q *}) :
{* P *} kernel.ite c th el ~> k₂ {* Q *}
```

We prove this transformation rule by the semantics of ite, which splits the execution into the two branches (cf. Section 3.3.5). We can reason about their respective active maps using h_1. From h_1 we can derive that deactivating threads based on condition c and state s results in an active map where no thread is active. Hence, no thread is active in the else branch and the execution of the else branch does not change the state. Futhermore, by negating the condition no thread gets deactivated. Hence, the then branch is executed on the same threads as the ite instruction itself. We can derive the execution of the then branch from h_2.

To proof the transformation from p_1 to p_2, we show the relation of k to itself by rhl_eq. We can remove the branching from the left kernel by applying known_branch_left. The validity of the condition id follows from the compute instruction, which we process using an instruction rule.

The validity of the postcondition follows from $s_1 = s_2$ after executing k, and the asymmetry of RHL on Parlang. By definition, the left kernel has a valid execution. Because Parlang kernels are deterministic, the right program also has a valid execution and the resulting memories are equal (under the assumption that there is at least one thread).

The benefit of defining a transformation rule is that it reduces the proof work of this particular transformation and can be used in other proofs as well. We

need to process the `compute` instruction to establish that the condition holds. However, we can ignore the complexity of kernel `k` by using `rhl_eq`.

The use case also shows the benefit of proving transformations instead of proving every version of a GPU program individually (e.g. using GPUVerify). The proof that p_1 or p_2 is non-interferent and barrier diverge free without any assumption requires an analysis of `k`. The proof of the transformation from p_1 to p_2 does not require an analysis of `k`, as illustrated above.

Chapter 5

Related Work

5.1 Semantics and Logics of GPU Programs

Betts et al. [2] base their tool (GPUVerify) on the *synchronous, delayed visibility* (SDV) semantics. Under the SDV semantics, all threads advance in lockstep (synchronous). Accesses to shared variables are logged and the writes to shared variables of one thread are only visible to the other threads after the next barrier (delayed visibility). The logs are used to detect interference at the barrier. Our semantics is inspired by SDV; however, Parlang is a big-step semantics and SDV is a small-step semantics. SDV has been mechanized in Isabelle [23]; however, the mechanization is independent of GPUVerify. That is, the output of the verifier is not checked.

Blom et al. [24] present a logic to prove functional correctness and data-race-freedom of GPU kernels. It is based on permission-based separation logic. Separation logic is an instance of Hoare logic where the assertions reason about dynamically allocated memory. Furthermore, which thread is allowed to read and write to which shared variable is tracked by permissions. At every barrier statement, the permissions can be redistributed among threads. An interference is detected if permissions are inconsistent. Similar to Hoare logic, the program must be annotated with pre- and postconditions. Additionally, annotations are required for group, thread and barrier specifications, which makes the proof fairly verbose. Based on their logic, they provide a prototype based on the verification tool VerCors [25] and experimental results of a single example.

Kojima et al. [21] present two different semantics for GPU kernels and show that they are equal under the assumption of non-interference. The first semantic is a lockstep big-step semantic, similar to the SDV semantics. However, a kernel often runs multiple sub-groups in parallel (which do not run in lockstep). Therefore, they present a second semantic that models all possible interleavings of threads. They show that the two semantics are equivalent for non-interfering

programs. Hence, for race-free programs, the simpler semantics can be used. For simplicity, we did not create an interleaving semantics for Parlang. Based on the lockstep semantics they created a non-relational Hoare logic with a few basic inference rules. The activeness of threads is an extra argument to the logic. This is as opposed to our approach, to include the active map in the assertions. The formalization has been mechanized in Coq in a separate paper.[1]

Collingbourne et al. [26] present another two semantics as part of the GPUVerify project. Similar to Kojima et al. one semantics is an interleaving semantics and the second one is a lockstep semantics. They also prove equivalence of the two semantics. Compared to Kojima et al. the approach is less formal and has not been mechanized (to the best of our knowledge).

None of the above define a relational Hoare logic for GPU programs. Instead, they concentrate on the verification of a single GPU program.

5.2 Verification Tools

Li et al. has developed PUG and released multiple versions. The original release [3] can verify data race freedom and user assertions to a limited extent. Variable assignments are translated into static single assignment (SSA) form and the resulting statements are directly translated into constraints which are handed of to an SMT solver, together with the assertions. The biggest limitation is that verification only terminates in reasonable time on 2–3 threads.

The successor PUG_{para} [27] can additionally check equivalence of two kernels. While PUG verifies for a fixed amount of threads, the newer release reasons about a single parameterized thread. It uses data flow analyses on shared variables to detect data races. To get around the limitations of the underlying SMT solver, PUG_{para} overapproximates, which may lead to missed bugs.

GPUVerify [2] uses the SDV semantics to detect data races and barrier divergences. It proves that it suffices to show data race and barrier divergence freedom for two arbitrary threads to conclude it for arbitrarily many threads. The data race and barrier divergence free program is transformed into a sequential program to simplify the verification of user-defined assertions using common analyses techniques.

Kojima et al. [28] use their lockstep semantic to develop a verification condition generator (VCG). They do not use the exact inference rules from their previous work and instead generate conditions from the kernel code in an SSA fashion. The description in the paper is fairly informal; however, an implementation exists. It rewrites the verification conditions for optimization purposes, before handing them off to an off-the-shelf SMT solver. They have successfully proven correctness of seven kernels. They do not attempt to compare two kernels. To

[1] The paper and the conference appear to be in Japanese. To the best of our knowledge, neither the paper nor information about the conference are available in English.

the best of our knowledge, the soundness of the VCG has not been formally proven.

Tripakis et al. [29] present an approach to check equality of two kernels by first checking that they are deterministic followed by checking equivalence based on the assumption of determinism. Their approach is fairly limited as it cannot deal with branches. Furthermore, their implementation can only check determinism but does not perform equivalence checks.

All the tools above target either CUDA or OpenCL. No verification tool for MCL has been reported.

5.3 Relational Hoare Logic

Benton [20] designed relational Hoare logic (RHL) as a variation of the traditional Hoare logic to relate two programs. The pre- and postconditions are adapted to relations, that can make assertions on the states of the two program. Benton shows relational Hoare logic on the syntax and denotational semantics of standard `while`-programs. Our logic is closely related; however, we define it on our new semantics (Parlang) that models GPU programs. Another key difference is that our logic is asymmetric, which simplifies the proofs without substantially weakening the expressiveness.

5.4 Optimization Tools and Automation

Relational Hoare logic is often referenced in the context of optimizations. Much research has been done on proving optimizations in compilers. These optimizations are usually proven once for arbitrary programs and then embedded into the compilation process.

Rhodium [30] is a tool to automatically prove the soundness of compiler optimizations. The idea is that a developer can extend the compiler with domain-specific optimizations. To express the optimization they have developed a domain-specific language that works on control flow graphs. The optimizations are generic (i.e. not specific to a program) and are given to a theorem prover to automatically prove soundness using abstract interpretation (cf. [31]). Our work can be used to prove an optimization of a particular program as well as a generic optimization. We do not make an effort to automatically apply optimizations (cf. Section 2.1.3).

Chapter 6

Conclusion and Future Work

We have created the abstract language Parlang to capture the essential behavior of GPU programs. Furthermore, we have used Parlang as a basis to define the operational semantics of MCL. The layered approach extends to our adaption of relational Hoare logic, which is first defined on Parlang and then lifted to MCL. The layered approach also suggests, how the proof of complex program transformations of real programs can be naturally broken down. Parlang proofs are concerned with the basic behavior of GPU programs. MCL proofs complement the former with aspects of the concrete language.

The layered approach makes it possible to use parts of our work individually. On the one hand, Parlang can be used to study GPU programs abstractly. On the other hand, MCL can be used to prove real programs. Parlang can also be the basis to formalize other programming languages. To that end, MCL serves as a template.

We have shown how to create an asymmetric version of the relational Hoare logic. This version is expressive enough to verify program transformations. Compared to requiring co-termination, the benefit is that we have additional assumptions, which simplify the proofs of transformation. This can be observed by the rule `rel_kernel_to_kernel` (cf. Section 3.4). However, the trade-offs of an asymmetric RHL certainly merit a more thorough analysis.

We have presented the verification of program transformations as a complementing method to verifying non-interference and barrier diverge freeness of individual GPU programs. Section 3.4.1 shows how the two methods can be used together. Furthermore, in Section 4.2 we demonstrate the advantage of proving program transformations, when verifying multiple program versions.

To make this work feasible in the available time we had to simplify the formal-

ization. Therefore, we see much potential to improve our work.

Our memory architecture only knows shared and thread-local memory. This is sufficient for our model because we focus on the verification of a single thread group, and hence, the scope of global and shared memory is the same. However, a more accurate approach is to distinguish between global and shared memory. Having a dedicated global memory would be the basis for Parlang to support the verification of multiple thread groups and the host program. We believe that this line of research is a valuable follow up to broaden the range of supported optimizations.

MCL is an advanced language. Our model only captures an approximation of MCL to illustrate the lift from Parlang to a concrete language and to prove a use case in Chapter 4. However, to increase the confidence in the model it should be sound with regards to the MCL specification. That is, MCL's operational semantics should not hold in any case that is not according to the MCL specification. Current limitations that break soundness include missing out of bounds checks for array accesses and the model of primitive types (e.g. missing overflow detection). Furthermore, our model is missing MCL features, for example, some expression operators and while loops. A complete model increases the application in practice. Towards that end, it would also be useful to read MCL programs from source code, instead of having to define them in Lean. This eliminates the risk of discrepancy between the definition in Lean and the actual program.

We have shown how to prove program transformations manually in Lean. This is a cumbersome process and requires experience in theorem proving. By creating Lean tactics the developer can be aided in the proof work. For common and simple optimizations, a tactic might even proof the transformation entirely automatically. An interesting approach is to investigate the integration of existing automated tools, such as SMT solvers.

All in all, we believe this work lays a good foundation to prove program transformations of GPU programs and we hope to see further work in this direction.

Bibliography

[1] H. Hijma, *Programming Many-Cores on Multiple Levels of Abstraction*. PhD thesis, Vrije Universiteit Amsterdam, 2015.

[2] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson, "GPU-Verify: A Verifier for GPU kernels," in *ACM SIGPLAN Notices*, vol. 47, pp. 113–131, 2012.

[3] G. Li, G. Gopalakrishnan, R. M. Kirby, and D. Quinlan, "PUG : A Symbolic Verifier of GPU Programs," 2012.

[4] T. Rauber and G. Ruenger, *Parallel programming : for multicore and cluster systems*. Berlin: Springer-Verlag, 2010.

[5] Nvidia, "Cuda toolkit documentation v10.1.168," Aug 2019.

[6] Khronos Group, "The OpenCL Specification v2.2," Aug 2019.

[7] Y. Lin and V. Grover, "Using CUDA Warp-Level Primitives," 2018.

[8] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr, "Profiling divergences in GPU applications," no. June 2012, pp. 775–789, 2013.

[9] D. Kirk and W.-m. Hwu, *Programming massively parallel processors : a hands-on approach LK - https://vu.on.worldcat.org/oclc/489718737*. Burlington, MA SE - xviii, 258 pages : illustrations ; 24 cm: Morgan Kaufmann Publishers, 2010.

[10] Z. Lin, X. Gao, H. Wan, and B. Jiang, "GLES: A practical GPGPU optimizing compiler using data sharing and thread coarsening," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[11] P. Hijma, R. V. van Nieuwpoort, C. J. H. Jacobs, and H. E. Bal, "Stepwise-refinement for performance: a methodology for many-core programming," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4515–4554, 2015.

[12] J. Avigad, L. D. Moura, and S. Kong, "Theorem Proving in Lean," 2017.

[13] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, 1988.

[14] P. Dybjer, "Inductive families," *Formal Aspects of Computing*, 1994.

[15] M. Carneiro, "Lean Mathlib," 2019.

[16] M. Sorensen and P. Urzyczyn, *Lectures on the Curry-Howard isomorphism*. Amsterdam Boston MA: Elsevier, 2006.

[17] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. D. Moura, "A Metaprogramming Framework for Formal Verification," vol. 1, no. September, 2017.

[18] K. Apt, A. Pnueli, F. de Boer, and E. Olderog, *Verification of Sequential and Concurrent Programs*. Texts in Computer Science, Springer London, 2009.

[19] G. Winskel, *The formal semantics of programming languages : an introduction*. Foundations of computing, Cambridge, Mass: MIT Press, 1994.

[20] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. 31, pp. 14–25, 2004.

[21] K. Kojima and A. Igarashi, "A Hoare Logic for GPU Kernels," vol. V, no. 212, 2014.

[22] G. Barthe, J. Crespo, and C. Kunz, "Unleashing relational program logics," *Software.Imdea.Org*, pp. 1–17, 2010.

[23] J. Wickerson, "Syntax and semantics of a gpu kernel programming language," *Archive of Formal Proofs*, Apr. 2014. `http://isa-afp.org/entries/GPU_Kernel_PL.html`, Formal proof development.

[24] S. Blom, M. Huisman, and M. Mihelˇ, "Science of Computer Programming Specification and verification of GPGPU programs," vol. 95, pp. 376–388, 2014.

[25] S. Blom and M. Huisman, "The vercors tool for verification of concurrent programs," in *FM*, vol. 8442 of *Lecture Notes in Computer Science*, pp. 127–131, Springer, 2014.

[26] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer, "Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels," no. 287767.

[27] G. Li, "Parameterized Verification of GPU Kernel Programs," 2013.

[28] K. Kojima and A. Imanishi, "Automated Verification of Functional Correctness of Race-Free GPU Programs," 2018.

[29] S. Tripakis, "Checking Equivalence of SPMD Programs Using Non- Interference," 2010.

[30] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, (New York, NY, USA), pp. 364–377, ACM, 2005.

[31] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Los Angeles, California), pp. 238–252, ACM Press, New York, NY, 1977.

Appendix A

Lean Variables

These variables are used in various definitions in this thesis. They were defined in the order from top to bottom.

```
variables
{ α β γ δ : Type }
{ σ : Type }
{ ι : Type }
{ τ : ι → Type }
[ decidable_eq ι ]
{ σ₁ σ₂ σ₃ : Type }
{ ι₁ ι₂ : Type }
{ τ₁ : ι₁ → Type }
{τ₂ : ι₂ → Type}
[decidable_eq ι₁]
[decidable_eq ι₂]
{ n : ℕ }
{ s t u : state n σ τ }
{ ac : vector bool n }
{ f f' : σ → bool }
{ m : memory τ }
{ i i' : ι }
{ val val' : τ i }
/- In Parlang -/
{ k₁ k₁' : kernel σ₁ τ₁ }
{ k₂ k₂' : kernel σ₂ τ₂ }
{ P Q R P' Q' I : ∀ n₁:ℕ, state n₁ σ₁ τ₁ → vector bool n₁ → ∀
↪  n₂:ℕ, state n₂ σ₂ τ₂ → vector bool n₂ → Prop }
/- In MCL -/
{ sig : signature }
{ sig₁ : signature }
{ sig₂ : signature }
{ k₁ : mclk sig₁ }
```

```
{ k₁' : mclk sig₁ }
{ k₂ : mclk sig₂ }
{ k₂' : mclk sig₂ }
{ P : parlang.state n₁ (memory $ parlang_mcl_tlocal sig₁)
 ↪  (parlang_mcl_shared sig₁) → vector bool n₁ → ∀ n₂:ℕ,
 ↪  parlang.state n₂ (memory $ parlang_mcl_tlocal sig₂)
 ↪  (parlang_mcl_shared sig₂) → vector bool n₂ → Prop }
{ P' : parlang.state n₁ (memory $ parlang_mcl_tlocal sig₁)
 ↪  (parlang_mcl_shared sig₁) → vector bool n₁ → ∀ n₂:ℕ,
 ↪  parlang.state n₂ (memory $ parlang_mcl_tlocal sig₂)
 ↪  (parlang_mcl_shared sig₂) → vector bool n₂ → Prop }
{ Q : parlang.state n₁ (memory $ parlang_mcl_tlocal sig₁)
 ↪  (parlang_mcl_shared sig₁) → vector bool n₁ → ∀ n₂:ℕ,
 ↪  parlang.state n₂ (memory $ parlang_mcl_tlocal sig₂)
 ↪  (parlang_mcl_shared sig₂) → vector bool n₂ → Prop }
{ Q' : parlang.state n₁ (memory $ parlang_mcl_tlocal sig₁)
 ↪  (parlang_mcl_shared sig₁) → vector bool n₁ → ∀ n₂:ℕ,
 ↪  parlang.state n₂ (memory $ parlang_mcl_tlocal sig₂)
 ↪  (parlang_mcl_shared sig₂) → vector bool n₂ → Prop }
```