UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

croissant.png

Msc. Thesis

Jacob Herbst (mwr148)

Verifiying eBPF programs in the Linux Kernel
Certification of Programs

Ivestigation of feasibility of checking Verification conditions for as part of the eBPF syscall

Advisor: Ken Friis Larsen
repository: github.com

2023-05-31

Abstract

This project investigates the possibility of using Proof Carrying Code (PCC) for the EBPF Linux subsystem. More specifically I focus on the validation of proofs in the context of PCC. I do so by considering the dependently typed language Logical Framework with Side Conditions (LFSC), how it can be used for proof-checking and an implementation in Rust. In doing so this project investigates both how suitable Rust is for implementing a PCC architecture that can run in the Linux kernel, aswell as considering if LFSC is at all useful when under the constraint of the Linux kernel.

Contents

1   Introduction

Extended Berkeley Package Filters (eBPF) is a subsystem in the Linux kernel, which allows users of the system to dynamically load eBPF bytecode into the kernel. The program can then be executed when certain events happen. This technology enables many interesting features, such as high-speed package filtering (which was the initial intend with the system) and Express Data Path (XDP) by circumventing the network stack of the operating system. Furthermore it can be used as system monitoring by access to kernel probes etc.

eBPF allows untrusted users to run arbitrary code in the kernel, which in itself is no problem if the program is non-malicious, but can be detremental if not. Programs that run in the Linux kernel must therefore be both safe and secure. an usafe program in the linux kernel might break the system alltogether, whilst a malicious users could get access to secrets. To prevent this the eBPF subsystem will perform an abstract interpretation of a program and then reject unsafe programs, before they are loaded into the kernel. This process is called the verifier. The verifier has been subject to multiple bugs in the past, which can lead to priveledge escalation[**TODO: ebpf priveledge**]. Furthermore the verifier will outright reject programs with loops, since the domain of the abstract interpretation have no way to tell if the program will terminate (and this ofcourse is not possible in general).

An alternative to verifying the safety of programs in general is to do a formal proof of safety. An automatic way to do this is to reduce the problem of program verification to satisfiability of programs. This is done by generating a formula that describe the properties of a program, a so called verification condition, and then checking the satisfiability of the negated formula. The process of generating a verification condition is the fairly cheap, however checking the validity of the formula can be quite a heavy task. Proof Carrying Code is an architecture and security mechanism first introduced by Necula[**necula**] in 1998 which moves the responsibility of proving that a certain program follows a set of security parameters to the user and then have the kernel check that this proof along with some additional checking. The general concept is described more in detail in**??**. All in all these tasks are far cheaper task than producing a proof.

In this project I investigate the feasibility of a proofchecker that can run inside the Linux kernel written in Rust. The proof checker leverages the Curry-howard correspondence and amounts to typechecking a dependently typed language, Logical Framework with Side Conditions (LFSC).

?? explains the necessary understanding PCC **??** explains how we enables the feature of a proof checker that can run in hte Linux Kernel. **??** gives a formal defintion of the LFSC language **??** describes the concrete syntax and gives an example **??** presents the overall design of the proof checker, whilst the concrete implementation is covered in **??**. Lastly I evaluate the performance of the proof checker and analyze the feasibility comparative to the verifier.

Enjoy.

2   **TODO** Linux

The Linux kernel is a monolithic kernel that provides not only the fundamental services necessary for a fully functioning operating system, but also a virtual interface for communication with hardware, including filesystems, network stacks, and device drivers, among others. This

interface is accessed through system calls, which serve as a standard means of communication between user space and kernel space. It is worth noting that although communication between the two spaces is primarily carried out through system calls, there exists an additional layer of abstraction in the form of communication through the filesystem.

Despite its monolithic architecture, the Linux kernel is characterized by its modular nature, allowing for the dynamic loading of kernel modules that can provide extended functionality and support for a diverse range of services. These modules function similarly to filesystems in their execution and can be loaded and unloaded as necessary.

In the following sections **??-??** we will describe what and why of ebpf and discuss the different ways to realize the PCC architecture for the eBPF subsystem.

3 TODO eBPF

The Linux kernel's ability to extend the kernel through Loadable Kernel Modules (LKMs) has been a useful feature. However, LKMs have inherent security risks and require root privileges, limiting their use to trusted users. A malicious LKM can destroy a kernel completely, especially considering that kernel modules may be proprietary. This creates a tradeoff between extensionality and trust, with LKMs providing little security.

BPF (Berkeley Package Filter) as it was originally presented is a system for effective filtering of network packages by allowing dynamically loaded package filters from userspace into the network stack in kernel space. the extended Berkeley Packet Filter (eBPF) was later introduced in the Linux kernel, offering a different approach to kernel extension. The eBPF virtual machine leverages the privileges of the kernel to oversee the entire system, enabling more powerful control. It is a just-in-time (JIT) compiled RISC instruction set running inside the kernel and it seeks to be secure by performing static analysis of the very limited language, which is desgined to not be turing complete.

The eBPF program loading process involves obtaining a program using an abstraction tool such as BCC or libbpf or writing the program by hand. The verifier then performs static analysis in the form of abstract interpretation using tristate numbers and cycle detection. The verifier checks criteria such as division by zero, no backward jumps (i.e., loops), and more. If the program is allowed by the verifier, it is JIT-compiled, making eBPF programs as fast as native code.

eBPF programs are event-driven, meaning they can be attached to a certain hook, and every time an event occurs, the program is triggered. For example, a program can be attached to a socket, and every time something is written to this socket, the program is triggered. Although an eBPF program lives in kernel space, it conceptually resides somewhere between user and kernel space. It can interact with both kernel and user space through two ways.

First, eBPF programs cannot call kernel functions directly as eBPF is designed to be kernel version agnostic (in reality, this is not always the case, due to the verifier). Instead, the eBPF subsystem provides a stable API of helper functions to provide functionality not immediately accessible in the limited instruction set. Second, the subsystem provides a collection of key-value stores called maps in a variety of different data structures such as ring buffers, arrays, etc. These data structures also reside in kernel space and are constructed through the bpf syscall,

allowing eBPF programs to read and write to a map. Users can also read and write to the maps using the bpf syscall.

3.1 The eBPF verifier

This section serves as a mapping of the eBPF verifier. The purpose of this is to serve as a basis for further discussion on why using a Proof Carrying Code approach to eBPF. The mapping is quite verbose and may not be of interest if one has a general understanding of how the verifier works.

3.1.1 The bpf syscall

All interaction between user and kernel regarding eBPF related matter uses the bpf syscall[1] and has the following signature:

```
asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Argument `cmd` is an integer that defines the intended interaction, for the purpose of this report we only care about the cmd `BPF_PROG_LOAD`. To be able to load a eBPF one of the following criteria must be met: either a program/user must be root or be bpf capable, or the kernel.unprivileged$_{\text{bpfdisabled}}$ kernel parameter must be set to 0, meaning regular users are capable of loading programs. This features has been disabled on many modern Linux distributions for security reasons, such as redhat, ubuntu and suse.[**<empty citation>**]

the `attr` argument is a union of structures that must be correspond to the argument type. For program loading this stuct notably contains the type of program to load, which could be socket programs, kernel probes, Express Data Path etc. The syscall will call the appropriate cmd after some sanity checks, such as wellformedness of the `bpf_attr` union.

3.1.2 `bfp_prog_load`

The `bpf_prog_load` will do a lot of checks related to capabilities and kernel configurations. These configurations includes memory alignment of the system etc. The specifics is irrelevant but it suggests that when designing a new BPF structure such checks should be considered.[2] For instance, we might consider a static design where only users with network capabilities might load network related eBPF or we could consider a dynamic structure where capabilities are also included in the security policy.

The program is then prepared for the verifier.[3]

---

[1]bpf() has syscall number 321

[2]An interesting sidenode is that eBPF programs must be between 1 and 1M instructions depending on user capabilities.

[3]should I specify what is going on here?

3.1.3   Static analysis `bpf_check`

The `bpf_check` is what we usually denotes as the verifier. It will perform the static analysis. Firstly the checking environment is setup. This is a large struct and thus is allocated and deallocated attached at each call to `bpf_check` and is thus also too large to see here but can be seen in appendix **??**.

1. Firstly subprograms and kernel functions[4] are added to the instructions of the ebpf program.

2. Afterwards the function `check_subprogs` is called, where some very basic testing is done, such as subprograms not being allowed to jump outside of its own address space. Control flow is here limited to subprogram and loops is in general not allowed. The last instruction must be either an exit or a jump to another subprogram.

3. Next `bpf_check` will check the control flow graph to detect loops in the code.

4. all the subprograms are then check according to their BPF TypeFormat (BTF), and the code is checked in a similar manner to the main program according to the abstract interpretation.

The following is a simplification of[**<empty citation>**]. A program must follow these requirements:

1. Registers may not be read unless they have previously been written. This is to ensure no kernel memory can be leaked.

2. Registers can either be scalars or pointers. after calls to kernel functions or when a subprogram ends, registers r1-r5 is forgotten and can then not be read before written. r6-r9 is callee saved and thus still available.

3. Reading and writing may only be done by registers marked by `ptr_to_ctx`, `ptr_to_stack` or `ptr_to_map`. These are bound and alignment checked.

4. stack space, for same reason as registers, may not be read before it has been written.

5. external calls are checked at entry to make sure the registers are appropriate wrt. to the external function.

To keep track of this the verifier will do abstract interpretation. the verification process tracks minimum and maximum values in both the signed and unsigned domain. It furthermore use `tnums` which is a pair of a mask and a value. The mask tracks bits that are unknown. Set bits in the value are known to be 1. The program is then traversed and updated modulo the instructions. For instance if register `r2` is a scalar and known to be in the range between `(0,IMAX)` then after abstractly interpreting a conditional jump `r2 > 42` the current state is split in two and the state where the condition is taken now have an updated range of `42 <= r2<= IMAX` etc. Pointers are handled in a similar manner however since pointer arithmetic is inherently dangerous modifying a pointer is very limited in eBPF. Additionally pointers may be interpreted as different types of pointers and are check wrt. the program type they occur in. For instance... TBD...

---

[4]why does ebpf.io say no kernel functions?

If all these requirements are met, then an eBPF program is loaded. This mapping ofcause is simplified a lot, but it shows that the current process of checking a valid eBPF program is not a simple task and thus a potential overhaul could be welcomming. The entire verification process (except for a few structs) is placed in `kernel/bpf/verifier.c` which at the time of writing is roughly 19000 lines of code, and these have in the past shown to errorprone.

3.2 eBPF and PCC

From the description of PCC in **??** and the description of the eBPF subsystem above, it is straightforward to see responsibility differences. We can compare the two pipelines as follows:

1. **Compilation and Cetification**: For PCC the untrusted program is both compiled and a certificate for safety policy compliance is generated. eBPF does not really "do" anything at this stage as source code is passed directly to the kernel using the syscall.

2. **Verification of certificate**: In PCC the consumer will check the validity of the certification wrt. the safety policy and the source program (possibly in native format), while eBPF will have to do a similar check but directly on the eBPF program. As mentioned the current eBPF verifier uses a abstract interpretation model with a tristate number domain, which is roughly equivalent in complexity.[5]

3. In both structures, once a certificate is checked the program is free to use possibly many times.

So why would we want to swap out the current structure vs eBPF?

4 **TODO** Proof Carrying Code

Proof Carrying Code (PCC) is a mechanism designed to ensure the safe execution of programs from untrusted sources.

The PCC architecture can be divided into two spaces, namely the code producer and the code consumer.

The code producer aims to execute some code at the expense of the code consumer. In case there is not perfect trust between code consumer and code producer, the code consumer must protect itself from potentially malicious or unsafe programs. To achieve this, the code consumer issues a collection of safety rules, referred to as the safety policy, which the code producer must adhere to. Depending on the specific domain, the safety policy could include constraints such as no out-of-bounds memory operations and termination of loops and the likes that might require programs to not corrupt the consumer. The code producer then uses the safety policy to certify the program in a compilation stage of the untrusted code. The certification process produces both the native code to be executed at the consumer and the certificate for the safety of the native code. This process can potentially be computationally heavy, and it is preferable for the code producer to perform the certification and compilation.

---

[5]is this even true? LOOK at verifier source code.

The next stage is the verification phase, where the producer hands over the results of the certification process to the consumer. The consumer then checks the validity of the proof in two ways. First, the safety proof must be valid modulo the safety policies, and second, the safety proof must correspond to the native code. This process should be quick and follow an algorithm trusted by the consumer. If both criteria are met, the consumer can mark the native code as safe and proceed to execute the program, possibly multiple times.

5   **TODO** Rust - In the kernel?

Rust is a modern systems programming language that was first introduced in 2010. It was designed to address common issues faced by developers in writing low-level, high-performance code, such as memory safety and thread safety.

One of the key features of Rust is its ownership model, which ensures that memory is managed efficiently and safely. Rust's ownership model is based on the concept of ownership and borrowing, which allows the compiler to track the lifetime of objects and manage memory without the need for a garbage collector. This ensures that common issues like null pointer dereferences, use-after-free errors and buffer overflows can occur, while at the same time being comparable to C in performance.

Rust's syntax is similar to that of C and C++, but it also includes modern language features like pattern matching, closures, and iterators. Rust also has a strong focus on performance and optimization, which makes it an ideal language for building high-performance applications and systems.

The borrow checker is a form of static analysis which ensures that a program complies with the ownership rules. The rules are 3 fold.

- Each value has an owner.

- There can only be one owner for each value.

- When the owner goes out of scope, its values are freed (dropped in Rust terminology).

if we for instance consider:

```rust
fn main() {
let x : i32 = 42;
}
```

then the value of 42 has an owner x. When the main() function ends then the owner x goes out of scope and the value is dropped. This specific type of x is i32 and will thus reside on the stack, however if we needed something that was heap allocated then we could create a value like:

```rust
fn main() {
let y : Box<i32> = Box::new(42);
}
```

then `y` will be a `Box` type, which is the simplest form of heap allocation. a call to `drop` will be performed. There are then 3 ways in which ownership can be transfered by either `move`, `copy` or `clone`. For primitive types (those that usually reside on the stack) can be copied from one variable to another, while the heap allocated values can be either `cloned`, which essentially works as a `mempy` or by moving it, this means that the snippet below will get a compile time error at line 6, because the variable that owns the box containing 42 is no longer owned by y.

```rust
fn main() {
let y : Box<i32> = Box::new(42);
let z = y.clone();
assert_eq!(y, z);
let a = y;
assert_eq!(y, z);
}
```

However cloning can be pretty inefficient and should most often be avoided, however Rust allows borrowing of values. Borrowing literally describes the action of receiving something with the promise of return. When borrowing a value the memory address of the value is referenced, by an & and is essentially just a pointer. It is however worth noting that a reference cannot be `NULL` and thus by mere construction eliminates `NULL` pointer problems. There are two types of references, exclusive and shared references. Exclusive references can mutate the borrowed value, while a shared reference may only read the reference. There are 3 rules that borrowing is subject to:

- There can exist a single exclusive reference or multiple shared references at a given time

- References must always be valid, which mean it is impossible the borrow a value after its owner has gone out of scope.

- A value cannot be modified whilst referenced.

These rules invariantly ensure that a reference is always as it seems to the borrower. If multiple mutable borrows were allowed at the same time or even at the same time as a shared reference, then one of the mutable borrows may destroy the reference for the others. An example where this is extremely obvious is when we consider a reference to a datastructure that might need to be reallocated, such as dynamic arrays. In such a situation the address of the old array will no longer be valid. This is also ensured by rule 3. This is what constitutes the basics of the Rust programming language and this different memory model has made Rust gain popularity in recent years, especially in the systems programming community, due to its combination of performance, safety, and ease-of-use. This popularity also includes the Linux Development community, where safety, security and reliability is mission critical. As of kernel 6.1 Rust is officially supported in the kernel (albeit fairly limited as described throughout this report). Because of these promises I wanted to see if Rust would be a suitable language to use for an in kernel proof checker

6 **TODO** Curry Howard Isomorphism

7 Deciding on a format

As mentioned in section**??** the code producer must hand over code in the appropriate format, as well as the certificate, so we must consider an appropriate format for the certificate. The process of checking the certificate should also be both fast and simple. I consider a PCC architecture which uses Verification condition generation on the eBPF most likely by using a derivation of Hoare Logic, since this propose an automatic way of generating formulas, which describes programs.[6] The process of proving the validity of such a verification condition however is not a simple task and is, dependending on the logic, undecidable. We will therefore require the code consumer to rely on the process of making this proof and then present the proof to the kernel. Checking the satisfiability of a formula can be done by a Satisfiability Modulo Theories (SMT) solver. In many SMT solvers it is possible to extract a proof that a certain formula is satisfiable. I have in this work considered two output formats/languages, Alethe and Logical Framework with Side Conditions (LFSC), supported by the CVC5 SMT solver. I will briefly describe why I have chosen to use the LFSC language over Alethe. Both formats follow the LISP family of languages and is therefore simple to parse. Alethe is designed to be easily readable by humans and is strutured as a box style proof.[7] This is not a property necessary for a PCC architecture where we want to automate the entire process. By this construction the Alethe format provides a set of basic inference rules of which there are 91[**<empty citation>**] on which proofs are built, for instance rule 20 denotes reflexivity often denoted as `refl` which describes syntactical equality between two terms modulo renaming of bound variables. This entails that an implementation must implement all rules necessary for a security policy and will thne not be easily extended in the future. LFSC on the otherhand is a metaframework that exploits the Curry-Howard isomorphism, explained in **??**. This metaframework allows for the security policy to be established by signatures, which can easily be extended. This is very much a property of interest, as eBPF might evolve to support new datastructures etc. Furthermore this approach can move bugs out of the in kernel certifier and into the specification. This will enable system administrators to quickly deploy fixes for a bug, by not allowing specific faulty signatures. It is hard to consider both time and memory used of the two languages, without actually implementing both of them. But if we consider the amount of code require for the two formats then there exists a rust implementation for an Alethe proof checker, called carcara[**<empty citation>**]. This implementation is ~13500 lines of code, and this specific implementation does not even support all theories present in CVC5, such as bitvectors. LFSC has a proof checker written in C++ and its code is merely 5000 lines, while the signatures may be arbitrarily large. The solution I present is ~3000 lines. CVC5 provides some signatures constituting <1500 lines. Hence LFSC seems like a better choice for an in kernel proof checker.

8 Logical Framework with Side Conditions

LFSC is a extention of the Edinbourgh Logical Framework (LF)[**<empty citation>**] and is a predicative typed lambda calculus with dependent types. This allow proof systems to be encoded as signatures, which amounts to a set of typing declarations. LFSC extends LF by

---

[6]should i go more indepth with this?
[7]example?

including sideconditions. In this context sideconditions is a operational semantic which might be evaluated during typechecking. I will in the following section describe the calculus and its typing along with the operational semantic of sideconditions.

8.1 Abstract Syntax

The syntax has 5 categories. At its core LFSC is a typed lambda calculus, so it consists of terms/objects, types and kinds. Terms are denoted by M, N and O. Types are denoted by A and B and used for classification of Terms. Kinds are denoted by K and is used to classify types. I use x to be a metavariable ranging over the set of variables that might occur in terms, c to denote constants in terms (free variables). a will range over constants in types. Sideconditions is denoted by S and T. Patterns describes pattern in side condition match cases and is denoted by P. I write keywords in **bold**. I write $[M_1/x_{1,\ldots,}M_n/x_n]N$, for simultanious substitution of M_1, M_2 , .. with free ocurrences of x_1, ~x_2 ~,... in N. We assume renaming, to avoid name clashing. I use * to denote a hole.

Figure **??** shows the syntactical categories. TODO: make the figure.

Terms maybe be either a constant c, a variable x, a type annotation M : A stating that term M must have type A, an abstraction λ x [:M] . N, which binds the term M to x in N, and lastly a term might be an application of term M to N, notice that application in terms is left-associative, so M N O is (M N) O.

Types may be a type constant a, an application of a type to a term, A M. a lambda abstraction ..., or a dependent product type. Pi types may contain sidecondtions in their binders.

Kinds may be either the **type** which classifies types or they may be a pi type.

Sideconditions may be:

1. an unbounded number, which may be either an integer or a rational.

2. a variable x

3. a **let** binding, setting x to S in T.

4. an application. Notice that we require that functions are fully applied and thus non-associative,

sideconditions may then also be one of the categories Compounds, Numerical and Sideeffects.

Compound sideconditions may be either a **fail** A, which raises an exception with type A, **match** expressions, which will match the scrutinee against patterns and evaluate corrsponding S_i in a similar manner to ML. **ifequal** compares S and T for syntactical equality.

Numerical sideconditions may be the binary operation addition, multiplication and division, these follow standard conventions. It may be a negation, a convertion from integer to rational or it may be one of the two branching constructs.

Sideeffects may be **ifmarked** a branch based on the marking of a varible. markvar which marks a variable, **do** S T, which is equivalent to **let** x = S T where x does not occur in T.

Patterns may be either a constructor applied to 0 or more arguments, but the constructor must always be fully applied.

8.2 Signatures and Contexts

In LFSC there is two construct we use to keep track of variables and constants. We have signatures, and contexts. Signatures are used to assign kinds and types to constants. This is what the metaframework revolves around. Contexts are used to assign types to variables. we write them as such and use Σ, Σ' to denote the concatenation of the two signatures Σ and Σ' and similarly for contexts.

$$\Sigma ::= \langle\rangle \mid \Sigma, a : K \mid \Sigma, c : A$$
$$\Gamma ::= \langle\rangle \mid \Gamma, x : A$$

The typesystem of LFSC is syntax directed meaning there we have only a single typing rule for each syntactical object. We achieve this by bidirectional typing. That means instead of stating that an expression must have a type, we can either construct a type from it (called inference) or we can check that an expression has a type.

$$\Sigma \checkmark \qquad (\Sigma \text{ is a valid signature})$$
$$\vdash_\Sigma \Gamma \qquad (\Gamma \text{ is a valid context in } \Sigma)$$
$$\Gamma \vdash_\Sigma K \qquad (K \text{ is a kind in } \Gamma \text{ and } \Sigma)$$
$$\Gamma \vdash_\Sigma M \Leftarrow A \qquad (M \text{ can be checked to have type } A \text{ in } \Gamma \text{ and } \Sigma)$$
$$\Gamma \vdash_\Sigma M \Rightarrow A \qquad (M \text{ can be synthesized to have type } A \text{ in } \Gamma \text{ and } \Sigma)$$

Valid signatures:

$$\text{EMPTY-SIG} \frac{}{\langle\rangle \checkmark}$$

$$\text{KIND-SIG} \frac{\Sigma \checkmark \qquad \vdash_\Sigma K \qquad a \notin dom(\Sigma)}{\Sigma, a : K \checkmark}$$

$$\text{TYPE-SIG} \frac{\Sigma \checkmark \qquad \vdash_\Sigma A : K \qquad c \notin dom(\Sigma)}{\Sigma, c : A \checkmark}$$

Figure 1: Valid signatures

Valid contexts:

8.3 Typing

For brevity we simply use $\vdash$ instead of $\vdash_\Sigma$ when its meaning is obvious.[8]

---

[8]what is the right word here?

$$\text{EMPTY-CTX} \; \frac{\Sigma \checkmark}{\vdash_\Sigma \langle\rangle}$$

$$\text{TYPE-CTX} \; \frac{\vdash_\Sigma \Gamma \qquad \Gamma \vdash_\Sigma A : Type \qquad x \notin dom(\Sigma)}{\vdash_\Sigma \Gamma, x : A}$$

Figure 2: Valid contexts

$$\text{TYPE} \; \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type} \Rightarrow \mathbf{kind}} \qquad \text{TYPEc} \; \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type^c} \Rightarrow \mathbf{kind}}$$

$$\text{LOOKUP-CTX} \; \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

$$\text{LOOKUP-KIND-SIG} \; \frac{\vdash \Gamma \qquad a : K \in \Sigma}{\Gamma \vdash a \Rightarrow K} \qquad \text{LOOKUP-TYPE-SIG} \; \frac{\vdash \Gamma \qquad c : A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$$

Figure 3: Typing rules for looking up types.

$$\text{ANN} \; \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash M : A \Rightarrow A}$$

$$\text{PI} \; \frac{\Gamma \vdash A \Leftarrow \mathbf{type} \qquad \Gamma, x : A \vdash B \Rightarrow \alpha \qquad \alpha \in \{\mathbf{type}, \mathbf{type^c}, \mathbf{kind}\}}{\Gamma \vdash \Pi x : A.\ B \Rightarrow \alpha}$$

$$\text{PI-SIDE} \; \frac{\Gamma \vdash S \Rightarrow A \qquad TODO \Rightarrow A \qquad \Gamma, x : A \vdash B \Rightarrow \mathbf{type}}{\Gamma \vdash \Pi x : \{S\ T\}.\ B \Rightarrow \mathbf{type^c}}$$

$$\text{TYPE-APP} \; \frac{\Gamma \vdash A \Rightarrow \Pi x : B.\ K \qquad \Gamma \vdash M \Leftarrow B}{\Gamma \vdash AM \Rightarrow [M/x]K}$$

$$\text{APP} \; \frac{\Gamma \vdash M \Rightarrow \Pi x : A.\ B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash MN \Rightarrow [M/x]N}$$

$$\text{APP-SC} \; \frac{\Gamma \vdash M \Rightarrow \Pi x : \{S\ T\}.\ B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash MN \Rightarrow [M/x]TODO}$$

$$\text{LAMANN} \; \frac{\Gamma \vdash A \Rightarrow \mathbf{type} \qquad \Gamma, x : A \vdash M \Rightarrow B}{\Gamma \vdash \lambda x : A.\ M \Rightarrow \Pi x : A.\ B}$$

$$\text{LAM} \; \frac{\Gamma, x : A \vdash M \Rightarrow B}{\Gamma \vdash \lambda x.\ M \Leftarrow \Pi x : A.\ B}$$

Figure 4: Bidirectional typing rules for LFSC

8.3.1 Operational Semantics of Sideconditions

Errors are not included in the operational semantics. Errors might occur when **fail** is evaluated, a match does not match any pattern, if **markvar** or **ifmarked** does not evaluate to a variable, if division by 0 is reached or a free identifier is found.

$$\text{IFMARKED} \quad \frac{\Gamma \vdash S \Rightarrow A \quad T_1 \Rightarrow B \quad T_2 \Rightarrow B}{\Gamma \vdash \mathbf{ifmarked}\ n\ S T_1 T_2 \Rightarrow B}$$

$$\text{MARKVAR} \quad \frac{\Gamma \vdash S \Rightarrow A}{\Gamma \vdash \mathbf{markvar}\ n\ S \Rightarrow A}$$

$$\text{LET} \quad \frac{\Gamma \vdash S \Rightarrow A \quad \Gamma, x : A \vdash T \Rightarrow B}{\Gamma \vdash \mathbf{let}\ x\ S\ T \Rightarrow B}$$

$$\text{DO} \quad \frac{\Gamma \vdash S \Rightarrow A \quad \Gamma \vdash T \Rightarrow B}{\Gamma \vdash \mathbf{do}\ S\ T \Rightarrow B}$$

Figure 5: Typing rules for sideeffects

$$\text{PROGAPP} \quad \frac{\begin{array}{c}\Gamma \vdash f \Rightarrow \{x_1 : A_1, \ldots, x_1 : A : A_n\}.S \\ \forall i \in \{1, \ldots, n\}.(\Gamma \vdash T_i \Leftarrow A_i) \\ \Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash S \Rightarrow B\end{array}}{\Gamma \vdash f(T_1, \ldots, T_n) \Rightarrow B}$$

$$\text{MATCH} \quad \frac{\Gamma \vdash S \Rightarrow A \quad \forall i \in \{1, \ldots, n\}.(\Gamma \vdash P_i \Rightarrow A \quad \Gamma, \mathbf{ctx}(P_i) \vdash T_i \Rightarrow B)}{\Gamma \vdash \mathbf{match}\ S\ (P_1, T_1) \ldots (P_n, T_n) \Rightarrow B}$$

$$\text{IFEQ} \quad \frac{\Gamma \vdash S_1 \Rightarrow A \quad \Gamma \vdash S_2 \Rightarrow A \quad \Gamma \vdash T_1 \Rightarrow B \quad \Gamma \vdash T_2 \Rightarrow B}{\Gamma \vdash \mathbf{ifeq}\ S_1\ S_2\ T_1\ T_2 \Rightarrow B}$$

$$\text{FAIL} \quad \frac{\Gamma \vdash A \Rightarrow \mathbf{type}}{\Gamma \vdash \mathbf{fail}\ A \Rightarrow A}$$

Figure 6: Typing rules for compound sideconditions

$$\text{INT} \quad \frac{}{\Gamma \vdash n \Rightarrow \mathbf{integer}}$$

$$\text{NEG} \quad \frac{\Gamma \vdash S \Rightarrow \mathbf{integer}}{\Gamma \vdash -S \Rightarrow \mathbf{integer}}$$

$$\text{Z-TO-Q} \quad \frac{\Gamma \vdash S \Rightarrow \mathbf{integer}}{\Gamma \vdash \mathbf{ztoq}\ S \Rightarrow \mathbf{rational}}$$

$$\text{BINOP} \quad \frac{\Gamma \vdash S \Rightarrow \mathbf{integer} \quad T \Rightarrow \mathbf{integer}}{\Gamma \vdash S \oplus T \Rightarrow \mathbf{integer}} \oplus \in \{+, *, /\}$$

$$\text{IFNEG} \quad \frac{\Gamma \vdash S \Rightarrow \mathbf{integer} \quad \Gamma \vdash T \Rightarrow A \quad \Gamma \vdash U \Rightarrow A}{\Gamma \vdash \mathbf{ifneg}\ S\ T\ U \Rightarrow A}$$

$$\text{IFZERO} \quad \frac{\Gamma \vdash S \Rightarrow \mathbf{integer} \quad \Gamma \vdash T \Rightarrow A \quad \Gamma \vdash U \Rightarrow A}{\Gamma \vdash \mathbf{ifzero}\ S\ T\ U \Rightarrow A}$$

Figure 7: Typing rules for numerical sideconditions

8.4 Concrete Syntax

8.4.1 Concrete syntax of LFSC

In this section I present the concrete syntax of LFSC.

$\langle commands \rangle ::= ('(' \langle command \rangle ')')^*$

$$\text{CST-O} \; \frac{}{\sigma_1; c \downarrow c; \sigma_1} \qquad \text{VAR-O} \; \frac{}{\sigma_1; x \downarrow x; \sigma_1} \qquad \text{NUM-O} \; \frac{}{\sigma_1; r \downarrow r; \sigma_1}$$

$$\text{CST-APP} \; \frac{\forall i \in \{1, \ldots, n\}.(\sigma_i; S_i \downarrow S'_i; \sigma_{i+1})}{\sigma_1; (c\ S_1 \ldots S_n) \downarrow (c\ S'_1 \ldots S'_n); \sigma_{n+1}}$$

$$\text{LET-O} \; \frac{\sigma_1; S \downarrow S'; \sigma_2 \quad \sigma_2; [S'/x]T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{let}\ x\ S\ T) \downarrow T'; \sigma_3} \qquad \text{DO-O} \; \frac{\sigma_1; S \downarrow S'; \sigma_2 \quad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{do}\ S\ T) \downarrow T'; \sigma_3}$$

$$\text{IFEQUAL-T} \; \frac{\sigma_1; S_1 \downarrow S'_1; \sigma_2 \quad \sigma_2; S_2 \downarrow S'_2; \sigma_3 \quad S'_1 \equiv_{\beta\eta} S'_2 \quad \sigma_3; T_1 \downarrow T'_2; \sigma_4}{\sigma_1; (\textbf{ifequal}\ S_1\ S_2\ T_1\ T_2) \downarrow T'_2; \sigma_4}$$

$$\text{IFEQUAL-F} \; \frac{\sigma_1; S_1 \downarrow S'_1; \sigma_2 \quad \sigma_2; S_2 \downarrow S'_2; \sigma_3 \quad S'_1 \not\equiv_{\beta\eta} S'_2 \quad \sigma_3; T_1 \downarrow T'_2; \sigma_4}{\sigma_1; (\textbf{ifequal}\ S_1\ S_2\ T_1\ T_2) \downarrow T'_2; \sigma_4}$$

$$\text{MATCH-O} \; \frac{\sigma_1; S \downarrow (c\ S_1 \ldots S_m); \sigma_2 \quad \exists i.P_i = (cx_1 \ldots x_m) \quad \sigma_2; [S'_1/x_1, \ldots, S_m/x_1]T_i \downarrow T'; \sigma_3}{\sigma_1; (\textbf{match}\ S\ (P_1 T_1) \ldots (P_n T_n)) \downarrow T'; \sigma_3}$$

$$\text{FUN-APP} \; \frac{\begin{array}{c} \forall i \in \{1, \ldots, n\}.(\Delta \vdash \sigma_i; S_i \downarrow S'_i; \sigma_{i+1}) \\ (f(x_1 : A_1 \ldots x_n : A_n) : B = T) \in \Delta \quad \Delta \vdash \sigma_{n+1}; [S'_1/x_1, \ldots S'_n/x_n]T \downarrow T'; \sigma_{n+2} \end{array}}{\sigma_1; (f\ S_1 \ldots S_n) \downarrow T'; \sigma_{n+2}}$$

$$\text{BINOP-O} \; \frac{\sigma_1; S \downarrow r_1; \sigma_2 \quad \sigma_2; T \downarrow r_2; \sigma_3 \quad r = r_1 \oplus r_2}{\sigma_1; S \oplus T \downarrow r; \sigma_3} \oplus \in \{+, *, /\}$$

$$\text{NEG-O} \; \frac{\sigma_1; S \downarrow r; \sigma_2}{\sigma_1; -S \downarrow r; \sigma_2} \qquad \text{ZTOQ-O} \; \frac{\sigma_1; S \downarrow z; \sigma_2 \quad r = z/z}{\sigma_1; \textbf{ztoq}\ S \downarrow r; \sigma_2}$$

$$\text{IFNEG–T} \; \frac{\sigma_1; S \downarrow r; \sigma_2 \quad r < 0 \quad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{ifneg}\ S\ T\ U) \downarrow T'; \sigma_3} \qquad \text{IFNEG–F} \; \frac{\sigma_1; S \downarrow r; \sigma_2 \quad r \geq 0 \quad \sigma_2; U \downarrow U'; \sigma_3}{\sigma_1; (\textbf{ifneg}\ S\ T\ U) \downarrow U'; \sigma_3}$$

$$\text{IFZERO–T} \; \frac{\sigma_1; S \downarrow r; \sigma_2 \quad r = 0 \quad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{ifzero}\ S\ T\ U) \downarrow T'; \sigma_3} \qquad \text{IFZERO–F} \; \frac{\sigma_1; S \downarrow r; \sigma_2 \quad r \neq 0 \quad \sigma_2; U \downarrow U'; \sigma_3}{\sigma_1; (\textbf{ifzero}\ S\ T\ U) \downarrow U'; \sigma_3}$$

$$\text{IFMARKED–T} \; \frac{\sigma_1; S \downarrow x; \sigma_2 \quad \sigma_2 x \quad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{ifmarked}\ S\ T\ U) \downarrow T'; \sigma_3} \qquad \text{IFMARKED–F} \; \frac{\sigma_1; S \downarrow x; \sigma_2 \quad \neg\sigma_2 x \quad \sigma_2; U \downarrow U'; \sigma_3}{\sigma_1; (\textbf{ifmarked}\ S\ T\ U) \downarrow U'; \sigma_3}$$

$$\text{MARKVAR-O} \; \frac{\sigma_1; S \downarrow x; \sigma_2}{\sigma_1; (\textbf{markvar}\ S) \downarrow x; \sigma_2[x \mapsto \neg\sigma_2 x]}$$

Figure 8: Operational semantics for side conditions

$$
\begin{array}{lll}
\langle \textit{command} \rangle & ::= & \text{'declare'}\ \langle \textit{ident} \rangle\ \langle \textit{type} \rangle \\
& | & \text{define}\ \langle \textit{ident} \rangle\ \langle \textit{term} \rangle \\
& | & \text{check}\ \langle \textit{term} \rangle \\
& | & \text{program}\ \langle \textit{args} \rangle\ \langle \textit{type} \rangle\ \langle \textit{sidecondition} \rangle \\
& | & \text{function}\ \langle \textit{args} \rangle\ \langle \textit{type} \rangle\ \langle \textit{sidecondition} \rangle \\
& | & \text{run}\ \langle \textit{sidecondition} \rangle \\
\\
\langle \textit{term} \rangle & ::= & \langle \textit{literals} \rangle \\
& | & \text{'('}\ \langle \textit{compound\_term} \rangle\ \text{')'}
\end{array}
$$

$$\langle \textit{literals} \rangle \quad ::= \ \langle \textit{integer} \rangle$$
$$\mid \ \langle \textit{rational} \rangle$$
$$\mid \ \langle \textit{ident} \rangle$$
$$\mid \ \langle \_ \rangle$$

$$\langle \textit{compound\_term} \rangle ::= \ \text{'@'} \ \langle \textit{ident} \rangle \ \langle \textit{term} \rangle \ \langle \textit{term} \rangle$$
$$\mid \ (\text{'!'} \mid \text{'forall'}) \ \langle \textit{ident} \rangle \ \langle \textit{type} \rangle \ \langle \textit{term} \rangle$$
$$\mid \ \text{':'} \ \langle \textit{type} \rangle \ \langle \textit{term} \rangle$$
$$\mid \ \text{''} \ \langle \textit{ident} \rangle \ \langle \textit{type} \rangle \ \langle \textit{term} \rangle$$
$$\mid \ (\text{'Í '} \text{'lam'}) \ \langle \textit{ident} \rangle \ \langle \textit{term} \rangle$$
$$\mid \ \text{'}\langle \textit{sidecondition} \rangle \langle \textit{sidecondition} \rangle \text{'('} \langle \textit{term} \rangle \text{')'}$$

$$\langle \textit{sidecondition} \rangle ::= \ \langle \textit{literals} \rangle$$
$$\mid \ \text{'(' compound\_sidecondition ')'}$$
$$\mid \ \text{'(' numeric ')'}$$
$$\mid \ \text{'(' sideeffect ')'}$$
$$\mid \ \text{'(' 'let'} \ \langle \textit{ident} \rangle \ \langle \textit{sidecondition} \rangle \ \langle \textit{sidecondition} \rangle \ \text{')'}$$

9 The in-kernel proof checker

In this section i present some implementation specific design decisions I have taken to reduce the amount of memory needed, the efficiency of the typechecking algorithm and some pure restrictions posed by the current status of Rust in the kernel. Duely note however than my current implementation does not actually run inside the kernel but rather is subject to experiments I have done along the way.

The implementation uses normalization by evaluation with De Bruijn indices and explicit substitutions.

9.0.1 De Bruijn Indicies

The reason I have decided to use De Brujin indices is two-fold. First, using debrujin indices uses less memory than explicit naming, the improvement might be negliable but is there nonetheless and secondly, using De Bruijn indices allows for easier consideration of α-equivalence between two terms, meaning that they have the same meaning, since α-equivalence amounts to syntactical equivalence with De Bruijn indices. Consider the types: $(\lambda x.\lambda y.xy)y$ then if we were to do direct substitution in $[x/y](\lambda y.xy)$ we would get $\lambda y.yy$ which changes the meaning of the term. De Bruijn indices instead swap each bound variable with a positive integer. The meaning of the integer n is then constituted by the $n^t h$ enclosing abstraction, Π or λ. This further reduce the need for a binding name. Specifically if we have the following $\lambda x.\lambda y.\lambda y.xy$ and $\lambda y.\lambda x.\lambda x.yx$ which are alpha-equivalent since they are the same function, but not syntactically identical. however in using De Bruijn notation we get: $\lambda\lambda 20$ for both, since the outermost binder in the inner application is described by the outermost lambda, whilst the argument for the application is 0, since it is captured by the innermost abstraction. Hence we see that they are now syntactical equivalent and capture avoiding since $(\lambda\lambda 10)y$ reduces to $\lambda\lambda 1y$. I will only consider De Bruijn notation for bound variables, this way we get around any capture avoiding complications. We could potentially also consider using De Bruijn indices

for free variables, however this would complicate the code as this would require lifting binders. TODO: give an example. Although it could be interesting to consider De Bruijn levels since these are not relative to the scope.

We also consider De Bruijn indices for other binders such as in the program definitions, for instance a program like:

```
(function sc_arith_add_nary ((t1 term) (t2 term)) term
  (a.+ t1 t2))
```

will get converted into

```
(function sc_arith_add_nary ((term) (term)) term
  (a.+ 1 0))
```

We can similarly do the same for pattern matching, when a constructor is applied to multiple arguments.

```
(match t
  ((apply t1 t2)
    (let t12 (getarg f t1)
      (ifequal t12 l tt (nary_ctn f t2 l)))))
  (default ff))
```

gets converted into:

```
(match t
  ((apply 2)
    (let (getarg f 1)
      (ifequal 0 l tt (nary_ctn f 1 l)))))
  (default ff))
```

Notice here that the arguments `t1` and `t2` is substituted by a 2, we need to save the number of arguments to not lose the meaning of the constructor, as they must be fully applied. In the example we likewise eliminated the binder of the "let".

9.0.2 Explicit substitutions

We have already touched upon substitution, but another matter at which we shall consider it is the sheer cost of direct substitution. When doing direct substitution on terms we cause an explosion in size of the term and thus wastes memory and execution time because we have to copy the struct at each occurence. In the substitution $[M/x]N$ then if M is large or if x occurs many times in N we can potentially generate a new term which are exponentially bigger than the two terms to begin with. Considering explicit substitution on the other hand allow us to not generate anything enecessarily large and keep the computation at a minimum. I consider a substitution which is lazy, meaning we use the result of a substitution when necessary and then proceed. TODO: example. I will not go into detail about how the abstract syntax and type system looks with explicit substitution.

9.0.3 Normalization by evaluation

9.1 Reading and formatting proofs

Section **??** describes the concrete syntax of LFSC, proofs generated by CVC5 will be in this format, however by the reasoning above we would prefer the format to be using De Bruijn indices. Therefore I propose a interface which is split in two. As presented in Figure **??**, we have a parser, converter and formatter pipeline in userspace and then we have a parser to get the correct form in kernel space. The parser in user space will parse the concrete syntax. The converter will then α-convert the AST and lastly a converter can realize the converted ast. This convertion could be a pretty printer or a serializer into some specific format that can easily be deserialized. This structure gives more leeway in terms of structure. For instance the Kernel can be picky about arbitrary nested parenthesis making it less errorprone to stack overflows, (In reality, the current implementation is stack safe wrt. nested parenthesis). I have looked into using a zero copy serialization framework, however i have not found one that has been easily usable in the kernel.

My first implementation was a handwritten lexer and recursive descent parser, however this implementation quickly got scrapped, when realizing how crates can be used in the Rust kernel development.

9.1.1 What restrictions is imposed by the Rust kernel?

In the Rust kernel development framework not a lot of functionality is exposed. The crates immediately exposed in the kernel is `alloc`, `core`, `kernel`, `compiler_builtins` and `macros`. The `macros` crate is tiny and exposes the ability to easily describe a LKM metadata. The `compiler_builtins` are a compiler built in functionality which usually resides in the standard library `std`. The builtins supported in the kernel at the moment is nothing more than a way to handle panics (exceptions). The `kernel` crate exposes the kernel APIs, such as character devices, file descriptors etc. The functionality of this crate is mostly intended for use in LKMs, which for time being is the inteded use for Rust. Rust is not considered to be part of the core kernel, which need to communicate which each other but rather for "leafs" in the kernel hierachy. The `alloc` and `core` crates constitutes most of the `std` library in Rust and is respectively the implementation of a memory allocator and core functionality. The `alloc` and `core` crates are often in embedded system and others where the is no operating system or kernel to provide the functionality of the standard library. The `core` crate exposes basic functionality such as primitive types, references etc. The `alloc` crate exposes memory allocations and in userspace uses some exposure of malloc, while in kernel space may use either `kmalloc` or `kvmalloc` to allocate physical and virtual memory inside the kernel. In its current form the `alloc` crate does not provide much functionality. Only simple allocation types such as `Box` are exposed and their API is conservative. The reason behind is that the kernel "apparently" has no way to handle Out-Of-Memory cases.[9] Thus most datastructures are simply not allowed, because they dont expose a secure way to allocate memory. Whenever a new allocation need to happen a `try_new()` function can be called, which will return a `Result` type with either a reference or an error. The only modifiable datastructures available is `Vec`, a dynamic array, this might take a toll on the performance. A discussion on the matter is presented in Section **??**. Furthermore the `alloc` crate is compiled with a `no_rc` feature

---

[9]What about the OOM killer?

meaning there is no way to use the reference counted pointers defined in Rust, because the maintainers of the Rust functionality in Linux have decided that it is unnecessary since the C part of the kernel already defines reference counting. To the best of my knowledge there is no clear exposure of this functionality however in any of the currently supported crates. It is however fairly easy to remove this restriction.

It is possible to compile crates that support a `no_std` feature (it relies on `alloc` and `core`) and that also does no memory allocations. From my investigation I have found the parser combinator library `nom` to be compilable in the kernel. I use this library for my parser.

9.2   Abstract syntax in Rust

Despite being similar to C and CPP in syntax, Rust provides a much richer typesystem that allow us to create enumerations which has fields aka Sum types. We might for instance define a construction for Identifiers as such:

```rust
pub enum Ident<Id> {
  Symbol(Id),
  DBI(u32)
}
```

An identifier can either be a Symbol if it is free or a De Bruijn index if it is bound. Terms are then defined almost identical to constructs described in **??**. The major difference comes from the way we represent binders.

```rust
pub enum BinderKind {
  Pi,
  Lam,
}
pub enum Term<Id> {
  Binder{ kind: BinderKind, var: Id,
          ty: Option<Box<Type<Id»>,
          body: Box<Term<Id» },
  // rest of terms
}
```

A binder is either a Π type or a λ abstraction, that abstract the var in the body. We use an option type as λ abstractions might contain an annotation but can have an annonymous type aswell. This structure is convenient in the frontend representation of the language as this allow for simpler α-normalization. In the backend language we however, split this structure into seperate constructors of the `AlphaTerm` enum.

```rust
pub enum AlphaTerm<Id> {
    Number(Num),
    Hole,
    Ident(Ident<Id>),
    Pi(Box<AlphaTerm<Id», Box<AlphaTerm<Id»),
```

```
        Lam(Box<AlphaTerm<Id»),
        AnnLam(Box<AlphaTerm<Id», Box<AlphaTerm<Id»),
        Asc(Box<AlphaTerm<Id», Box<AlphaTerm<Id»),
        SC(AlphaTermSC<Id>, Box<AlphaTerm<Id»),
        App(Box<AlphaTerm<Id», Box<AlphaTerm<Id»),
}
```

We define a similar structure for the rest of the language. We parameterize `AlphaTerm` by `Id` which is the data representation of symbols. In the specific implementation we consider a `&str`, which is a reference to a fixed sized string. We use this type over a `String` type because it is more efficient and there is no need for a term to own the string. Having terms parameterized by the Identifer type allow for easily convertion to using De Bruijn levels instead of string identifiers.

9.2.1 Parsing lFSC

We use `nom` for parsing. `nom` is a parser combinator library that has evolved over the years from being mainly driven by macros to in version 7 using composable closures. It is mainly focused around parsing bytes and hereby also `str`. The interfacing is a little confusing at times because there are many ways to call and compose parsers. I have settled for a structure that look mostly like the following:

```
pub fn parse_file(it: &str) -> IResult<&str, Vec<StrCommand» {
    delimited(ws, many0(parse_command), eof)(it)
}
```

That is, we have our input string, `it`, which is parsed with a parser. We define the parser for a file by compostion. `delimited` takes 3 parsers, parse the first, the second and then the third and return the result of the second. This style is the one propsed from the `nom` maintainers[**nom combinators**]. We can parse term binders as such:

```
fn parse_binder(it: &str) -> IResult<&str, Term<&str» {
    alt((
        map(
            preceded(alt((reserved("let"),reserved("@"))),
                          tuple((parse_ident, parse_term, parse_term))),
            |(var, val, body)|  binder!(let var, val, body)
        ),
        map(
            preceded(alt((reserved("pi"),reserved("!"))),
                        tuple((parse_ident, parse_term, parse_term))),
            |(var, ty, body)| binder!(pi, var : ty,  body),
        ),
        ...
    ))(it)
}
```

We parse the different aspects of a binder, indentifier, binding term and the bound term and the construct the appropriate binder. Notice here that *let x = M in N* is syntactical sugar for $(\lambda x.N)M$ and is not the same *let* as in side conditions. We might be able to do some fancy combination of conditional compilation and macros to reuse this code, but for now we settle on the kernel parser being a copy of the userspace parser with identifier parsing removed in binders.

9.2.2 Converction from terms

With front end language, we can pretty simply convert it the language into the backend language. We traverse the AST and uses an environment to update symbols appropriately. The lookup is simply a collection on names that need be substituted. The environment is simply a vector if &str. When a new binder is found we push it to the end of a the vector. When we meet a symbol we can then look up if it should be converted into a binder.

```
fn lookup_(vars: &[&str], var: &str) -> Option<u32> {
    vars.iter().rev()
                .position(|&x| x == var)
                .map(|x| (x as u32))
}
```

and specifically map the option as follow:

```
pub(crate) trait Lookup<'a> {
    fn lookup(vars: &[&'a str], var: &'a str) -> Self;
}

impl<'a> Lookup<'a> for StrAlphaTerm<'a> {
    fn lookup(vars: &[&'a str], var: &'a str) -> Self {
        lookup_(vars, var).map(|x| Ident(DBI(x)))
                            .unwrap_or(Ident(Symbol(var)))
    }
}
```

One thing to note however is that this approach is errorprone. Consider the expression: $\lambda x.((\lambda y.xy) : (\lambda z.z))$ then we push x to the vars environment, to update the body of the abstraction and then we have two branches of the ascription, the type and the term. When transforming the type, we push y to vars, then we replace x with the index 1. Then we replace and y with 1. We then get to transforming the term of the ascription and because vectors are a mutable structure, when pushing z it will lie at vars[2]. For a simple solution, I define a a function local inspired by the effectful function local of the Reader monad.

```
fn local<'a, 'b, Input, Output>
    (fun: impl Fn(Input, &mut Vec<&'a str>) -> Output + 'b,
     vars: &'b mut Vec<&'a str>
    ) -> Box<dyn FnMut(Input) -> Output + 'b>
{
```

```rust
    Box::new(move |term| {
        let len = vars.len();
        let aterm = fun(term, vars);
        vars.truncate(len);
        aterm
    })
}
```

We create a closure which takes in a term, the closure will call `fun` with the term and the environment as arguments and then it will truncate the environment to its size before `fun` was called.

We can then use the function as such:

```rust
  Term::Ascription { ty, val } => {
      let mut alpha_local = local(alpha_normalize, vars);
      let ty = alpha_local(*ty);
      let val = alpha_local(*val);
      Asc(Box::new(ty), Box::new(val))
  },
```

Following these rules we simply convert the AST.

9.2.3 Serialization

To feed the transformed AST to the kernel we imagine a function that can convert this into a format the kernel can read. I have not focused on this part and thus have no implementation for it at the moment. Ideally we would want to serialize the data into a binary format that is easy to deserialize. I have spent some time looking into good libraries for this and formats such as Cap'n Proto or rkyv, however they are not implemented with `no_std` that support `no_oom_handling` and are thus not feasible without much further work. We could also introduce a specific binary format which could then be parsed using nom, which has decent support for zero copy, given the right circumstances. Again this would require a fairly deep knowledge of when zero copy is supported in Nom. The most simple solution would be to implement a pretty-printer.

9.3 Typechecking LFSC

In this section i describe the implementation that corresponds to Section **??** through **??**. I present how the code is structured and why I have decided to do so.

9.3.1 Values

as mentioned, we consider typechecking using normalization by evaluation. To define what an evaluation look like we need another type. We define them as such:[10]

---

[10]Notice here that Z and Q should actually have unbounded integers as fields. I have not looked into a solution that is compatible with the kernel

```rust
pub enum Value<'a, T: Copy> {
    Pi(RT<'a, T>, Closure<'a, T>),
    Lam(Closure<'a, T>),
    Box,
    Star,
    ZT,
    Z(i32),
    QT,
    Q(i32, i32),
    Neutral(RT<'a, T>, Rc<Neutral<'a, T>>),
    Run(&'a AlphaTermSC<T>, RT<'a, T>),
    Prog(Vec<RT<'a, T>>, &'a AlphaTermSC<T>),
}
```

A value might be one of the abstractions in the term language, as these cannot be reduced
further. It can be a □or a ⋆ where ⋆ is *kind* and □is a sort classifying kinds. It can then be the
value of a Z or Q or it can be the base types: Z and Q. Neutral expressions, consists of an RT
which is the type describing it, and a the neutral expression it describe. The RT typesynonym
is a reference counted pointer to a value. The reason we use reference counting is to reduce the
overall memory needed. It allow us to only define a value once isntead of having to potentially
cloning it again and again. This may not be immediately obvious for the simpler types, but for
the complex values that contain closures which captures term this may get costly quickly. We
use reference counter over compile time references because we dont immediately know the
owner of a value and thus also not the lifetime of it. Considering that most of the functions I
am gonna describe produces values, the value will be handed to the caller of the function, but
in some cases the owner may be the environment or we would have to clone values from the
context. Further because of the lifetime guarantee there is no way to create a value and return
a reference to it.

Values can then also be Or it can be a program or a run command.[11]

Neutral types are:

```rust
#[derive(Debug, Clone)]
pub enum Neutral<'a, T: Copy>
{
    Var(T),
    DBI(u32),
    Hole(RefCell<Option<RT<'a, T>>>),
    App(Rc<Neutral<'a, T>>, Normal<'a, T>),
    // SC
}

#[derive(Debug, Clone)]
pub struct Normal<'a, T: Copy>(pub Rc<Type<'a, T>>, pub Rc<Value<'a, T>>);
```

---

[11]would it make sense for programs and run commands to be neutral?, and should holes be neutral? they dont have
an associated type.

9.3.2 Contexts

The context has been the most complicated part of this implementation. As described in **??** we consider two levels of environments. Signatures are used for the global context while *Context* is used for the local context. They have a similar interface but internally works quite differently. A global context is defined as such:

```rust
pub struct GlobalContext<'a, K: Copy> {
    pub kind: RT<'a, K>,
    keys: RefCell<Vec<K>>,
    values: RefCell<Vec<TypeEntry<'a, K>>>,
}
```

The kind field is simply meant to be a □and is only place like this for ease of use. The kind then has a `keys` and a `values` field. These are vectors (although in the future should probably be either a hashmap or btree or the likes of that.)

10 **TODO** Experiments

- Take some examples, (for instance from Mads Thesis and check the proofs and see if the new format will let the updated ebpf syscall allow programs to load)

- obviously also check that it will reject programs that either does not match the program (if this can even happen by design), or if the proof is unsatisfiable.

- We really want the functionality to be non-blocking. Is this easy or difficult in Rust and in the Kernel?

11 **TODO** Evaluation

- Videnskabsteori: Hypothesis etc.

  - Correctness

  - Expressiveness

  - Performance

  - Ease of use?

  - maybe more???

12 Conclusion