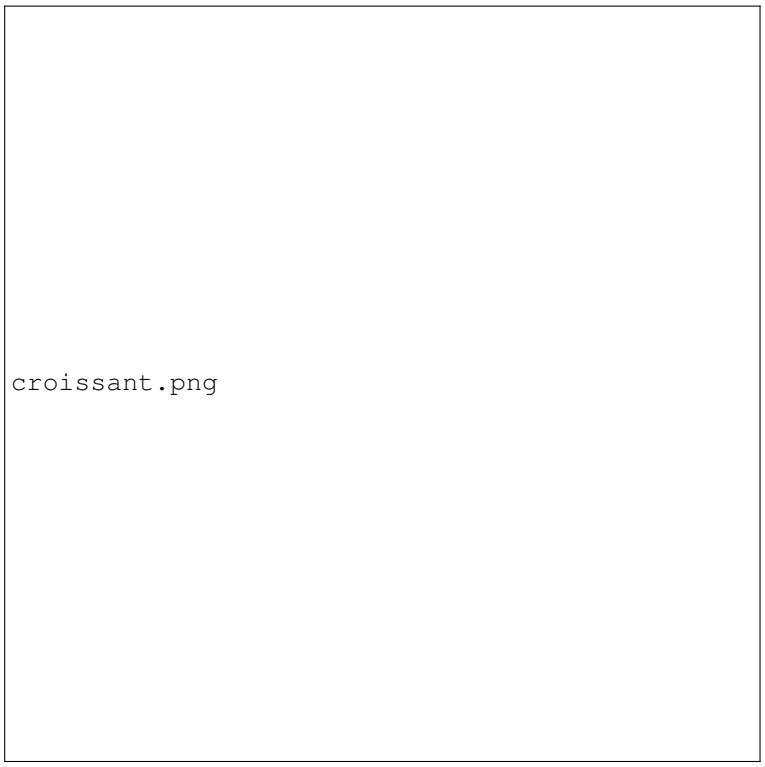
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Msc. Thesis

Jacob Herbst (mwr148)

Verifying eBPF programs in the Linux Kernel

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

repository: `github.com`

Advisor: Ken Friis Larsen

2023-05-31

Abstract

eBPF (extended Berkley Package Filter) is a Linux functionality that allow users to run code of a limited assembly like language inside the Linux kernel. This can potentially be dangerous in the hands of a malicious or even uncareful users. To combat this, the Linux kernel provides a static analysis of eBPF programs to reject harmful programs. This static analysis have in the past shown to be unsound, meaning harmful programs have been rejected. In this project we investigate the feasibility of making a logical proof checker that runs inside the kernel, implemented in Rust. We do so by considering the Logical Framework with Side Conditions (LFSC) language. The motivation behind such a proof checker is to be able to formally prove the correctness of eBPF programs. We consider the feasibility not only as a standalone tool but as a hypothetical part of a larger Proof Carrying Code (PCC) architecture. By this we provide an analysis of how eBPF programs are loaded and what the static analysis consists of and hereby describe a vision for a PCC architecture. The main focus of the report is to describe and evaluate the implementation of LFSC with respect to eBPF, PCC and Rust in 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 unsafe 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[1]. 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]

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

[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?

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...

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.

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

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.

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

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:

```
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:

```
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.

```
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

[6] should i go more indepth with this?

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 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

[7] example?

by P. I write keywords in **bold**. I write $[M_1/x_{1,\ldots,Mn}/x_n]N$, 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. $L : R$ denotes that L has type R.

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

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. All assertions has one of the following forms.

$$\begin{aligned}
\Sigma\ \checkmark \quad & (\Sigma \text{ is a valid signature}) \\
\vdash_\Sigma \Gamma \quad & (\Gamma \text{ is a valid context in } \Sigma) \\
\Gamma \vdash_\Sigma K \quad & (K \text{ is a kind in } \Gamma \text{ and } \Sigma) \\
\Gamma \vdash_\Sigma M \Leftarrow A \quad & (M \text{ can be checked to have type } A \text{ in } \Gamma \text{ and } \Sigma) \\
\Gamma \vdash_\Sigma M \Rightarrow A \quad & (M \text{ can be infered to have type } A \text{ in } \Gamma \text{ and } \Sigma)
\end{aligned}$$

The validity of signatures is as follows, the empty signature is valid. Then Signature may contain either a value or a type level constant of a type or kind respectively.

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

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

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

Figure 1: Valid signatures

An empty context is valid under Σ given the validity of Σ. For a $x : A$ to be part of a valid context, A must be of kind * and x must not occur in the domain of the signature.

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

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

Figure 2: Valid contexts

8.3 Typing

Figure 3 describes type inference with respect to the signatures and contexts. Given that Γ and Σ is valid, we can infer the build in types **type** and **type^c** to be **kind**. Notice here that **type** and **type^c** are kinds which describes types often denoted as * in the literature, whereas **kind**

describes kinds. constants can be inferred either as a kind or a type respcetively if they appear in the signature, whereas object level variables must occur in the context.

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

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

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

Figure 3: Typing rules for looking up types.

Figure 4 describes the bidirectional typing of LFSC. For brevity we simply use $\vdash insteadof \vdash_\Sigma asitisassumed\Sigma$ is valid. *ANN* states that given object *M* can be checked to have type *A* then the annotation can be inferred to type *A*.

Both *TYPE-APP* and *APP* states that the rator (left construct) must be a function type and that the operand (right construct) can be checked to be the type of domain of the function type, then an application can be inferred as the substitution of the operand with the *x* in the construct describing the range of the function. *APP-SC LAMANN* states that a bound variable *x* has type *A* in *M* can be inferred as a $\Pi x : A.B$ given that *A* is a **type** and that *M* can be inferred as *B* with $x : A$ added to the environment. Lambda abstractions is the only type that we cannot directly infer but instead must be checked to be a function type. A lambda is valid function type if the body *M* can be inferred as *B* with $x : A$ added to the context. concrete integers and rationals must trivially be their respective types.

Typechecking of sideconditions occurs in *PI-SIDE* of Figure **??**. We follow the structure of the language reference [**<empty citation>**]. Numbers, and variables follows similar rules as for the term language. Similarly Let expressions $\mathbf{let}xST$ work as its term counterpart. Applications are a bit more tricky. Applications must be fully applied but at its core construct it should follows.

$$\text{SCAPP}\ \frac{\Gamma \vdash S \Rightarrow \Pi x : A.B \quad \Gamma \vdash T \Rightarrow C \quad x \notin FV(B)}{\Gamma \vdash S\ T \Rightarrow B}$$

Again the operator should be a pi type. But we further require that *x* does not occur free in *B*. This is essentially to ensure that parameters to a side condition program must not occur in the other parameters or in ty return type. Below the 3 cateogrical constructs of side conditions are represented.

The inference of *DO* is similar to the rule for let except we dont bind anything. *MARKVAR* will simply return the type the inner sidecondition. *IFMARKED* will check the "marked" sidecondition *S* and that it is a symbolic value aswell as checking that the two branches have the same type.

Compound sideconditions may be a **fail**, described by the *FAIL* rule, which simply typechecks the inner value. The reason **fail** must take an argument is that we have no apparent way to polymorphically check its type from the rules. For *IFEQ* S_1 and S_2 must have the same type and the branches T_1 and T_2 must equally have the same type. Match statement work similar to other functional languages and given the scrutinee S_1 can be inferred as A then for all match

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

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

$$\text{PI-SC}\ \frac{\Gamma \vdash S \Rightarrow \mathbf{type} \quad M \Rightarrow \mathbf{type} \quad \Gamma \vdash B \Rightarrow \mathbf{type}}{\Gamma \vdash \Pi x : \{S\ M\}.\ B \Rightarrow \mathbf{type^c}}$$

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

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

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

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

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

Figure 4: Bidirectional typing rules for LFSC

$$\text{IFMARKED}\ \frac{\Gamma \vdash S \Rightarrow TODOA \quad T_1 \Rightarrow B \quad T_2 \Rightarrow B}{\Gamma \vdash \mathbf{ifmarked}\ n\ ST_1T_2 \Rightarrow B}$$

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

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

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

Figure 5: Typing rules for sideeffects

cases P_i must also be inferreable as *A*, whilst all branches T_i must be inferreable to the same type *B*. $\mathbf{ctx}(P_i)$ describes the context created from P_i. concretely $\mathbf{ctx}(P_i) = \langle\rangle, x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$, when $P_i = cx_1\ x_2\ \ldots\ x_n$

Numerical Sideconditions are fairly straight forward and follows standard mathematical rules. The rules are represented for **integer** but work similarly for rational, except for *Z-TO-Q*. *IFNEG* and *IFZERO* are similarly to other branching constructs.

8.3.1 Operational Semantics of Sideconditions

The operational semantics is show in Figure 8. The operational semantics are under the judgement of $\Delta \vdash \sigma\sigma_1; S \downarrow T; \sigma_n$, where Δ describes all program definitions σ_1 and σ_n, describes states mapping symbols to markings. The Δ is elided in all cases where it is unused. Errors are not included in the operational semantics. Errors might occur when **fail** is evaluated, a scrutinee does not match any pattern, a **markvar** or **ifmarked** does not evaluate to a variable or

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

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

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

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

Figure 6: Typing rules for compound sideconditions

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

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

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

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

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

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

Figure 7: Typing rules for numerical sideconditions

division by 0. *CST-O*, *VAR-O* and *NUM-O* simply evaluate to itself and the store is unchanged. *CST-APP* applies a constant to n sideconditions, update the store with respect to all of them and the resulting value is the constant applied to each updated *S'*. *LET-O* and *DO-O* evaluates *S* and then evaluates *T* with the updated store and in the case of *LET-O* by substituting occurences of *x* in T. the two rules *IFEQUAL-T* and *IFEQUAL* describes a standard semantic for equality checks. Two terms S_1' and S_2' are considered equivalent by syntactical equivalence. Match constructs evaluate the scrutinee and matches the result with one of the patterns. If a pattern matches then the given branch is evaluated. *FUN-APP* refers to the application of a sidecondition program. f must be a program in Δ and all arguments are evaluated and then results are substituted into the body of the program T and it is evaluated. *BINOP-O* and *NEG-O* works similar to any other language. *ZTOQ-O* evaluate S to an integer and then make the rational z/z. *IFNEG* and *IFZERO* rules are very similar, based on the evaluation of S, either branch *T* or branch *U* is evaluated. *IFMARKED* is again similar however the branching depends on the marking of variable *x* in the store. *MARKVAR-O* is the only rule that can update the store. *n* is the flag that need to be swapped. The language support 32 flags, corresponding to a 32 bit integer where each bit defines a flag.

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

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

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

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

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

$$\text{MATCH-O}\ \frac{\sigma_1; S \downarrow (c\ S_1 \ldots S_m); \sigma_2 \quad \exists i.P_i = (cx_1 \ldots x_m) \quad \sigma_2; [S_1'/x_1, \ldots, S_m/x_1]T_i \downarrow T'; \sigma_3}{\sigma_1; (\textbf{match}\ S\ (P_1T_1) \ldots (P_nT_n)) \downarrow T'; \sigma_3}$$

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

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

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

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

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

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

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

Figure 8: Operational semantics for side conditions

8.4 Concrete Syntax

Although the concrete syntax of LFSC is not terribly important we give a brief introduction to make the understanding of the examples presented later easier to understand. LFSC implements the core language and sideconditions as S-expressions.

At the toplevel lfsc allows the following commands, and derivatives thereof. They all take a single argument.

- define: Defines a term.
- declare: Declare a type.
- program/function: is used to define sideconditions.
- check: check will simply check a the type of the provided argument.

For the term language the following symbols are used:

Abstract Syntax	Concrete Syntax
Π x : A . B	! x A B
A : M	: M N
x : M N	# x M N
x N	\ x N
let x M N	@ x M N
{ S T }	^ S T

The sidecondition language directly uses the keywords marked with bold in the previous sections.

8.5 A Small Example

To show a that the formula $P \wedge \neg P$ is not satisfiable, we get the following program.

```
(define cvc.p (var 0 Bool))
(check
(# a0 (holds (and cvc.p (and (not cvc.p) true)))
(: (holds false)
(resolution _  _  _
(and_elim _  _  1 a0)
(and_elim _  _  0 a0) ff cvc.p))))
```

The program first defines a variable `cvc.p` which is defined by `var` which is used to define free constants, `cvc.p` describes the *P* in the formula described above. Then for the actual proof we have the check, which consist of an annotated lambda, stating that `a0` should bind with the type that $P \wedge (\neg P \wedge \top)$ holds. The reason `true` is also included, is because considers applications as n-ary functions and these will be represented as a null-terminated curried form of higher order application. the body of the lambda is then an annotation stating that line 5-7 should have the type holds false. We notice that there are quite a number of holes. these will get filled out as we go. `resolution` and `and_elim` look as follows:

```
(declare resolution (! c1 term
                    (! c2 term
                    (! c term
                    (! p1 (holds c1)
                    (! p2 (holds c2)
                    (! pol flag
```

```
                    (! l term
                    (! r (^ (sc_resolution c1 c2 pol l) c) (holds c)))))))))))
(declare and_elim (! f1 term
                  (! f2 term
                  (! n mpz
                  (! p (holds f1)
                  (! r (^ (nary_extract f_and f1 n) f2)  (holds f2)))))))
```

We can from these declarations see that the 3 first holes should match parameters `c1, c2, c`, these occur later in the type and can therefore be "derived" from these. similarly for `and_elim` the hole used for argument `f1` will be filled by argument `p` because the types must match. Since the fourth argument of the `and_elim` applications are `a0`, we can syntactically compare `a0` and `(holds f1)`, letting `f1 = (and cvc.p (and (not cvc.p) true))`. Notice here that both applications of `and_elim` is provided with 4 arguments. However, from rules *APP* and *PI-SIDE* the inner pi, will be run after application is checked. Hence when all types are checked `nary_extract` is run. `nary_extract` looks like:

```
(function nary_extract ((f term) (t term) (n mpz)) term
  (match t
    ((apply t1 t2)
      (mp_ifzero n
        (getarg f t1)
        (nary_extract f t2 (mp_add n (mp_neg 1))))))
)
```

It will extract the `n`-th element of an `f` application in `t`. specifically `and` is defined as follows:

```
(declare f_and term)
(define and (# t1 term (# t2 term (apply (apply f_and t1) t2))))
```

so when $\text{nary}_{\text{extract}}$ is called with `f_and`, `f1 = (and cvc.p (and (not cvc.p) true))` and `1`, then by beta reduction we have: `f1 = (apply (apply f_and cvc.p) (and (not cvc.p) true))`. The only branch matches this scrutinee and we check if n is 0. Since it is not we recursively call extract. In the next iteration we get: `t = (apply (apply f_and (not cvc.p)) true))`, now `n` is also 0 and we call `getarg` defined as follows:

```
(function getarg ((f term) (t term)) term
  (match t ((apply t1 t2) (ifequal t1 f t2 (fail term)))))
```

`t` now is `(apply f_and (not cvc.p))`. Since `t1` and `f` is both `f_and` we therefore get `(not cvc.p)` back. This is then checked agains `f2` (in the run case of `and_elim`) and the body `(holds f2)` is then checked. Similarly happens for the other application of `and_elim`. By now we have established all the arguments to `resolution`, `c1 = (not cvc.p)`, `c2 = cvc.p`, `c` is still a hole and `p1` and `p2` is `(holds (not cvc.p))` and `(holds cvc.p)` respectively. `sc_resolution` is a little more complicated.

```
(function sc_resolution
```

```
    ((c1 term) (c2 term) (pol flag) (l term)) term
  (nary_elim f_or
    (nary_concat f_or
      (nary_rm_first_or_self f_or
         (nary_intro f_or c1 false)
         (ifequal pol tt l (apply f_not l))
          false)
      (nary_rm_first_or_self f_or
          (nary_intro f_or c2 false)
          (ifequal pol tt (apply f_not l) l)
           false)
    false)
  false))
```

at the inner most applications we have calls to `nary_intro` which lifts a value into `n`-ary form, specifically the two calls will turn `c1` into `(or c1 false)` and vice versa. `nary_rm_first_or_self` will then check if the result of `nary_intro` is equivalent to the `ifequal` call. and return the fourth argument `false` if that is the case otherwise it will remove the first occurence of `f_or`. Specifically, the application in line 4, will be (or (not cvc.p) false) (apply f_{not} cvc.p) false and the result will then be `false`. For line 5 we equally get false. `nary_concat` will then also return `false`, and we then clearly cannot eliminate any `f_or` from the expression thus the result becomes `false` and we match hole `f_2` in `resolution` with the result and get `holds false` which is equivlaent to the annotated type. Hence the check is done and valid.

This is a small formula, however already here we see that programs can become pretty complicated however the sideconditions can be useful in more complicated.[8] We will not dwell much at the sideconditions as the empirical experiments does not contain any. This is explained in Section **??**.

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

[8] I cannot come up with a smart example

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 Brujin indices instead swap each bound variable with a positive integer. The meaning of the integer n is then constituted by the n^th 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. 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 substititution, by lookup, when necessary and then proceed. Specifically we use closures which close a Γ TODO: example. I will not go into detail about how the abstract syntax and type system looks with explicit substitution, as there is no significant difference with having constructs for explicit substitution as presented in **??** as opposed to storing it in Γ.

9.0.3 Normalization by evaluation

Normalization by evaluation is a process first introduced by Berger and Schwichtenberg[**<empty citation>**] for efficient normalization of simply typed calculus, but it has since been refined for more other systems in Barendregt's lambda cube. This implementation is inspired by Christensens tutorial "Implementing Dependent Types in Haskell"[**<empty citation>**]. The technique ties a connection between syntax and semantics.

The process of evaluation might dependent on a context but for programming languages this amounts to either compilation to machine code and then execution or by an interpreter. In both evaluation gives meaning to said program. For instance if the result of an interpretation is a number then the number constitutes the meaning. The meaning may not be concrete, but can also be functions. LFSC have values for the built in types **type**, **kind** and function types such as Π types. Evaluation in general is only sensible for closed terms as free variables.

The process of Normalization on the other hand is to transform a program into its normal form. Letting $nf(t)$ denote the normal form of term t with $\Gamma \vdash t : A$, then the following properties must hold:

- $\Gamma \vdash nf(t) : A$
- $\Gamma \vdash nf(nf(t)) = nf(t)$
- $nf(t) = t$

That is the normal form has the same type as the original term, the normal form is idempotent and cannot be further normalized and the meaning doesnt change when normalizing a term. However many such functions follow these properties. Informally, we consider the normal form to be expressions which contains no redexes. A redex is function type applied directly to an argument. Specifically the normalization is considered with respect to β-conversion. However the process of β-reduction is slow. instead we interpret the understanding of finding a normal form can as evaluating open terms, however the result of evaluation will not be a value buth rather a modified term with possibly unknown values. We denote these as neutral expressions. A neutral expression in general may be free variables or application where the function is neutral.

To perform normalization by evaluation we therefore must extend the set of possible values to also include neutrals. We the require a reflection function $T \longrightarrow T$ giving mening to terms, the evaluator per se. We then further a reification function $T \longrightarrow T$, the inverse of the evaluator that gives normal forms of type T. A normal form is then obtained by reflection followed by reification. For typed derivation of lambda calculus the reification process is syntax directed buth rather inductive on the type of the normal form. Section **??** goes into more details about this.

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 meta-data. 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 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 and interfacing can thus be a little tricky. It is easy to remove this restriction, but may make a potential PCC implementation harder to get merged into the upstream Linux.

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:

```
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.

```
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 or it can be a locally defined variable in a let. 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.

```
pub enum AlphaTerm<Id> {
    Number(Num),
    Hole,
```

[9] What about the OOM killer?

```
    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 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. 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 Convertion 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 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>
{
    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:

```
  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]

```
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 $\square$ or a $\star$ where $\star$ is *kind* and $\square$ is a sort classifying kinds, these specifically correspond to **kind** and **type**. Notice that we dont consider a $\mathbf{type}^{\mathbf{c}}$ as it by construction in the implementation will never clash with **type** 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

[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

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 instead 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. Although we cant fully utilize the borrowing system, reference counting in Rust makes for lot cleaner code. The reference counted smartpointer looks as follows:

```
pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
    phantom: PhantomData<RcBox<T>>,
}
```

An Rc is nothing more than a struct, which contains a pointer to the inner value that is referenced, along with a phantom field. The fanthom field is merely there to keep strong static typing in a similar way to a phantom type in Haskell. The ptr in this struct points to the following struct:

```
#[repr(C)]
struct RcBox<T: ?Sized> {
    strong: Cell<usize>,
    weak: Cell<usize>,
    value: T,
}
```

Which contains the values and the counts for strong and week reference counts. Whenever the Rc is then cloned we simply take the RcBox inside of Rc and increment the pointer, and construct a new Rc struct. The ease of use then comes from the drop trait which will either decrement the inner RcBox and drop the Rc or it will drop both if the strong count is 0. Hence, allowing us to only specify when we want a new reference, but dont need to decrement or drop manually.

Abstractions Π and λ contain a closure which at its core is a function of type $RT \longrightarrow RT$, which closes over a local context and the body of the term it was constructed from. The Pi value further contain its domain and a boolean value. This boolean value denotes if the variable bound occurs free in the body. This is extremely important for performance reasons which we describe more in detail in Section **??**.

Values can then also be neutral expressions, which can be either a neutral variable a global or local scope, It can then be a hole, or an application of a neutral term to a normal form. The constructor Neutral for values is a Value Neutral pair. The value constitutes the type of the neutral. Which is what allows for reification.

```
#[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>),
}
```

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

Lastly Values can be programs and run commands. Programs can not be constructed by reflection but must exist in the same type to be part of the signature. Likewise Run cannot be directly constructed but will always be a domain of a Pi value.

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 Σ are used for the global context while *Context* Γ is used for the local context. They have a similar interface but internally works quite differently. A global context is defined as such:

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

The kind field is simply meant to be a $\square and is only placed like this for ease of use. The kind then has a `keys` and a `values`$ *checker, since the global environment is passed around as a shared reference in all but one function,* `handle_comma`

Both the global and the local context contain type-entries of the following form:

```
pub enum TypeEntry<'a, Key: Copy> {
    Def { ty: RT<'a, Key>, val: RT<'a, Key> },
    IsA { ty: RT<'a, Key>, marks: RefCell<u32> },
    Val ...
}
```

Notice here that this does not directly correspond to our definition in **??**. The `IsA` construct corresponds directly to $x : A$. The other two are defined purely for ease of use. The `Def` constructor is used for definitions such that we can express $c = M : A$ in the signatures and hereby stating that *c* is a term *M* with type *A*. This is purely useful when considering evaluation or for let expressions. The `Val` is used in extending the environment in evaluation. Again for simplicity we reuse the contexts instead of having a seperate environment.

The global context exposes the following functions:

```
pub fn insert(&self, key: K, ty: RT<'a, K>)
pub fn define(&self, name: K, ty: RT<'a, K>, val: RT<'a, K>)
pub fn get_value(&self, key: &K) -> ResRT<'a, K>
pub fn get_type(&self, key: &K) -> ResRT<'a, K>
```

If one tries to get the value of a IsA type, they get a neutral expression consisting of the stored type `ty` and a neutral symbol.

The local context exposes a much similar interface. The underlying datastructure is however different. The local context is implemented as a linked list using reference counted pointers and much more closely represent the concatenation of Γ presented in **??**.

```
pub enum LocalContext<'a, K: BuiltIn> {
    Nil,
    Cons(TypeEntry<'a, K>, Rlctx<'a, K>),
}
```

Most functions we use such as `eval`, `infer` etc, needs to have access to both Σ and Γ and thus for simplicity we define the following wrapper.

```
struct EnvWrapper<'global, 'term, T: Copy> {
    pub lctx: Rlctx<'term, T>,
    pub gctx: Rgctx<'global, 'term, T>,
    pub allow_dbi: u32,
}
```

The Rlctx is a type synonym for a reference counted Γ. Whereas Rgctx is a standard reference to Σ.

9.3.3 Commands

We define a single function to handle a specific command and then apply this on an iterator of all commands. The function starts by constructing the environment wrapper, with the current Σ and an empty Γ.

- Declarations are first checked that the symbol we want to bind $\alpha \notin dom(\Sigma)$

and then infers the type to make sure $\alpha : K$ or $\alpha : A$. We then evaluate the expression and insert it, as an `IsA`.

- Definitions is similarly first typechecked, and the ty must not be of a kindlevel. They are then stored in the global environment as `Def` where the value is the evaluation of the term.
- Checks is nothing more than infering the type to check for well-typedness.
- Programs are complicated for multiple reasons. Firstly we check that the return type of a program is a type, either built in or a declared inductive datatype. We then check each argument against the empty Γ and add them to a Γ' which will be used for checking the body. Because we have an environment wrapper defined we must drop it, before we can mutably borrow it to add define a program, before we can check the body. We must do it in this specific order, as programs may be recursive and thus we need access to the type of the program before we check the body.

  ```
  let env = EnvWrapper::new(Rc::new(LocalContext::new()), gctx, 0);
      ... other cases ...
      Command::Prog { cache: _chache, id, args, ty, body } => {
        ... typesignature check ...
        let lctx = tmp_env.lctx.clone();
        drop(tmp_env); // drop before we can push
        let typ = Rc::new(Value::Prog(args_ty.clone(), body));
        gctx.define(id, res_ty.clone(), typ);

        let env = EnvWrapper::new(lctx, gctx, 0); // make new
        let body_ty = env.infer_sc(body)?;
        ... sameness check }
  ```

  We then lastly check that the body has the same type as the return value. Notice further that we must create a `Prog` type instead of a `Pi` type as we cannot construct a `&AlphaTerm`. This has the neat sideeffect that we dont have to check for `Pi` types that actually describe programs when checking terms.

9.3.4 Inferring types.

We define an `infer` function for each of the constructs in the language, The functions are implemented as inherent implementations (concrete associated functions) and has the types:

```
impl<'global, 'ctx, T> EnvWrapper<'global, 'ctx, T>
where T: BuiltIn
{
    pub fn infer(&self, term: &'ctx AlphaTerm<T>) -> ResRT<'ctx, T>
    pub fn infer_sc(&self, sc: &'ctx AlphaTermSC<T>) -> ResRT<'ctx, T> {
    fn infer_sideeffect(&self, sc: &'ctx AlphaSideEffectSC<T>) -> ResRT<'ctx,
     T>
    fn infer_compound(&self, sc: &'ctx AlphaCompoundSC<T>) -> ResRT<'ctx, T>
    fn infer_num(&self, sc: &'ctx AlphaNumericSC<T>) -> ResRT<'ctx, T>
}
```

We here define that the typeparameter `T` must implement the Trait `BuiltIn`. The trait is bound by other traits: pub trait BuiltIn: Eq + Ord + Hash + Copy. It must be PartialEq to be able to look up the `T` in the environment. It must also be copy. We uses this stricter trait than clone, as it allows for quick "copying" and is a satisfied criteria for both &str and u32's that could be used for De Bruijn levels. Hash and Ord is not strictly necessary but is required to use Hashmaps or Btrees. Using such types is also described in Section **??**. The BuiltIn trait itself defines how the builtin types **type**, **mpz** and **mpq** is defined. For &str this is simply a stringification of the literals, for u32 prepresented De bruijn indices these may be 0,1,2.

The functions follow closely the rules in **??**. The function patternmatches on term, and for λ sideconditions and holes the inference fails. when inferring a `Pi` we check if the domain is a sidecondition, if that is the case we check them respectively and create a Run type. In case it is not a sidecondition, we simply infer the domain to have **type** and then evaluate it, to get its value. We can then update the local environment stating that DBI 0 in the localcontext `IsA` val type.

```
AlphaTerm::Pi(a, b) => {
    let val =
        if let SC(t1, t2) = &**a {
            let t1_ty = self.infer_sc(t1)?;
            self.check(t2, t1_ty.clone())?;
            Rc::new(Type::Run(t1, t1_ty, self.lctx.clone()))
        } else {
            self.infer_as_type(a)?;
            self.eval(a)?
        };
    self.update_local(val).infer_sort(b)
},
```

The most compliated rule Application. Instead of interpreting multiple continious application as curried form we use a flat approach in which we evaluate each argument in a loop. First we evaluate the function point, this is saved in a mutable variable updated each iteration of the loop. Each argument is checked against the domain of the Π type, if the variable bound by the Π is free, then we evaluate it or we return an arbitrary value. Lastly the body is evaluated, with either the new value or default added to the local environment. the result is bound to `f_ty`. This happens for each iteration and lastly we return the bound value. In case the argument is a Hole, we do not check it, as it is trivially the correct type at this point. We add it to the environment and evaluate the body.

```
App(f, args) => {
```

```
    let mut f_ty = self.infer(f)?;
    for n in args {
        f_ty = if let Type::Pi(free,a,b) = f_ty.borrow() {
         if Hole == *n {
             let hole = Rc::new(Neutral::Hole(RefCell::new(None)));
             b(Rc::new(Type::Neutral(a.clone(), hole)), self.gctx)?
         } else {
            self.check(n, a.clone())?;
             let x = if *free { self.eval(n)? } else { a.clone() };
            b(x, self.gctx)?
         }
        } else {
            return Err(TypecheckingErrors::NotPi)
        }
    };
    Ok(f_ty)
```

similar inference is done for the sidecondition language.

9.3.5 Checking types

We define two functions for typechecking. One takes a term and a value/type t2 and check against it. The other takes a sideconditions as first argument, but essentially does the same. We match on term and if it an anonymous lambda then we check *LAM*-rule of Figure **??**, otherwise we infer the type of the term to t1 and check t2 and t1 for "sameness". This process is two-fold. Firstly we do a value comparison between t1 and t2. to their canonical form using reification as decribed in **??** and check for equality. Generally we cannot compare values, as functions such as Π is implemented using Closures, which cannot easily be compared. The main reason for the `ref_compare` call, is to fill in holes. The function return a boolean of the equality between the two values, and as long as the values are one of the simple types or syntactically the same neutrals. If one of the arguments is a hole it is filled with the other value. This is done using the interior mutability of Refcells. We cannot fill the holes after reading back values as a hole might occur in multiple places and we must ensure that it is filled with the same value. However empirically, the $\text{ref}_{\text{compare}}$ will return `true` most of the time. In case it returns 0. We readback (a more idimatic term than reification) the values into their normal form and compare them.

```
pub fn convert(&self,
            t1: RT<'term, T>,
            t2: RT<'term, T>,
            tau: RT<'term, T>) -> TResult<(), T>
{
    if ref_compare(t1.clone(), t2.clone()) { return Ok(()) }
    let e1 = self.readback(tau.clone(), t1)?;
    let e2 = self.readback(tau, t2)?;
    if e1 == e2 {
        Ok(())
    } else {
        Err(TypecheckingErrors::Mismatch(e1, e2))
    }
}
```

9.3.6 Readback (Reification)

As mentioned previously, reading back semantical objects into the term language is type directed. `read_back` takes a type and a value as arguments. The we first check if the value is neutral. Is it the case, then we call `read_back` neutral.

Neutral values have enough information to be read back without a type. Variables can be read back directly, holes either get read back as its inner value or as a term language hole. Application will read back the function point and then argument point. And construct an application from it. Be aware here that the argument point is a `Normal` type. It will thus call `read_back` with its type and val.

If the value of `read_back` is not neutral, we patternmatch on the type. If the type is Z or Q then we can read back integers and rationals respectively. All built in types can be readback to the built in terms. Pi types can be read back by reading back the domain, then evaluate the body and read the body back. Run commands can be $\text{read}_{\text{back}}$ as the second argument and constructing and `SC` type by cloining the actual sidecondition call.

9.3.7 evaluation

We define evaluation on two levels, both on terms and on the functional sidecondition language. Evaluation of the term language is straightforward. Sideconditions and holes cannot be evaluated. Application have a similar structure to `infer`. We consider applications in applicative order evaluation with a loop over the arguments. The function `do_app` is then called with the function and the argument evaluated

```
pub fn do_app(&self, f: RT<'ctx, T>, arg: RT<'ctx, T>) -> ResRT<'ctx, T>
{
match f.borrow() {
    Value::Lam(closure) => closure(arg, self.gctx),
    Value::Neutral(f, neu) => {
        if let Value::Pi(_,dom, ran) = f.borrow() {
            Ok(Rc::new(Value::Neutral(
                ran(arg.clone(), self.gctx)?,
                Rc::new(Neutral::App(neu.clone(), Normal(dom.clone(), arg))))
    ))
        } else {
            todo!("This should be an error")
        }
    }
    _ => todo!("This should be an error"),
}
}
```

Functions can be either a concrete lambda abstraction in which we simply evaluate the closure. A Function can also be unknown, in which case, we check that the type of the neutral value is a function type. We construct a new neutral value, where the type of the neutral value is range of the Π and the value is an application of the unknown value onto the normal expression (dom, arg). stating that arg has type dom. For instance if we consider the application (f x y), where f :: a -> b -> c, then we construct: Neu(b -> c, f (x : a)) by the first applcation and Neu(c, (f (x : a)) (y : b)) If we then want to read back, we will get (f x y) back since it is already in normal form.

Evaluation of Π terms are also interesting. For standard Π constructs, we simply evaluate the domain, construct a closure around the body, check if the bound variable is free in the range and construct a Pi value. However if the domain of a Π is a sidecondition, we evaluate the sidecondition, the target and check for equivalence. Lastly the result is inserted in the local context and the body is evaluated. This is what enables execution of sideconditions in the typechecking.

[11]

10 Experiments

The purpose of this project is to check the feasibility of an in-kernel proof-checker that can replace the eBPF verifier. Thus we want to evaluate the implementation with respect to the domain. To do so, we consider a simple verification condition generator based on the weakest precondition predicate transformers. Specifically we consider a limited subset of eBPF consisting only of the following instructions:

$$
\begin{aligned}
&r_d := src \\
&r_d := r_d \oplus src \\
&r_d := -src \\
&\oplus \in \{+, -, **, /, mod, xor, \&, |,
\end{aligned}
$$

r_d denotes an arbitrary register. And src may be either, a register or a constant value of either 32 or 64 bits. Instructions can be a move from src into r_d, a negation of a src value into r_d or it can be one of the binary operators $\oplus$ where & and | is binary con- and disjunction, and « » is logical left and right shifts. With this we want to do some positive testing by create valid programs and then check the proofs. We consider only valid programs as we cannot create proofs for satisfiable formulas, but rather satisfying assignments. We then use quickcheck to generate arbitrary instructions with the following property. Register r_8 should be greater than 0 and smaller than some arbitrary value. This simulate a sequence of instructions followed by a memory access. This is a situation the eBPF verifier has been shown to be faulty at previously. We ensure this property by evaluating the program. If the program satisfy the property we then add an prolog and epilog to the program. To mimic a program that must be valid with respect to the verifier. We create the verification condition, with the post condition that $0 \leq r_0 < n$, where in the arbitrary upperbound. We convert the representation of the verification condition into smt2 and discharged to cvc5. If the negation of the verification condition is unsatisfiable by cvc5, then we can discharge an LFSC proof. To check the performance of this implementation versus the C++ implementation *lfscc* we can run both through hyperfine to get a comparative runtime.

We can then also check the runtime speed of loading an eBPF program into the kernel for comparison.

In creating this "experiment" some of the work should be attributed to Ken Friis Larsen, Mads Obitsøe and Matilde Broløs. To discharge eBPF, we use Larsens `https://github.`

[11] should i mention anything about evaluation of sideconditions?

`com/kfl/ebpf-tools` and a collection of FFI-bindings for Haskell to work with eBPF by Obitsøe. The generation of code and evaluation is part of ongoing research by all three and I. Lastly convertion (or printing) of the verification condition takes heavy inspiration from Obitsøes Thesis. The actual verification condition is made specifically for this. For the actual benchmarking, we create programs of size 1, 10, 100 and 1000. One small caviat that needs mention is that sometimes cvc5 will give a satisfiable result instead an unsatisfiable result. The course of this is either a problem with the verification condition generator or that a program in fact is not valid. This could for instance happen with division by 0. With more time, it would be ideal to test the generator. For now we settle on generating a new program of same size and try again. In the process it showed to be unfeasible to prove the validity of programs of size 1000 in cvc5 on my i7-1165G7 CPU with 16 GB of ram.

10.1 Speed

In the first iteration of the code, the performance of the implementation presented in this report were atrocious. From table **??** it should be clear that, my implementation *lfscr* were extremely slow. For a single instruction, the performance close, here we essentially just check all the signatures distributed by cvc5**??** along with a small proof. Already for 10 instructions my implementation was using 4 times as long to check. And for a 100 instruction straight line program, this difference was close to 140 times slower.

instructions	1	10	100
lfscc	9.0 ± 3.3 ms	9.0 ± 3.2 ms	59.6 ± 0.7 ms
lfscr	12.7 ± 0.7 ms	36.6 ± 1.3 ms	8.3 ± 0.9 s

Inspecting the proofs, which can be found at the repository, in the vcgen folder as, $\text{benchmark}_{\text{n.plf}}$, one can notice that there are more than 1000 local bindings for a 100 line program. This lead be to belive that using cons lists would maybe not be optimal, switching to `Vec` gave a marginal improvement. being a couple of seconds faster than the first implementation.

instructions	1	10	100
Cons list	12.7 ± 0.7 ms	36.6 ± 1.3 ms	8.3 ± 0.9 s
Vec	18.2 ± 1.7 ms	51.8 ± 1.9 ms	6.4 ± 0.1 s

This was still early in the development and the current implementation still uses cons lists, as they provide a higher level of confidence in correctness. Although the speedup is good it was not the massive boost we need. But with a working faster cons list solution it might be worth revisiting the a `Vec` implementation.

10.1.1 Massive speedup

Analysing the code with perf, it got clear that most of the time was used in evaluating applications, namely about 60 percent of the time spend was in `eval` and `do_app`. There is nothing inherently wrong in that proofs are mainly just applications and terms might get big. By analysing the *lfscc* implementation it got clear that my implementation did unecessary complications. Considering the example from **??**, `and_elim` is a 4 argument symbol, of

which `p` is used to destruct the `holds` of the 4th argument and fill f1. In the example `a0 = (holds (and cvc.p (and (not cvc.p) true)))` and while the typechecking that $a0 \Longleftarrow holdsf1$ in necessary, the following call to `eval` to bind `p` for the range of the function is uneccesary since `p` does not occur free in the range. This pattern appear often in LFSC proofs. often Π types will include a parameter that does not occur free in the body, but merely exist to destruct a pattern onto a unfilled hole. So including a calculation of whether a bound variable occurs in the body and then checking the condition before evaluation can save massive amount of computation.

```
let x = if *free { self.eval(n)? } else { a.clone() };
```

This line (along with the actual function for calculating `free`) is enough to make *lfscr* 43 times faster and relatively compareable to *lfscc*. Specifically we get:

instructions	1	10	100
lfscc	8.4 ± 3.2 ms	10.7 ± 1.7 ms	59.2 ± 2.9 ms
lfscr	5.4 ± 1.9 ms	11.7 ± 0.6 ms	193.0 ± 4.6 ms

Meaning that now *lfscc* is merely 3 times faster than *lfscr*. *lfscc* does everything all at once, meaning lexing/parsing and inference and evaluation all occurs in the same function. By this it seems to reduce a lot of overhead and the function `check` which does all of this, also implements tail calls by using goto statements to the top of the function. No such constructs is immediately available in rust, since they are inherently unsafe and we might not get performance completely on par with *lfscc*, especially not using safe rust.

ADDENDUM These benchmarks were done before, i realized that *lfscc* can be build in both a debug and release version. In the release version it is consistently 2-3 times faster than the results presented here. This suggest that a proof checker can indeed be effeciently implemented, but the approach done in this project is not ideal.

10.1.2 formal checking vs static analysis.

TODO

10.2 Memory

We should consider the memory usage of the implementation in two manners. Firstly the size of proofs, plays a key role in the feasibility of using proof carrying code. A proof for a single instruction proof (actually 4 with pre initialization and the final check), has a 2.7KB size, while 10 isntructions is 8.6 and 100 instructions gives 109KB, so the proofs, atleast for straight-line programs, scales linearly (or close) with roughly 1KB per instruction. Encoding the proofs in a more compact binary format could make these sizes even smaller, however these sizes in themselves are not alarming and could still see use in devices with limited memory.

On the other hand we should also look at how much memory the typechecker uses. Running both *lfscc* and *lfscr* with the 1,10 and 100 line proofs, we get the following memory usage:

Program size	1	10	100
peak memory	1.3MB	1.8MB	5.7MB
peak RSS	9MB	15.7MB	25.3MB
temporary allocations:	50.13 %	46 %	40 %

From these results we see that the program does not use a massive amount of memory, at a single point in time we allocate 5.7MB for a 100 line program and for the entire of a program uses 25.3MB.[12] What is most interesting is that 40 % of allocations are temporary and for smaller programs even higher. This suggests that we do some uneccesary computations. This especially become noticable, when similar diagnostics is done for *lfscc* For the 100 line program only 2,9MB memory is used at its peak, while it uses 10MB overall and has only 6% of allocations are temporary. One thing to keep in mind however is that about 1/4 of allocations are leaked. This is not ideal, but for very shortlived programs such as *lfscc* it is not a big deal. Having a proof checker to run in the kernel however, memory leaks is problematic. In any case, we can again see that we can check large proofs without much resources needed. But that a "all in one" solution presented by *lfscc* could be worth prototyping in either pure C or in Rust.

11 Evaluation

In the previous section we described a method of for evaluating performance, and although the implementation discussed in this report is reasonable in both runtime and memory usage, the C++ implementation suggest that a lot more efficient approach exists. However this implementation does have a couple of features that are worth taking into consideration aswell. It is implemented completely in safe Rust, meaning we cannot have any illegal memory that potentially crashes the program. This might be the most desireable property for a program that is designed to run inside the kernel, as "proofs" could exploit such a vulnerability. Equally an implementation should be robust in the amount of time it takes to check the proof. We showed before the performance difference in checking if the occurence of a variable was free could improve the performance from by 43 times. This is easily done for pi types which is not immediately available for users and especially since function in general are small, however similar problems can arise from let-bindings inside of a check. Here malicious users can slow down/block the system with unecessarily large, expressions that are not needed. There is no easy way to solve such problems, as terms with let bindings might be deeply nested and contain 100's or even 1000's of nested local bindings. Thus it can then further get costly to do a range check and it requires a non-online approach of typecheking, which my implementation supports but *lfscc* does not. No immediate solution to this present itself.

My implementation however has an advantage over *lfscc* that checking the proof has not been tampered with is straight forward and already implemented unintentionally. In its current state, the LFSC proofs discharged from cvc5. always contains the following pattern:

```
... POTENTIAL BINDINGS ...
(# a0 (holds x)
(: (holds false)
... ACTUAL PROOF...
```

[12] Note that this memory also include some heaptrack overhead.

here `x` is the formula that unsatisfied by cvc5. Given that an in kernel verification condition generator outputs `Value` types, then all the functionality for normalizing both and comparing them for equality is already implemented.

The experiment has not only provided useful insight into the performance of the implementation but it also establishes confidence that the proof checker works as expected and follow the semantics presented in Section **??**. Checking the signatures along with the generated proofs suggests that mostly all parts of the typechecker is correct. All matters of the term language is covered, and most of the side condition language is also checked. At the moment my implementation also typecheck the functions `markvar` and `if_marked`, but the evaluate of these are still left undone. The main reason for this is that there is currently no signatures distributed by cvc5 that includes their use and they might in fact be uneccesary. The sideconditions could be tested more thoroughly as only a single larger tests has been conducted, in the $P \wedge \neg P$ unsatisfiability proof from **??**. Despite, the example test a large part of the sidecondition language, both constant and program application, match constructs, branching and numerical functions. One point where the implementation is inherently wrong is the usage of i32's for the representation of integers and rationals, in fact these should be unbounded integers. This is not a problem for bitvector proofs, but only for arithmetic logics. I have however left the representation as is for now, as i have not been able to find a library that efficiently implements ubounded integers and rationals. Even though the implementation does not run in the kernel, the implementation only uses the `core` and `alloc` crate along with `nom`, which I have been succesful in compiling and running a simple example of in a kernel module. Hence there is nothing theoretical stopping from compiling into the kernel. The major work in this should be make every allocation fallible by using the `try_new` counter parts to `new` allocations, and implementation a simple from trait to easily converting allocation errors into typechecking errors. Hereby the `?` shortcut can be used, and no types should changes as they already implement Result types.

12 Is PCC a good idea?

Even with a Rust implementation that shows okay performance, promises memory safety and no unexpected errors that can crash the program, the answer is not definite. It might still not be feasible to use LFSC for an in kernel proof checker as part of a larger proof carrying code architecture, since a lot of questions are still unanswered. The eBPF verifier does a lot more than just validating instructions of a bytecode format. It check validity in memory alignment, user-rights, does simple program optimizations, such as deadcode elimination and much more. Although some of these can be embedded into a proof, code optimizations is inherent to the bytecode not the proof and user-rights should be checked seperately anyway. None of this discards the use of PCC but merely accept it as an alternative to some of responsibility of the verifier.

The more pressing matter is that of execution time. Although typechecking LFSC is decidable, it is still a dependent language in which typechecking can invoke evaluation and having this in the kernel essentially means that untrusted sources may execute code inside the kernel. LFSC has no recursion (if we regard the fact that sideconditions are, but these must be trusted to terminate) and hence typechecking will be decidable and terminate. However there is no mechanism stopping a malicious user to construct complex proofs that can block the kernel, not indefinitely, but for a long time, a good old school denial of services. The verifier solves

the matter by only allowing a certain amount of instructions. This is much easier for eBPF as instructions are atomic, but for applications, they may be arbitrarily nested and specifying a limit will require deeper analysis. For instance the LFSC proof for the validity of a 100 line eBPF program has a nesting dept 1850, and this could potentially appear in many let bindings. If we can reach performance similar to that of *lfscc* then this problem does not seem to be as big of an issue. But the performance of the implementation presented here can be problematic.

Despite all this the implementation is rather small and consist of only 2400 lines of code compared to 19000 in the verifier. Bugs are hence less likely to appear.

13 Conclusion

I have in this report given a brief overview of how the eBPF verifier works, and we have discussed some concerns with the current eBPF verifier. This motivates the intend of making a proof carrying code architecture as part of the Linux kernel. We have described how we can use the reasonably new feature of Rust inside the kernel, to make such an architecture. Although not complete, we have described an implementation of a typechecker for the dependently typed LFSC language which uses the curry howard isomorphism to construct logics. The typechecker in its current form, is not compileable inside of the kernel but uses only features of the Rust language, with some slight modifications, such as having to enable the Reference counted smart pointer. The implementation of the typechecker is reasonably simple in structure, with only 2400 lines of code and uses purely safe features in rust and thus provide necessary safety to run inside the kernel. We have further done some experiments that generates arbitrary straight line eBPF programs and performs a naive verification condition on them to generate LFSC proofs with cvc5. We have used these experiments to ensure the correctness of the implementation aswell as providing a framework for benchmarking the performance. In this process we have found the implementation described in this report to be inferior in performance to the *lfscc* implementation from the cvc5 by up top 20 times slower execution time, which suggest a modular approach using normalization by evaluation is not the ideal approach, but this needs more investigation in the future. As a side effect of the experiments a realization arised, in which untrusted users can clog a kernel by cluttering proofs with unneccesary information. At the moment, we know not a good solution for this. The work done is inconclusive with respect to proof carrying code inside the linux kernel but without addressing some of the shortcommings presented here, it may not be feasible to replace the eBPF verifier.

14 Future work

We have presented only a small part of a proof carrying code architecture, so naturally for a final conclusion on the matter discussed in this work an in-kernel verification condition generator should be defined as well as a definitive communication of these two parts. We suggest such a proof checker represents the logic in the form of LFSC. This is however dependent on getting a faster proof checker and to solve the problem of potential clogging.

The implementation discussed, should be modified to run inside the kernel. Because of the preliminary work, this should be a minor task.

Lastly, a more comprehensive analysis and design between PCC and the eBPF verifier should

be conducted. Specifically, what parts of the verifier should co-exist with PCC. One such is example is checking of user-rights etc.

References

[1] Manfred Paul. *CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification*. 2020. URL: https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification (visited on 05/25/2023).