École Polytechnique Fédérale de Lausanne

Grounded Verification of Recursive Trace Systems

by Benjamin Bürki
MSc in Cyber Security, EPFL

Master Project Report

Supervisor and responsible:

Prof. Bryan Ford

EPFL IC IINFCOM DEDIS
BC 210 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 23, 2026

Abstract

This project is situated in formal verification and studies how Grounded Deduction can be used to reason about recursive, trace-producing computations. The focus is a small HRM-style machine whose processor, input device, and output device are defined by mutually recursive functions and whose execution produces observable event traces.

The main difficulty is that before proving safety or liveness of such traces, we must first justify that the recursively defined execution actually produces a well-defined object. Classical verification frameworks can reason about program correctness, but definedness, termination, and trace-level behaviour are often handled as separate proof obligations or encoded indirectly.

The high-level idea of this work is to use the GD principle of *habeas quid*: before using a computed object in a proof, we first prove that it is grounded. The verification is therefore organized in layers: first prove that finite observations of execution produce grounded finite traces, then reason about safety and liveness on those traces.

A key design decision is to make execution produce explicit events. The processor delegates input and output actions to separate device functions, which append their own events to the trace and return control through first-order continuation tags. This keeps the model recursive and modular while making the observable behaviour explicit.

The evaluation shows that the design supports the intended proof pattern. Finite executions yield grounded finite traces, trace-processing functions terminate on finite traces, and finite safety is a grounded predicate. Infinite safety is obtained by checking all finite prefixes, while liveness is stated on infinite traces and inherited by finite traces through prefix reasoning.

Contents

Abstract **2**

1 Introduction **5**
1.1 Context and Motivation . . . 5
1.2 Approach . . . 6
1.3 Project Statement, Results, and Contributions . . . 7

2 Background **8**
2.1 Formal Verification . . . 8
2.2 Traces, Safety, and Liveness . . . 9
2.3 Grounded Deduction and Habeas Quid . . . 9
2.4 Recursive Definitions and Finite Observations . . . 10
2.5 HRM as a Minimal Running Example . . . 10

3 Design **12**
3.1 Design Goals . . . 12
3.2 Overall Architecture . . . 13
3.3 State . . . 14
3.4 Well-Formedness Assumptions . . . 14
3.5 Events and Traces . . . 15
3.6 Continuations . . . 17
3.7 Fuel . . . 17
3.8 Processor Definition . . . 18
3.9 Devices . . . 19
3.9.1 Input Device . . . 19
3.9.2 Output Device . . . 20
3.10 Example: Copy Program . . . 20
3.11 Application-Level Specification . . . 21
3.11.1 Causal Finite Safety . . . 22
3.11.2 Recursive Checker . . . 22
3.12 Finite and Infinite Traces . . . 23
3.12.1 Infinite Safety Through Finite Prefixes . . . 24

3.12.2 Liveness on Infinite Traces . . . 24
3.12.3 Finite Liveness Through Prefixes . . . 25
3.13 How the Design Fits Together . . . 26

4 Formalization and Evaluation **27**
4.1 Evaluation Method . . . 27
4.2 Reusable Habeas Quid Lemmas . . . 28
4.3 Habeas Quid of Finite Execution . . . 30
4.4 Habeas Quid of Finite Safety . . . 32
4.5 Safety Examples . . . 33
4.6 Infinite Safety Through Finite Prefixes . . . 35
4.7 Habeas Quid of Liveness . . . 35
4.8 Finite Liveness Through Prefixes . . . 37
4.9 Discussion . . . 37
4.10 Limitations . . . 38
4.11 Summary . . . 38

5 Related Work **39**
5.1 Classical Program Verification . . . 39
5.2 Trace-Based Verification . . . 40
5.3 Grounded Deduction and Grounded Arithmetic . . . 41
5.4 Position of This Work . . . 41

6 Conclusion **43**
6.1 Future Work . . . 44

Bibliography **46**

Chapter 1

Introduction

1.1 Context and Motivation

Formal verification is concerned with proving that a program or system satisfies a mathematical specification. Unlike testing, which observes the behaviour of a program on selected executions, formal verification aims to reason about all executions allowed by a model. This distinction becomes important as soon as the system under study contains loops, recursion, interaction with an environment, or internal state. In such systems, the number of possible behaviours is often too large to explore exhaustively by testing alone. Logical reasoning and automated reasoning have therefore become central tools in the construction and analysis of reliable software systems. Harrison emphasizes that the complexity of computer systems has made automated reasoning an important component of program construction and programming-language design [2].

A verification proof depends strongly on the adequacy of the model being used. Before proving that a system is correct, one must decide what object represents an execution and what properties should be stated about that object. For a terminating computation, it may be sufficient to reason about an initial state and a final value. For an interactive or streaming computation, however, the observable behaviour is not only a final result. It is the sequence of events produced over time.

The central challenge addressed in this project is the verification of a small interactive system whose behaviour is described by recursive, mutually recursive, and trace-producing definitions. The system is inspired by the Human Resource Machine (HRM) programming model, where a processor repeatedly executes simple instructions such as `INBOX`, `OUTBOX`, and `JUMP`. Even in this small setting, execution is not merely a pure function from inputs to outputs. It consumes inputs, updates an internal state, interacts with input and output devices, and generates an observable trace of events.

Classical program logics provide powerful tools for reasoning about programs, but they do not make all aspects of this problem equally explicit. Hoare logic, for example, expresses correctness

through triples of the form

$$\{P\}\ C\ \{Q\},$$

where a program command C transforms states satisfying a precondition P into states satisfying a postcondition Q. Hoare's original presentation emphasizes the derivation of program properties from axioms and inference rules, while also distinguishing correctness of the result from the separate issue of successful termination [3]. Separation logic extends this style of reasoning with resource-sensitive assertions and is especially effective for programs manipulating shared mutable data structures [6].

These methods are highly successful in their intended domains. The focus of this project is different. Here, the execution itself is represented as a recursively generated trace. The first question is not whether the trace is safe or live, but whether the recursive definition produces a well-defined trace at all.

1.2 Approach

This project studies the problem using Grounded Deduction (GD), and more specifically the arithmetic setting of Grounded Arithmetic (GA). The guiding principle of GD is that one must first "have a thing" before using it in further reasoning. This principle is expressed by *habeas quid* obligations: premises requiring a term or proposition to be grounded before it is used in a proof.

This is useful for recursive definitions. Instead of treating the result of a recursive definition as automatically available, GD requires a proof that the object being used is grounded. Recent work on GA describes this as a way to allow unconstrained recursive definitions while preventing non-grounded or circular objects from being used unsafely in proofs [1]. Kehrli's formalization of GA in Isabelle/Pure further shows that this style of reasoning can be mechanized, although habeas quid premises introduce proof obligations that often require automation [4].

The approach of this project is to model a small HRM-style machine directly as a family of recursive functions. Execution is observed through finite fuel: for each finite fuel value, the system should produce a finite trace. The first proof obligation is therefore the habeas quid of execution:

$$\frac{\Gamma \vdash I\ \mathsf{InputStream} \qquad \Gamma \vdash s\ \mathbb{N}}{\Gamma \vdash f_S(I, s)\ \mathsf{Trace}}$$

This judgment says that, for a valid input stream I and finite fuel s, the execution $f_S(I, s)$ denotes a finite trace.

Once finite traces are grounded, safety and liveness can be stated as trace properties. Safety is first defined on finite traces: every output already produced must be justified by the input history available before that output. Infinite safety is obtained by requiring every finite prefix of an infinite

trace to be safe. Liveness is stated on infinite traces, because it is an eventuality property: outputs required by the specification and not yet produced must eventually appear. Finite traces inherit liveness only when they are prefixes of live infinite executions.

1.3 Project Statement, Results, and Contributions

The main statement of this project is the following: *Grounded Deduction provides a natural framework for verifying recursive, trace-producing system models by separating the proof that executions are grounded from the proof that their traces satisfy safety and liveness properties.*

The main results are threefold. First, the project defines a small HRM-style model with mutually recursive components for the processor, input device, and output device. Second, it proves the habeas quid of finite execution: for every well-formed initial execution configuration and finite fuel value, the system produces a finite trace. Third, it defines trace-based safety and liveness properties and shows how finite and infinite traces are related by prefix reasoning.

The key connection is dual:

Safety:	finite safety is lifted to infinite traces by checking every prefix,
Liveness:	infinite liveness is inherited by finite prefixes.

The contribution is not a new HRM program or a complete verifier for all possible programs. Rather, it is a focused case study showing how GD can organize the verification of a recursive trace-producing model. The resulting proof pattern is:

$$\text{habeas quid of execution} \quad \Longrightarrow \quad \text{grounded finite traces} \quad \Longrightarrow \quad \text{safety and liveness through prefix reasoning.}$$

This pattern is especially relevant for interactive or streaming systems, where executions are better understood as observable behaviours over time than as final return values.

Chapter 2

Background

This chapter introduces the concepts needed to understand the rest of the report. It fixes the vocabulary used later: formal verification, traces, safety and liveness, Grounded Deduction, habeas quid, finite observations of recursive computations, and the small HRM-style programming model used as a running example.

2.1 Formal Verification

Formal verification is the use of mathematical reasoning to prove that a program or system satisfies a specification. A specification describes the behaviour that the system is expected to have, and verification aims to show that every execution allowed by the model satisfies this specification.

This differs from testing. Testing observes the behaviour of a program on a finite set of executions. It can reveal bugs, but it cannot usually prove their absence. Formal verification instead reasons about all executions represented by the model. This is especially important for systems involving loops, recursion, state, or interaction with an environment, since such systems may have infinitely many possible behaviours.

A verification proof therefore depends on two ingredients. First, one needs a model of the system: this determines what counts as an execution. Second, one needs a property to prove about those executions. For a terminating program, an execution may be represented by an initial state and a final state. For an interactive system, however, the observable behaviour is often better represented as a trace of events produced over time.

2.2 Traces, Safety, and Liveness

A trace is a sequence of events representing the observable behaviour of a system. A finite trace represents a finite observation of execution, while an infinite trace represents unbounded behaviour.

Two standard classes of trace properties are safety and liveness [5]. Informally, safety says that "something bad never happens", while liveness says that "something good eventually happens". In a trace-based setting, safety can often be refuted by a finite bad prefix: once a wrong event appears, the violation is already observable. Liveness is different, because it concerns the future. A finite trace may not yet contain a desired event, but this does not prove that the event will never appear.

This distinction is central in this report. Finite traces are used to express and check safety properties. Infinite traces are used to express eventuality properties such as liveness. The connection between the two is made through the prefix relation. We write

$$T \preceq T'$$

to mean that T is a prefix of T'. Intuitively, T' extends T without changing the events already observed.

For an infinite trace τ, finite observations are written as

$$\mathsf{prefix}(\tau, n).$$

This denotes the finite trace containing the first n events of τ. Thus, even when a property is stated over an infinite execution, it can often be related back to finite observations.

2.3 Grounded Deduction and Habeas Quid

Grounded Deduction (GD) is a logical framework in which expressions are used in proofs only after they have been established as grounded. The guiding principle is *habeas quid*: before reasoning with a thing, one must first have the thing. In other words, before a term is used as a value in a proof, there must be a proof that it denotes a well-defined object.

This is especially relevant for recursive definitions. In many classical or constructive settings, recursive functions must be restricted or accompanied by a termination argument before they can be used as total functions. GD takes a different perspective: recursive definitions may be introduced, but later uses of their results require explicit groundedness obligations. If a recursive definition does not denote a value, the corresponding habeas quid proof cannot be completed.

Grounded Arithmetic (GA) is the arithmetic setting of this idea. It supports reasoning about

natural-number computations while making groundedness explicit. In this report, GD is used as a disciplined way to reason about recursively defined executions. Before proving safety or liveness of a trace, we first prove that the recursive execution actually produces a finite trace.

Judgments are written in the form

$$\Gamma \vdash t\ A,$$

meaning that under assumptions Γ, the term t is established to have the kind or property A. For example,

$$\Gamma \vdash T\ \mathsf{Trace}$$

means that T is a well-defined finite trace. Similarly,

$$\Gamma \vdash P(T)\ B$$

means that the predicate $P(T)$ is a well-defined boolean. This does not say that $P(T)$ is true; it only says that the predicate is grounded and can be used in later reasoning.

2.4 Recursive Definitions and Finite Observations

The systems studied in this report are recursive: execution is described by functions that call each other. Such recursive definitions may describe nonterminating behaviour. To reason about them in a grounded way, we use finite observations.

A finite observation is obtained by bounding execution with a natural number parameter, called fuel. Fuel does not mean that the original system terminates. It only limits how much of the execution is observed. For each finite fuel value, the execution should produce a finite trace.

This gives the basic habeas quid shape used later:

$$\frac{\Gamma \vdash I\ \mathsf{InputStream} \qquad \Gamma \vdash s\ \mathbb{N}}{\Gamma \vdash f_S(I,s)\ \mathsf{Trace}}$$

The details of the execution function f_S are part of the design chapter. The background point is only that finite fuel turns recursive execution into a finite object suitable for grounded reasoning.

2.5 HRM as a Minimal Running Example

Human Resource Machine (HRM) is a small programming model based on simple instructions and an implicit processor state. Programs receive values from an inbox, send values to an outbox, and use jumps to repeat behaviour. This makes HRM a useful running example: it is simple enough to

model completely, but still contains input, output, state, recursion through loops, and observable behaviour.

This report uses an HRM-inspired fragment rather than the full game language. The relevant instructions are:

$$\texttt{INBOX}, \quad \texttt{OUTBOX}, \quad \texttt{JUMP}.$$

The instruction `INBOX` consumes an input value, `OUTBOX` produces an output value, and `JUMP` changes the program counter.

The design chapter defines precisely how this HRM-style machine is represented using recursive functions, state, continuations, and traces. The background purpose of HRM is only to provide a concrete setting in which the GD proof structure can be studied.

Chapter 3

Design

This chapter presents the design of the HRM model and of the trace-based verification framework used in the project. The goal of the design is not to faithfully reproduce every feature of the Human Resource Machine game, but to construct a small formal system that captures the verification problems of interest: recursive execution, interaction with input and output devices, finite observations, and trace-level safety and liveness.

The chapter is organized around the main design decisions. First, the system is modeled as a trace-producing machine rather than as a function returning only a final value. Second, execution is parameterized by finite fuel, which makes finite observations grounded. Third, the processor and devices are separated into mutually recursive components. Fourth, safety and liveness are stated at the application level on traces, rather than directly on the internal machine state.

3.1 Design Goals

The design has four main goals.

The first goal is to obtain a model that is simple enough to reason about formally. HRM programs can contain many instructions and later levels introduce additional storage mechanisms, but the core input/output behaviour already appears in a very small fragment. For this reason, the model focuses on the instructions

`INBOX, OUTBOX, JUMP.`

This fragment is sufficient to model looping input/output programs such as copy and duplicate, while keeping the operational semantics manageable.

The second goal is to make execution observable. Instead of treating execution as a black-box

transformation from an input stream to an output stream, the model records events as they occur. This is important because the properties studied in this report are trace properties: safety concerns the outputs that have already appeared, and liveness concerns outputs that must appear in future extensions.

The third goal is to make definedness explicit. Since the system is described by recursive functions, the first proof obligation is not that the trace is correct, but that the recursive execution produces a finite trace for finite fuel. This is the habeas quid of the execution model.

The fourth goal is to separate the internal execution model from the application-level specification. The processor state is needed to execute the machine, but safety and liveness should be stated on the observable trace. This keeps the specification independent of low-level implementation details such as the program counter or accumulator.

3.2 Overall Architecture

The system is divided into three conceptual components:

$$f_P \quad \text{processor}, \qquad f_I \quad \text{input device}, \qquad f_O \quad \text{output device.}$$

The processor is responsible for interpreting the current instruction. The input device is responsible for consuming input values and producing input events. The output device is responsible for emitting output values and producing output events.

The master execution function is defined as

$$f_S(I, s) = f_P(I, s, S_0, [], \mathsf{nil}),$$

where I is the encoded input stream, s is the fuel, S_0 is the initial state, $[]$ is the empty trace, and nil is the initial continuation tag.

The model is mutually recursive: the processor may call the input or output device, and the devices return control to the processor. Conceptually, execution has the following shape:

$$f_P \to f_I \to f_P \to f_O \to f_P$$

for a simple copy program. The input device does not directly call the output device. After each device action, control returns to the processor, which decides the next instruction according to the program counter.

This separation is a design choice. One could define a single monolithic step function that

handles all instructions and device actions at once. However, the separated design better reflects the structure of an interactive system: the processor, input device, and output device are distinct conceptual components, even though their definitions are mutually recursive.

3.3 State

The machine state is defined as

$$S = \langle pc, acc, n_{in}, n_{out} \rangle.$$

Each component has a specific role:

pc	program counter, indicating the current instruction,
acc	accumulator, holding either a data value or empty,
n_{in}	number of input events already produced,
n_{out}	number of output events already produced.

The program counter determines which instruction the processor executes next. The accumulator stores the current value being manipulated by the program. The input and output counters provide sequence numbers for events. These sequence numbers are useful because the same data value may appear multiple times; for example, the input value 5 may occur as both the first and third input. The event indices distinguish these occurrences.

The state belongs to the execution model. It is not the object on which the final application-level specification is stated. Safety and liveness are stated on the trace. However, the state is necessary for defining how the machine produces the trace.

3.4 Well-Formedness Assumptions

The execution functions are defined under standard well-formedness assumptions. These assumptions are not safety properties; they are the conditions under which the operational rules are meaningful.

First, the input I is treated as an input stream rather than as an arbitrary finite list. Thus, whenever the input device is called, the operations $\mathsf{head}(I)$ and $\mathsf{tail}(I)$ are grounded:

$$\frac{\Gamma \vdash I \ \mathsf{InputStream}}{\Gamma \vdash \mathsf{head}(I) \ \mathsf{Data}} \qquad \frac{\Gamma \vdash I \ \mathsf{InputStream}}{\Gamma \vdash \mathsf{tail}(I) \ \mathsf{InputStream}}$$

Second, the program is assumed to be well formed. Whenever the processor dispatches on a program counter, the instruction lookup is defined:

$$\frac{\Gamma \vdash S\ \mathsf{State}}{\Gamma \vdash \mathsf{instr}(pc(S))\ \mathsf{Instr}}$$

In particular, jump targets are valid program counters.

Third, `OUTBOX` is executed only when the accumulator contains a data value. This ensures that the expression $acc(S)$ used by the output device is grounded as a data value:

$$\frac{\Gamma \vdash S\ \mathsf{State} \qquad acc(S) \neq \mathsf{empty}}{\Gamma \vdash acc(S)\ \mathsf{Data}}$$

Fourth, continuation tags are assumed to be consistent with the instruction being resumed. The tag $\mathsf{input}(d)$ resumes an `INBOX` instruction, and the tag done resumes an `OUTBOX` instruction. This prevents impossible configurations such as resuming an `OUTBOX` instruction with an input value.

Finally, execution configurations are assumed to satisfy a state-trace consistency invariant. If the current state is S and the current trace is T, then

$$n_{in}(S) = |\mathsf{Inputs}(T)|, \qquad n_{out}(S) = |\mathsf{Outputs}(T)|.$$

This invariant connects the counters stored in the state with the event indices stored in the trace. It is what allows the input and output devices to append well-indexed events.

3.5 Events and Traces

The observable behaviour of the system is represented by a trace. A finite trace is a list of events:

$$T = [e_0, e_1, \ldots, e_n].$$

The event datatype contains input, output, and control-flow events:

$$\begin{array}{rcl} e & ::= & EI(k,d) \\ & | & EO(k,d) \\ & | & EP_{\mathsf{jump}}(pc, pc'). \end{array}$$

The event $EI(k,d)$ records that the k-th input value was d. The event $EO(k,d)$ records that the k-th output value was d. The event $EP_{\mathsf{jump}}(pc, pc')$ records a jump from program counter pc to program counter pc'.

A trace is well indexed when the indices stored in input and output events agree with their position in the corresponding input or output history. Formally,

$$\mathsf{WellIndexed}(T)$$

means that for every decomposition

$$T = T_0 + +[EI(i,d)] + +T_1,$$

we have

$$i = |\mathsf{Inputs}(T_0)|,$$

and for every decomposition

$$T = T_0 + +[EO(j,d)] + +T_1,$$

we have

$$j = |\mathsf{Outputs}(T_0)|.$$

This invariant rules out traces where, for example, the first output event is labelled $EO(10,d)$. The execution model preserves the invariant through the state-trace consistency condition

$$n_{in}(S) = |\mathsf{Inputs}(T)|, \qquad n_{out}(S) = |\mathsf{Outputs}(T)|.$$

The initial state and empty trace satisfy this condition. The input device emits $EI(n_{in}(S),d)$ and then increments n_{in}; the output device emits $EO(n_{out}(S),d)$ and then increments n_{out}. Therefore, each new input or output event receives exactly the index corresponding to its position in the current trace history.

In the rest of the report, the judgment

$$\Gamma \vdash T\ \mathsf{Trace}$$

means that T is a grounded finite trace satisfying this well-indexing invariant.

The trace is the central application-level object of the design. It records what an external observer can see: which inputs were consumed, which outputs were produced, and which relevant control-flow events occurred. Internal state changes are not all recorded directly. For example, storing a value in the accumulator is an internal update, whereas consuming an input or producing an output is an event.

A central design decision is that each component produces its own events. The input device appends input events. The output device appends output events. The processor appends processor events such as jumps. This makes the trace structure modular and becomes important if the model is later extended with more components or threads.

3.6 Continuations

The model uses a first-order continuation tag:

$$c ::= \mathsf{nil} \mid \mathsf{input}(d) \mid \mathsf{done}.$$

This is a defunctionalized form of continuation-passing style. The continuations are not higher-order functions; they are data constructors that tell the processor how to resume after a device action.

The tag nil means that the processor is starting a new instruction. The tag $\mathsf{input}(d)$ means that the input device has returned the value d to the processor. The tag done means that the output device has completed an output action.

This design keeps the model first-order while still making control flow explicit. It also avoids hiding the interaction between processor and devices inside an implicit call stack. The processor always resumes by inspecting the continuation tag and the current program counter.

3.7 Fuel

Execution is parameterized by a natural number s, called fuel. Fuel limits the number of recursive execution steps. The base case is

$$f_P(I, 0, S, T, c) = T.$$

When no fuel remains, execution returns the trace accumulated so far.

Fuel is not intended to prove that the original HRM program terminates. Instead, it provides finite observations of possibly nonterminating behaviour. This is essential for GD: for each finite fuel value, the execution should produce a finite trace, and this can be proved by induction on the fuel.

The main habeas quid goal for execution is:

$$\frac{\Gamma \vdash I \ \mathsf{InputStream} \qquad \Gamma \vdash s \ \mathbb{N}}{\Gamma \vdash f_S(I, s) \ \mathsf{Trace}}$$

The proof is structurally aligned with the design. Every recursive call consumes strictly smaller fuel, and every event append preserves the trace type.

The design aims for prefix monotonicity:

$$s \le s' \quad \Longrightarrow \quad f_S(I, s) \preceq f_S(I, s').$$

This means that running with more fuel extends the trace without modifying the events already observed. This comparison only makes sense when the input stream is kept fixed. In other words, the sequence

$$f_S(I,0), \quad f_S(I,1), \quad f_S(I,2), \quad \ldots$$

represents successive finite observations of the same execution environment. Increasing fuel does not change the input source; it only allows the same execution to run for longer.

3.8 Processor Definition

The processor function has the form

$$f_P(I,s,S,T,c).$$

It receives the remaining input I, remaining fuel s, current state S, current trace T, and continuation tag c.

The base case is:

$$f_P(I,0,S,T,c) = T.$$

For positive fuel, the processor inspects the instruction at the current program counter. In the case where the continuation is nil, the main dispatcher is:

$$\begin{aligned} f_P(I,s+1,S,T,\mathsf{nil}) \quad = \quad & \mathsf{case}\ \mathsf{instr}(pc(S))\ \mathsf{of} \\ & \texttt{INBOX} \to f_I(I,s,S,T) \\ & \texttt{OUTBOX} \to f_O(I,s,S,T) \\ & \texttt{JUMP}\ p \to f_P(I,s,S[pc := p], T++[EP_{\mathsf{jump}}(pc(S),p)], \mathsf{nil}) \end{aligned}$$

The jump case is handled directly by the processor because it is an internal control-flow instruction. The processor appends a jump event, updates the program counter, and recursively continues with smaller fuel.

The input and output cases are delegated to the corresponding devices. The processor does not itself append EI or EO events in these cases.

In this design, fuel is consumed by processor dispatches. When the processor executes an `INBOX` or `OUTBOX` instruction, it calls the corresponding device with already-decreased fuel. The device then executes its action and returns control to the processor without decreasing fuel again. Thus, a processor-device interaction counts as one execution step, not two.

The processor also handles continuation tags returned by devices. If the current instruction is `INBOX` and the continuation is $\mathsf{input}(d)$, then the input device has returned a value. The processor

stores it in the accumulator and advances the program counter:

$$f_P(I, s+1, S, T, \mathsf{input}(d)) = f_P(I, s, S', T, \mathsf{nil}),$$

where

$$S' = S[acc := d,\ pc := pc(S)+1].$$

Similarly, if the current instruction is `OUTBOX` and the continuation is done, then the output device has completed the output action. The processor clears the accumulator and advances the program counter:

$$f_P(I, s+1, S, T, \mathsf{done}) = f_P(I, s, S', T, \mathsf{nil}),$$

where

$$S' = S[acc := \mathsf{empty},\ pc := pc(S)+1].$$

3.9 Devices

The processor delegates input and output effects to two device functions, f_I and f_O. Both devices follow the same pattern: they perform one observable action, append the corresponding event to the trace, and return to the processor with the appropriate continuation tag.

3.9.1 Input Device

The input device is responsible for consuming one input value and recording the corresponding event in the trace. Let

$$d = \mathsf{head}(I), \qquad I' = \mathsf{tail}(I), \qquad k = n_{in}(S).$$

Then:

$$f_I(I, s, S, T) = f_P(I', s, S', T ++ [EI(k, d)], \mathsf{input}(d)),$$

where

$$S' = S[n_{in} := k+1].$$

Thus, f_I consumes the next input value, increments the input counter, appends the event $EI(k, d)$, and returns control to the processor with the continuation tag $\mathsf{input}(d)$.

3.9.2 Output Device

The output device is responsible for emitting the current accumulator value and recording the corresponding event in the trace. Let

$$d = acc(S), \qquad k = n_{out}(S).$$

Then:

$$f_O(I, s, S, T) = f_P(I, s, S', T ++ [EO(k, d)], \mathsf{done}),$$

where

$$S' = S[n_{out} := k + 1].$$

Thus, f_O reads the accumulator, increments the output counter, appends the event $EO(k, d)$, and returns control to the processor with the continuation tag done.

3.10 Example: Copy Program

Consider the simple copy program:

```
0 : INBOX
1 : OUTBOX
2 : JUMP 0
```

Suppose the input stream begins with

$$5, 7, \ldots$$

and the initial state has $pc = 0$, an empty accumulator, $n_{in} = 0$, and $n_{out} = 0$.

The execution proceeds as follows. The processor sees `INBOX` at program counter 0 and calls the input device. The input device reads 5 and appends:

$$EI(0, 5).$$

It returns to the processor with continuation $\mathsf{input}(5)$. The processor stores 5 in the accumulator and advances to program counter 1.

The processor then sees `OUTBOX` and calls the output device. The output device reads the accumulator and appends:

$$EO(0, 5).$$

It returns to the processor with continuation done. The processor clears the accumulator and advances to program counter 2.

At program counter 2, the processor executes `JUMP 0`, appends:

$$EP_{\mathsf{jump}}(2,0),$$

and returns to program counter 0. The next input value is then processed in the same way.

A possible trace prefix is:

$$[EI(0,5),\ EO(0,5),\ EP_{\mathsf{jump}}(2,0),\ EI(1,7),\ EO(1,7)].$$

This example illustrates the intended control flow:

$$f_P \to f_I \to f_P \to f_O \to f_P.$$

The input device does not call the output device directly. The processor always retakes control and dispatches the next instruction according to the program counter.

3.11 Application-Level Specification

The model separates execution from specification. The execution functions define which traces the HRM machine produces. The application specification defines which traces are considered safe or live.

Let

$$\mathsf{Inputs}(T)$$

be the list of input values appearing in the finite trace T, and let

$$\mathsf{Outputs}(T)$$

be the list of output values appearing in T. Control-flow events are ignored by these functions.

An application specification is represented by a function

$$F : \mathsf{InputHistory} \to \mathsf{OutputHistory}.$$

The value $F(xs)$ is the output sequence allowed by the application after seeing input history xs.

For example:

$$\begin{aligned} F_{\mathsf{copy}}([x_0,x_1]) &= [x_0,x_1], \\ F_{\mathsf{dup}}([x_0,x_1]) &= [x_0,x_0,x_1,x_1]. \end{aligned}$$

3.11.1 Causal Finite Safety

The intended finite safety property is causal: an output must be justified by the inputs that occurred before the output was produced. A convenient way to express this is to require every finite prefix of the trace to be safe:

$$P_{\mathsf{safe},F}(T) \equiv \forall T'.\ T' \preceq T \Longrightarrow \mathsf{Outputs}(T') \preceq F(\mathsf{Inputs}(T')).$$

This is stronger than checking only the final trace

$$\mathsf{Outputs}(T) \preceq F(\mathsf{Inputs}(T)).$$

The final check is useful as a compact snapshot, but by itself it does not guarantee that each output was justified at the time it appeared. The prefix formulation captures the intended safety: the machine may compute, but it must not fabricate outputs from future inputs.

For well-indexed traces, safety can equivalently be stated event-wise. To make the statement readable, we first name the condition that an output is expected by the application specification:

$$\mathsf{Expected}_F(j, T_0, d) \ \equiv\ j = |\mathsf{Outputs}(T_0)| \ \wedge\ \mathsf{nth}(F(\mathsf{Inputs}(T_0)), j) = \mathsf{Some}(d).$$

Then finite safety can be written as:

$$\begin{aligned} P_{\mathsf{safe},F}(T) \ \equiv\ \forall T_0, T_1, j, d.\ T = T_0 ++ [EO(j,d)] ++ T_1 \\ \Longrightarrow \mathsf{Expected}_F(j, T_0, d). \end{aligned}$$

Here T_0 is the part of the trace before the output event, and T_1 is the part after it. The condition $j = |\mathsf{Outputs}(T_0)|$ ensures that the index carried by $EO(j,d)$ agrees with the number of outputs already produced. The second part of $\mathsf{Expected}_F$ then checks that d is the value required by the specification at exactly that output position.

3.11.2 Recursive Checker

The same finite safety property can also be written as a recursive checker over finite traces:

$$P_{\mathsf{safe},F}([]) = \mathsf{true},$$

$$P_{\mathsf{safe},F}(T ++ [e]) = P_{\mathsf{safe},F}(T) \wedge \mathsf{Check}_F(e, T).$$

The local check is defined by cases. For an output event, the newly extended output history must

still be allowed by the current input history:

$$\mathsf{Check}_F(EO(j,d),T) = \big(j = |\mathsf{Outputs}(T)|\big) \wedge \big(\mathsf{Outputs}(T) ++ [d] \preceq F(\mathsf{Inputs}(T))\big).$$

For an input event, the input history changes. The existing outputs must remain compatible with the new input history:

$$\mathsf{Check}_F(EI(i,d),T) = \big(i = |\mathsf{Inputs}(T)|\big) \wedge \big(\mathsf{Outputs}(T) \preceq F(\mathsf{Inputs}(T ++ [EI(i,d)]))\big).$$

For control-flow events, no input or output history changes:

$$\mathsf{Check}_F(EP_{\mathsf{jump}}(pc,pc'),T) = \mathsf{true}.$$

If the application specification F is prefix-monotone with respect to input history, then the input case is automatically preserved. However, keeping the case explicit makes the checker valid for a more general class of specifications.

3.12 Finite and Infinite Traces

Finite traces and infinite traces have different roles in the design.

Finite traces are grounded objects. They can be inspected by terminating functions, and predicates over them can have habeas quid proofs. For example, extracting inputs, extracting outputs, looking up events, and checking the prefix relation all terminate on finite lists.

Infinite traces represent unbounded executions. They are needed for liveness, because liveness is about what eventually happens. A finite trace can show that a required output has not yet appeared, but it cannot show that the output will never appear.

The connection between finite and infinite traces is made through prefixes. For an infinite trace τ, the finite trace

$$\mathsf{prefix}(\tau,n)$$

contains the first n events of τ. This lets us express infinite properties using finite observations.

3.12.1 Infinite Safety Through Finite Prefixes

Safety is first defined on finite traces. Infinite safety is then obtained by requiring every finite prefix of the infinite trace to be safe:

$$P^{\infty}_{\mathsf{safe},F}(\tau) \equiv \forall n.\, P_{\mathsf{safe},F}(\mathsf{prefix}(\tau, n)).$$

This definition is compatible with GD because each fixed prefix is finite. For each n, the following habeas quid obligation is finite:

$$\frac{\Gamma \vdash \tau\ \mathsf{Trace}^{\infty} \qquad \Gamma \vdash n\ \mathbb{N}}{\Gamma \vdash \mathsf{prefix}(\tau, n)\ \mathsf{Trace}}$$

Then the finite safety checker can be applied:

$$\frac{\Gamma \vdash \mathsf{prefix}(\tau, n)\ \mathsf{Trace} \qquad \Gamma \vdash F\ \mathsf{Spec}}{\Gamma \vdash P_{\mathsf{safe},F}(\mathsf{prefix}(\tau, n))\ B}$$

Thus, infinite safety is not checked by traversing an infinite trace. It is expressed as a universal property over finite prefixes, and each finite prefix has its own habeas quid proof.

3.12.2 Liveness on Infinite Traces

Liveness is stated on infinite traces, because it is an eventuality property. To make the definition readable, we first define the obligation generated by a finite prefix:

$$\mathsf{Required}_F(j, n, d, \tau) \ \equiv\ \mathsf{nth}(F(\mathsf{Inputs}(\mathsf{prefix}(\tau, n))), j) = \mathsf{Some}(d).$$

The predicate $\mathsf{Required}_F(j, n, d, \tau)$ means that, after observing the first n events of τ, the specification requires output number j to have value d.

However, an output obligation that has already been fulfilled should not be required again. We therefore define:

$$\mathsf{ProducedBefore}(j, d, \tau, n) \ \equiv\ \exists m < n.\, \mathsf{nth}(\tau, m) = EO(j, d).$$

Then the unfulfilled obligations at position n are:

$$\mathsf{Unfulfilled}_F(j, n, d, \tau) \ \equiv\ \mathsf{Required}_F(j, n, d, \tau) \wedge \neg \mathsf{ProducedBefore}(j, d, \tau, n).$$

Liveness is then:

$$P_{\mathsf{live},F}(\tau) \equiv \forall j,n,d.\, \mathsf{Unfulfilled}_F(j,n,d,\tau) \\ \Longrightarrow \exists m \geq n.\, \mathsf{nth}(\tau,m) = EO(j,d).$$

In words, whenever a finite prefix of the infinite trace contains an output obligation that has not yet been fulfilled, the corresponding output event must eventually appear later in the same trace.

3.12.3 Finite Liveness Through Prefixes

A finite trace cannot satisfy full liveness by itself. It may stop before a required output has had time to appear. Therefore, finite liveness is understood relative to an infinite extension.

Let T be a finite prefix of an infinite trace τ. The clearest case is:

$$T = \mathsf{prefix}(\tau, n).$$

If τ satisfies liveness, then every unfulfilled obligation already visible in T is eventually satisfied later in τ.

Formally:

$$\frac{T = \mathsf{prefix}(\tau,n) \qquad P_{\mathsf{live},F}(\tau)}{P^{\mathsf{finite}}_{\mathsf{live},F}(T,\tau,n)}$$

where:

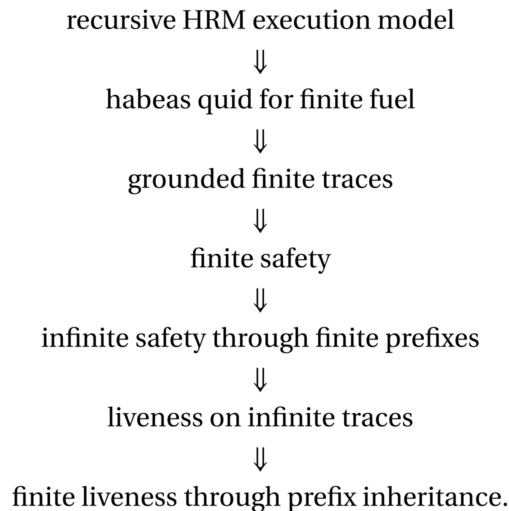
$$P^{\mathsf{finite}}_{\mathsf{live},F}(T,\tau,n) \equiv \forall j,d.\, \Big(\mathsf{nth}(F(\mathsf{Inputs}(T)),j) = \mathsf{Some}(d) \wedge EO(j,d) \notin T\Big) \Longrightarrow \exists m \geq n.\, \mathsf{nth}(\tau,m) = EO(j,d).$$

The proof is direct. Since $T = \mathsf{prefix}(\tau,n)$, every unfulfilled output required by $F(\mathsf{Inputs}(T))$ is also an unfulfilled obligation at position n of τ. By $P_{\mathsf{live},F}(\tau)$, this required output appears at some future position $m \geq n$.

Thus, finite liveness is not a property checked from T alone. It is a relative property: T inherits liveness because it is a prefix of an infinite trace that satisfies $P_{\mathsf{live},F}$.

3.13 How the Design Fits Together

The design can be summarized as a layered structure:



The execution model defines how traces are produced. Fuel ensures that finite executions are grounded. The habeas quid proof establishes that a finite trace exists as a grounded object.

Safety is then stated first on finite traces. Infinite safety is obtained by checking finite safety on every finite prefix of an infinite trace. Liveness goes in the opposite direction: it is stated on infinite traces, and finite traces inherit liveness when they are prefixes of live infinite traces.

This separation is the main design principle of the project. Definedness is proved before correctness properties are stated. The internal state explains how the HRM machine executes, but safety and liveness are stated on the observable trace.

Chapter 4

Formalization and Evaluation

This chapter presents the formalization and evaluation of the design introduced in Chapter 3. The evaluation is proof-oriented: we show that the definitions from the design generate meaningful proof obligations, and that these obligations can be discharged for the main trace properties.

The chapter has two roles. First, it makes explicit the proof obligations generated by the model. Second, it evaluates the design by proving the main habeas quid judgments and by checking that the safety and liveness predicates behave as intended on representative traces.

The central proof pattern is:

$$\text{habeas quid of execution} \quad \Longrightarrow \quad \text{grounded finite traces} \quad \Longrightarrow \quad \text{safety and liveness through prefix reasoning.}$$

4.1 Evaluation Method

The evaluation follows the layers of the design.

The first layer is the execution model. We must show that the mutually recursive functions defining execution produce a finite trace whenever they are run with finite fuel. This is the habeas quid of the execution model.

The second layer consists of trace-processing functions. Once a finite trace exists, we must show that functions such as input extraction, output extraction, lookup, and prefix checking terminate on it. These are routine but important GD obligations, because later predicates depend on these functions being grounded.

The third layer is finite safety. We must show that the safety predicate is a grounded boolean on finite traces. This is different from proving that a particular trace is safe. In GD, we first prove that

the predicate can be evaluated, and only then reason about whether it evaluates to true.

The fourth layer connects finite and infinite traces. Infinite safety is reduced to all finite prefixes, while liveness is stated as an eventuality property over infinite traces. Finite traces do not satisfy full liveness in isolation, but they can inherit liveness from a live infinite extension.

4.2 Reusable Habeas Quid Lemmas

Many proof obligations in GD are repetitive. Instead of proving the same groundedness facts in every case, the formalization relies on reusable lemmas. These lemmas are not the main result of the project, but they make the later habeas quid proofs manageable.

In the rest of the report, a judgment

$$\Gamma \vdash T \ \mathsf{Trace}$$

means that T is a grounded finite trace and that it satisfies the well-indexing invariant introduced in Chapter 3. In particular, input and output event indices must agree with their position in the corresponding input or output history.

The basic trace-extension lemma is therefore not that any event can be appended to a trace. The appended event must have the correct index:

$$\frac{\Gamma \vdash T \ \mathsf{Trace} \qquad \Gamma \vdash e \ \mathsf{Event} \qquad \Gamma \vdash \mathsf{IndexOK}(e, T)}{\Gamma \vdash T ++ [e] \ \mathsf{Trace}}$$

The predicate $\mathsf{IndexOK}$ is defined by cases:

$$\mathsf{IndexOK}(EI(i, d), T) \equiv i = |\mathsf{Inputs}(T)|,$$

$$\mathsf{IndexOK}(EO(j, d), T) \equiv j = |\mathsf{Outputs}(T)|,$$

and

$$\mathsf{IndexOK}(EP_{\mathsf{jump}}(pc, pc'), T) \equiv \mathsf{true}.$$

This lemma is used whenever the execution model appends an event. In the input case, the device appends $EI(k, d)$, where $k = n_{in}(S)$. In the output case, the device appends $EO(k, d)$, where $k = n_{out}(S)$. The well-formedness invariant of configurations ensures that

$$n_{in}(S) = |\mathsf{Inputs}(T)|, \qquad n_{out}(S) = |\mathsf{Outputs}(T)|.$$

Therefore, the events produced by the input and output devices satisfy $\mathsf{IndexOK}$. Jump events satisfy

$\mathsf{IndexOK}$ automatically, because they do not contribute to the input or output history.

The trace-processing functions also generate reusable habeas quid lemmas:

$$\frac{\Gamma \vdash T\ \mathsf{Trace}}{\Gamma \vdash \mathsf{Inputs}(T)\ \mathsf{List}} \qquad \frac{\Gamma \vdash T\ \mathsf{Trace}}{\Gamma \vdash \mathsf{Outputs}(T)\ \mathsf{List}}$$

and

$$\frac{\Gamma \vdash T\ \mathsf{Trace} \qquad \Gamma \vdash e\ \mathsf{Event}}{\Gamma \vdash \mathsf{lookup}(e, T)\ B}$$

The extraction and lookup lemmas are proved by structural induction on the finite trace T. The empty trace is immediate. In the non-empty case, the function inspects the head event and recursively continues on the tail. Since the tail of a finite trace is finite, the induction hypothesis applies.

The indexed trace-extension lemma follows from the definition of well-indexed traces. The premise $\mathsf{IndexOK}(e, T)$ ensures that the newly appended event receives the correct input or output index, while all earlier events keep the same indices as before.

The prefix relation on finite lists is handled similarly:

$$\frac{\Gamma \vdash xs\ \mathsf{List} \qquad \Gamma \vdash ys\ \mathsf{List}}{\Gamma \vdash xs \preceq ys\ B}$$

The application specification also requires a groundedness assumption. We write

$$\Gamma \vdash F\ \mathsf{Spec}$$

to mean that F is grounded on finite input histories:

$$\frac{\Gamma \vdash xs\ \mathsf{List}}{\Gamma \vdash F(xs)\ \mathsf{List}}$$

Finally, finite prefixes of infinite traces are grounded finite traces:

$$\frac{\Gamma \vdash \tau\ \mathsf{Trace}^{\infty} \qquad \Gamma \vdash n\ \mathbb{N}}{\Gamma \vdash \mathsf{prefix}(\tau, n)\ \mathsf{Trace}}$$

These lemmas are used throughout the evaluation. The habeas quid of execution uses the indexed trace-extension lemma. The habeas quid of finite safety uses the input/output extraction lemmas, the specification lemma, and the prefix-checking lemma. Infinite safety uses the finite-prefix lemma together with the finite safety habeas quid. Liveness uses the same finite-prefix and specification lemmas to show that the liveness proposition is well formed for each finite prefix and

trace position appearing in the statement.

4.3 Habeas Quid of Finite Execution

The first main proof obligation is the habeas quid of the execution function. Since execution proceeds through intermediate processor configurations, we prove a slightly stronger statement for the processor function.

Let

$$\mathsf{WFConfig}(I, S, T, c)$$

mean that the current execution configuration is well formed. This packages the assumptions required by the execution rules: I is an input stream, $\mathsf{instr}(pc(S))$ is defined whenever the processor dispatches, jump targets are valid program counters, continuation tags match the instruction being resumed, output instructions are reached only when the accumulator contains a data value, and T is a well-indexed finite trace. It also includes the state-trace consistency invariant

$$n_{in}(S) = |\mathsf{Inputs}(T)|, \qquad n_{out}(S) = |\mathsf{Outputs}(T)|.$$

The strengthened habeas quid statement is:

$$\frac{\Gamma \vdash I\ \mathsf{InputStream} \qquad \Gamma \vdash s\ \mathbb{N} \qquad \Gamma \vdash \mathsf{WFConfig}(I, S, T, c)}{\Gamma \vdash f_P(I, s, S, T, c)\ \mathsf{Trace}}$$

The desired result for the system function follows as a special case, because $f_S(I, s)$ starts from the initial well-formed configuration:

$$f_S(I, s) = f_P(I, s, S_0, [], \mathsf{nil}).$$

Thus:

$$\frac{\Gamma \vdash I\ \mathsf{InputStream} \qquad \Gamma \vdash s\ \mathbb{N} \qquad \Gamma \vdash \mathsf{WFConfig}(I, S_0, [], \mathsf{nil})}{\Gamma \vdash f_S(I, s)\ \mathsf{Trace}}$$

The proof is by induction on the fuel s.

Base case. When $s = 0$, the processor returns the trace accumulated so far:

$$f_P(I, 0, S, T, c) = T.$$

Since $\mathsf{WFConfig}(I, S, T, c)$ includes that T is a finite trace, the result is grounded.

Inductive step. Assume that the statement holds for fuel s. We prove it for fuel $s+1$ by case analysis on the continuation tag c.

If $c = \mathsf{nil}$, the processor dispatches on the current instruction.

In the `INBOX` case, well-formedness gives that I is an input stream. Therefore,

$$\mathsf{head}(I) \qquad \text{and} \qquad \mathsf{tail}(I)$$

are grounded. Let

$$d = \mathsf{head}(I), \qquad I' = \mathsf{tail}(I), \qquad k = n_{in}(S).$$

The input device appends the event $EI(k, d)$. By state-trace consistency,

$$k = n_{in}(S) = |\mathsf{Inputs}(T)|.$$

Therefore, $\mathsf{IndexOK}(EI(k, d), T)$ holds, and the indexed trace-extension lemma gives:

$$\Gamma \vdash T ++ [EI(k, d)] \ \mathsf{Trace}.$$

The resulting configuration is again well formed, and the recursive call uses fuel s. Therefore, the induction hypothesis applies.

In the `OUTBOX` case, well-formedness gives that the accumulator contains a data value. Hence $acc(S)$ is grounded. Let

$$d = acc(S), \qquad k = n_{out}(S).$$

The output device appends $EO(k, d)$. By state-trace consistency,

$$k = n_{out}(S) = |\mathsf{Outputs}(T)|.$$

Therefore, $\mathsf{IndexOK}(EO(k, d), T)$ holds, and the indexed trace-extension lemma gives that the extended trace is finite. The recursive call uses fuel s, so the induction hypothesis applies.

In the `JUMP` case, program well-formedness gives that the target p is a valid program counter. The processor appends

$$EP_{\mathsf{jump}}(pc(S), p)$$

and updates the program counter to p. Jump events do not contribute to the input or output history, so

$$\mathsf{IndexOK}(EP_{\mathsf{jump}}(pc(S), p), T)$$

holds automatically. The indexed trace-extension lemma therefore gives that the extended trace is finite. The next configuration is well formed, and the recursive call uses fuel s. Therefore, the induction hypothesis applies.

If $c = \mathsf{input}(d)$, the processor is resuming after an `INBOX` action. Well-formedness ensures that this continuation is compatible with the current instruction. The processor stores d in the accumulator, advances the program counter, and recursively calls f_P with fuel s. No event is appended in this resume step, so the trace remains finite. The resulting configuration is well formed, and the induction hypothesis applies.

If $c = \mathsf{done}$, the processor is resuming after an `OUTBOX` action. Well-formedness ensures that this continuation is compatible with the current instruction. The processor clears the accumulator, advances the program counter, and recursively calls f_P with fuel s. Again, no event is appended in this resume step, so the trace remains finite. The resulting configuration is well formed, and the induction hypothesis applies.

Thus, every possible execution branch either returns the current trace or makes a recursive call with strictly smaller fuel while preserving trace groundedness. This establishes the habeas quid of finite execution.

4.4 Habeas Quid of Finite Safety

The finite safety predicate from Chapter 3 is defined by a recursive checker:

$$P_{\mathsf{safe},F}([]) = \mathsf{true},$$

$$P_{\mathsf{safe},F}(T ++ [e]) = P_{\mathsf{safe},F}(T) \wedge \mathsf{Check}_F(e, T).$$

The local checker Check_F is defined by cases on the new event. For an output event, the newly extended output history must still be allowed by the current input history, and the output index must agree with the current output position:

$$\mathsf{Check}_F(EO(j,d), T) = \big(j = |\mathsf{Outputs}(T)|\big) \wedge \big(\mathsf{Outputs}(T) ++ [d] \preceq F(\mathsf{Inputs}(T))\big).$$

For an input event, the input index must agree with the current input position, and the existing outputs must remain compatible with the new input history:

$$\mathsf{Check}_F(EI(i,d), T) = \big(i = |\mathsf{Inputs}(T)|\big) \wedge \big(\mathsf{Outputs}(T) \preceq F(\mathsf{Inputs}(T ++ [EI(i,d)]))\big).$$

For a control-flow event:

$$\mathsf{Check}_F(EP_{\mathsf{jump}}(pc, pc'), T) = \mathsf{true}.$$

The habeas quid statement is:

$$\frac{\Gamma \vdash T\ \mathsf{Trace} \qquad \Gamma \vdash F\ \mathsf{Spec}}{\Gamma \vdash P_{\mathsf{safe},F}(T)\ B}$$

Here $\Gamma \vdash F\ \mathsf{Spec}$ means that F is grounded on finite input histories:

$$\frac{\Gamma \vdash xs\ \mathsf{List}}{\Gamma \vdash F(xs)\ \mathsf{List}}$$

The proof is by structural induction on the finite trace T. The empty trace case is immediate because

$$P_{\mathsf{safe},F}([]) = \mathsf{true}.$$

For the inductive case $T ++[e]$, the induction hypothesis gives

$$\Gamma \vdash P_{\mathsf{safe},F}(T)\ B.$$

It remains to show that $\mathsf{Check}_F(e, T)$ is a grounded boolean. This follows by case analysis on e. Input and output cases use the habeas quid lemmas for Inputs, $\mathsf{Outputs}$, F, list length, equality, and $\preceq$. The jump case is immediate because the checker returns true.

Thus, $P_{\mathsf{safe},F}(T)$ is a grounded boolean for every finite trace T.

This proof is important because it separates two questions:

$$\begin{array}{ll} \text{Definedness:} & P_{\mathsf{safe},F}(T) \text{ is a boolean,} \\ \text{Truth:} & P_{\mathsf{safe},F}(T) = \mathsf{true}. \end{array}$$

In GD, the first question must be answered before the second one can be used in a proof.

4.5 Safety Examples

The next step is to evaluate whether the safety predicate has the intended meaning. We use two specifications: copy and duplicate.

For the copy specification,

$$F_{\mathsf{copy}}(xs) = xs.$$

Consider the trace:

$$T_1 = [EI(0,5), EO(0,5), EP_{\mathsf{jump}}(2,0), EI(1,7)].$$

Then:

$$\mathsf{Inputs}(T_1) = [5,7], \qquad \mathsf{Outputs}(T_1) = [5].$$

The compact snapshot check is:

$$[5] \preceq [5,7],$$

which is true. Moreover, the output $EO(0,5)$ is justified by the input history available before it was produced.

Now consider:

$$T_2 = [EI(0,5), EO(0,7)].$$

Here:

$$\mathsf{Inputs}(T_2) = [5], \qquad \mathsf{Outputs}(T_2) = [7].$$

The safety check is:

$$[7] \preceq [5],$$

which is false. The trace is unsafe because it produces an output not justified by the input history.

For the duplicate specification,

$$F_{\mathsf{dup}}([x_0, x_1, \ldots]) = [x_0, x_0, x_1, x_1, \ldots].$$

The trace

$$T_3 = [EI(0,5), EO(0,5)]$$

is safe, because:

$$[5] \preceq [5,5].$$

The second copy has not yet been produced, but this is not a safety violation. It is a liveness obligation.

By contrast, the trace

$$T_4 = [EI(0,5), EO(0,5), EO(1,5), EO(2,5)]$$

is unsafe, because:

$$[5,5,5] \not\preceq [5,5].$$

The third output is not justified by the single input value.

These examples show why prefix-based safety is appropriate. It accepts partial but correct output histories, and it rejects outputs that go beyond the application specification.

4.6 Infinite Safety Through Finite Prefixes

Infinite safety is defined by reducing it to finite safety on all finite prefixes:

$$P^{\infty}_{\mathsf{safe},F}(\tau) \equiv \forall n.\, P_{\mathsf{safe},F}(\mathsf{prefix}(\tau, n)).$$

This definition fits GD because each fixed prefix is finite. The relevant habeas quid rule is:

$$\frac{\Gamma \vdash \tau\ \mathsf{Trace}^{\infty} \qquad \Gamma \vdash n\ \mathbb{N}}{\Gamma \vdash \mathsf{prefix}(\tau, n)\ \mathsf{Trace}}$$

Once this finite prefix is obtained, the finite safety habeas quid applies:

$$\frac{\Gamma \vdash \mathsf{prefix}(\tau, n)\ \mathsf{Trace} \qquad \Gamma \vdash F\ \mathsf{Spec}}{\Gamma \vdash P_{\mathsf{safe},F}(\mathsf{prefix}(\tau, n))\ B}$$

Thus, infinite safety is not checked by traversing an infinite stream. Instead, it is a universal statement over finite observations, and each finite observation has its own habeas quid proof.

4.7 Habeas Quid of Liveness

We define the obligation generated by a finite prefix:

$$\mathsf{Required}_F(j, n, d, \tau) \ \equiv\ \mathsf{nth}(F(\mathsf{Inputs}(\mathsf{prefix}(\tau, n))), j) = \mathsf{Some}(d).$$

The predicate $\mathsf{Required}_F(j, n, d, \tau)$ means that, after observing the first n events of the infinite trace τ, the specification requires output number j to have value d.

An obligation may already have been fulfilled before position n. We therefore define:

$$\mathsf{ProducedBefore}(j, d, \tau, n) \ \equiv\ \exists m < n.\, \mathsf{nth}(\tau, m) = EO(j, d).$$

The predicate $\mathsf{ProducedBefore}(j, d, \tau, n)$ is grounded because it only searches the finite prefix before position n.

The unfulfilled obligations are:

$$\mathsf{Unfulfilled}_F(j, n, d, \tau) \ \equiv\ \mathsf{Required}_F(j, n, d, \tau) \wedge \neg \mathsf{ProducedBefore}(j, d, \tau, n).$$

Liveness is then stated as:

$$P_{\mathsf{live},F}(\tau) \;\equiv\; \forall j,n,d.\ \mathsf{Unfulfilled}_F(j,n,d,\tau) \\ \qquad\qquad \Longrightarrow \exists m \geq n.\ \mathsf{nth}(\tau,m) = EO(j,d).$$

This is not a finite checker. Unlike $P_{\mathsf{safe},F}(T)$, which is a recursive boolean predicate over a finite trace, $P_{\mathsf{live},F}(\tau)$ is a proposition about an infinite trace. Its habeas quid proof therefore does not compute liveness. It only shows that the liveness proposition is well formed.

The habeas quid statement is:

$$\frac{\Gamma \vdash \tau\ \mathsf{Trace}^{\infty} \qquad \Gamma \vdash F\ \mathsf{Spec}}{\Gamma \vdash P_{\mathsf{live},F}(\tau)\ \mathsf{Prop}}$$

Here Prop means that $P_{\mathsf{live},F}(\tau)$ is a well-formed proposition, not that it is a terminating boolean computation.

The proof checks the groundedness of the expressions that occur in the liveness statement. For any fixed n, the prefix of an infinite trace is finite:

$$\frac{\Gamma \vdash \tau\ \mathsf{Trace}^{\infty} \qquad \Gamma \vdash n\ \mathbb{N}}{\Gamma \vdash \mathsf{prefix}(\tau,n)\ \mathsf{Trace}}$$

Since this prefix is a finite trace, input extraction is grounded:

$$\frac{\Gamma \vdash \mathsf{prefix}(\tau,n)\ \mathsf{Trace}}{\Gamma \vdash \mathsf{Inputs}(\mathsf{prefix}(\tau,n))\ \mathsf{List}}$$

Then F can be applied to this finite input history, and the expression

$$\mathsf{nth}(F(\mathsf{Inputs}(\mathsf{prefix}(\tau,n))),j)$$

is well defined.

The predicate $\mathsf{ProducedBefore}(j,d,\tau,n)$ is also grounded, because it quantifies only over positions $m < n$, that is, over a finite prefix.

Similarly, for any fixed m, the event at position m of the infinite trace is well defined:

$$\frac{\Gamma \vdash \tau\ \mathsf{Trace}^{\infty} \qquad \Gamma \vdash m\ \mathbb{N}}{\Gamma \vdash \mathsf{nth}(\tau,m)\ \mathsf{Event}}$$

Therefore, the habeas quid of liveness establishes only that the liveness statement is meaningful:

each finite prefix, each unfulfilled required output, and each future event lookup used in the formula is grounded. It does not prove that liveness holds for a particular trace. Proving $P_{\mathsf{live},F}(\tau)$ is a separate, program-specific proof obligation.

4.8 Finite Liveness Through Prefixes

Finite traces do not satisfy full liveness in isolation. Liveness is an eventuality property over an infinite trace, so a finite trace can only inherit liveness from an infinite extension.

Assume that

$$T = \mathsf{prefix}(\tau, n) \qquad \text{and} \qquad P_{\mathsf{live},F}(\tau).$$

Then every unfulfilled obligation already visible in T is eventually satisfied later in τ:

$$\forall j, d. \left(\mathsf{nth}(F(\mathsf{Inputs}(T)), j) = \mathsf{Some}(d) \land EO(j,d) \notin T\right) \Longrightarrow \exists m \geq n.\ \mathsf{nth}(\tau, m) = EO(j,d).$$

The proof is immediate from the definition of $P_{\mathsf{live},F}$. Since $T = \mathsf{prefix}(\tau, n)$, every unfulfilled obligation computed from T is exactly an unfulfilled obligation generated at position n of τ. Liveness of τ then provides a future position $m \geq n$ where the required output event appears.

Thus, finite liveness is an inheritance principle, not a finite checker.

4.9 Discussion

The formalization and evaluation support the main design claim. The execution model produces finite traces for finite fuel. The functions used to inspect finite traces are grounded. The safety predicate is itself grounded before it is used as a correctness property. Infinite safety is reduced to all finite prefixes, and liveness is expressed as an eventuality property over infinite traces.

The evaluation also clarifies the different directions in which safety and liveness use prefixes. Safety starts from finite traces: once finite safety is grounded, infinite safety is obtained by requiring all finite prefixes to be safe. Liveness starts from infinite traces: once an infinite trace satisfies $P_{\mathsf{live},F}$, each finite prefix inherits the guarantee that its unfulfilled obligations will be fulfilled later in the infinite trace.

4.10 Limitations

The evaluation is intentionally limited to the small HRM fragment used in the design. The goal is not to prove that all HRM programs are automatically safe and live. Rather, the goal is to show that the GD proof structure is coherent for a recursive, trace-producing model.

There are three main limitations.

First, liveness proofs are program-specific. The predicate $P_{\mathsf{live},F}$ states what must be shown, but the proof depends on the program and the specification.

Second, prefix monotonicity assumes that the input stream I is fixed when fuel is increased. If the environment were nondeterministic, an additional environment-consistency or fairness assumption would be required.

Third, the model only includes a small instruction set. Extending it with memory cells, arithmetic operations, or multiple interacting threads would require additional state components, event types, and invariants.

4.11 Summary

This chapter evaluated the formal design by proving the main groundedness and trace-correctness obligations. The results validate the intended proof pattern:

$$\text{habeas quid of finite execution} \quad \Longrightarrow \quad \text{grounded finite traces} \quad \Longrightarrow \quad \text{finite safety.}$$

Infinite safety is then obtained by checking finite safety on every finite prefix:

$$P^{\infty}_{\mathsf{safe},F}(\tau) \equiv \forall n.\ P_{\mathsf{safe},F}(\mathsf{prefix}(\tau, n)).$$

Liveness goes in the opposite direction: it is stated on infinite traces, and finite traces inherit guarantees for their unfulfilled obligations when they are prefixes of live infinite traces.

The main output of the project is not an automated verifier, but a clear proof structure for recursive trace-producing systems in GD.

Chapter 5

Related Work

This chapter briefly positions the project with respect to existing approaches in program verification. The goal is not to provide a complete survey, but to clarify how this work differs from standard state-based verification, trace-based verification, and previous work on Grounded Deduction and Grounded Arithmetic.

5.1 Classical Program Verification

Classical program verification provides mature methods for proving that programs satisfy formal specifications. Hoare logic is one of the foundational examples: it expresses correctness using triples of the form

$$\{P\}\ C\ \{Q\},$$

where P is a precondition, C is a command, and Q is a postcondition. The usual reading is that, if execution of C starts in a state satisfying P and terminates, then the final state satisfies Q. This style of reasoning is powerful because it decomposes program verification into local proof rules for assignments, sequencing, conditionals, and loops.

This project has a different focus. The HRM model is not primarily verified as a relation between an initial state and a final state. Instead, execution is viewed as a process that produces an observable trace of events. The main object of verification is therefore not only the final machine state, but the trace generated over time.

Classical approaches can encode traces, for example by adding a ghost variable to the program state and maintaining invariants about it. However, in this report the trace is not an auxiliary object: it is the central semantic object. Safety and liveness are both stated directly on traces.

Another difference concerns definedness. In standard partial-correctness reasoning, one often proves that if a program terminates, then the result is correct. Termination is then handled separately when total correctness is needed. In this project, GD makes this separation explicit through habeas quid obligations. Before proving that a trace is safe or live, we first prove that the recursive execution produces a grounded finite trace.

Separation logic is another important extension of Hoare-style reasoning. It is especially effective for programs manipulating shared mutable heap structures, because it provides concise assertions about ownership and disjointness of memory resources. This project does not focus on heap ownership. The central issue is instead the definedness and trace behaviour of recursive execution. Thus, separation logic and this work address different verification concerns.

5.2 Trace-Based Verification

Trace-based reasoning is widely used to specify observable behaviour. Instead of describing only initial and final states, a trace records the sequence of events produced during execution. This is especially natural for interactive systems, communication protocols, operating systems, and reactive programs, where correctness depends on what happens over time.

The distinction between safety and liveness is also commonly formulated using traces. A safety property rules out bad finite prefixes: if safety is violated, some finite observation already demonstrates the error. A liveness property says that some desired event eventually happens; such a property cannot usually be refuted by a finite prefix alone, because the desired event may still occur later.

This report follows the trace-based view. Finite traces are used to define finite safety. Infinite safety is obtained by requiring every finite prefix to be safe. Liveness is stated on infinite traces, and finite traces inherit liveness when they are prefixes of live infinite traces.

What is specific to this work is the combination of trace reasoning with Grounded Deduction. The trace is not simply assumed to exist. It is produced by recursive functions, so the first proof obligation is to establish the habeas quid of finite execution:

$$\Gamma \vdash f_S(I, s)\ \mathsf{Trace}.$$

Only after this definedness proof do we reason about safety and liveness.

Bounded model checking is also related because it reasons about finite executions up to a bound. However, its goal is usually algorithmic search for counterexamples or bounded proofs using solvers. In this project, fuel is not used to drive a model checker. It is used to obtain grounded finite observations of a recursive execution in GD.

5.3 Grounded Deduction and Grounded Arithmetic

The closest related work is Grounded Deduction (GD) and Grounded Arithmetic (GA). GD was introduced to reason consistently in the presence of unrestricted recursive definitions. Instead of forbidding such definitions at the definitional level, GD restricts how their results may be used in proofs. A term or proposition must be shown to be grounded before it can be used as an ordinary object of reasoning.

GA applies this idea to arithmetic and computation. It allows recursive definitions while requiring habeas quid proofs when their results are used. This is the main logical idea used in the project. The HRM execution functions are recursive and mutually recursive, but finite executions are justified by proving that finite fuel produces finite traces.

Kehrli's Isabelle/GA formalization is closely related because it studies how grounded reasoning can be mechanized. It develops GA inside Isabelle/Pure and shows that many routine habeas quid obligations can be supported by automation. The present project is smaller and more specific. It does not aim to build a general GA formalization; instead, it studies one trace-producing system model and the proof obligations generated by that model.

The contribution of this project is therefore a case study: it shows how GD can structure the verification of a recursive system whose result is not a single number or final state, but a trace of observable events.

5.4 Position of This Work

The work in this report combines ideas from several areas, but its scope is specific. From classical verification, it inherits the goal of proving program properties mathematically. From trace-based verification, it inherits the idea that observable behaviour should be represented as sequences of events. From GD, it inherits the requirement that objects must be grounded before they are used in proofs.

The result is a small proof structure for recursive, trace-producing systems:

$$\text{habeas quid of execution} \quad \Longrightarrow \quad \text{finite trace} \quad \Longrightarrow \quad \text{safety and liveness through prefix reasoning.}$$

This project does not replace existing verification frameworks. Hoare logic and separation logic remain better suited for many state-based and heap-manipulating programs. Bounded model checking remains better suited for automated counterexample search. The goal here is different: to show how GD can make the definedness of recursively generated traces explicit before proving trace properties about them.

The main trade-off is verbosity. GD exposes proof obligations that other systems may hide inside termination checks, semantic assumptions, or typing disciplines. However, this explicitness is also the benefit: the report separates the proof that a trace exists from the proof that the trace satisfies safety and liveness.

Chapter 6

Conclusion

This report studied how Grounded Deduction can structure the verification of a small recursive, trace-producing system. The system is inspired by the HRM programming model and focuses on a minimal instruction fragment: `INBOX`, `OUTBOX`, and `JUMP`. Even in this small setting, the model contains the main ingredients of interest: recursive execution, mutual recursion between processor and devices, finite observations through fuel, and observable behaviour represented as traces.

The main result is a proof structure that separates definedness from trace correctness. Before proving that an execution is safe or live, we first prove the habeas quid of execution: for every well-formed initial execution configuration and every finite fuel value, the recursive execution function produces a finite trace. This makes the trace a grounded object on which later predicates can be stated.

Once finite traces are grounded, safety is stated as a trace property. Finite safety requires every output already produced to be justified by the input history available before that output. Infinite safety is then obtained by prefix reasoning: an infinite trace is safe when every finite prefix is safe. This fits naturally with GD, because each fixed prefix is a finite object with its own habeas quid proof.

Liveness is treated in the opposite direction.Full liveness is stated on infinite traces, because it is an eventuality property: every required output that has not already appeared must eventually appear. A finite trace cannot satisfy this property in isolation, since it may stop before the required output occurs. Instead, finite traces inherit liveness when they are prefixes of live infinite traces. Thus, safety and liveness are connected to prefixes in dual ways:

Safety:	finite safety is lifted to infinite traces by checking all prefixes,
Liveness:	infinite liveness is inherited by finite prefixes.

The formalization and evaluation show that this structure works as intended. Finite execution

has a habeas quid proof by induction on fuel. Trace-processing functions are grounded by structural induction on finite traces. The finite safety predicate is a grounded boolean, and the copy and duplicate examples show that the predicate supports specifications where one input may justify one or several outputs. Liveness remains program-specific, but the report identifies the right form of the property and explains how finite traces obtain liveness through live infinite extensions.

The project does not aim to provide a complete verifier for HRM programs. Instead, it is a focused case study showing how GD can organize reasoning about recursive, trace-producing computations. The resulting proof pattern is:

$$\text{habeas quid of execution} \quad \Longrightarrow \quad \text{grounded finite traces} \quad \Longrightarrow \quad \text{safety and liveness through prefix reasoning.}$$

6.1 Future Work

A natural extension of this project is to make the modular structure more general and more explicit. The current HRM model already contains non-local mutual recursion: the processor calls the input or output device, the device returns control to the processor, and the global trace is produced by their interaction. However, the model is still specialized to HRM-style state, instructions, and continuation tags.

A cleaner future formulation would describe a system as a collection of pure components operating over a shared global history. Let H_s denote the global history produced up to step s. Each component receives this prior history and computes a contribution to the next step:

$$C_i : \mathsf{History} \to \mathsf{Contribution}.$$

A wrapper function then combines the component contributions and extends the history:

$$H_{s+1} = \mathsf{extend}\big(H_s,\ C_1(H_s),\ C_2(H_s), \ldots, C_n(H_s)\big).$$

The key restriction is that components may inspect only the already-grounded history H_s, while H_{s+1} is obtained by extending that history. This would make the recursion well structured for GD: the induction hypothesis gives that H_s is grounded, each component is applied to a grounded history, and the next history is grounded by an append or extension lemma.

In such a formulation, safety and liveness could also be expressed more conveniently. Safety would be a finite-history predicate:

$$\mathsf{Safe}_F(H)$$

stating that no event in H violates the specification using only the prior history available when the event occurred. Infinite safety would again be defined by checking all finite prefixes.

Liveness could be expressed through obligations generated from finite histories. Instead of writing liveness directly as a nested expression involving inputs, prefixes, and list indexing, one could define:

$$\mathsf{Required}_F(H)$$

as the set of obligations created by the finite history H. Then liveness would say that every obligation generated at a finite prefix is eventually fulfilled in the infinite history:

$$\forall n. \forall o \in \mathsf{Required}_F(\mathsf{prefix}(\tau, n)). \exists m \geq n. \mathsf{Fulfills}(\mathsf{nth}(\tau, m), o).$$

This would make the specification more modular and more readable: safety says that events do not violate the current history, while liveness says that obligations generated by the current history are eventually fulfilled.

This direction would clarify the broader role of GD. GD is not only useful for the specific HRM fragment studied in this report. It is useful for systems where several modules are defined recursively and non-locally, but where finite histories can still be justified by habeas quid proofs. The HRM processor, input device, and output device can therefore be seen as a small instance of a more general modular history-based architecture.

Several limitations remain. The instruction set is intentionally small, and the liveness proof must be carried out for each program and specification. The infinite trace semantics is described through prefix reasoning rather than a complete coinductive development. Extending the model with memory cells, arithmetic operations, richer control flow, or multiple interacting components would require additional events, state components, and invariants.

Nevertheless, the project demonstrates that GD provides a useful perspective for recursive system models. It forces the verification to begin with the existence of the object being studied. Only after the trace is grounded do we reason about whether it satisfies safety and liveness. This is the main lesson of the report: for recursive trace-producing systems, proving that the trace exists is not a minor technical detail, but the first step of the verification itself.

Bibliography

[1] Bryan Alexander Ford. “Have a Thing? Reasoning around Recursion with Dynamic Typing in Grounded Arithmetic”. Manuscript. 2026.

[2] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. ISBN: 978-0-521-89957-4.

[3] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[4] Sascha Kehrli. “Formalizing Grounded Arithmetic atop Isabelle/Pure”. Master’s thesis. École Polytechnique Fédérale de Lausanne, 2025.

[5] Leslie Lamport. “Proving the Correctness of Multiprocess Programs”. In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977), pp. 125–143.

[6] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS 2002. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.