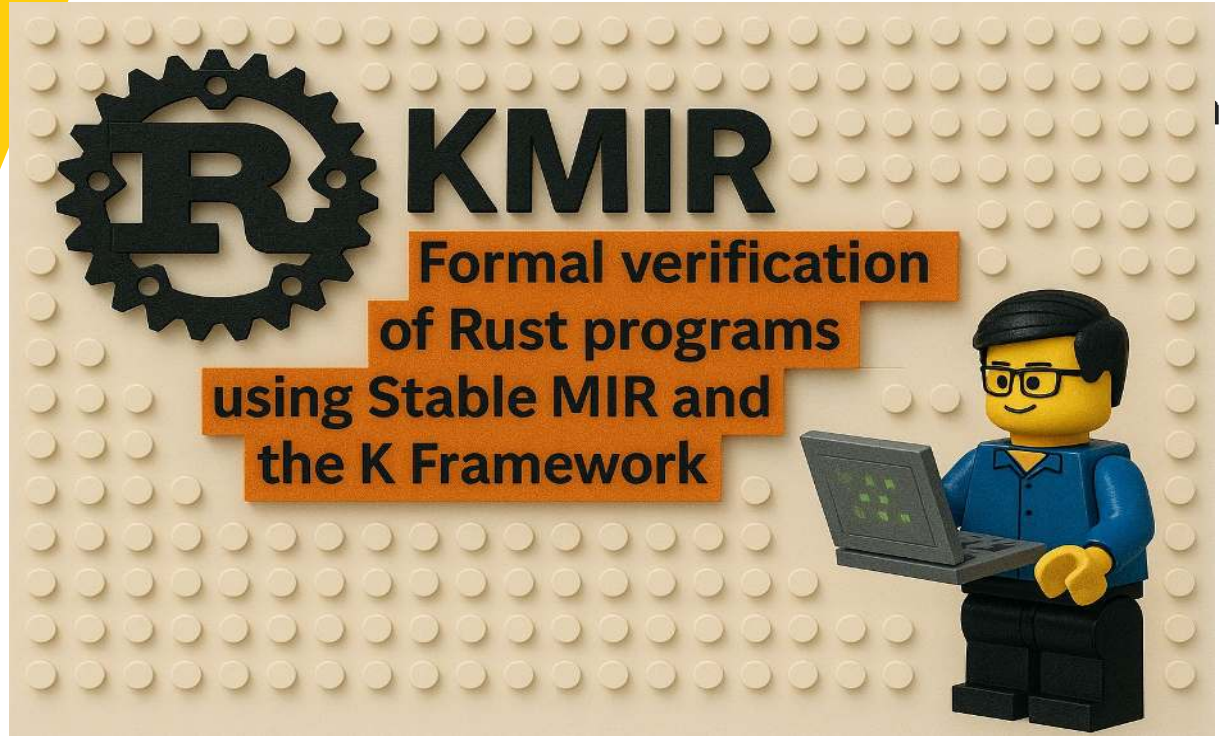


Symbolic Execution & MIR

Rust Sydney, 2025 04 09

Jost Berthold

Runtime Verification Inc.

Presentation Overview

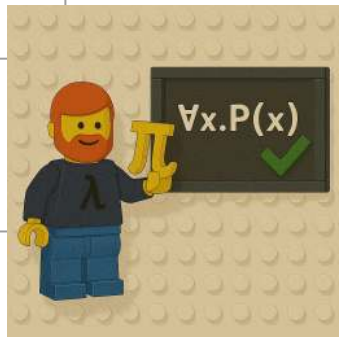
runtime verification

- Intro & Motivation
- KMIR: Symbolic Execution for Rust
- How it Works
  - Stable MIR and stable-mir-json
  - Semantics of (Stable) MIR in K
  - Verification of Properties in KMIR
- KMIR: Status, Limitations, Summary

Why Formal Verification Matters

runtime verification

Method	Confidence	Problem
Unit tests	I checked my internal functions	What about my public endpoints?
Integration tests	I checked my external functions	What about the inputs I haven't checked?
Property tests / fuzzing	I checked my functions for random inputs	What about the inputs I haven't checked?
Formal Verification	I checked my program (exhaustively) against a mathematical specification	What if my specification is incomplete?

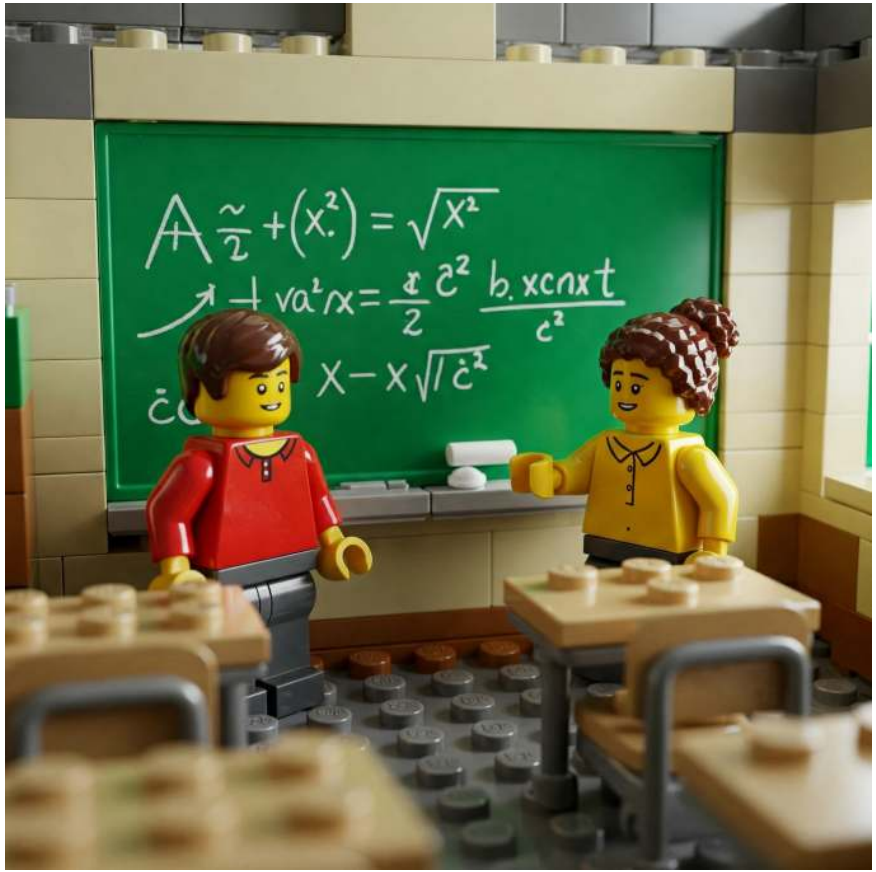


Formal Verification for Rust

runtime verification

- Rust has great safety guarantees!
  - Thread and memory safety via ownership
    However, there is `unsafe`…
  - Various other checks (accessing uninitialised values, incomplete matches, etc.)
- Rust's safety guarantees are not absolute.
  - Testing has limited scope
  - *Safe* code can still do the wrong thing.
- Beyond Testing: **Formal Verification**
  - Formal verification provides guarantees (through *exhaustive* exploration)

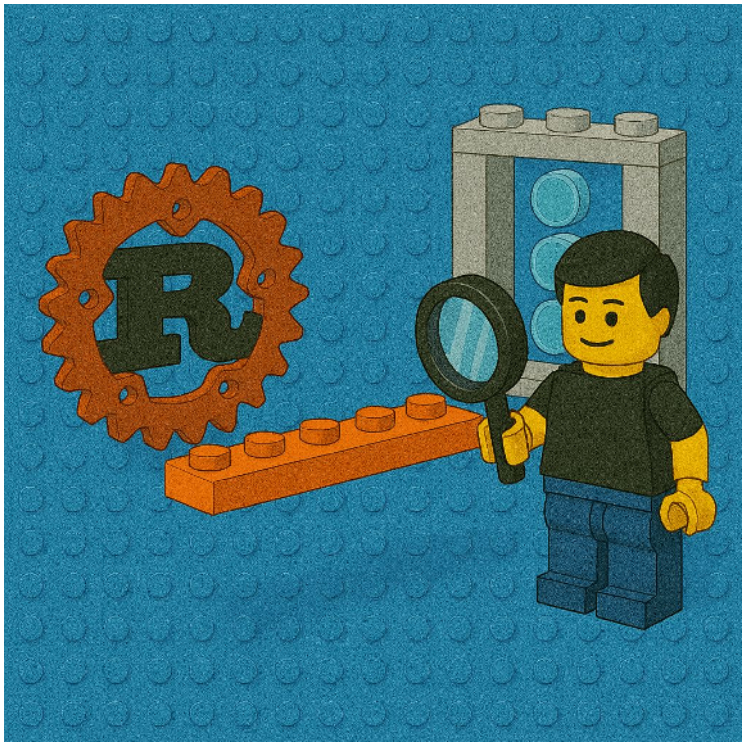
Goal: Verification tool for Rust programs including unsafe code.



KMIR: Symbolic Execution for Rust

runtime verification

- Execution and verification of Rust programs using Stable MIR and the K Framework.
- Focus: Symbolic execution for program verification.
- Enables verification of both safe and unsafe Rust code (by using intermediate code).
- Bridges the gap between Rust's compiler internals and formal verification tools.
- Open Source (BSD3): https://github.com/runtimeverification/mir-semantics



Who are we?

runtime verification

The work on KMIR is part of my job at Runtime Verification Inc.

Runtime Verification Inc. is a software quality assurance company aimed at using formal methods to perform security audits on virtual machines and smart contracts on public blockchains.

It is dedicated to improving the safety, reliability, and correctness of software systems in the blockchain field (and other fields, too!)

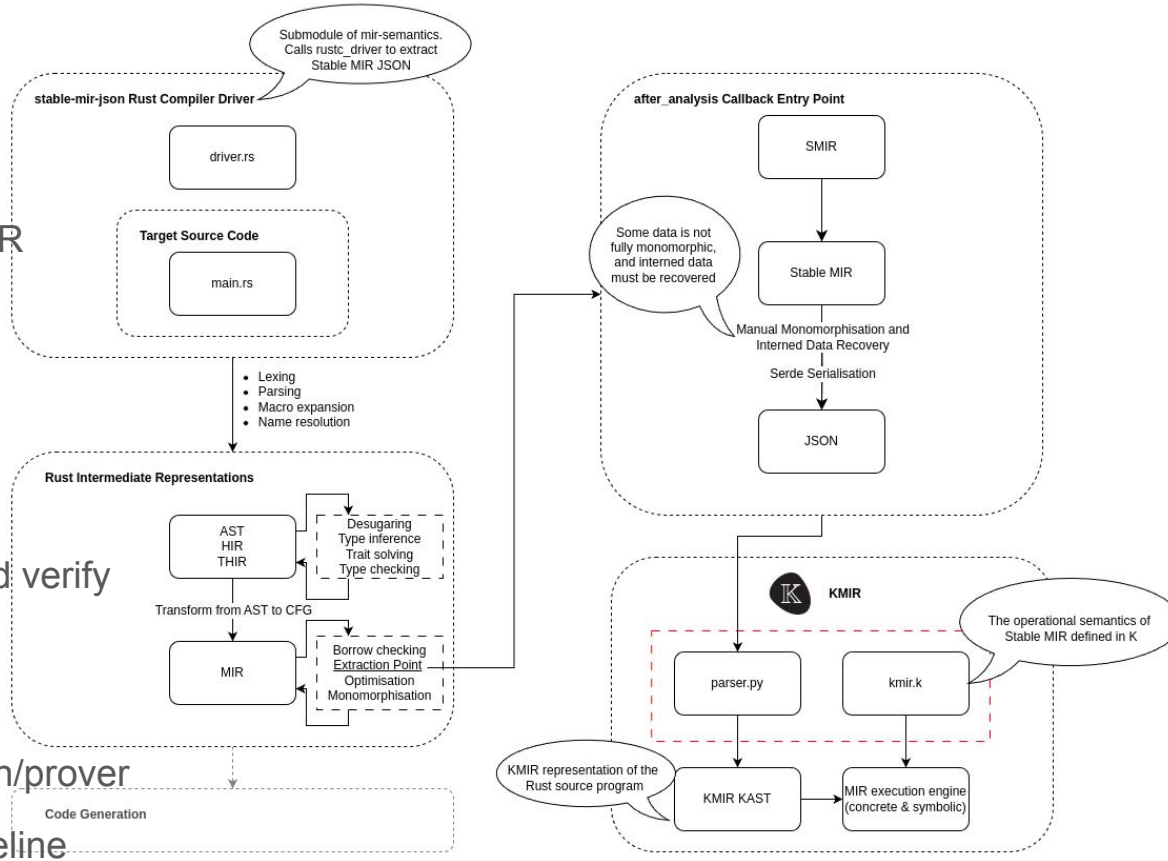
- Several blockchains using Rust smart contracts (Stellar, Solana)
- Open Source development at Runtime Verification

How does it work?

kmir's Workflow and Pipeline

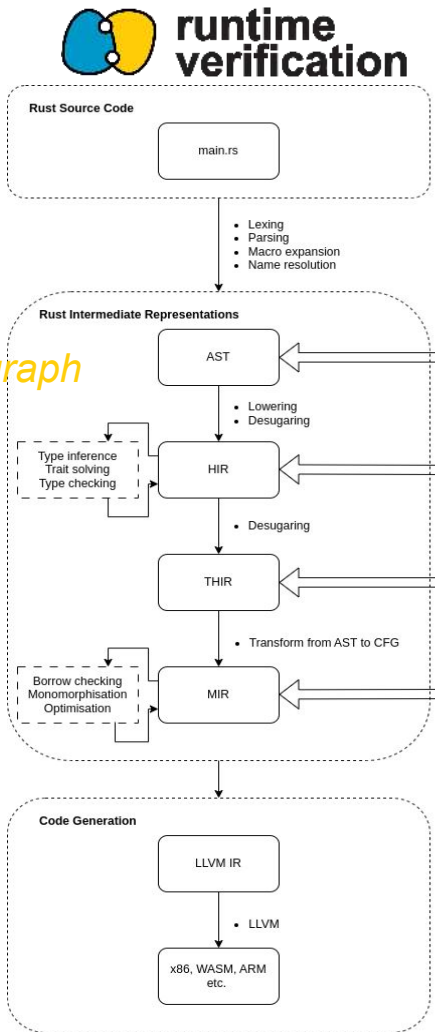
runtime verification

- Modified `rustc` to extract MIR
  - Call-back `after_analysis`
  - Externalises MIR to Stable MIR
  - Adds additional context data and several lookup tables
- Semantics of Stable MIR (in K)
  - Meaning of extracted code
  - K Framework can execute and verify properties.
- `kmir` tool (in Python)
  - Drives K Framework execution/prover
  - Will orchestrate the entire pipeline



MIR: The Core of Rust's Compiler

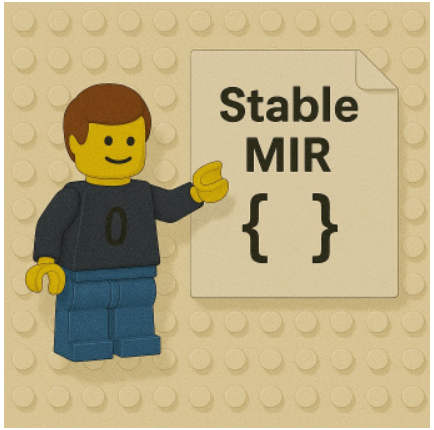
- Rust's Mid-Level Intermediate Representation (MIR)
  - An internal representation of Rust code within the compiler
  - Follows source AST, HIR (high-level IR) and type-checking (THIR)
  - Function bodies represented with their *basic blocks* in a *control flow graph*
  - Target for borrow-checking, monomorphisation, and optimisations
- Why MIR for Verification?
  - Simpler structure (easier to analyse) than Rust source code
  - Already type-checked and borrow-checked
  - Last target-independent stage in the Rust compilation



stable-mir-json: MIR Serialization

runtime verification

- Why *Stable* MIR?
  - Textual MIR incomplete, not intended for programmatic consumption
  - Internal MIR changes quickly with the compiler
  - Project stable_mir to provide a stable API for MIR https://github.com/rust-lang/project-stable-mir
- `stable-mir-json`
  - A Rust compiler plugin that extracts Stable MIR into a structured JSON format
  - External format requires extracting some additional data (beyond internal Stable MIR)
  - Provides a data format for MIR-based tools that run outside the compilation pipeline



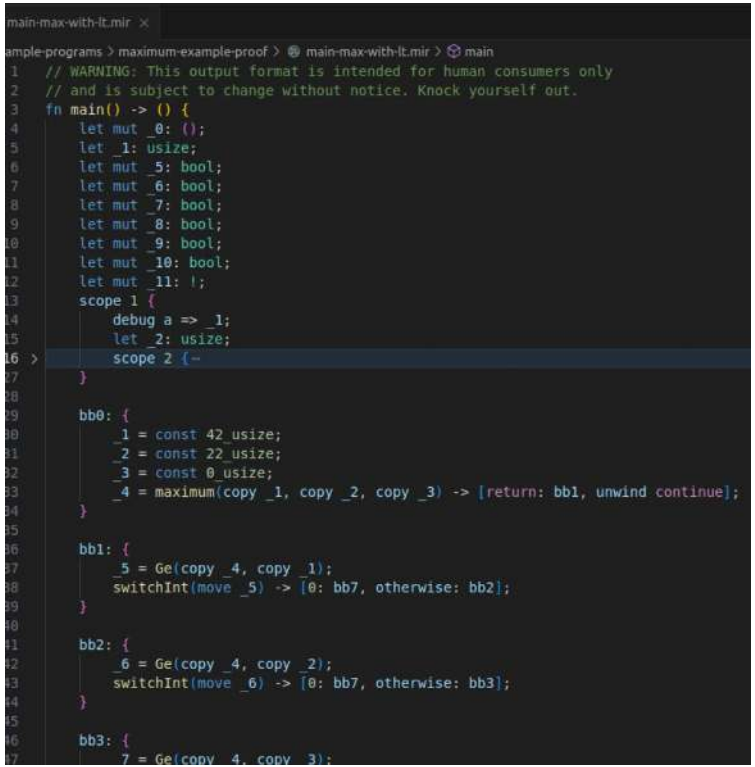
Demo 1: stable-mir-json

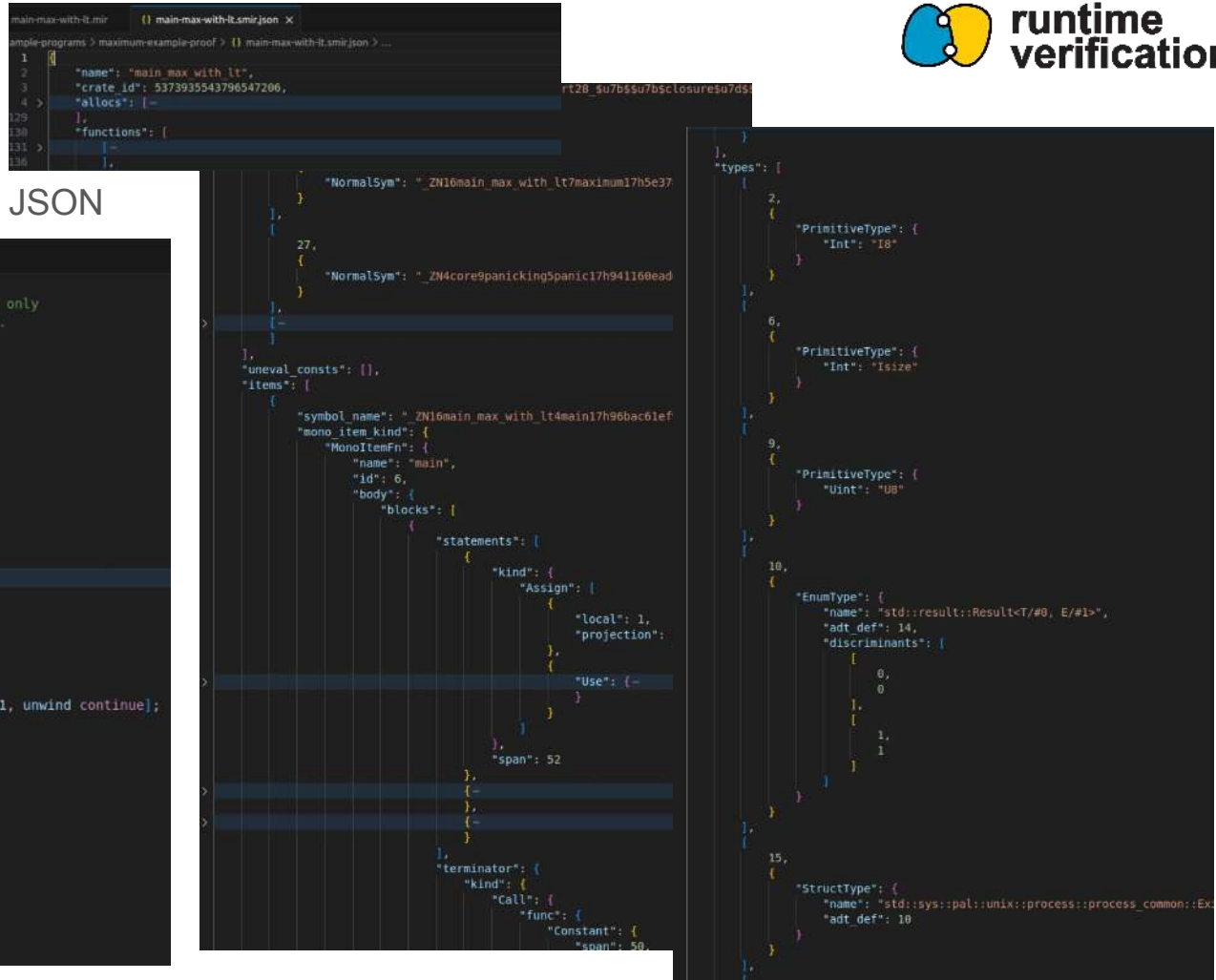
- Steps:
  - Write a few simple Rust functions calling stdlib code and each other
  - `rustc --emit mir` for textual MIR
  - `stable-mir-json` for MIR JSON
    - `stable-mir-json --dot some.rs -Zno-codegen`
    - `stable-mir-json --dot some.rs -Zno-codegen`

Demo 1: stable-mir-json

runtime verification

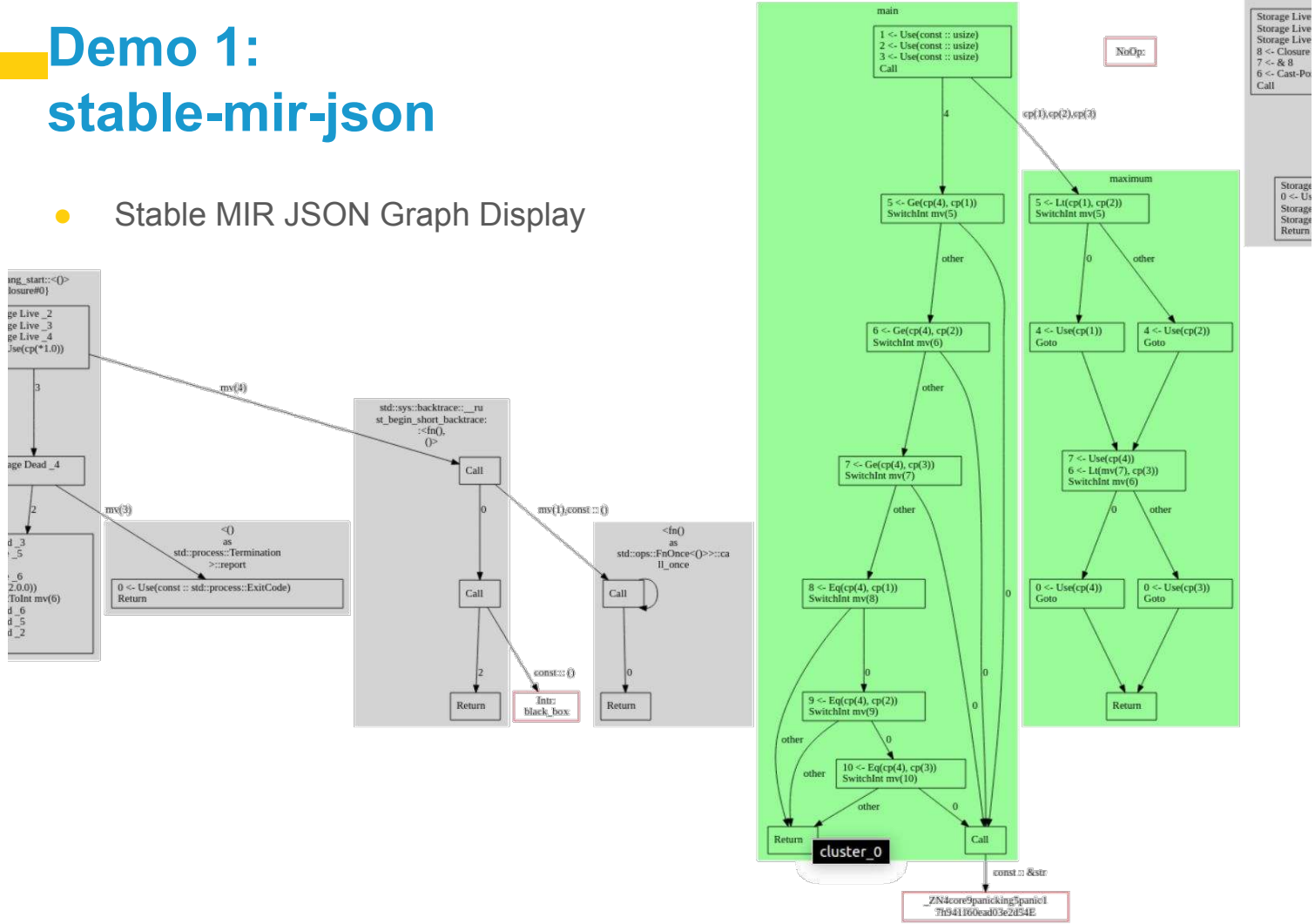
- Textual MIR vs Stable MIR JSON





Demo 1: stable-mir-json

- Stable MIR JSON Graph Display



Semantics of Stable MIR

runtime verification

- Executing (Stable) MIR statements and terminators
- Computing results of basic primitive operations
  - Representing (primitive and aggregate) data at a high level
- Execution as *rewrite operations* to a *global execution state*
  - Program input is Stable MIR AST
  - Execution of given function as entry point (default: `main`)
- Semantics modelled using the *K Framework*

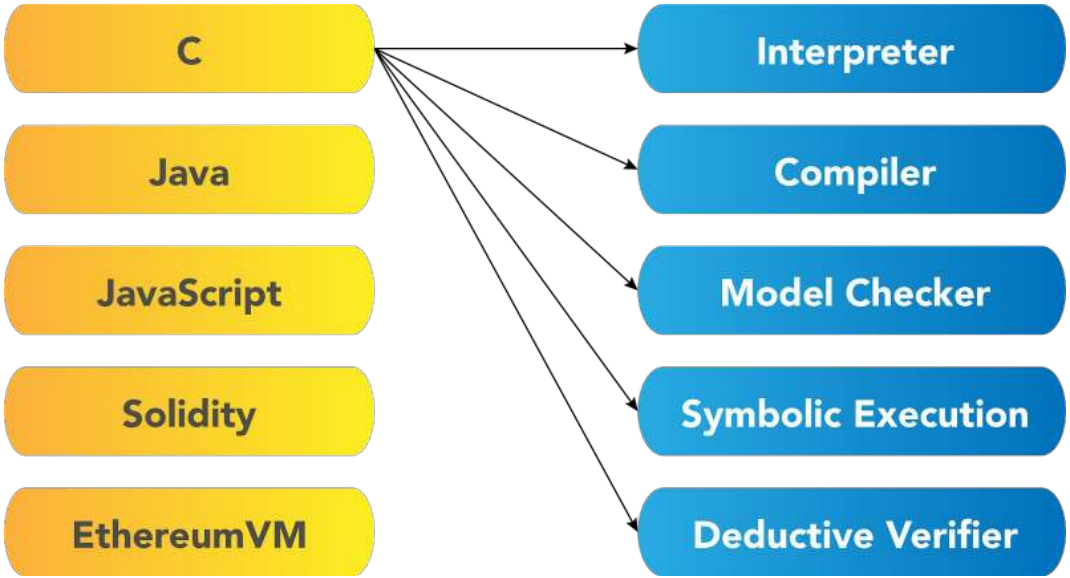
What is K Framework?

runtime verification

- https://kframework.org maintained by Runtime Verification / Pi^2
- K is an *operational semantics framework* based on rewriting.
  - Specify your language or system as a K definition.
  - The K compiler derives a number of tools (parser, printer, interpreter, prover)
- Project started >15 years ago, building on earlier rewriting systems
- K's logical foundation is Matching Logic
- Given a K specification, there are two main backends you can use:
  - LLVM backend is for *concrete execution*, generating a fast interpreter.
  - Haskell backend is for *symbolic execution*: verification engine and model checker

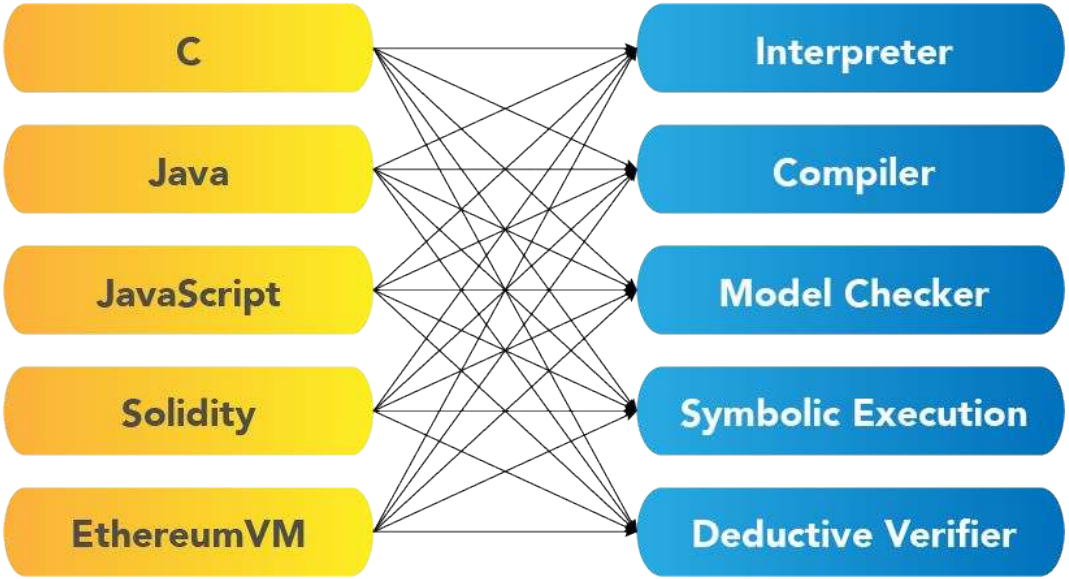
The Problem: Too Many Tools

runtime verification



The Problem: Too Many Tools

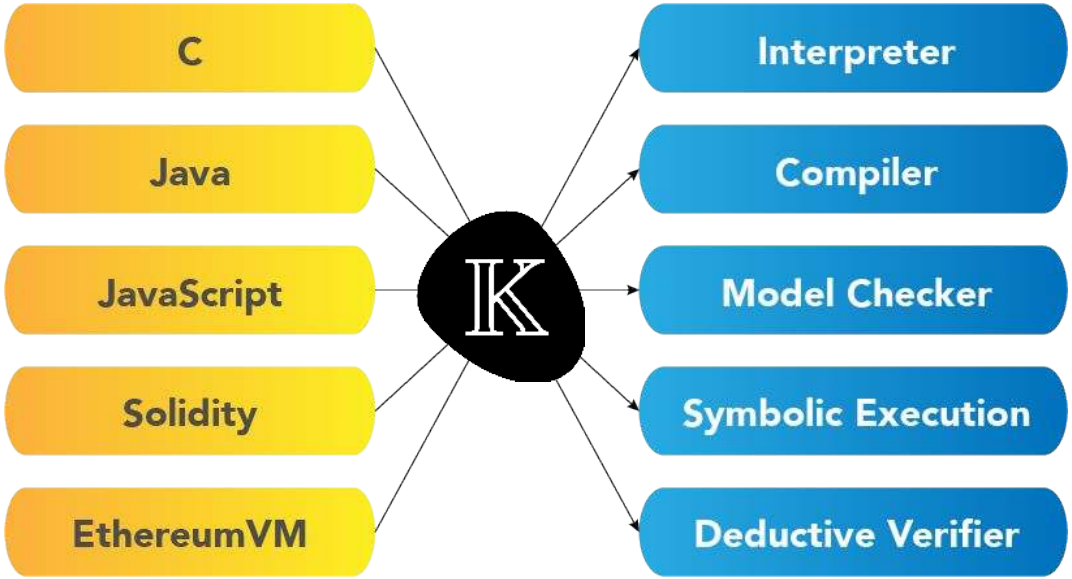
runtime verification



The K Approach

runtime verification

- Develop each language definition and each tool once:



- Updates to tools benefit *all* the languages

Semantics for Stable MIR in K

runtime verification

- MIR AST as K data structures
  - 1:1 with `stable_mir` data structures (and additional context data)
  - Annotations drive a JSON parser
- Global configuration of running program
  - Control cell (`<k>`), call stack, local data
  - all program items (for call/return)
  - type metadata (for computation)
- Rewrite rules to model MIR execution
  - Driven by `<k>` cell (containing MIR)
  - Other cells manipulated as effects
  - Unchanged irrelevant cells omitted

```
module KMIR-CONFIGURATION
 imports KMIR-AST
 imports INT-SYNTAX
 imports BOOL-SYNTAX
 imports RT-VALUE-SYNTAX

 syntax RetVal ::= return( Value )
                 | "noReturn"

 syntax StackFrame ::= StackFrame(caller:Ty,              // index of caller function
                                  dest:Place,             // place to store return value
                                  target:MaybeBasicBlockIdx,// basic block to return to
                                  UnwindAction,           // action to perform on panic
                                  localsList)             // return val, args, local variables

 configuration <kmir>
                 <k> #init($PGM:Pgm) </k>
                 <retVal> noReturn </retVal>
                 <currentFunc> ty(-1) </currentFunc> // to retrieve caller
                 // unpacking the top frame to avoid frequent stack read/write operations
                 <currentFrame>
                   <currentBody> .List </currentBody>
```

```
rule <k> #execTerminator(terminator(terminatorKindReturn, _SPAN)) ~> _
       => #setLocalValue(DEST, #decrementRef(LOCAL0)) ~> #execBlockIdx(TARGET)
     </k>
     <currentFunc> _ => CALLER </currentFunc>
     //<currentFrame>
       <currentBody> _ => #getBlocks(FUNCS, CALLER) </currentBody>
       <caller> CALLER => NEWCALLER </caller>
       <dest> DEST => NEWDEST </dest>
       <target> someBasicBlockIdx(TARGET) => NEWTARGET </target>
       <unwind> _ => UNWIND </unwind>
       <locals> ListItem(LOCAL0:TypedValue) _ => NEWLOCALS </locals>
     //</currentFrame>
     // remaining call stack (without top frame)
     <stack> ListItem(StackFrame(NEWCALLER, NEWDEST, NEWTARGET, UNWIND, NEWLOCALS)) STACK => STACK </stack>
     <functions> FUNCS </functions>
   requires CALLER in_keys(FUNCS)
```

Demo 2: Executing MIR

Steps:

- Using programs from Demo 1
- `stable-mir-json my-program.rs`
- `kmir run my-program.smir.json --depth <N>`

Demo 2: Executing MIR

runtime verification

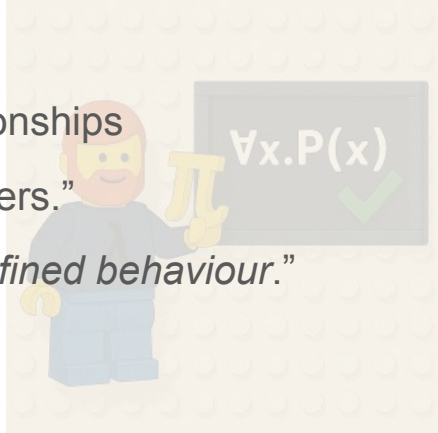
- Example
  - `SwitchInt` branch
  - on a BoolVal
  - in local `_5`
- Note the data model
  - High level values for primitive types (`u64` values here)
- Not shown:
  - Type mappings and metadata

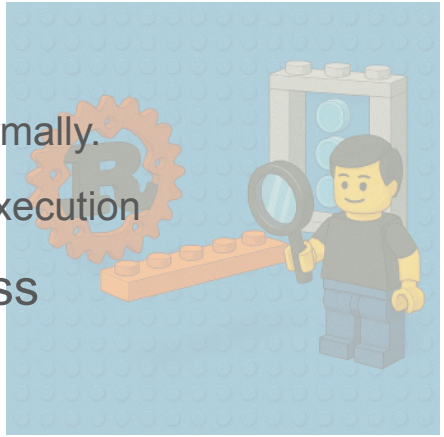
```
<kmir>
 <k>
   #execTerminator ( terminator (... kind: terminatorKindSwitchInt (... discr: operandMove ( place (... local: local ( 5 ) , projec>
 </k>
 <retVal>
   noReturn
 </retVal>
 <currentFunc>
   ty ( 25 )
 </currentFunc>
 <currentFrame>
   <currentBody>
     ListItem ( basicBlock (... statements: statement (... kind: statementKindAssign (... place: place (... local: local ( 5 ) , pro>
     ListItem ( basicBlock (... statements: statement (... kind: statementKindAssign (... place: place (... local: local ( 4 ) , pro>
     ListItem ( basicBlock (... statements: statement (... kind: statementKindAssign (... place: place (... local: local ( 4 ) , pro>
     ...
   </currentBody>
   <caller>
     ty ( -1 )
   </caller>
   <dest>
     place (... local: local ( 4 ) , projection: .ProjectionElems )
   </dest>
   <target>
     someBasicBlockIdx ( basicBlockIdx ( 1 ) )
   </target>
   <unwind>
     unwindActionContinue
   </unwind>
   <locals>
     ListItem ( newLocal ( ty ( 26 ) , mutabilityMut ) )
     ListItem ( typedValue ( Integer ( 42 , 64 , false ) , ty ( 26 ) , mutabilityNot ) )
     ListItem ( typedValue ( Integer ( 22 , 64 , false ) , ty ( 26 ) , mutabilityNot ) )
     ListItem ( typedValue ( Integer ( 0 , 64 , false ) , ty ( 26 ) , mutabilityNot ) )
     ListItem ( newLocal ( ty ( 26 ) , mutabilityNot ) )
     ListItem ( typedValue ( BoolVal ( false ) , ty ( 29 ) , mutabilityMut ) )
     ListItem ( newLocal ( ty ( 29 ) , mutabilityMut ) )
     ListItem ( newLocal ( ty ( 26 ) , mutabilityMut ) )
   </locals>
 </currentFrame>
 <stack>
   ListItem ( StackFrame ( ty ( -1 ) , place (... local: local ( -1 ) , projection: .ProjectionElems ) , noBasicBlockIdx , unwindActi>
   ...
```

Property Specification in K

runtime verification

- What are Properties?
  - Formal statements about program behavior and input/output relationships
    - “This function computes the maximum of three integral numbers.”
    - “For all *valid* inputs, function f executes *without* causing *undefined behaviour*.”
- How to specify properties in KMIR
  - We execute a program (fragment) with *symbolic* data
  - … and formulate expectations about the result, in a K *claim*
  - `kmir gen-spec` generates a claim that a program terminates normally.
  - Other properties can use configuration fragments extracted from execution
- Currently a manual process – automation work in progress





Demo 3: Verification of a Property in KMIR

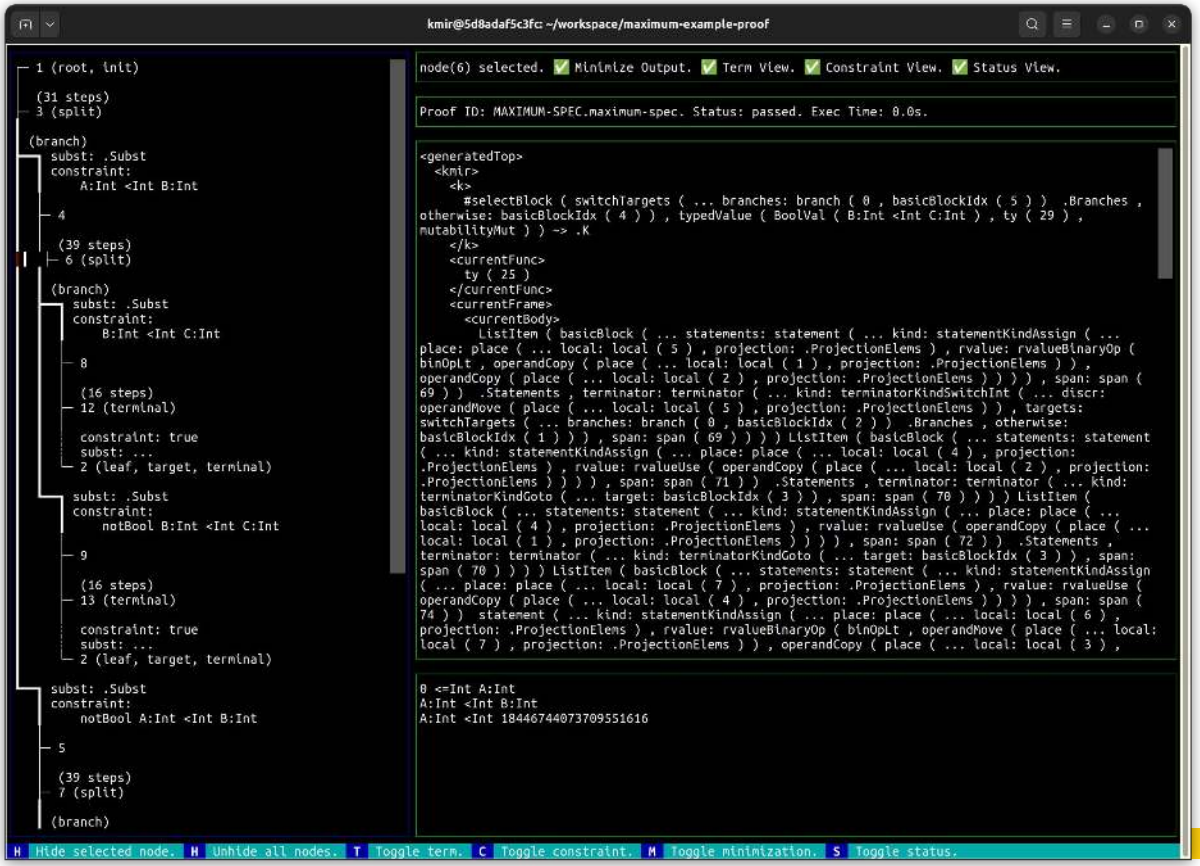
(semi-manual at the moment)

- Steps
  - Use the JSON output from Demo 1.
  - Generate claim using `gen-spec`, or/and `maximum-spec.k`
  - `kmir prove run` to verify property
  - `kmir prove view` to view proof

Demo 3:
Verification of a Property in KMIR

runtime verification

- Execution branches depending on the symbolic values
  - `kmir prove view` to inspect
  - Each branch carries its own *path conditions* (bottom)
- A successful proof means:
  - *All* branches reach a configuration consistent with the target state in the claim (“all-path reachability”).
  - The configurations unify,
  - accumulated path conditions imply target conditions.



KMIR: Status and Limitations

runtime verification

- KMIR is under active development
  - Supports most primitive types and structs
  - Support for enums/arrays in progress
  - Next on the list: Support for heap operations and pointers/addresses
  - Automation of claim generation and custom start symbols in progress
  - No multi-crate support yet
- Limitations
  - No support for inline ASM (obvious)
  - High-level data model (difficult to support `transmute`)

Summary: The three pillars of KMIR

runtime verification

- **Stable MIR JSON**:
  - Extracts Stable MIR from `rustc` into a simple JSON format
- **Semantics of Stable MIR** in K
  - Models Stable MIR operational semantics with K Framework
  - Program execution as rewrites to a global configuration
  - Symbolic execution of Rust and property proofs (reachability logic)
- **KMIR frontend**:
  - Drives the Stable MIR execution and property proofs
  - Uses K's Python bindings

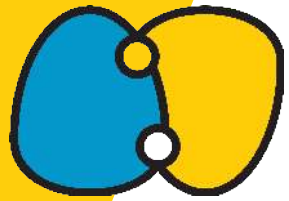
Learn More

runtime verification

- Source code on Github
  - https://github.com/runtimeverification/mir-semantics
  - https://github.com/runtimeverification/stable-mir-json
- Docker image:
  - https://hub.docker.com/r/runtimeverificationinc/kmir/tags
- K Framework Documentation
  - https://kframework.org

runtime verification

- https://github.com/runtimeverification/mir-semantics
- https://github.com/runtimeverification/stable-mir-json
- https://hub.docker.com/r/runtimeverificationinc/kmir/tags
- https://kframework.org



Questions?

https://runtimeverification.com/

@rv_inc

https://discord.com/invite/CurfmXNtbN

contact@runtimeverification.com

Picture Warehouse

runtime verification