An Introductory Practical Guide to Rust Bindings for the Z3 Solver

Mehrad Haghshenas*

June 2025

Acknowledgement

Some of the responses and content in this document were developed with the assistance of *LLMs*. Likewise, in some cases where clear explanations were provided in the official documentation, they were directly used. This document is not intended to introduce novel material. The primary goal was to centralize information and provide illustrative examples to create a beginner-friendly practical guide in using the Rust bindings for Z3.

*https://mehrad31415.github.io/

This page has been intentionally left blank.

1 Introduction

Rust has high-level and low-level bindings for the Z3 solver. The *z3 crate* provides high-level bindings, and the *z3-sys crate* provides the low-level bindings to the Z3 solver. The *z3* crate is built on top of the *z3-sys* crate and encapsulates it. The *z3-sys* crate provides the raw, unsafe, low-level C API that Z3 exposes. More often than not, the *z3* crate should be used. The *z3-sys* crate is used only when some Z3 feature isn't wrapped into high-level bindings in the *z3* crate. Another time *z3-sys* is directly used is when writing your custom high-level API for Z3. There is no indication that the *z3-sys* crate offers better performance; thus, the *z3* crate will be used unless a limitation is confronted.

Both *z3* and *z3-sys* crates have dependencies on Z3. They can be added by writing the following in `Cargo.toml`:

```
[dependencies]
z3 = "0.12"
z3-sys = "0.8"
```

There are 3 ways for the crates to currently find Z3 on the local computer:

1. By default, it will look for a system-installed copy of Z3. On macOS, this will be via Homebrew (`brew install z3`). To reiterate, to use the crates on a macOS computer, run `brew install z3` in the terminal. Confirm that Z3 has been installed by running `which z3` in the terminal. After that, write the following lines in the `Cargo.toml` file of the workspace:[1]

```
[dependencies]
z3 = { git="https://github.com/prove-rs/z3.rs.git" }
z3-sys = { git="https://github.com/prove-rs/z3.rs.git" }
```

2. Adding the *bundled* feature will use *CMake* to build a locally bundled copy of Z3. This copy is provided via a git submodule, nevertheless the time to build the project is considerably higher. This will look like `z3 = {version="0.12", features = ["bundled"]}` for example.[2]

3. Enabling the *vcpkg* feature will use *vcpkg* to build and install a copy of Z3. This will look like `z3 = {version="0.12", features = ["vcpkg"]}` for example.[3]

The source code related to the *z3* crate can be found in the *src* subdirectory. The following is a list of the rust modules:

1. `lib.rs`
2. `config.rs`
3. `context.rs`
4. `solver.rs`
5. `ast.rs`
6. `sort.rs`
7. `symbol.rs`

[1] For some reason writing `z3 = "0.12"` or `z3-sys = "0.8"` in the `Cargo.toml` will give the error *fatal error: 'z3.h' file not found*. Note that the *z3* crate internally uses the *z3-sys* crate, which in turn requires a *z3.h* during the build time. You may need to ensure Z3 headers are discoverable via `PKG_CONFIG_PATH` or `Z3_SYS_INCLUDE_DIR`.

[2] Note that `static-link-z3` is a deprecated alias for `bundled`. Therefore, writing `z3 = { version = "0.12", features = ["static-link-z3"] }` is the same, but again it gets stuck at *Blocking waiting for file lock on build directory*. This is because `static-link-z3` - similar to `bundled` - links Z3 statically; i.e., the native *libz3* code is compiled directly into your binary and there is no need for Z3 to be installed on the system at runtime. Nevertheless, building the project takes a longer amount of time.

[3] `CMake` builds Z3 from source and gives full control over compilation options, making it ideal for customization. `vcpkg` installs prebuilt Z3 libraries with minimal setup, making it convenient for quick integration, especially on Windows which lacks a native package manager for `C/C++` libraries.

8. `ops.rs`
9. `datatype_builder.rs`
10. `rec_func_decl.rs`
11. `func_decl.rs`
12. `func_entry.rs`
13. `func_interp.rs`
14. `model.rs`
15. `pattern.rs`
16. `goal.rs`
17. `tactic.rs`
18. `probe.rs`
19. `optimize.rs`
20. `params.rs`
21. `statistics.rs`
22. `version.rs`

By looking at the *lib.rs* module and the defined module tree system, the following functionalities are provided by the public API of the *z3* crate:

1. `AstKind, GoalPrec, SortKind, DeclKind` from the *z3-sys* crate are re-exported. See sections `AstKind`, `SortKind`, `DeclKind`, and `GoalPrec` for details.
2. `get_global_param, reset_all_global_params, set_global_param` from the *params* module are re-exported. See section `params` for explanations.
3. `StatisticsEntry, StatisticsValue` from the *statistics* module are re-exported. See section `statistics` for explanations.
4. `full_version, version, Version` from the *version* module are re-exported. See section `version` for explanations.
5. Everything marked *pub* inside the `lib.rs` file is accessible. This includes any struct or enum declared as pub. Although their full implementations (e.g., methods) are defined in other modules, any pub methods in those impl blocks are also accessible by external crates. See section `lib` for explanations.
6. Everything public inside the *ast* module is accessible.
7. Everything public inside the *datatype_builder* module is accessible.

We will begin by explaining points one to four. After that, we will discuss the public data structures defined in *lib.rs*, and then proceed to explain each module in turn, covering the implementations of the structs and enums introduced in *lib.rs*.

2 *z3-sys* re-exports

The `AstKind, GoalPrec, SortKind, DeclKind` from *z3-sys* are accessible:

1. **AstKind:** In Z3, everything is represented as an AST (Abstract Syntax Tree). `AstKind` tells you what kind of AST node you're working with. It corresponds to `Z3_ast_kind` in the C API,

defining the different kinds of Z3 AST (abstract syntax trees). Specifically, it is used to define *terms, formulas, types*. It has the following variants:

- **Numeral:** Used whenever the node is a literal number or bit-vector literal.
- **App:** Function or operator applications, including constant symbols. For example, every function application like `f(a,b)`, predicate like the constant *true*, or built-in operator like `x + y` is an `App` node.
- **Var:** Bound variables inside quantifiers (using *de Bruijn* indices). In `forall x. P(x)`, the occurrence of `x` is represented as a `Var` AST. Nevertheless, Z3 doesn't store the name `x`; it uses an index like `Var(0)`.
- **Quantifier:** *forall* and *exists* nodes. Wraps the whole quantifier; for example, `z3::ast::Quantifier::forall`.
- **Sort:** Type ASTs themselves, like Int, Bool, or user-declared sorts. When you inspect an AST that represents a type, you get `AstKind::Sort`.
- **FuncDecl:** Function-declaration ASTs (uninterpreted functions & constructors). Symbols when you declare a function or a datatype constructor.
- **Unknown:** Anything otherwise!

Here is an example of how the abstract syntax kind can be retrieved:

```rust
use z3::{
    ast::{self, Ast},
    Config, Context,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    // Build a small term:  x + 1
    let x = ast::Int::new_const(&ctx, "x");
    let one = ast::Int::from_i64(&ctx, 1);
    let term = x + one;

    // Inspect its kind
    match term.kind() {
        z3::AstKind::App => println!("func app."),
        z3::AstKind::Numeral => println!("num."),
        other => println!("Unexpected {:?}", other),
    }
}
```

2. **SortKind:** This corresponds to `Z3_sort_kind` in the C API and represents the different kind of Z3 *types*. This enum has the following variants:

   (a) **Uninterpreted:** This type is used to model abstract types with no built-in semantics (e.g., generics or opaque types). Do not use this type to model self-defined enums or structs. To reiterate, even unit structs (e.g., `struct MyUnitStruct;`) should be modeled as *Datatypes*, not uninterpreted sorts. **When to choose each sort:**

   Use ***Uninterpreted*** when the type is truly opaque and you only need an identity:

   - **Generic parameters.** For example in Rust if we have `fn f<T>(x: T) ...` the generic will translate to `(declare-sort T 0)` in Z3.

- **Opaque handles.** For example file descriptors, DB keys, capability tokens, ... which do not have constructors, translate well to uninterpreted types like `(declare-const fd File)`.

  – Rust (interface only)

```rust
/// An OS file descriptor; the function caller never sees its fields.
pub struct FileHandle { _priv_field: () }

/// Close the handle; returns true on success.
pub fn close(fd: FileHandle) -> bool { /* kernel call */ }
```

  – SMT-LIB

```
(declare-sort FileHandle 0)
(declare-fun close (FileHandle) Bool)

; Concrete handle values the program might manipulate
(declare-const fd_stdin  FileHandle)
(declare-const fd_temp   FileHandle)
```

  No constructors or pattern matching are necessary—the verifier only needs to reason about *identity* - (in)equality - and the effects of the operations such as `close`.

- **Trait objects or Implementations.** The caller sees only a behaviour contract, not a shape.

  – Rust

```rust
trait Drawable { fn draw(&self); }

fn render(obj: &dyn Drawable) {           // caller sees only &dyn Drawable
    obj.draw();
}

// returning an opaque iterator via `impl Trait`
fn counter() -> impl Iterator<Item = i32> {
    0..10
}
```

  – SMT-LIB

```
; A type with no visible structure
(declare-sort Drawable 0)

; Functions that use it
(declare-fun render (Drawable) Bool)

; The concrete iterator type returned by `counter` is hidden,
; so we model it as another uninterpreted sort.
(declare-sort CounterIter 0)
(declare-fun next  (CounterIter) Int)
```

- **Phantom types.** Compile-time markers that never exist at runtime (e.g., `PhantomData<T>`)).

  – Rust

```rust
use std::marker::PhantomData;

// Distinguishes IDs that are all just u64 at runtime.
```

```
struct Id<Tag> {
    raw: u64,
    _p: PhantomData<Tag>,
}

struct UserTag;
struct OrderTag;

type UserId  = Id<UserTag>;
type OrderId = Id<OrderTag>;
```

– SMT-LIB

```
; Each phantom specialisation becomes an opaque sort
(declare-sort UserId  0)
(declare-sort OrderId 0)

; The runtime payload can be modelled separately if needed
(declare-fun rawUserId  (UserId)  Int)
(declare-fun rawOrderId (OrderId) Int)
```

Use *Datatype* when the type has a fixed set of constructible values or internal structure:

- **Unit structs.** `struct A;` in Rust translates to `(declare-datatypes () ((A A)))`.
- **Fielded structs.** `struct Pt { x: Int, y: Int }` in Rust translates to `(declare-datatypes () ((Pt (mkPt (x Int) (y Int)))))`.
- **Enums/variants.** `enum E { V1, V2(i32) }` in Rust translates to one datatype with two constructors `V1` and `V2`.
- **Option/Result-like sum types.** Needed for pattern matches.
- **Recursive data structures.** Lists, trees, etc. *Datatypes* give you built-in induction and selectors.

```
use z3::{
    ast::{Ast, Dynamic},
    Config, Context, Sort,
};
fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let my_sort = Sort::uninterpreted(&ctx, "MySort".into());
    let a = Dynamic::new_const(&ctx, "a", &my_sort);
    let b = Dynamic::new_const(&ctx, "b", &my_sort);

    let eq = a.eq(&b);
    println!("Expression eq: {}", eq);

    let _eq = a._eq(&b);
    println!("Expression _eq: {}", _eq);
}

```

(b) **Bool:** This type is used for two-valued logic.

```
fn main() {
    use z3::{ast::Bool, Config, Context, Solver};

    let cfg = Config::new();
```

```
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // Create a boolean variable and assert it
    let b = Bool::new_const(&ctx, "b");
    solver.assert(&b);

    // Check if the assertion is satisfiable
    match solver.check() {
        z3::SatResult::Sat => println!("The assertion is satisfiable."),
        z3::SatResult::Unsat => println!("The assertion is unsatisfiable."),
        z3::SatResult::Unknown => println!("The satisfiability is unknown."),
    }
}
```

(c) **Int:** This type is used to model integers.

```
fn main() {
    use z3::{ast::Ast, ast::Int, Config, Context, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // Create integer variables
    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    solver.assert(&x.gt(&Int::from_i64(&ctx, 0))); // x > 0
    solver.assert(&y.gt(&Int::from_i64(&ctx, 0))); // y > 0
    solver.assert(&Int::add(&ctx, &[&x, &y])._eq(&Int::from_i64(&ctx, 10))); // x + y
    ↪  = 10
    solver.assert(&x.lt(&Int::from_i64(&ctx, 7))); // x < 7

    // Check if the constraints are satisfiable
    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!(
                "Solution found: x = {}, y = {}",
                model.eval(&x, true).unwrap(),
                model.eval(&y, true).unwrap()
            );
        }
        _ => println!("No solution found."),
    }
}
```

One extra note for the `model.eval(..., model_completion)`: When the *model_completion* flag is set to true, Z3 will always return a value for the given expression. This means that even if that value wasn't explicitly assigned in the model or there were no explicit constraints related to that variable, arbitrary values will be assigned to the expression which are consistent with the constraints. However, when set to false, Z3 will return `None` unless the variable or expression was explicitly assigned a value by the solver or used in an assertion constraint. This stricter mode is useful when you want to know exactly which parts of your formula the model truly depends on. For example, if you query an *unused* variable with `model_completion = false`, you'll get `None`, whereas setting it to true will give you some default value, even though it plays no role in the actual solution.

(d) **Real:** Represents mathematical real numbers with exact algebraic precision, allowing you to write constraints that reason about fractions, square-roots, and polynomial roots without the rounding errors of floating-point arithmetic – i.e., Z3 internally treats *reals* as exact rational values (like 1/3 stays as 1/3, not 0.333). This is not suited for hardware or low-level binary math—use `BV` (bit-vectors) instead.

```rust
fn main() {
    use z3::{ast::Ast, ast::Real, Config, Context, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // Declare real variables
    let x = Real::new_const(&ctx, "x");
    let y = Real::new_const(&ctx, "y");

    // Constraints: x + y = 3/2, x * y = 1/2
    solver.assert(&(x.clone() + y.clone())._eq(&Real::from_real(&ctx, 3, 2))); // x + y = 1.5
    solver.assert(&(x.clone() * y.clone())._eq(&Real::from_real(&ctx, 1, 2))); // x * y = 0.5

    // Solve and retrieve values
    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            let x_val = model.eval(&x, true).unwrap().as_real().unwrap();
            let y_val = model.eval(&y, true).unwrap().as_real().unwrap();
            println!(
                "Solution: x = {}/{}, y = {}/{}",
                x_val.0, x_val.1, y_val.0, y_val.1
            );
        }
        _ => println!("No solution"),
    }
}
```

(e) **BV:** The theory of bit-vectors allows modeling the precise semantics of unsigned and of signed two-complements arithmetic. Bit-vectors model fixed-size binary numbers, like 32-bit or 64-bit integers. Use this for low-level operations such as bitwise logic, shifts, and overflow behavior. This is ideal for reasoning about hardware.

```rust
fn main() {
    use z3::{ast::Ast, ast::BV, Config, Context, SatResult, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // Create 8-bit variables: a = 255, b = 1
    let a = BV::from_u64(&ctx, 255, 8);
    let b = BV::from_u64(&ctx, 1, 8);

    // Add them (wraps around due to 8-bit width)
    let sum = a.bvadd(&b);

    // Assert sum == 0 (255 + 1 wraps to 0 in 8-bit)
    solver.assert(&sum._eq(&BV::from_u64(&ctx, 0, 8)));
```

```
17
18      // Verify and print model
19      match solver.check() {
20          SatResult::Sat => {
21              let model = solver.get_model().unwrap();
22              println!("Result: {}", model.eval(&sum, true).unwrap()); // result = #x00
                ↪  (hexadecimal with 8 bits)
23          }
24          _ => panic!("Unsatisfiable"),
25      }
26  }
```

(f) **Array:** Used to stored key-value mappings. Each array term is an immutable function from an *index* sort (the key) to a *range* sort (the value). The underlying *select/store* axioms capture the behavior of read and write operations, while *extensionality* ensures that two arrays are equal iff they agree on all indices. Beyond simple maps, boolean-valued arrays encode *sets*.

```
1   fn main() {
2       use z3::{
3           ast::{Array, Ast, Int},
4           Config, Context, Solver,
5       };
6
7       let cfg = Config::new();
8       let ctx = Context::new(&cfg);
9       let solver = Solver::new(&ctx);
10
11      // array : Int → Int
12      let array = Array::new_const(&ctx, "mem", &z3::Sort::int(&ctx),
        ↪  &z3::Sort::int(&ctx));
13
14      // Write: mem[2] := 10
15      let mem1 = array.store(&Int::from_i64(&ctx, 2), &Int::from_i64(&ctx, 10));
16
17      // Read: r = mem1[2] ; r2 = mem1[3]
18      let r = mem1.select(&Int::from_i64(&ctx, 2));
19      let r2 = mem1.select(&Int::from_i64(&ctx, 3));
20
21      // Constraints: r=10, r2 != 10
22      solver.assert(&r._eq(&Int::from_i64(&ctx, 10).into()));
23      solver.assert(&r2._eq(&Int::from_i64(&ctx, 10).into()).not());
24
25      assert_eq!(solver.check(), z3::SatResult::Sat);
26      let model = solver.get_model().unwrap();
27      let mem1_in_model = model.eval(&mem1, true).unwrap();
28      println!("mem1_in_model = {}", mem1_in_model);
29  }
```

(g) **Datatype:** Note that self-defined types are best represented via *Datatype* sort. Specifically, this sort is used to model enums and structs.

```
1   fn main() {
2       use z3::{ast::Ast, ast::Datatype, Config, Context, DatatypeBuilder, Solver};
3
4       let cfg = Config::new();
5       let ctx = Context::new(&cfg);
6       let solver = Solver::new(&ctx);
7
8       // Define an enum Color { Red, Green, Blue }
9       let color_enum = DatatypeBuilder::new(&ctx, "Color")
```

```
        .variant("Red", vec![]) // No fields (unit variant)
        .variant("Green", vec![]) // No fields
        .variant("Blue", vec![]) // No fields
        .finish();

    // Create variables of Color type
    let x = Datatype::new_const(&ctx, "x", &color_enum.sort);
    let y = Datatype::new_const(&ctx, "y", &color_enum.sort);

    // Assert x is Red
    let red_value = color_enum.variants[0].constructor.apply(&[]);
    solver.assert(&x._eq(&red_value.as_datatype().unwrap()));

    // Assert y is not Red (must be Green or Blue)
    let is_red = color_enum.variants[0].tester.apply(&[&y]);
    solver.assert(&is_red.as_bool().unwrap().not());

    match solver.check() {
        z3::SatResult::Sat => {
            println!("Satisfiable: x is Red, y is not Red");
            let model = solver.get_model().unwrap();
            let x_value = model.eval(&x, true).unwrap();
            let y_value = model.eval(&y, true).unwrap();
            println!("x: {:?}", x_value);
            println!("y: {:?}", y_value);
        }
        z3::SatResult::Unsat => {
            println!("Unsatisfiable");
        }
        z3::SatResult::Unknown => {
            println!("Unknown result");
        }
    }
}
```

(h) **Relation:** Represents finite relations for Datalog-style fixed-point engines. Relations in Z3 are predicate symbols that can be used in Datalog-style logic programming. They differ from regular functions because they're designed to work with Z3's fixedpoint solver rather than the standard SMT solver. They follow least-fixedpoint semantics - a relation is empty unless explicitly defined by facts or rules. **Note that Z3's fixedpoint engine (for Datalog/Horn clauses) is not fully supported in the high-level Rust binding of Z3.**

(i) **FiniteDomain:** Used to model small, fixed-size enumerations (like an enum with a known number of opaque values). By contrast, *enums* in *Datatype* sorts require giving each element a distinct name. A FiniteDomain sort of size N provides exactly N distinct values (0 through N–1) with no further structure. **The high-level Rust binding doesn't currently expose *finite_domain_sort* directly.**

(j) **FloatingPoint & RoundingMode:** IEEE-754 floats and their rounding modes. Z3 supports all five IEEE-754 rounding modes, each as a Float AST of the RoundingMode sort:

- `Float::round_nearest_ties_to_even(&ctx)` How this works is that it rounds to the nearest representable floating-point value. When halfway between two values, rounds to the even number. (RNE – default mode)
- `Float::round_nearest_ties_to_away(&ctx)` How this works is that it rounds to the nearest representable floating-point value. When halfway between two values, rounds to the number away from zero. (RNA)

- `Float::round_toward_zero(&ctx)` Always rounds the number to the number closest to zero. (RTZ)
- `Float::round_toward_negative(&ctx)` Always rounds the number to the number closest to $-\infty$. (RTN)
- `Float::round_toward_positive(&ctx)` Always rounds the number to the number closest to $+\infty$. (RTP)

In the high-level Rust binding of Z3, `Float::round_nearest_ties_to_even(&ctx)` and `Float::round_nearest_ties_to_away(&ctx)` are unavailable. These modes can be accessed via `z3_sys`.

```rust
use z3::{ast::Ast, ast::Float, Config, Context, Solver};

use z3::{ast::Ast, ast::Float, Config, Context, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // Create -1.5 as a 64-bit float
    let x = Float::from_f64(&ctx, -100.5);
    let y = Float::from_f64(&ctx, 100.55);

    // Define rounding modes
    let rtz = Float::round_towards_zero(&ctx);
    let rtn = Float::round_towards_negative(&ctx);
    let rtp = Float::round_towards_positive(&ctx);

    // Apply rounding
    let x_rtz = rtz.add(&x, &y);
    let x_rtn = rtn.add(&x, &y);
    let x_rtp = rtp.add(&x, &y);

    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!("RTZ: {}", model.eval(&x_rtz, true).unwrap());
            println!("RTN: {}", model.eval(&x_rtn, true).unwrap());
            println!("RTP: {}", model.eval(&x_rtp, true).unwrap());
        }
        _ => println!("No solution"),
    }
}
```

Ironically, I always get the same result when using the rounding modes... why? The question arises as to why use Float over Real (or BV). First, special Values like `NaN`, $+\infty$, $-\infty$ are native to Float, but impossible in Real or BV. Also, rounding semantics like `RTZ` for truncate and `RTP` for ceiling are available in Float.

(k) **Seq:** used to model vectors and ordered collections. In general, it indicates sequences of elements (often String theory).

```rust
use z3::{ast::String, Config, Context, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
```

```
    let solver = Solver::new(&ctx);

    let s1 = String::from_str(&ctx, "abc").unwrap();
    let s2 = String::from_str(&ctx, "bc").unwrap();

    let substr = s1.contains(&s2);
    solver.assert(&substr);

    let starts_with = String::from_str(&ctx, "a").unwrap().prefix(&s1);
    solver.assert(&starts_with);

    assert_eq!(solver.check(), z3::SatResult::Sat);
    let model = solver.get_model().unwrap();
    println!("Valid: s1 = {}, s2 = {}", model.eval(&s1, true).unwrap(),
    ↪  model.eval(&s2, true).unwrap());
}
```

(l) **RE:** Regular expression sort provides a representation of regular languages over sequences (often strings). Use it whenever you need to constrain a sequence variable by membership in a regular language—e.g., does this URL path match `/api/v0-9+/items`?

```
// is there a string that matches the regex "ab" one-or-more times and has length 1?
↪  No!
fn main() {
    use z3::{
        ast::{Ast, Int, Regexp, String},
        Config, Context, SatResult, Solver,
    };

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let s = String::new_const(&ctx, "s");

    // Build the regex: one-or-more "ab"
    let re_literal = Regexp::literal(&ctx, "ab");
    let one_or_more_ab = re_literal.plus();

    // wraps Z3_mk_seq_in_re
    let matches = s.regex_matches(&one_or_more_ab);

    solver.assert(&matches);
    solver.assert(&s.length()._eq(&Int::from_i64(&ctx, 1)));
    match solver.check() {
        SatResult::Sat => {
            let model = solver.get_model().unwrap();
            let s_value = model.eval(&s, true).unwrap();
            println!("Found a string that matches the regex: {}", s_value);
        }
        SatResult::Unsat => {
            println!("No string matches the regex.");
        }
        SatResult::Unknown => {
            println!("The satisfiability of the regex is unknown.");
        }
    }
}
```

(m) **Unknown:** Should not be used unless there is a type that does not fall under any of the

previous categories. It represents an error condition rather than a legitimate sort type for users. Z3 uses numeric identifiers $0 - 12$ & 1000 to represent different sort kinds in its internal AST representation.

```
typedef enum {
    Z3_UNINTERPRETED_SORT = 0,
    Z3_BOOL_SORT,                 // 1
    Z3_INT_SORT,                  // 2
    Z3_REAL_SORT,                 // 3
    Z3_BV_SORT,                   // 4
    Z3_ARRAY_SORT,                // 5
    Z3_DATATYPE_SORT,             // 6
    Z3_RELATION_SORT,             // 7
    Z3_FINITE_DOMAIN_SORT,        // 8
    Z3_FLOATING_POINT_SORT,       // 9
    Z3_ROUNDING_MODE_SORT,        // 10
    Z3_SEQ_SORT,                  // 11
    Z3_RE_SORT,                   // 12
    Z3_UNKNOWN_SORT = 1000        // deliberately out-of-band
} Z3_sort_kind;
```

3. **DeclKind:** Represents the different types of interpreted function declaration. This enum corresponds directly to the `Z3_decl_kind` constants in Z3's C API and defines the fundamental operations that Z3 can reason about in various mathematical theories. Specifically, `DeclKind` identifies the *semantic meaning* of function declarations in Z3; i.e., when you create expressions, each operation (like addition, conjunction, array access, etc.) has an associated declaration kind that tells Z3 how to reason about that operation. Z3's *DeclKind* encompasses over 200 different operations. The following is a summary of all variants in the *DeclKind* type:

   - **Boolean constants and connectives:** TRUE, FALSE, EQ, DISTINCT, ITE, AND, OR, IFF, XOR, NOT, IMPLIES, OEQ

```
fn main() {
    use z3::{ast::Bool, Config, Context};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let p = Bool::new_const(&ctx, "p");
    let q = Bool::new_const(&ctx, "q");

    // DeclKind::AND - conjunction
    let _and_expr = Bool::and(&ctx, &[&p, &q]);

    // DeclKind::OR - disjunction
    let _or_expr = Bool::or(&ctx, &[&p, &q]);

    // DeclKind::NOT - negation
    let _not_p = p.not();

    // DeclKind::IMPLIES - implication
    let _implies_expr = p.implies(&q);

    // DeclKind::ITE - if-then-else
    let _ite_expr = p.ite(&q, &Bool::from_bool(&ctx, false));
}
```

   - **Arithmetic operators:** ANUM, AGNUM, UMINUS, ADD, SUB, MUL, DIV, IDIV, REM, MOD, LE, LT, GE, GT, TO_REAL, TO_INT, IS_INT, POWER

```rust
fn main() {
    use z3::{ast::Ast, ast::Int, Config, Context, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    // DeclKind::ADD - addition
    let _sum = Int::add(&ctx, &[&x, &y]);

    // DeclKind::MUL - multiplication
    let _product = Int::mul(&ctx, &[&x, &y]);

    // DeclKind::LE - less than or equal
    let le_constraint = x.le(&Int::from_i64(&ctx, 10));

    // DeclKind::GT - greater than
    let gt_constraint = y.gt(&Int::from_i64(&ctx, 0));

    solver.assert(&le_constraint);
    solver.assert(&gt_constraint);
    solver.assert(&_product._eq(&_sum));

    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!("Model found:");
            println!("x: {}", model.eval(&x, false).unwrap());
            println!("y: {}", model.eval(&y, false).unwrap());
        }
        z3::SatResult::Unsat => {
            println!("No solution found.");
        }
        z3::SatResult::Unknown => {
            println!("Solver could not determine satisfiability.");
        }
    }
}
```

- **Arrays and sets:** STORE, SELECT, CONST_ARRAY, ARRAY_MAP, ARRAY_DEFAULT, AS_ARRAY, ARRAY_EXT, SET_UNION, SET_INTERSECT, SET_DIFFERENCE, SET_COMPLEMENT, SET_SUBSET

```rust
fn main() {
    use z3::{
        ast::{Array, Int},
        Config, Context
    };

    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let array = Array::new_const(&ctx, "arr", &z3::Sort::int(&ctx),
    ↪  &z3::Sort::int(&ctx));
    let index = Int::new_const(&ctx, "i");
    let value = Int::new_const(&ctx, "v");

```

```
    // DeclKind::SELECT - array read
    let read_value = array.select(&index);

    // DeclKind::STORE - array write
    let updated_array = array.store(&index, &value);

    // DeclKind::CONST_ARRAY - constant array
    let const_array = Array::const_array(&ctx, &z3::Sort::int(&ctx),
    ↪  &Int::from_i64(&ctx, 42));
}
```

- **Bit-vector operations:** BNUM, BIT0, BIT1, BNEG, BADD, BSUB, BMUL, BSDIV, BUDIV, BSREM, BUREM, BSMOD, CONCAT, SIGN_EXT, ZERO_EXT, EXTRACT, REPEAT, BSHL, BLSHR, BASHR, ROTATE_LEFT, ROTATE_RIGHT, BREDOR, BREDAND, BAND, BOR, BXOR, BNAND, BNOR, BXNOR, BCOMP

```
fn main() {
    use z3::{ast::BV, Config, Context};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let a = BV::new_const(&ctx, "a", 8);
    let b = BV::new_const(&ctx, "b", 8);

    // DeclKind::BADD - bit-vector addition
    let _sum = a.bvadd(&b);

    // DeclKind::BAND - bitwise AND
    let _and_result = a.bvand(&b);

    // DeclKind::BSHL - left shift a for 2 bits
    let _shifted = a.bvshl(&BV::from_u64(&ctx, 2, 8));

    // DeclKind::ULT - unsigned less than
    let _comparison = a.bvult(&b);

    // DeclKind::EXTRACT - bit extraction
    let _extract = a.extract(7, 4); // Extract bits 7-4 inclusive
}
```

- **Sequences and regexes:** SEQ_UNIT, SEQ_EMPTY, SEQ_CONCAT, SEQ_LENGTH, SEQ_CONTAINS, SEQ_PREFIX, SEQ_SUFFIX, SEQ_EXTRACT, SEQ_REPLACE, SEQ_AT, SEQ_INDEX, SEQ_TO_RE, SEQ_IN_RE, STR_TO_INT, INT_TO_STR, RE_PLUS, RE_STAR, RE_OPTION, RE_CONCAT, RE_UNION, RE_RANGE, RE_LOOP, RE_INTERSECT, RE_EMPTY_SET, RE_FULL_SET, RE_COMPLEMENT

```
fn main() {
    use z3::{
        ast::{Ast, String},
        Config, Context,
    };

    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let s1 = String::from_str(&ctx, "hello").unwrap();
    let space = String::from_str(&ctx, " ").unwrap();
    let s2 = String::from_str(&ctx, "world").unwrap();

```

```
    // DeclKind::SEQ_CONCAT - concatenation
    let concat = z3::ast::String::concat(&ctx, &[&s1, &space, &s2]);

    // DeclKind::SEQ_LENGTH - length
    let length = concat.length();

    // DeclKind::SEQ_CONTAINS - substring check
    let _contains = s1.contains(&String::from_str(&ctx, "ell").unwrap());

    let solver = z3::Solver::new(&ctx);
    solver.assert(&length._eq(&z3::ast::Int::from_i64(&ctx, 11)));

    match solver.check() {
        z3::SatResult::Sat => println!("The concatenated string is 'hello world' with length 11."),
        z3::SatResult::Unsat => println!("The constraints are unsatisfiable."),
        z3::SatResult::Unknown => println!("The satisfiability of the constraints is unknown."),
    }
}
```

- **Datatypes and relations:** DT_CONSTRUCTOR, DT_RECOGNISER, DT_ACCESSOR, DT_UPDATE_FIELD, RA_STORE, RA_EMPTY, RA_IS_EMPTY, RA_JOIN, RA_UNION, RA_WIDEN, RA_PROJECT, RA_FILTER, RA_NEGATION_FILTER, RA_RENAME, RA_COMPLEMENT, RA_SELECT, RA_CLONE

```
fn main() {
    use z3::{ast::Int, Config, Context, DatatypeBuilder};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    // Define a Point datatype
    // Point = { mk_point: (x: Int, y: Int) }
    let point = DatatypeBuilder::new(&ctx, "Point")
        .variant(
            "mk_point",
            vec![
                ("x", z3::DatatypeAccessor::Sort(z3::Sort::int(&ctx))),
                ("y", z3::DatatypeAccessor::Sort(z3::Sort::int(&ctx))),
            ],
        )
        .finish();

    // DeclKind::DT_CONSTRUCTOR - create instance
    // point.variants[0] is the constructor "mk_point"
    let p = point.variants[0]
        .constructor
        .apply(&[&Int::from_i64(&ctx, 10), &Int::from_i64(&ctx, 20)]);

    // DeclKind::DT_ACCESSOR - field access
    // point.variants[0].accessors[0] is the accessor for "x"
    // We can use it to get the value of x from the point instance p
    let _x_val = point.variants[0].accessors[0].apply(&[&p]);
}
```

- **Pseudo-Boolean constraints:** Z3 represents linear constraints over Boolean literals as *pseudo-Boolean* (PB) atoms. Two special cases are **cardinality** constraints—"at most" and "at least" k true literals—while the general form allows integer weights. In the Rust API

the cardinality helpers `PB_AT_MOST` and `PB_AT_LEAST` do not appear as separate calls; they are expressed with the weighted predicates `pb_le` and `pb_ge` by giving every literal weight 1. Concretely:

- `PB_AT_MOST` — encoded as `Bool::pb_le(&ctx, &[(&b1,1), ...], k)`.
- `PB_AT_LEAST` — encoded as `Bool::pb_ge(&ctx, &[(&b1,1), ...], k)`.
- `PB_LE` — `Bool::pb_le(&ctx, &[(&b1,w1), ...], k)`.
- `PB_GE` — `Bool::pb_ge(&ctx, &[(&b1,w1), ...], k)`.
- `PB_EQ` — `Bool::pb_eq(&ctx, &[(&b1,w1), ...], k)`.

```rust
use z3::{ast::Bool, Config, Context, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // Three Boolean decision variables
    let x = Bool::new_const(&ctx, "x");
    let y = Bool::new_const(&ctx, "y");
    let z = Bool::new_const(&ctx, "z");

    /* At most two are true - x + y + z <= 2
    let at_most_two = Bool::pb_le(&ctx, &[(&x, 1), (&y, 1), (&z, 1)], 2);
    solver.assert(&at_most_two);

    /* Weighted: 3·x + 2·y + 1·z >= 4
    let weighted_ge = Bool::pb_ge(&ctx, &[(&x, 3), (&y, 2), (&z, 1)], 4);
    solver.assert(&weighted_ge);

    match solver.check() {
        z3::SatResult::Sat => {
            let m = solver.get_model().unwrap();
            println!("x = {}", m.eval(&x, true).unwrap().as_bool().unwrap());
            println!("y = {}", m.eval(&y, true).unwrap().as_bool().unwrap());
            println!("z = {}", m.eval(&z, true).unwrap().as_bool().unwrap());
        }
        other => println!("Solver returned {:?}", other),
    }
}
```

- **Floating-point operations and modes:** FPA_NUM, FPA_PLUS_INF, FPA_MINUS_INF, FPA_NAN, FPA_PLUS_ZERO, FPA_MINUS_ZERO, FPA_ADD, FPA_SUB, FPA_NEG, FPA_MUL, FPA_DIV, FPA_REM, FPA_ABS, FPA_MIN, FPA_MAX, FPA_FMA, FPA_SQRT, FPA_ROUND_TO_INTEGRAL, FPA_EQ, FPA_LT, FPA_GT, FPA_LE, FPA_GE, FPA_IS_NAN, FPA_IS_INF, FPA_IS_ZERO, FPA_IS_NORMAL, FPA_IS_SUBNORMAL, FPA_IS_NEGATIVE, FPA_IS_POSITIVE, FPA_RM_NEAREST_TIES_TO_EVEN, FPA_RM_NEAREST_TIES_TO_AWAY, FPA_RM_TOWARD_POSITIVE, FPA_RM_TOWARD_NEGATIVE, FPA_RM_TOWARD_ZERO, FPA_FP, FPA_TO_FP, FPA_TO_FP_UNSIGNED, FPA_TO_UBV, FPA_TO_SBV, FPA_TO_REAL, FPA_TO_IEEE_BV, FPA_BVWRAP, FPA_BV2RM

```rust
fn main() {
    use z3::{ast::Float, Config, Context};

    let cfg = Config::new();
```

```
    let ctx = Context::new(&cfg);

    let x = Float::from_f64(&ctx, 1.5);
    let y = Float::from_f64(&ctx, 2.3);

    // DeclKind::FPA_RNE - rounding mode for floating-point arithmetic
    let rne = Float::round_towards_zero(&ctx);

    // DeclKind::FPA_ADD - floating-point addition
    let sum = rne.add(&x, &y);

    // DeclKind::FPA_MUL - floating-point multiplication
    let product = rne.mul(&x, &y);

    // DeclKind::FPA_LT - floating-point less than
    let comparison = x.lt(&y);
}
```

- **Proof objects and special rules:** Z3 can generate detailed proof objects that justify why formulas are valid or unsatisfiable; for example: PR_UNDEF, PR_TRUE, PR_ASSERTED, PR_GOAL, PR_MODUS_PONENS, PR_REFLEXIVITY, PR_SYMMETRY, PR_TRANSITIVITY, PR_TRANSITIVITY_STAR, PR_MONOTONICITY, PR_QUANT_INTRO, PR_BIND, PR_DISTRIBUTIVITY, PR_AND_ELIM, PR_NOT_OR_ELIM, PR_REWRITE, PR_REWRITE_STAR, PR_PULL_QUANT, PR_PUSH_QUANT, PR_ELIM_UNUSED_VARS, PR_DER, PR_QUANT_INST, PR_HYPOTHESIS, PR_LEMMA, PR_UNIT_RESOLUTION, PR_IFF_TRUE, PR_IFF_FALSE, PR_COMMUTATIVITY, PR_DEF_AXIOM, PR_DEF_INTRO, PR_APPLY_DEF, PR_IFF_OEQ, PR_NNF_POS, PR_NNF_NEG, PR_SKOLEMIZE, PR_MODUS_PONENS_OEQ, PR_TH_LEMMA, PR_HYPER_RESOLVE

- **Miscellaneous:** These are *catch-all* values that do not correspond to standard interpreted functions; for example `INTERNAL, UNINTERPRETED`. For `UNINTERPRETED`, Z3 treats the symbol purely as a fresh function over its domain and range sorts. For `INTERNAL`, it is used only by Z3's own re-writers and tactic pipelines; user code should never receive or create this kind. An example of `UNINTERPRETED` can be found in the following;

```
fn main() {
    use z3::{
        ast::{Ast, Dynamic, Int},
        Config, Context, Solver,
    };

    let ctx = Context::new(&Config::new());
    let solver = Solver::new(&ctx);

    let f = z3::FuncDecl::new(&ctx, "f", &[&z3::Sort::int(&ctx)],
    ↪  &z3::Sort::int(&ctx));
    let x = Int::new_const(&ctx, "x");

    solver.assert(&f.apply(&[&x])._eq(&Dynamic::from_ast(&Int::add(
        &ctx,
        &[&x, &Int::from_i64(&ctx, 1)],
    )))); // f(x) = x+1

    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!("Model found:");
```

```
22              println!("{}", model);
23          }
24          z3::SatResult::Unsat => {
25              println!("No solution found.");
26          }
27          z3::SatResult::Unknown => {
28              println!("Solver could not determine satisfiability.");
29          }
30      }
31  }
```

4. **GoalPrec:** Defines the precision of a given goal. Some goals can be transformed using over/under approximations. The following variants are present:

   - **Precise:** No approximation has been applied—both satisfiable and unsatisfiable answers for the goal are exactly preserved.
   - **Over:** In over-approximation, we make the formula looser. Therefore, unsat answers are preserved but sat answers are not; meaning that if we prove that the over-approximation is unsatisfiable, we can conclude that the original expression was unsatisfiable as well. This is useful if we want to keep the proofs valid. For example, adding a disjunct, eliminating a conjunct, or extending the range of numbers for an expression are examples of over-approximating the original formula.
   - **Under:** In under-approximation, we make the formula tighter. Therefore, sat answers are preserved but unsat answers are not; i.e., proving the satisfiability of the under-approximation results into showing that the original expression is satisfiable. For example, adding a conjunct, eliminating a disjunct, or limiting the range of numbers for an expression are examples of under-approximating the original formula.
   - **UnderOver:** Both under- and over-approximations have been applied (e.g. in different parts of the goal). Neither sat nor unsat answers are fully preserved, so the answer is *garbage*.

   The following is a useful image I picked up from the following link:

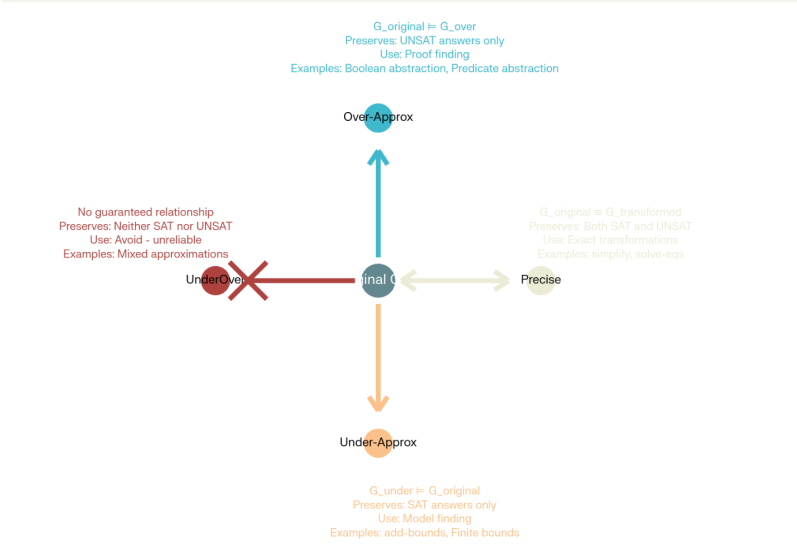

Figure 1: Z3 GoalPrec Approximation Types.

3 *params* re-exports

The `get_global_param, reset_all_global_params, set_global_param` from the *params* module are re-exported. Z3's *global parameters* are shared key–value store of configuration options that affect all created Z3 contexts; for example, timeout, smt.string_solver, etc.

- **Set** a parameter with `set_global_param`(k, v), e.g. `set_global_param("timeout","1000")` means *any new context will time out after 1000 ms.* This change applies to all subsequently created Z3 contexts. Existing contexts and solvers are unaffected.
- **Get** its current value with `get_global_param`(k), which returns `Some(String)` if you've set it, or `None` to use Z3's built-in default.
- **Reset** everything to Z3's built-in defaults with `reset_all_global_params()`, after which every `get_global_param` returns `None` again. This does *not* affect any existing tactics, solvers, or contexts. Built-in defaults are hard-coded—e.g. the default for `timeout` is `4294967295` (no limit), for `model` is `false`, etc.

In short, `set_global_param` and `reset_all_global_params` manipulate a process-wide table that every new Config, Context, solver, tactic, etc. copies when it is created; once a Context exists, later changes to the global table do not reach it. Let's explain this with the following example:

```rust
use z3::{ast::Ast, ast::Int, reset_all_global_params, set_global_param, Config, Context,
↪  Solver};

fn main() {
    set_global_param("timeout", "1"); // 1ms timeout

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    solver.assert(&(Int::mul(&ctx, &[&x, &y]))._eq(&Int::from_i64(&ctx, 100)));

    match solver.check() {
        z3::SatResult::Sat => println!("SAT"),
        z3::SatResult::Unsat => println!("UNSAT"),
        z3::SatResult::Unknown => println!("TIMEOUT/UNKNOWN"),
    }

    // Reset and try again with infinite timeout
    reset_all_global_params();

    let cfg2 = Config::new(); // inherits the now-empty table
    let ctx2 = Context::new(&cfg2);
    let solver2 = Solver::new(&ctx2);
    let x2 = Int::new_const(&ctx2, "x");
    let y2 = Int::new_const(&ctx2, "y");

    solver2.assert(&(Int::mul(&ctx2, &[&x2, &y2]))._eq(&Int::from_i64(&ctx2, 100)));

    match solver2.check() {
        z3::SatResult::Sat => println!("SAT after reset"),
        z3::SatResult::Unsat => println!("UNSAT after reset"),
        z3::SatResult::Unknown => println!("TIMEOUT/UNKNOWN after reset"),
    }
}
```

The output of the above code is the following:

```
TIMEOUT/UNKNOWN
SAT after reset
```

There are multiple factors to pinpoint here:

1. **Global-parameter snapshotting.**
   (a) `set_global_param("timeout","1")` stores the pair `timeout = 1` in Z3's *process-wide* parameter table.
   (b) `let cfg1 = Config::new();` copies *all* global entries into `cfg1`. From now on that `Config` carries its own private copy `timeout = 1`.
   (c) `Context::new(&cfg1)` propagates the 1 ms limit to the first solver.
   (d) `reset_all_global_params();` wipes the global table, but does *not* mutate `cfg1`.
   (e) Re-using `cfg1` for a second context therefore copies the same 1 ms limit again.
   (f) Both solvers inherit the same tiny time-out and report `unknown`.

   Therefore, re-using the same `cfg1` will lead to both outputs showing *TIMEOUT/UNKNOWN*. Likewise, if we create a new `cfg2`, but erase the `reset_all_global_params()` line, the new configuration will inherit the process-wide parameter table, thus, inheriting the *1ms* timeout. *Fix:* after resetting the globals, create a fresh `Config` (call it `cfg2`) and build the second context from that. The new configuration will see an empty parameter table and therefore inherit the default "no time-out".

2. **Contexts are not interchangeable.** Every AST node records the `Context` that created it. Mixing nodes from two different contexts in the same expression triggers an assertion inside the Rust bindings. In practice this means you must rebuild `x`, `y`, and any other ASTs from the first context, after you switch to `ctx2`.

4 *statistics* re-exports

The `StatisticsEntry, StatisticsValue` from the *statistics* module are re-exported. Z3 collects various runtime metrics in a `Statistics` object. Each entry is represented by:

`StatisticsEntry` A single key–value pair within `Statistics`:

`key: String` The name of the statistic.

`value: StatisticsValue` The associated value which can be either a non-negative integer counter (`UInt(u32)`) or a floating-point measurement (`Double(f64)`).

```rust
use z3::{Config, Context, Solver, StatisticsEntry, StatisticsValue};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = z3::ast::Int::new_const(&ctx, "x");
    solver.assert(&x.gt(&z3::ast::Int::from_i64(&ctx, 0))); // x > 0
    let _ = solver.check();


    let stats = solver.get_statistics();

    // Example keys to fetch
    let example_keys = ["max memory", "time"];
```

```
    for &key in &example_keys {
        match stats.value(key) {
            Some(StatisticsValue::UInt(n)) => println!("{:15} = {} (UInt)", key, n),
            Some(StatisticsValue::Double(d)) => println!("{:15} = {} (Double)", key, d),
            None => println!("{:15} = <not reported>", key),
        }
    }
    println!("\nAll statistics:");
    for StatisticsEntry { key, value } in stats.entries() {
        println!("{} = {:?}", key, value);
    }
}
```

5 *version* re-exports

The `full_version, version, Version` from the *version* module are re-exported. Z3 exposes the library version both as a structured tuple via the *version()* function and as a formatted string via *full_version()*. The Rust bindings provide:

`struct Version` Holds the four components of the Z3 library version:

- `major: u32` Major version number.
- `minor: u32` Minor version number.
- `build_number: u32` Internal build counter.
- `revision_number: u32` Revision identifier (patch/commit identifier).

`fn version() -> Version` Queries Z3 via `Z3_get_version`, filling each `Version` field. Returns a `Version` struct populated with the library's current version.

`fn full_version() -> &'static str` Calls `Z3_get_full_version` to obtain a C-string containing the full version string (including build date, git hash, etc.). Returns it as a Rust `&str`.

```
use z3::{full_version, version};

fn main() {
    // Get structured version components
    let ver = version();
    println!("Structured version: {:?}", ver);

    // Get full version string
    let full_ver = full_version();
    println!("Full version: {}", full_ver);

    let ver1 = version();
    let ver2 = version();
    assert_eq!(ver1, ver2, "Version changed between calls!"); // Never fails
}
```

6 *lib* module

The following are the public enum and struct definitions in the *lib* module:

- ***Config***

  Purpose: Holds configuration parameters for a Z3 `Context`. Internally wraps a `Z3_config` and a list of key/value pairs.

  Fields (private)

  - `kvs`: `Vec<(CString, CString)>` of raw key/value pairs.
  - `z3_cfg`: raw `Z3_config` handle.

  When to use

  - Before creating any Z3 `Context` to tune context or solver behavior.

  How to use

```rust
use z3::{Config, Context};

fn main() {
    let mut cfg = Config::new();

    cfg.set_param_value("timeout", "1000");        // 1 second timeout
    cfg.set_proof_generation(true);                // enable proofs
    cfg.set_model_generation(true);                // enable model production


    let ctx = Context::new(&cfg);
    let solver = z3::Solver::new(&ctx);
}
```

  Tips

  - All parameters must be set *before* creating the `Context`. In other words, calling setters after `Context::new()` has *no effect* on the already existing contexts.
  - Use `set_param_value` for arbitrary flags not covered by the high-level API. Note that you need to define the configuration as *mutable* to change the parameters. The following is a set of the legal parameters:[4]
    - **auto_config (bool) (default: true)** – Uses heuristics to automatically select the appropriate solver and configure it optimally. When enabled, Z3 analyzes the problem structure and chooses the best solving strategy.
    - **ctrl_c (bool) (default: true)** – Although, `ctrl_c` is listed as a legal parameter, setting the parameter does not work. Specifically, `cfg.set_bool_param_value("ctrl_c", false);` does not work (maybe depracated?).
    - **debug_ref_count (bool) (default: false)** – Enables debugging support for AST reference counting, which may be useful for debugging memory management issues.
    - **dot_proof_file (string) (default: proof.dot)** – Specifies the filename for outputting graphical proofs in *DOT* format. It is used when proof generation is enabled to visualize proof trees.

[4] A list of available parameters in *Z3* can be obtained by running the `z3 -pd` command in the terminal.

* **dump_models (bool) (default: false)** – Automatically dumps models whenever check-sat returns satisfiable.
* **encoding (string) (default: unicode)** – Sets the internal string encoding used by Z3.
* **model (bool) (default: true)** – Enables model generation for solvers when encountering satisfiable expressions. This is equivalent to `set_model_generation(true)`.
* **model_validate (bool) (default: false)** – When enabled Z3 will *double-check* every model it produces against the original assertions before returning `Sat`. Internally, after finding a candidate assignment, Z3 re-evaluates all asserted formulas with that model. When model validation is enabled and a model fails to validate, Z3 will produce a warning message but still return the model.
* **proof (bool) (default: false)** – Enables proof generation for unsatisfiable formulas. This is equivalent to `set_proof_generation(true)`.
* **rlimit (unsigned int) (default: 0)** – This parameter imposes a hard cap on Z3's *basic operation* count (e.g. AST construction, simplification, solving step). Once the solver has performed more than `rlimit` operations, it aborts the check and returns `Unknown`. Unlike `timeout`, this bound is completely deterministic and preferred for reproducible behavior. Lastly, 0 means *unlimited* operation cap.
* **smtlib2_compliant (bool) (default: false)** – When this flag is `true`, Z3 enforces strict *SMT-LIB 2.0* syntax. Any unsupported language extensions are rejected, and only constructs defined in the official *SMT-LIB 2.0* standard may appear. This is useful when you need maximal portability of your benchmarks or want to ensure compatibility with other SMT-LIB 2-compliant solvers.

```rust
use z3::{ast::Int, Config, Context, Solver};

fn main() {
    let mut cfg = Config::new();
    cfg.set_param_value("smtlib2_compliant", "true"); // Enforce strict standard
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    solver.assert(&x.gt(&Int::from_i64(&ctx, 0))); // x > 0
    solver.assert(&y.lt(&z3::ast::Int::add(&ctx, &[&x, &Int::from_i64(&ctx, 3)]))); // y < x + 3

    let smt2_output = solver.to_smt2();
    println!("{}", smt2_output);
}
```

* **stats (bool) (default: false)** – Enables collection and reporting of solving statistics.
* **timeout (unsigned int) (default: 4294967295)** – Sets maximum solving time in *milliseconds*. This is less deterministic than `rlimit` and can cause non-reproducible behavior. Note that if the solver does not produce a result before the timeout, the `Unknown` will be outputted. This is equivalent to `cfg.set_timeout_msec(5)`.
* **trace (bool) (default: false)** – Enables trace message logging for debugging.
* **trace_file_name (string) (default: z3.log)** – Specifies the output file for trace messages.

* **type_check (bool) (default: true)** – Enables type checking of formulas and terms.
* **unsat_core (bool) (default: false)** – Enables generation of unsatisfiable cores; i.e., it identifies minimal subsets of constraints that cause unsatisfiability. Without tracked assertion labels, `unsat_core()` returns empty even when enabled . Therefore, proper labeling is essential. For example the following code showcases this:

```rust
fn main() {
    use z3::{ast::Bool, Config, Context, SatResult, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);
    let p = Bool::from_bool(&ctx, true);
    let q = Bool::from_bool(&ctx, false);

    let lp = Bool::fresh_const(&ctx, "p_label");
    let lq = Bool::fresh_const(&ctx, "q_label");

    solver.assert_and_track(&p, &lp);
    solver.assert_and_track(&q, &lq);

    assert_eq!(solver.check(), SatResult::Unsat);

    // Extract which labels caused the conflict
    let core = solver.get_unsat_core();
    for l in core {
        println!("Conflict came from: {}", l);
    }
}
```

* **well_sorted_check (bool) (default: false)** – Performs additional type checking and more thorough type validation (alias for `type_check`).

```rust
use z3::{ast, ast::Ast, Config, Context, Solver};

fn main() {
    let mut cfg = Config::new();

    cfg.set_timeout_msec(5000); // 5 second timeout
    cfg.set_proof_generation(true); // Enable proof generation
    cfg.set_model_generation(true); // Enable model generation

    cfg.set_param_value("rlimit", "1000000"); // Resource limit
    cfg.set_param_value("stats", "true"); // Enable statistics
    cfg.set_param_value("unsat_core", "true"); // Enable unsat cores

    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = ast::Int::new_const(&ctx, "x");
    let y = ast::Int::new_const(&ctx, "y");
    solver.assert(&x.gt(&ast::Int::from_i64(&ctx, 0)));
    solver.assert(&y.lt(&ast::Int::from_i64(&ctx, 10)));
    solver.assert(&x._eq(&z3::ast::Int::mul(
        &ctx,
        &[&y, &ast::Int::from_i64(&ctx, 2)],
    )));

    // Check satisfiability
```

```
    match solver.check() {
        z3::SatResult::Sat => {
            println!("Satisfiable!");
            if let Some(model) = solver.get_model() {
                println!("Model: {}", model);
            }
        }
        z3::SatResult::Unsat => println!("Unsatisfiable!"),
        z3::SatResult::Unknown => println!("Unknown result"),
    }
}
```

- ***Context***

  Purpose: Manages all Z3 objects (ASTs, solvers, models, etc.) and global configuration options. Each `Context` defines an isolated logical environment. We may use multiple contexts. Contexts are independent; objects cannot be shared across them but can be translated.

  Fields (private)

  - `z3_ctx`: raw `Z3_context` pointer.

  When to use

  - After configuring `Config` and before creating ASTs or solvers.
  - When you need multiple independent logical environments (e.g. parallel queries).

  How to use

```
use z3::{ast, ast::Ast, Config, Context, Solver};

fn main() {
    let mut cfg = Config::new();
    cfg.set_model_generation(true);

    // create context
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = ast::Int::new_const(&ctx, "x");
    solver.assert(&x.gt(&ast::Int::from_i64(&ctx, 0))); // x > 0

    // new context
    let ctx2 = Context::new(&cfg);
    let solver2 = Solver::new(&ctx2);

    let x_in_ctx2 = x.translate(&ctx2);
    solver2.assert(&x_in_ctx2.lt(&ast::Int::from_i64(&ctx2, 10))); // x < 10
}
```

  Tips

  - `Context` is neither `Send` nor `Sync`: do not share it across threads.
  - All ASTs, solvers, and models belong to their creating `Context`. Mixing terms will cause errors; instead, you can translate as seen in the above code.

- ***ContextHandle***

  Purpose: A thread-safe handle for interrupting Z3 computations (solving, model building, etc.) from another thread.

Fields (private)

- `ctx`: reference to the parent `Context` that this handle will interrupt.

When to use

- When you need to enforce timeouts or user-initiated cancellation from a different thread.
- In long-running queries or tactics where you want external control to abort.
- To recover from hangs in custom tactics or complex quantifier instantiations.

How to use

```rust
use std::{thread, time::Duration};
use z3::{ast::Int, Config, Context, SatResult, Solver};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut cfg = Config::new();
    cfg.set_param_value("timeout", "100000");
    let ctx    = Context::new(&cfg);
    let solver = Solver::new(&ctx);
    let x      = Int::new_const(&ctx, "x");

    for i in 0..50_000 {
        solver.assert(&x.lt(&Int::from_i64(&ctx, i)));
    }

    thread::scope(|s| {
        let handle = ctx.handle();
        s.spawn(move || {
            thread::sleep(Duration::from_millis(100));
            handle.interrupt();
            eprintln!("Interrupt requested");
        });

        match solver.check() {
            SatResult::Unknown => println!("Solver was interrupted (Unknown)"),
            SatResult::Sat     => println!("Solver unexpectedly returned Sat"),
            SatResult::Unsat   => println!("Solver unexpectedly returned Unsat"),
        }
    });

    Ok(())
}
```

Tips

- Calling `interrupt()` sets an internal flag; it does not force immediate exit, but the next check in Z3 will abort work.
- Always hold your `ContextHandle` alive as long as you need the ability to cancel.
- If you call `Context::interrupt()` directly, it affects all handles for that context. In other words, interruption state is stored at the context level, not per-handle.

- ***Symbol***

Purpose: Represents the Z3 *symbol* type used to name term and type constructors (identifiers for constants, functions, and sorts).

Variants

- `Int(u32)`
- `String(String)`

When to use

- When declaring constants or functions: `Symbol::String("x".into())`.
- When inspecting symbols returned by Z3 APIs (e.g. `FuncDecl::name()`).

How to use:

```rust
use z3::{
    ast::Bool,
    Config, Context, FuncDecl, Sort, Symbol,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let sym_str: Symbol = "Node".into();    // String variant
    let sym_num: Symbol = 1u32.into();      // Int variant

    // Declare an uninterpreted sort named "Node"
    let node_sort: Sort = Sort::uninterpreted(&ctx, sym_str.clone());

    // Declare a constant c: Node
    let c_decl: FuncDecl = FuncDecl::new(&ctx, "c", &[], &node_sort);
    let c = c_decl.apply(&[]); // c : Node

    // Declare a function f1: Node -> Int
    let f1_decl: FuncDecl =
        FuncDecl::new(&ctx, sym_num.clone(), &[&node_sort], &Sort::int(&ctx));
    let f1_app = f1_decl.apply(&[&c]).as_int().unwrap();

    println!("f1(c) = {}", f1_app);
    println!("Function name: {}", f1_decl.name()); // f1_decl.name() returns a
    ↪ String
}
```

- ***Sort***

Purpose: Wrapper around a Z3 sort (`Z3_sort`) in a given `Context`, representing a type in the logic.

Fields (private)

- `ctx`: reference to the owning `Context`.
- `z3_sort`: raw Z3 sort handle.

When to use

- When constructing AST nodes and types.

How to use

```rust
use z3::{Config, Context, Sort, SortKind};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

```

```
    // Create some sorts
    let bool_s = Sort::bool(&ctx);
    let bv8_s = Sort::bitvector(&ctx, 8);

    // Inspect
    assert_eq!(bool_s.kind(), SortKind::Bool);
    assert_eq!(bv8_s.kind(), SortKind::BV);
}
```

Tips

- Only for **in-crate users**, never construct `Sort` directly; always use *Sort::new()* to handle refcounting properly. For outside users, please use code as shown in the above example.

- ***SortDiffers***

  Purpose: Error type indicating two `Sort` values are not the same kind.

  Fields (private)

  - `left`, `right`: the mismatched `Sort` values.

  When to use

  - In custom type-checking to report meaningful errors on mismatch.

  How to use

```
use z3::{Sort, SortDiffers};

fn check_same_sort<'a>(a: &'a Sort, b: &'a Sort) -> Result<(), SortDiffers<'a>> {
    if a.kind() == b.kind() {
        Ok(())
    } else {
        Err(SortDiffers::new(a.clone(), b.clone()))
    }
}

fn main() {
    let cfg = z3::Config::new();
    let ctx = z3::Context::new(&cfg);
    let bo = z3::Sort::bool(&ctx);
    let bv = z3::Sort::bitvector(&ctx, 8);
    if let Err(e) = check_same_sort(&bo, &bv) {
        eprintln!("Sorts differ {:?}", e);
    };
}
```

- ***IsNotApp***

  Purpose: Error type indicating an `Ast` is not an `App` node; i.e., not a function application.

  Fields (private)

  - `kind`: the actual `AstKind` encountered.

  When to use

  - To signal that pattern-matching on `App` failed.

  How to use

```
use z3::ast::{Ast, Bool, Int};
use z3::IsNotApp;

```

```rust
fn get_app_args<'a, T: Ast<'a>>(ast: &T) -> Result<&T, IsNotApp> {
    if ast.is_app() {
        return Ok(ast);
    }
    Err(IsNotApp::new(ast.kind()))
}

fn main() {
    let cfg = z3::Config::new();
    let ctx = z3::Context::new(&cfg);
    let term = z3::ast::Int::from_i64(&ctx, 42); // constants are apps
    match get_app_args(&term) {
        Err(e) => eprintln!("Expected App, found {:?}", e.kind()),
        _ => (),
    };

    // forall x. x > 0  this is a Quantifier node, not an App
    let x = z3::ast::Int::new_const(&ctx, "x");
    let body = x.gt(&Int::from_i64(&ctx, 0));
    let forall: Bool = z3::ast::forall_const(&ctx, &[&x], &[], &body);
    match get_app_args(&forall) {
        Err(e) => eprintln!("Expected App, found {:?}", e.kind()),
        _ => (),
    };
}
```

- ***Solver***

  Purpose: It is a *combined solver*, internally maintaining both a *non-incremental* solver and an *incremental* solver, switching behavior based on usage. It is attached to a `Context` and wraps a raw `Z3_solver`. Also supports `push`/`pop`, `check`, and proof / model extraction.

  Fields: (private)

  - `ctx`: reference to the owning `Context`.
  - `z3_slv`: raw solver handle (`Z3_solver`).

  When to use

  - General Sat solving and proof generation.
  - Quick feasibility checks during symbolic execution.
  - Running under a specific tactic or logic via `Solver::new_for_logic`.

  How to use

```rust
use z3::{ast, Config, Context, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let _solver = Solver::new(&ctx);
    let solver2 = Solver::new_for_logic(&ctx, "QF_LIA").unwrap(); // if the logic is
    ↪  unknown, it will give None, otherwise it will return Some(Solver)

    // Build a simple constraint: x > 0 ∧ x < 5
    let x = ast::Int::new_const(&ctx, "x");
    let c1 = x.gt(&ast::Int::from_i64(&ctx, 0));
    let c2 = x.lt(&ast::Int::from_i64(&ctx, 5));

```

```
    solver2.assert(&c1);
    solver2.assert(&c2);

    match solver2.check() {
        z3::SatResult::Sat => {
            let m = solver2.get_model().unwrap();
            println!("Model: x = {}", m.eval(&x, true).unwrap());
        }
        z3::SatResult::Unsat => println!("Unsat"),
        z3::SatResult::Unknown => println!("Unknown"),
    }
}
```

- ***Model***

  Purpose: Finite mapping from constants/functions to values that satisfies the assertions proven `Sat` by a `Solver`. Models can also be obtained from an `Optimize` context via `Model::of_optimize`.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_mdl`: raw model handle (`Z3_model`).

  When to use

  - After `solver.check() == Sat` to inspect witness values. It can also be used when the solver spits *unknown*. In the case of *unknown*, a model will not be generated or it will be generated; however, it is not guaranteed to satisfy the constraints.
  - In counter-example generation or test-case extraction.

  How to use

```
fn main() {
    use z3::{ast::Ast, ast::Int, Config, Context, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");


    let _sum = Int::add(&ctx, &[&x, &y]);
    let _product = Int::mul(&ctx, &[&x, &y]);
    let le_constraint = x.le(&Int::from_i64(&ctx, 10));
    let gt_constraint = y.gt(&Int::from_i64(&ctx, 0));

    solver.assert(&le_constraint);
    solver.assert(&gt_constraint);
    solver.assert(&_product._eq(&_sum));

    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!("Model found:");
            println!("x: {}", model.eval(&x, false).unwrap());
            println!("y: {}", model.eval(&y, false).unwrap());
        }
```

```
        z3::SatResult::Unsat => {
            println!("No solution found.");
        }
        z3::SatResult::Unknown => {
            println!("Solver could not determine satisfiability.");
        }
    }
}
```

- ***Optimize***

  Purpose: Optimization engine built on top of Z3's `Z3_optimize` object. It supports hard & soft constraints along with multi-objective (Pareto) optimization.[5]

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_opt`: raw optimize handle (`Z3_optimize`).

  When to use

  - Maximizing or minimizing arithmetic terms under constraints.
  - For Pareto optimization of several objectives.
  - Ability to parse SMT-LIB2 optimization problems via `from_string`.

  How to use

```rust
use z3::{ast, Config, Context, Optimize, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let opt = Optimize::new(&ctx);

    // Create integer variables
    let x = ast::Int::new_const(&ctx, "x");
    let y = ast::Int::new_const(&ctx, "y");

    // Hard constraints (must be satisfied)
    opt.assert(&x.ge(&ast::Int::from_i64(&ctx, 0))); // x >= 0
    opt.assert(&y.ge(&ast::Int::from_i64(&ctx, 0))); // y >= 0
    opt.assert(&(&x + &y).le(&ast::Int::from_i64(&ctx, 20))); // x + y <= 20

    // Soft constraints (weighted penalties for violation)
    opt.assert_soft(
        &x.le(&ast::Int::from_i64(&ctx, 8)), // x <= 8 (preferred)
        10,                                  // Weight = 10 (higher weights make
        ↪ constraints \harder" to violate)
        Some("group1".into()),               // Constraint group name
    );

    opt.assert_soft(
        &y.le(&ast::Int::from_i64(&ctx, 5)), // y <= 5 (preferred)
        5,                                   // Weight = 5
        Some("group2".into()),
    );

```

[5] Pareto optimization is a mathematical framework for solving problems with multiple conflicting objectives where improving one objective inevitably worsens another.

```
    // Optimization objectives
    let profit = &x * ast::Int::from_i64(&ctx, 3) + &y * ast::Int::from_i64(&ctx,
    ↪  2); // 3x + 2y
    opt.maximize(&profit); // Maximize profit

    match opt.check(&[]) {
        SatResult::Sat => {
            let model = opt.get_model().unwrap();

            // Evaluate variables
            let x_val = model.eval(&x, true).unwrap().as_i64().unwrap();
            let y_val = model.eval(&y, true).unwrap().as_i64().unwrap();
            let profit_val = model.eval(&profit, true).unwrap().as_i64().unwrap();

            // Check which soft constraints were satisfied
            let soft1_satisfied = x_val <= 8;
            let soft2_satisfied = y_val <= 5;

            println!("Optimal solution found:");
            println!("- x = {}, y = {}", x_val, y_val);
            println!("- Profit = {}", profit_val);
            println!("Soft constraints:");
            println!(
                "- x <= 8: {} (weight: 10)",
                if soft1_satisfied { "yes" } else { "no" }
            );
            println!(
                "- y <= 5: {} (weight: 5)",
                if soft2_satisfied { "yes" } else { "no" }
            );
        }
        SatResult::Unsat => println!("No solution exists"),
        SatResult::Unknown => println!("Could not determine solution"),
    }
}
```

Tips

- Objectives are returned in the same order you added them.
- Combine hard constraints (`assert`) and soft constraints (`assert_soft`) to model *best effort* goals.

- ***FuncDecl***

  Purpose: Describes the signature of a constant or n-ary function in Z3 (name, argument sorts, and result sort). A constant is just a zero-arity `FuncDecl`.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_func_decl`: raw `Z3_func_decl` handle.

  When to use

  - When declaring new symbols: constants, uninterpreted functions, or datatype constructors.

  How to use

```
use z3::{
    ast::{self, Ast},
```

```
    Config, Context, FuncDecl, Sort, Symbol,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let int = Sort::int(&ctx);
    let bool = Sort::bool(&ctx);

    // Declare a function  f : Int * Int -> Bool
    let f_sym = Symbol::String("f".into());
    let f = FuncDecl::new(&ctx, f_sym, &[&int, &int], &bool);
    let x = f.arity(); // f.arity() == 2
    assert_eq!(x, 2);

    // Apply it to concrete arguments
    let a = ast::Int::new_const(&ctx, "a");
    let b = ast::Int::new_const(&ctx, "b");
    let app = f.apply(&[&a, &b]);

    // Check the type of the application
    let app_sort: Sort<'_> = app.get_sort();
    assert_eq!(app_sort, Sort::bool(&ctx));
}
```

- ***FuncInterp***

  Purpose: Holds the concrete interpretation of a function symbol inside a `Model` (i.e. which value it returns for every argument tuple).

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_func_interp`: raw `Z3_func_decl` handle.

  When to use

  - After checking that the solver is *Sat*, use `model.get_func_interp(&func)` to inspect how an uninterpreted function was satisfied.

  How to use

```
use z3::{
    ast::{self, Ast, Dynamic},
    Config, Context, FuncDecl, Sort, Symbol,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = z3::Solver::new(&ctx);

    let int_sort = Sort::int(&ctx);
    let f_sym = Symbol::String("f".into());
    let f = FuncDecl::new(&ctx, f_sym, &[&int_sort], &int_sort);

    let zero = ast::Int::from_i64(&ctx, 0);
    let one = ast::Int::from_i64(&ctx, 1);

    let zero_dyn: Dynamic = zero.clone().into();
    let one_dyn: Dynamic = one.clone().into();
```

```
20
21      // Example constraints: f(0) = 1  and  f(1) = 0
22      solver.assert(&f.apply(&[&zero])._eq(&one_dyn));
23      solver.assert(&f.apply(&[&one])._eq(&zero_dyn));
24
25      assert_eq!(solver.check(), z3::SatResult::Sat);
26
27      let model = solver.get_model().unwrap();
28
29      if let Some(x) = model.get_func_interp(&f) {
30          println!("Interpretation of f: {:?}", x);
31      };
32  }
```

- ***FuncEntry***

  Purpose: Represents one row in a `FuncInterp` table (a concrete argument tuple and its mapped value).

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_func_entry`: raw `Z3_func_entry` handle.

  When to use

  - Iterate over explicit mappings and the function interpretation using `x.get_entries()`, which returns a vector of `FuncEntry` representing the function's explicit input-output entries, where `x` is a `FuncInterp`. Note that when you inspect a function interpretation in Z3 using `get_entries()`, you may see only a single entry even if the function is defined for more inputs. This is expected behavior: Z3 models functions as a finite map (explicit entries) plus a default else value. *Only those input tuples that are explicitly constrained in your problem will appear as entries.*

  How to use

```
1   use z3::{
2       ast::{self, Ast, Dynamic},
3       Config, Context, FuncDecl, Sort, Symbol,
4   };
5
6   fn main() {
7       let cfg = Config::new();
8       let ctx = Context::new(&cfg);
9       let solver = z3::Solver::new(&ctx);
10
11      let int_sort = Sort::int(&ctx);
12      let f_sym = Symbol::String("f".into());
13      let f = FuncDecl::new(&ctx, f_sym, &[&int_sort], &int_sort);
14
15      let zero = ast::Int::from_i64(&ctx, 0);
16      let one = ast::Int::from_i64(&ctx, 1);
17
18      let zero_dyn: Dynamic = zero.clone().into();
19      let one_dyn: Dynamic = one.clone().into();
20
21      // Example constraints: f(0) = 1  and  f(1) = 0
22      solver.assert(&f.apply(&[&zero])._eq(&one_dyn));
23      solver.assert(&f.apply(&[&one])._eq(&zero_dyn));
24
```

```
    assert_eq!(solver.check(), z3::SatResult::Sat);

    let model = solver.get_model().unwrap();

    if let Some(x) = model.get_func_interp(&f) {
        let entries = x.get_entries();
        for entry in entries {
            println!("entry: {:?}", entry);
        }
        println!("Interpretation of f: {:?}", x);
    };
}
```

- ***RecFuncDecl***

  Purpose: Function declaration that can carry a (possibly recursive) definition inside the solver via `add_def`. Therefore using this type we can define functions that have a definition unlike `FuncDecl` which was used to define uninterpreted functions.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_func_decl`: raw `Z3_func_decl` handle.

  When to use

  - Encoding recursive definitions such as factorial, list length, or user-defined predicates.
  - Also usable for non-recursive definitions if you plan to give them a symbolic definition instead of adding assertions.

  How to use

```
use z3::RecFuncDecl;
use z3::{
    ast::{self, Ast, Dynamic},
    Config, Context, Solver, Sort, Symbol,
};

fn main() {
    let config = Config::new();
    let ctx = Context::new(&config);
    let solver = Solver::new(&ctx);
    let n_sort = Sort::int(&ctx);
    let fact_sym = Symbol::String("fact".into());
    let fact = RecFuncDecl::new(&ctx, fact_sym, &[&n_sort], &n_sort);

    let n = ast::Int::new_const(&ctx, "n");
    let one = ast::Int::from_i64(&ctx, 1);
    let cond = n.le(&one);
    let n_minus_1 = ast::Int::sub(&ctx, &[&n, &one]);
    let rec = fact.apply(&[&n_minus_1]).as_int().unwrap();
    let n_times_fact = ast::Int::mul(&ctx, &[&n, &rec]);
    let body = cond.ite(&one, &n_times_fact);

    fact.add_def(&[&n], &body); // attach definition
    let five = ast::Int::from_i64(&ctx, 5);
    let one_twenty = ast::Int::from_i64(&ctx, 120);
    let one_twenty_dynamic: Dynamic = one_twenty.clone().into();
    solver.assert(&fact.apply(&[&five])._eq(&one_twenty_dynamic));
    match solver.check() {
```

```
        z3::SatResult::Sat => {
            println!("The factorial of 5 is 120.");
        }
        z3::SatResult::Unsat => {
            println!("The assertion failed.");
        }
        z3::SatResult::Unknown => {
            println!("The result is unknown.");
        }
    }
}
```

Tips

- After `add_def`, the solver treats calls to the function like an unfoldable definition (no axiom explosion).

- ***DatatypeBuilder***

  Purpose: Builder pattern for declaring an algebraic data type (ADT) in Z3 with variants, constructors, testers, and field accessors. It is used to model structured data such as enums, records, recursive trees, or mutually recursive types.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `name`: `Symbol` for the datatype.
  - `constructors`: specification of each variant `(variant_name, [(field, DatatypeAccessor)])`.[6]

  When to use

  - Modelling Rust-style enums, structs, recursive trees, or mutually recursive types.

  How to use

```
use z3::{Config, Context, DatatypeAccessor, DatatypeBuilder, DatatypeSort, Sort};

fn main() {
    let config = Config::new();
    let ctx = Context::new(&config);

    // equivalent to:
    //     enum OptionInt {
    //     IntNone,
    //     IntSome { int_value: i32 },
    // }
    let option_int: DatatypeSort<'_> = DatatypeBuilder::new(&ctx, "OptionInt")
        .variant("IntNone", vec![])
        .variant(
            "IntSome",
            vec![("int_value", DatatypeAccessor::Sort(Sort::int(&ctx)))],
        )
        .finish(); // returns DatatypeSort

    let option_int_sort: Sort<'_> = option_int.sort;
    println!("OptionInt sort: {:?}", option_int_sort);

    let v: &Vec<z3::DatatypeVariant<'_>> = &option_int.variants;
```

[6] More precisely, $Vec < (String, Vec < (String, DatatypeAccessor) >) >$.

```rust
    println!("variants: {:?}", v);

    let c1: &z3::DatatypeVariant<'_> = &v[0];
    let c1_constructor = c1.constructor.apply(&[]);
    let c1_tester = &v[0].tester;
    let c1_accessor = &v[0].accessors;
    println!("constructor for IntNone: {:?}", c1_constructor);
    println!("tester for IntNone: {:?}", c1_tester);
    println!("accessors for IntNone: {:?}", c1_accessor);

    let c2 = &v[1];
    let c2_constructor = c2.constructor.apply(&[&z3::ast::Int::from_i64(&ctx, 42)]);
    let c2_tester = &v[1].tester;
    let c2_accessor = &v[1].accessors;
    println!("constructor for IntSome(42): {:?}", c2_constructor);
    println!("tester for IntSome: {:?}", c2_tester);
    println!("accessors for IntSome: {:?}", c2_accessor);

    let option_bool = DatatypeBuilder::new(&ctx, "OptionBool")
        .variant("BoolNone", vec![])
        .variant(
            "BoolSome",
            vec![("bool_value", DatatypeAccessor::Sort(Sort::bool(&ctx)))],
        )
        .finish(); // returns DatatypeSort

    let option_bool_sort = option_bool.sort;
    println!("OptionBool sort: {:?}", option_bool_sort);

    let w = &option_bool.variants;
    println!("variants: {:?}", w);

    let is_bool = &w[0].tester; // recogniser for OptionBool
    println!("is_bool recogniser: {:?}", is_bool);
}
```

Tips

- Call `finish()` exactly once; it registers the sort and yields a `DatatypeSort` with ready-made constructors/testers.
- Variants may be mutually recursive: use `DatatypeAccessor::Datatype(other_symbol)` for fields.

- ***DatatypeAccessor***

  Purpose: Describes the type of a field inside a variant — either a plain `Sort` or a reference to another (possibly recursive) datatype.

  Variants (public)

  - `Sort(Sort)` – field has this concrete sort.
  - `Datatype(Symbol)` – field is another ADT identified by symbol.

  When to use

  - While constructing a datatype with `DatatypeBuilder`.
  - To express recursive/self-referential fields.

- ***DatatypeVariant***

Purpose: Holds the Z3 artifacts for one variant of the ADT.

Fields: (public)

- `constructor`: `FuncDecl` to build the variant.
- `tester`: `FuncDecl` predicate *is* to check if the a value is of the respective type.
- `accessors`: field-projection `FuncDecl`s.

When to use

- Constructing values: `DatatypeSortType.variants[1].constructor.apply([&value])`.
- Pattern-matching: `DatatypeSortType.variants[0].tester.apply([&value])`.
- Accessing fields inside the variant: `DatatypeSortType.variants[1].accessors[0].apply([&value])`.

```rust
use z3::{
    ast::{self, Ast},
    Config, Context, DatatypeAccessor, DatatypeBuilder, DatatypeSort, Sort,
};
fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = z3::Solver::new(&ctx);

    // enum OptionInt { IntNone, IntSome { int_value: Int } }
    let option_int: DatatypeSort<'_> = DatatypeBuilder::new(&ctx, "OptionInt")
        .variant("IntNone", vec![])
        .variant(
            "IntSome",
            vec![("int_value", DatatypeAccessor::Sort(Sort::int(&ctx)))],
        )
        .finish(); // register the sort DatatypeSort.sort and .variants[...]

    // Create two constants of this sort
    let x = ast::Datatype::new_const(&ctx, "x", &option_int.sort);
    let y = ast::Datatype::new_const(&ctx, "y", &option_int.sort);

    // Assert x is None and y == Some(3)
    let none_tester = &option_int.variants[0].tester;                              //
    ↪  tester for IntNone
    solver.assert(&none_tester.apply(&[&x]).as_bool().unwrap());

    let some_ctor = &option_int.variants[1].constructor;                           //
    ↪  constructor for IntSome
    let three = ast::Int::from_i64(&ctx, 3);
    let some_three = some_ctor.apply(&[&three]);
    solver.assert(&y._eq(&some_three.as_datatype().unwrap()));

    assert_eq!(solver.check(), z3::SatResult::Sat);

    // Extract the value inside the Some variant
    let accessor = &option_int.variants[1].accessors[0];                           //
    ↪  the int_value accessor
    let extracted = accessor.apply(&[&y]).as_int().unwrap();
    let model = solver.get_model().unwrap();
    let v = model.eval(&extracted, true).unwrap().as_i64().unwrap();
    assert_eq!(v, 3);
}
```

- ***DatatypeSort***

  Purpose: Result of `DatatypeBuilder::finish()`; represents the actual ADT sort plus all its variants.

  Fields: (public)

  - `sort`: `Sort` object for typing terms.
  - `variants`: `Vec<DatatypeVariant>`.

```rust
use z3::{
    ast::{self, Ast},
    datatype_builder::create_datatypes,
    Config, Context, DatatypeAccessor, DatatypeBuilder, DatatypeSort, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    run_z3_example(&ctx);
}

fn run_z3_example(ctx: &Context) {
    let solver = z3::Solver::new(ctx);

    // Build: enum Tree { Leaf(i32), Node { left: Tree, right: Tree } }
    let tree_dt: DatatypeSort<'_> = DatatypeBuilder::new(ctx, "Tree")
        .variant(
            "Leaf",
            vec![("value", DatatypeAccessor::Sort(Sort::int(ctx)))],
        )
        .variant(
            "Node",
            vec![
                ("left", DatatypeAccessor::Datatype("Tree".into())),
                ("right", DatatypeAccessor::Datatype("Tree".into())),
            ],
        )
        .finish();

    // Constants
    let t1 = ast::Datatype::new_const(ctx, "t1", &tree_dt.sort);
    let t2 = ast::Datatype::new_const(ctx, "t2", &tree_dt.sort);

    // Construct t1 = Node(Leaf(1), Leaf(2))
    let leaf_ctor = &tree_dt.variants[0].constructor;
    let one = ast::Int::from_i64(ctx, 1);
    let two = ast::Int::from_i64(ctx, 2);
    let leaf1 = leaf_ctor.apply(&[&one]).as_datatype().unwrap();
    let leaf2 = leaf_ctor.apply(&[&two]).as_datatype().unwrap();

    let node_ctor = &tree_dt.variants[1].constructor;
    let node12 = node_ctor.apply(&[&leaf1, &leaf2]).as_datatype().unwrap();
    solver.assert(&t1._eq(&node12));

    // Assert t2 is a Leaf
    let is_leaf = &tree_dt.variants[0].tester;
    solver.assert(&is_leaf.apply(&[&t2]).as_bool().unwrap());

```

```
    assert_eq!(solver.check(), z3::SatResult::Sat);

    // Access fields of Node
    if let Some(m) = solver.get_model() {
        // t1 should be a Node
        let is_node = &tree_dt.variants[1].tester;
        assert!(m
            .eval(&is_node.apply(&[&t1]).as_bool().unwrap(), true)
            .unwrap()
            .as_bool()
            .unwrap());
    }

    // and mutually recursive datatypes
    let even_b = DatatypeBuilder::new(&ctx, "Even")
        .variant("Zero", vec![]) // Even::Zero
        .variant(
            // Even::ESucc { pred : Odd }
            "ESucc",
            vec![("pred", DatatypeAccessor::Datatype("Odd".into()))],
        );

    // Odd datatype
    let odd_b = DatatypeBuilder::new(&ctx, "Odd").variant(
        // Odd::OSucc { pred : Even }
        "OSucc",
        vec![("pred", DatatypeAccessor::Datatype("Even".into()))],
    );

    let [_even_sort, _odd_sort]: [DatatypeSort; 2] =
        create_datatypes(vec![even_b, odd_b]).try_into().unwrap();
}
```

Tips

- Use `constructor` to build values, `tester` to discriminate, and `accessors` to project fields.

- ***Params***

  Purpose: Key–value map of configurations used to configure many Z3 components (simplifiers, tactics, solvers, optimize, etc.)

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_params`: raw `Z3_params` handle.

  When to use

  - Fine-tune solver behaviour.
  - Pass custom parameters to a tactic chain.

  How to use

```
use z3::{Config, Context, Params, Tactic};

fn main() {
    let config = Config::new();
    let ctx = Context::new(&config);

    let mut p = Params::new(&ctx);
```

```
    p.set_u32("timeout", 2_000); // 2-second timeout
    p.set_bool("mbqi", false); // disable MBQI

    let tactics = Tactic::new(&ctx, "smt");
    let solver = tactics.solver();
    solver.set_params(&p);
}
```

Tips

- Call all `set_*()` *before* passing the params to a tactic/solver.

- ***SatResult***

  Purpose: Represents the three possible outcomes of a satisfiability check – `Solver::check()` / `Optimize::check()`. `Unknown` covers timeout, interrupt, or incomplete theory.

  Variants: (public) `Unsat` — `Unknown` — `Sat`

  How to use

```
use z3::{Config, Context, Params, SatResult, Tactic};

fn main() {
    let config = Config::new();
    let ctx = Context::new(&config);

    let mut p = Params::new(&ctx);
    p.set_u32("timeout", 2_000); // 2-second timeout
    p.set_bool("mbqi", false); // disable MBQI

    let tactics = Tactic::new(&ctx, "smt");
    let solver = tactics.solver();
    solver.set_params(&p);

    match solver.check() {
        SatResult::Sat => println!("Model = {:?}", solver.get_model().unwrap()),
        SatResult::Unsat => println!("Proof  = {:?}", solver.get_proof().unwrap()),
        SatResult::Unknown => println!("Why?   = {:?}",
        ↪  solver.get_reason_unknown().unwrap()),
    }
}
```

  Regarding `Optimize::check()` the following code snippet presents an example:

```
use z3::ast::Ast;
use z3::{ast::Int, Config, Context, Optimize, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let opt = Optimize::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    // 0 <= x <= 10, 0 <= y <= 10, x + y <= 11
    opt.assert(&x.ge(&Int::from_i64(&ctx, 0)));
    opt.assert(&x.le(&Int::from_i64(&ctx, 10)));
    opt.assert(&y.ge(&Int::from_i64(&ctx, 0)));
    opt.assert(&y.le(&Int::from_i64(&ctx, 10)));
    opt.assert(&Int::add(&ctx, &[&x, &y]).le(&Int::from_i64(&ctx, 11)));
```

```
18
19      // maximize x, maximize y
20      opt.maximize(&x);
21      opt.maximize(&y);
22
23      match opt.check(&[]) {
24          SatResult::Sat => {
25              let model = opt.get_model().unwrap();
26              let x_val = model.eval(&x, true).unwrap();
27              let y_val = model.eval(&y, true).unwrap();
28              println!("Optimal solution: x = {}, y = {}", x_val, y_val);
29          }
30          SatResult::Unsat => {
31              println!("No solution exists");
32          }
33          SatResult::Unknown => {
34              println!("Solver returned unknown: {:?}", opt.get_reason_unknown());
35          }
36      }
37  }
```

- ***Pattern***

  Purpose: Explicit trigger for quantifier instantiation; influences the heuristic used by Z3's E-matching engine.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_pattern`: raw `Z3_pattern` handle.

  When to use

  - Manually control which subterms trigger quantifier instantiation.
  - Improve performance by avoiding automatic (and sometimes suboptimal) pattern inference.

  How to use

```
1   use z3::{ast, ast::Ast, Config, Context, Pattern, Solver};
2
3   fn main() {
4       let config = Config::new();
5       let ctx = Context::new(&config);
6       let solver = Solver::new(&ctx);
7
8       let x = ast::Int::new_const(&ctx, "x");
9       let y = ast::Int::new_const(&ctx, "y");
10      let pat = Pattern::new(&ctx, &[&ast::Int::add(&ctx, &[&x, &y])]);
11
12      let forall = ast::forall_const(
13          &ctx,
14          &[&x, &y],
15          &[&pat],
16          &ast::Int::add(&ctx, &[&x, &y])._eq(&ast::Int::add(&ctx, &[&y, &x])),
17      );
18      solver.assert(&forall);
19  }
```

- ***ApplyResult***

Purpose: Captures the list of sub-goals produced by applying a `Tactic` to a `Goal`.

Fields: (private)

- `ctx`: owning `Context`.
- `z3_apply_result`: raw `Z3_apply_result` handle.

When to use

- Inspect intermediate goals after rewriting / simplification.
- Feed the resulting sub-goals to different tactics or solvers.

How to use

```rust
use z3::{
    ast::{Bool, Int},
    ApplyResult, Config, Context, Goal, Solver, Tactic,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    // (produce_models = true, produce_unsat_cores = true, produce_proofs = false)
    let goal = Goal::new(&ctx, true, true, false);

    // x + y > 10  AND  x < 0
    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let c1 = z3::ast::Int::add(&ctx, &[&x, &y]).gt(&Int::from_i64(&ctx, 10));
    let c2 = x.lt(&Int::from_i64(&ctx, 0));

    goal.assert(&c1);
    goal.assert(&c2);

    println!("=== Original Goal ===");
    for f in goal.get_formulas::<Bool>() {
        println!("  {:?}", &f);
    }

    // simplify (rewrites equations) then solve-eqs (solves simple equalities)
    let simplify = Tactic::new(&ctx, "simplify");
    let solve_eqs = Tactic::new(&ctx, "solve-eqs");

    // first simplify, then solve-eqs
    let pipeline = simplify.and_then(&solve_eqs);

    // apply the tactic pipeline to the goal
    let ar: ApplyResult<'_> = pipeline
        .apply(&goal, None)
        .expect("tactic application failed");

    // solve each sub-goal separately
    println!("\n=== Solving each sub-goal ===");
    for (i, subg) in ar.list_subgoals().enumerate() {
        println!(" Sub-goal {} formulas:", i);
        let solver = Solver::new(&ctx);
        for f in subg.get_formulas() {
            println!("{:?}", f);
            solver.assert(&f);
```

```
        }
        let res = solver.check();
        println!(" Sub-goal {} is {:?}", i, res);
    }
}
```

- ***Tactic***

  Purpose: Encapsulates a reasoning strategy. Tactics can be composed to form pipelines or converted into a solver.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_tactic`: raw `Z3_tactic` handle.

  When to use

  - Custom preprocessing before feeding a core solver.
  - Domain-specific strategies (bit-blasting, omega, nnf, ...).

  How to use

```
use std::time::Duration;
use z3::{Config, Context, Tactic};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let t1 = Tactic::new(&ctx, "simplify");
    let t2 = Tactic::new(&ctx, "smt");
    let combo = t1.and_then(&t2).try_for(Duration::from_secs(5)); // 5-second limit

    let _solver: z3::Solver<'_> = combo.solver(); // tactic-backed solver
}
```

  Tips

  - When you create a solver with tactics, the specified tactics are the *only* strategies used in checking the statisfiability of the formula. These tactics are used in order. Whereas if you created a solver with the *new* function, all default tactics would have been available.
  - Use *Tactic::list_all()* to see a list of all tactics and use *Tactic::new()* to create a single one.

- ***Goal***

  Purpose: Represents an (immutable) set of formulas that can be solved or transformed by tactics and solvers.[7]

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_goal`: raw `Z3_goal` handle.

  When to use

  - Break a large set of assertions into independent sub-goals.
  - Analyse goals with `Probe` before choosing a tactic.

[7] See the code example in the `ApplyResult` section for how `Goal` is used.

Tips

- Use `Goal::new(ctx, models, unsat_cores, proofs)` to create a new goal.
  - `models` – **true** tells Z3 to store a *model* whenever the goal (or the solver that consumes it) is `Sat`. Set to **false** if you never need models and want to save memory/time.
  - `unsat_cores` – **true** enables collection of an *unsat-core* (minimal subset of assertions causing `Unsat`) when the goal is unsatisfiable. Disable if you do not care about explanations.
  - `proofs` – **true** enables production of full formal *proof objects*. Proof generation can be expensive; keep **false** unless you need proof terms.

- ***Probe***

Purpose: A fast metric that inspects a goal (number of variables, is-linear, etc.). A Probe is essentially a little *sensor* you can stick on a Goal to ask Z3 for a quantitative property of your formulas. Think of it like asking, "How big is this goal?" or "How many bit-vector terms does it contain?" or "Does it mix arithmetic and arrays?"

Fields: (private)

- `ctx`: owning `Context`.
- `z3_probe`: raw `Z3_probe` handle.

When to use

- Choose between tactics; i.e., selecting a tactic based on goal-characteristics.
- Collect statistics for adaptive pipelines.

How to use

```rust
use z3::{ApplyResult, Config, Context, Goal, Probe, Tactic};

fn main() {
    let ctx = Context::new(&Config::new());
    let goal = Goal::new(&ctx, true, false, false);

    // Probe the size of the goal
    let size = Probe::new(&ctx, "size").apply(&goal);

    let size_tactic = Tactic::new(&ctx, "bit-blast");
    let base = Tactic::new(&ctx, "simplify");
    let pipeline = if size > 10.0 {
        base.and_then(&size_tactic)
    } else {
        base
    };

    let ar: ApplyResult = pipeline.apply(&goal, None).unwrap();

    for g in ar.list_subgoals() {
        println!("Subgoal: {}", g);
    }
}

```

Tips

- Write `Probe::list_all(&ctx)` to get a list of all the probes.

- ***Statistics***

  Purpose: Holds profiling data (counters (UInt) or timings (Double)) produced by a solver or tactic pipeline.

  Fields: (private)

  - `ctx`: owning `Context`.
  - `z3_stats`: raw `Z3_stats` handle.

  When to use

  - After `solver.check()` to see time spent in each module.
  - Compare tactic pipelines quantitatively.

  How to use

```rust
use z3::{ast::Int, Config, Context, SatResult, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    //  x + y > 10, x < 5
    solver.assert(&Int::add(&ctx, &[&x, &y]).gt(&Int::from_i64(&ctx, 10)));
    solver.assert(&x.lt(&Int::from_i64(&ctx, 5)));

    match solver.check() {
        SatResult::Sat => println!("SAT"),
        SatResult::Unsat => println!("UNSAT"),
        SatResult::Unknown => println!("UNKNOWN"),
    }

    // Print solver statistics
    let stats = solver.get_statistics();
    println!("\n=== Z3 Solver Statistics ===");
    for x in stats.entries() {
        println!("{:?}", x);
    }
}
```

  Tips

  - After running the above code, the following statistics will be provided in the terminal:

```
StatisticsEntry { key: "elim-unconstrained", value: UInt(5) }
StatisticsEntry { key: "solve-eqs-steps", value: UInt(2) }
StatisticsEntry { key: "solve-eqs-elim-vars", value: UInt(2) }
StatisticsEntry { key: "num allocs", value: UInt(29699) }
StatisticsEntry { key: "rlimit count", value: UInt(177) }
StatisticsEntry { key: "max memory", value: Double(17.76) }
StatisticsEntry { key: "memory", value: Double(17.76) }
StatisticsEntry { key: "time", value: Double(0.007) }
```

- The statistic `elim-unconstrained` (5) indicates that Z3 removed five variables deemed irrelevant to the final result; `solve-eqs-steps` (2) and `solve-eqs-elim-vars` (2) show it took two elimination steps to remove two variables via equation solving; `num allocs` (29706) tracks the total number of internal memory allocations; `rlimit count` (171) records how many times Z3 hit its resource-limit checks (used for timeouts and fairness); `max memory` and `memory` (17.74 MB) report peak and current memory usage; and `time` (0.004 s) is the total solve time in seconds. These entries help you diagnose performance bottlenecks and resource consumption in your queries.
- Depending on the logic and the query, the statistics can present different keys; e.g., *propagations, mk clause, etc.*

Before moving on to the individual modules of the *z3* crate, we will provide extra explanation of the core abstractions: `Goal`, `Tactic`, `Solver`, `Probe`, and `Statistics`.

- **Goal vs. Tactic vs. Solver**
  - **Goal** A *bag of formulas*. Think "workspace": you `assert` formulas into it, but it does not solve anything.
  - **Tactic** A *transformer* that consumes a goal and returns an `ApplyResult` (usually a list of simpler sub-goals). Examples: `"simplify"`, `"qe"`.
  - **Solver** A complete decision procedure that ultimately returns `Sat`/`Unsat`/`Unknown`. Internally a solver *contains* a tactic pipeline (either the Z3 default or one you supply via `tactic.solver()`).
  - **Relation**

    `Goal` $->$ `Tactic` $\Rightarrow$ new `Goal` ... $->$ finally fed to a `Solver`
- **Probe vs. Statistics**
  - **Probe** Query *before* solving. A probe inspects a goal and accordingly based on the data is used to drive tactic selection: `Tactic::when(p, heavy, light)`.
  - **Statistics** Collected *after* solving. A statistics object is a map of counters/timers. Useful for profiling and performance debugging.[8]

7 *version* module

This module has been extensively discussed in Section 5. To reiterate, the following public items are accessible:

- `Version`**:** This struct holds the four-part Z3 version. The first part is *major*, which is the first (leftmost) number, and represents significant changes that break backward compatibility or represent major architectural changes. The second part is equivalent to *minor* in *semantic versioning*, and represents significant enhancements that maintain backward compatibility. The third part is *Build number*, which represents a recompilation of the same source for different platforms or compilers. The last number is *Revision number*, often linked to fully interchangeable fixes to shipped versions. This is defined as follows:

```
pub struct Version {
    major: u32,
    minor: u32,
    build_number: u32,
    revision_number: u32, }
```

[8] **Mnemonic**: *Probe* = "peek first"; *Statistics* = "summary after".

- `pub fn version() -> Version`: Internally calls `Z3_get_version` via FFI (Z3 C API) and returns the Z3 `Version`. Specifically, it first creates a zero initialized version with `Version::default()`. Then, passes the four field addresses to the raw C API `Z3_get_version` and returns the now-populated version.
- `pub fn full_version() -> &'static str`: Internally calls `Z3_get_full_version()` and returns the full version string as `&'static str`.

```rust
fn main() {
    use z3::version;
    use z3::full_version;

    let ver = version();
    println!("Z3 version: {:?}", ver);

    let ver2 = full_version();
    println!("Z3 full version: {:?}", ver2);
}
```

The previous code gives the following output on my computer:

```
Z3 version: Version { major: 4, minor: 14, build_number: 1, revision_number: 0 }
Z3 full version: "4.14.1.0"
```

8 *tactic* module

The following public items are accessible:

- `impl ApplyResult`:
  - `pub fn list_subgoals(self) -> impl Iterator<Item = Goal>`: Iterate over the sub-goals produced by the application of a tactic on a *goal*. The following code snippet produces multiple sub-goals after applying the tactic on the original goal:

```rust
use z3::{
    ast::{Bool, Int},
    Config, Context, Goal, Tactic,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let z = Int::new_const(&ctx, "z");
    let one = Int::from_i64(&ctx, 1);
    let zero = Int::from_i64(&ctx, 0);

    // (x > 1) \/ (y > 1) \/ (z > 1)
    let big_or = Bool::or(&ctx, &[&x.gt(&one), &y.gt(&one), &z.gt(&one)]);

    // (x < 0) \/ (y < 0)
    let other_or = Bool::or(&ctx, &[&x.lt(&zero), &y.lt(&zero)]);

    // Goal
    let g = Goal::new(
        &ctx, /*models=*/ false, /*unsat_core=*/ false, /*proofs=*/ false,
    );
    g.assert(&big_or);
    g.assert(&other_or);
```

```
28
29      // simplify then split the disjunctions
30      let tactic = Tactic::new(&ctx, "simplify").and_then(&Tactic::new(&ctx,
        ↪  "split-clause"));
31
32      // apply the tactic to the goal to get the apply result sub-goals
33      let result = tactic.apply(&g, None).unwrap();
34
35      // Print every sub-goal produced
36      for (i, sg) in result.list_subgoals().enumerate() {
37          println!("Sub-goal {}:\n{}\n", i, sg);
38      }
39  }
```

- `impl Tactic`: The `tactic` API lets you programmatically compose Z3 *tactics*.[9]

  Tactics vs. Logic: In short, **Logics** are *declarative*; i.e., they specify what theories to use whereas **Tactics** are *procedural*; i.e., they specify how to solve problems.

  Logic: A *declaration of the theory fragment* you intend to use. For example, `QF_BV` (quantifier-free bit-vectors), `QF_LIA` (quantifier-free linear integer arithmetic), `AUFLIRA` (Arrays + Uninterpreted Functions + Linear Real Arithmetic), etc. Choosing a logic tells Z3 *which sorts, functions, and decision procedures* are allowed and will enable specialized back-ends.

  Tactic: A *programmable strategy* that rewrites or solves goals step by step. Examples include `simplify`, `bit-blast`, `qe`, `solve-eqs`, or whole pipelines like `qfbv`. They let you *control exactly how Z3 transforms and dispatches* your formulas to its core solver. Without a logic declaration, Z3 falls back to a general-purpose *smt* tactic.

  – `pub fn list_all(ctx: &Context) -> impl Iterator<Item = Result<&str, Utf8Error>>`[10]: Enumerate every tactic name known to Z3.

```
1  fn main() {
2      use z3::{Config, Context, Tactic};
3
4      let ctx = Context::new(&Config::new());
5      for name in Tactic::list_all(&ctx).flatten() {
6          println!("{name}");
7      }
8  }
```

  A more detailed explanation about each of the tactics can be found on the *Z3* guide.

  – `pub fn new(ctx: &Context, name: &str) -> Tactic`: Create a tactic by its name (e.g. `"smt"`, `"solve-eqs"`). Please look at the example at the end of the previous page. Note that the string must be a valid tactic name.

  – `pub fn create_skip(ctx: &Context) -> Tactic`: A tactic that returns the goal unchanged. This no-op may sound useless, but it is essential whenever building compound strategies. Combinators such as `or_else`, `cond`, or `if` need a safe fall-back that leaves the goal intact when a probe fails or a branch condition is false.

  – `pub fn create_fail(ctx: &Context) -> Tactic`: A tactic that always fails. It gives a way to deliberately abort a pipeline when a probe or condition is not met. Specifically, because tactics are combined, sometimes we need to deliberately *short-circuit* the pipeline.

[9] Algorithmic recipes for preprocessing or solving goals.

[10] Note that these signatures are meant as a detailed overview and may not exactly match the function definitions; they may omit lifetime annotations and other details.

```rust
fn main() {
    use z3::{ApplyResult, Config, Context, Goal, Probe, Tactic};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let g = Goal::new(&ctx, true, false, false);

    let p = Probe::new(&ctx, "size");
    let p_le_10 = p.le(&Probe::constant(&ctx, 10.0));

    let simplify_heavy = Tactic::new(&ctx, "ctx-solver-simplify");

    // if size <= 10 then run heavy simplifier, else skip
    let pipeline = Tactic::cond(&ctx, &p_le_10, &simplify_heavy,
    ↪  &Tactic::create_skip(&ctx));

    let ar: ApplyResult = pipeline.apply(&g, None).unwrap();
    for sub in ar.list_subgoals() {
        println!("Sub-goal: {:?}", sub);
    }
}
```

- `pub fn repeat(ctx: &Context, t: &Tactic, max: u32) -> Tactic`: Keep applying tactic `t` until fix-point (until the goal is not modified anymore) or *max* iterations has been reached.
- `pub fn try_for(&self, timeout: Duration) -> Tactic`: Wrap self with a wall-clock timeout. Abort `tactic` if it doesn't terminate within the period specified by `timeout`.
- `pub fn and_then(&self, then_tactic: &Tactic) -> Tactic`: Sequential composition - first `self`, then `then_tactic` is applied on each sub-goal produced by the application of the original tactic on the original goal.
- `pub fn or_else(&self, else_t: &Tactic) -> Tactic`: Fallback - use `self` on the given goal, only on failure apply `else_t` on the given goal.
- `pub fn probe_or_else(&self, p: &Probe, t: &Tactic) -> Tactic`: Probe-guarded choice between `self` and `t`. If probe `p` holds on the goal, use `self`, otherwise `t`.
- `pub fn when(&self, p: &Probe) -> Tactic`: Run `self` only if `p` is true, else behave like `skip`.
- `pub fn cond(ctx: &Context, p: &Probe, t1: &Tactic, t2: &Tactic) -> Tactic`: Probe-based if-then-else between two tactics. *if `p` then `t1` else `t2`.*
- `pub fn fail_if(ctx: &Context, p: &Probe) -> Tactic`: Produce a failing tactic when `p` is false.
- `pub fn apply(&self, goal: &Goal , params: Option<&Params>) -> Result<ApplyResult, String>`: Run the tactic on a goal (with optional parameters). If the tactic succeeds, returns *Ok(ApplyResult)*.
- `pub fn solver(&self) -> Solver`: Convert the tactic into a solver that allows adding assertions and checking their satisfiability. Internally a solver contains a tactic pipeline as its processing strategy.

```rust
use std::time::Duration;
use z3::{
    ast::{Ast, Bool, Int},
    Config, Context, Goal, Params, Probe, Tactic,
};

fn main() {
    let cfg = Config::new();
```

```rust
    let ctx = Context::new(&cfg);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    for i in Tactic::list_all(&ctx) {
        println!("Tactic: {:?}", i);
    }
    // Build a goal and assert constraints
    let goal = Goal::new(&ctx, false, false, false);
    goal.assert(&Int::add(&ctx, &[&x, &y])._eq(&Int::from_i64(&ctx, 10))); // x + y = 10
    goal.assert(&x.ge(&Int::from_i64(&ctx, 0))); // x >= 0
    goal.assert(&y.ge(&Int::from_i64(&ctx, 0))); // y >= 0

    // tactics
    let simp = Tactic::new(&ctx, "simplify");
    let solve_eqs = Tactic::new(&ctx, "solve-eqs");

    // try simplification for 100 ms
    let simp_timeout = simp.try_for(Duration::from_millis(100));

    // repeat simplification up to 5 times or until fix-point
    let simp_repeat = Tactic::repeat(&ctx, &simp_timeout, 5);

    // sequentially simplify then solve equations
    let simplify_then_solve = simp_repeat.and_then(&solve_eqs);

    // fallback: if that pipeline fails, just use the default SMT tactic
    let default_smt = Tactic::new(&ctx, "smt");
    let pipeline = simplify_then_solve.or_else(&default_smt);

    // check if the goal size is larger than a threshold
    let size = Probe::new(&ctx, "size");
    let threshold = Probe::constant(&ctx, 2.0);
    let size_large = size.gt(&threshold);
    // if size_large holds, use pipeline; otherwise, skip (no-op)
    let smart_tactic = pipeline.when(&size_large);

    // if the goal is not a linear real arithmetic problem, use a different tactic
    let non_linear = Probe::new(&ctx, "is-lra");
    let nlsat = Tactic::new(&ctx, "nlsat");
    let cond_tactic = Tactic::cond(&ctx, &non_linear, &nlsat, &smart_tactic);

    // fail if already solved (size == 0), to demonstrate fail_if
    let size_zero = Probe::new(&ctx, "size").eq(&Probe::constant(&ctx, 0.0));
    let guarded_tactic = Tactic::fail_if(&ctx, &size_zero).and_then(&cond_tactic);

    // apply the composed tactic to our goal
    let result = guarded_tactic
        .apply(&goal, Some(&Params::new(&ctx)))
        .expect("Tactic application failed");

    let all_formulas: Vec<Bool> = result
        .list_subgoals()
        .flat_map(|goal| {
            let formulas: Vec<Bool> = goal.get_formulas().iter().cloned().collect();
            formulas.into_iter()
        })
        .collect();
```

```rust

    // Convert the tactic into a solver and check satisfiability
    let solver = guarded_tactic.solver();
    solver.assert(&Bool::and(&ctx, &all_formulas));
    match solver.check() {
        z3::SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!(
                "sat; model: x = {}, y = {}",
                model.eval(&x, true).unwrap(),
                model.eval(&y, true).unwrap(),
            );
        }
        z3::SatResult::Unsat => println!("unsat"),
        z3::SatResult::Unknown => println!("unknown"),
    }
}
```

9 *symbol* module

Z3 uses a unified `Symbol` type to represent identifiers for functions, sorts, constants, and datatypes. Internally, a symbol may be an integer (e.g. `Symbol::Int(0)`) or a string (e.g. `Symbol::String("x")`). The following methods are accessible:

- `pub fn as_z3_symbol(&self, ctx: &Context) -> Z3_symbol`: This method is called internally whenever you invoke a Z3 API function that needs a symbol argument. It turns your Rust-side Symbol into a Z3-internal symbol that the Z3 C API expects.

```rust
use z3::{ast::Int, Config, Context, FuncDecl, Solver, Sort, Symbol};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let sym_id: Symbol = 42u32.into(); // -> Symbol::Int(42)
    let sym_f = Symbol::from("f"); // -> Symbol::String("f")
    let _sym_x = Symbol::from(String::from("x")); // -> Symbol::String("x")

    let raw_f = sym_f.as_z3_symbol(&ctx); // Z3_mk_string_symbol
    let raw_id = sym_id.as_z3_symbol(&ctx); // Z3_mk_int_symbol
    println!("Raw symbols: {raw_f:?}, {raw_id:?}");

    let int = Sort::int(&ctx);
    let bool = Sort::bool(&ctx);

    // FuncDecl::new(ctx, name, domain_sorts, &range_sort)
    let f_decl = FuncDecl::new(&ctx, sym_f.clone(), &[&int, &int], &bool);

    let x = Int::new_const(&ctx, "x"); // uses a string literal
    let y = Int::new_const(&ctx, "y");

    let fxy = f_decl.apply(&[&x, &y]).as_bool().unwrap();

    let forty_two = Int::from_i64(&ctx, 42);
    let f42y = f_decl.apply(&[&forty_two, &y]).as_bool().unwrap();

    let solver = Solver::new(&ctx);
    solver.assert(&fxy); // assert  f(x, y)
    solver.assert(&f42y.not()); // assert ¬f(42, y)

```

```
    println!("sat? {:?}", solver.check());
    if solver.check() == z3::SatResult::Sat {
        let model = solver.get_model().unwrap();
        println!("Model: {model:?}");
    } else {
        println!("No solution found.");
    }
}
```

- The following `From` implementations are provided:
  - `From<u32>` → `Symbol::Int`
  - `From<String>` → `Symbol::String`
  - `From<&str>` → `Symbol::String`

10 *statistics* module

The `Statistics`, `StatisticsValue`, and `StatisticsEntry` types have been extensively discussed before, specifically in section 4. The `Statistics` holds profiling data produced by a solver or tactic pipeline. The following are additional items not covered in the previous sections:

- `impl Statistics`
  - `pub fn value(&self, key: &str) -> Option<StatisticsValue>` Look up the statistic named `key`, returning its value if present.
  - `pub fn entries(&self) -> impl Iterator<Item = StatisticsEntry>` Iterate over every (key, value) pair in the statistics set.

```
fn main() {
    use z3::{Config, Context, Solver, StatisticsValue};

    let ctx = Context::new(&Config::new());
    let solver = Solver::new(&ctx);
    let stats = solver.get_statistics();
    if let Some(StatisticsValue::Double(t)) = stats.value("memory") {
        println!("Solver uses {:.2} memory", t);
    }
    for entry in stats.entries() {
        println!("{}: {:?}", entry.key, entry.value);
    }
}
```

11 *sort* module

`Sort` is the Z3 *type* object. The different kind of Z3 types has been extensively covered in section 2. The following represent the methods provided by the high-level Rust Z3 API for working with sorts:

- `pub fn get_z3_sort(&self) -> Z3_sort`: Exposes the raw C `Z3_sort` handle.
- `pub fn uninterpreted(ctx: &Context, name: Symbol) -> Sort`: Build a user-defined sort such as *type parameters & generics*. For full details, please visit section 2.
- `pub fn bool(ctx: &Context) -> Sort`: Boolean sort.
- `pub fn int(ctx: &Context) -> Sort`: Mathematical integers.
- `pub fn real(ctx: &Context) -> Sort`: Mathematical real numbers.
- `pub fn float(ctx: &Context, ebits: u32, sbits: u32) -> Sort`: IEEE-754 generic float (`ebits` exponent, `sbits` significand).

- `pub fn float32(ctx: &Context) -> Sort`: 32-bit IEEE float (ebits = 8 & sbits = 24).
- `pub fn double(ctx: &Context) -> Sort`: 64-bit IEEE double (ebits = 11 & sbits = 53).
- `pub fn string(ctx: &Context) -> Sort`: Z3's sequence/Unicode string sort.
- `pub fn bitvector(ctx: &Context, sz: u32) -> Sort`: Fixed-width bit-vector of n bits.
- `pub fn array(ctx: &Context, domain: &Sort, range: &Sort) -> Sort` : *domain* $\rightarrow$ *range* array sort.
- `pub fn set(ctx: &Context, elt: &Sort) -> Sort`: Finite set (*array-like*) with Boolean range.
- `pub fn seq(ctx: &Context, elt: &Sort) -> Sort`: Sequence (*ordered collections*) of type `elt`.
- `pub fn enumeration( ctx: &Context, name: Symbol, enum_names: &[Symbol], ) -> (Sort, Vec<FuncDecl>, Vec<FuncDecl>)`: Creates an enum along with its associated constructors and testers. This function is fundamental for modeling discrete, finite sets of values where each value is distinct and mutually exclusive. It receives a symbol representing the name of the enumeration sort itself and an array of symbols representing the names of individual variants as inputs. The outputs are the enum type, constructor function declarations, and testers.

```rust
use z3::{
    ast::{Ast, Dynamic},
    Config, Context, SatResult, Solver, Sort, Symbol,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let (color_sort, constructors, testers) = Sort::enumeration(
        &ctx,
        Symbol::from("color"),
        &[
            Symbol::from("Red"),
            Symbol::from("Green"),
            Symbol::from("Blue"),
        ],
    );

    let red_constructor = &constructors[0]; // Creates Red values
    let green_constructor = &constructors[1]; // Creates Green values
    let blue_constructor = &constructors[2]; // Creates Blue values

    let is_red = &testers[0];
    let is_green = &testers[1];
    let is_blue = &testers[2];

    let red_value = red_constructor.apply(&[]);
    let green_value = green_constructor.apply(&[]);
    let blue_value = blue_constructor.apply(&[]);

    let x = Dynamic::new_const(&ctx, "x", &color_sort);

    let solver = Solver::new(&ctx);
    solver.assert(&x._eq(&red_value));

    match solver.check() {
```

```
        SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!("x = {:?}", model.eval(&x, true));
            let is_red = is_red.apply(&[&x]);
            println!("x is Red: {:?}", model.eval(&is_red, true));
        }
        _ => {
            println!("No solution found");
        }
    }
}
```

- `pub fn kind(&self) -> SortKind`: Return the low-level Z3 C API type.
- `pub fn float_exponent_size(&self) -> Option<u32>`: Returns the exponent width wrapped inside `Some` if this is a float sort and if not, *None*.
- `pub fn float_significand_size(&self) -> Option<u32>`: Returns the significand width wrapped inside `Some` if this is float sort and if not, *None*.
- `pub fn is_array(&self) -> bool`: `True` for `Array` and `Set` sorts and `false` otherwise.
- `pub fn array_domain(&self) -> Option<Sort>`: Returns the domain sort of an array/set.
- `pub fn array_range(&self) -> Option<Sort>`: Returns the range sort of an array/set (`Bool` is always returned for sets).

SortDiffers Utility: The `SortDiffers` struct is used to represent and report a sort mismatch error when two `Sort` instances cannot be meaningfully compared or combined. This typically arises in typed AST construction when an operation (e.g., equality, function application, or conditional branches) involves terms of incompatible sorts. It holds two fields: `left` and `right`, both of type `Sort`, and implements `Display` to format a clear error message. For example, comparing an `Int` and a `Bool` will produce a `SortDiffers` error: "`Cannot compare nodes, Sort does not match. Nodes contain types Int and Bool`." This type is useful for debugging and enforcing sort correctness in higher-level Z3 abstractions.

- `impl SortDiffers`:
  - `pub fn new(left: Sort, right: Sort) -> Self`: Package two mismatched sorts into an error.
  - `left(&self) -> &Sort`, `right(&self) -> &Sort`: Access the offending sorts for diagnostics.

```
fn main() {
    use z3::{Config, Context, Sort, SortKind};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let s = Sort::array(&ctx, &Sort::int(&ctx), &Sort::bool(&ctx));

    assert_eq!(s.kind(), SortKind::Array);
    assert!(s.is_array());
    assert_eq!(s.array_domain(), Some(Sort::int(&ctx)));
    assert_eq!(s.array_range(), Some(Sort::bool(&ctx)));

    let fp = Sort::double(&ctx);
    assert_eq!(fp.float_exponent_size(), Some(11));
    assert_eq!(fp.float_significand_size(), Some(53));

    match assert_equal(&s, &fp) {
        Ok(_) => println!("Sorts are equal"),
        Err(e) => println!("Sorts differ: {:?}", e),
```

```
    };
}

use z3::{Sort, SortDiffers};
fn assert_equal<'a>(a: &'a Sort, b: &'a Sort) -> Result<(), SortDiffers<'a>> {
    if a != b {
        return Err(SortDiffers::new(a.clone(), b.clone()));
    }
    Ok(())
}
```

Tips on numeric sorts in Z3

- Integer-like vs. Bit-Vector vs. Real vs. Float
  - `Int` — the sort of *mathematical integers* $(\ldots, -2, -1, 0, 1, 2, \ldots)$. Operations follow pure number-theory rules: addition never overflows and division is exact.
  - `BitVector n` — an n-bit *machine word*. Arithmetic wraps modulo 2^n exactly like CPU registers; you can also extract individual bits or slices. Use this sort for low-level purposes for example when modeling hardware.
  - `Real` — the sort of *rationals*. Represents mathematical real numbers with exact algebraic precision, allowing you to write constraints that reason about fractions, square-roots, and polynomial roots without the rounding errors of floating-point arithmetic – i.e., Z3 internally treats *reals* as exact rational values (like 1/3 stays as 1/3, not 0.333).
  - `Float(e,s)` — an *IEEE-754 floating point* value with a fixed number of exponent bits (e) and significand bits (s). Arithmetic follows IEEE rounding rules and can distinguish +0 from –0, `NaN`, `Inf`, etc.
- `float` vs `float32` vs `double`
  - `float(e,s)` — generic constructor; you choose any exponent–significand width.
  - `float32` — shorthand for the 32-bit IEEE single precision format ($e = 8$ bits, $s = 24$ bits).
  - `double` — shorthand for 64-bit IEEE double precision format ($e = 11$, $s = 53$).
- Exponent vs. Significand (IEEE-754) A floating-point number is stored as

$$(\text{sign}) \times 2^{\text{exponent}} \times (1.\textit{significand bits})$$

  - **Exponent bits** (e) decide *how large or small* the number can be (range of magnitudes).
  - **Significand bits** (s) decide *how many significant digits* you can keep (precision of number).

  In short, more exponent bits ⇒ wider range; more significand bits ⇒ finer precision.

12 *solver* module

This module provides the associated functions to the `Solver` struct data type defined in the *lib.rs* module:

- `pub fn new(ctx: &Context) -> Solver:` Create Z3's *combined* solver. When you call `Solver::new(&ctx)` Z3 actually builds *two* engines under the hood; an *incremental* and *non-incremental* solver. The default solver is the *non-incremental* one; i.e., `solver1`. The moment you start working incrementally—calling `Solver::push()`/`Solver::pop()` or asserting new constraints after a satisfiability check with `Solver::assert()` or `Solver::assert_and_track()` without an intervening `Solver::reset()`—Z3 transparently switches to `solver2`, a fully incremental SMT engine. You can tweak this behaviour via the parameters `solver2_timeout`, `solver2_unknown`, and `ignore_solver1`. More often than not, using the *new* function is how you want to build a *solver*, unless you want a solver with custom tactic pipeline.

What is *non-incremental*? You just call `assert()` and `check()` once without modifying the solver again. `solver1` chooses a tactic based on the logic of your assertions. If it matches a known logic, Z3 picks a specialized tactic (e.g., `qflia`, `qfbv`, etc.) Otherwise, it runs the general tactic chain: `(and-then simplify smt)`. This means: first simplify, then run the general SMT solver.

What is *incremental*? If you use `push()`, `pop()`, or call `assert()/assert_and_track` again after a check without a `reset`, the solver upgrades to `solver2`, which behaves like a stateful SMT solver with support for backtracking. It always uses the general-purpose `smt` tactic and maintains the assertion stack. The solver cannot go back to the non-incremental version.

One-shot solving (uses `solver1`)

```
fn main() {
    use z3::{ast::Int, Config, Context, SatResult, Solver};
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    solver.assert(&x.gt(&Int::from_i64(&ctx, 10)));
    solver.assert(&x.lt(&Int::from_i64(&ctx, 20))); // not an assertion after the check
    assert_eq!(solver.check(), SatResult::Sat);
}
```

Incremental solving (switches to `solver2`)

```
fn main() {
    use z3::{ast::Ast, ast::Int, Config, Context, Solver};
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let five = Int::from_i64(&ctx, 5);
    let ten = Int::from_i64(&ctx, 10);
    solver.push();
    solver.assert(&x._eq(&y));
    solver.assert(&x.gt(&ten));
    solver.assert(&y.lt(&five));
    match solver.check() {
        z3::SatResult::Sat => println!("x == y"),
        z3::SatResult::Unsat => println!("x != y"),
        z3::SatResult::Unknown => println!("unknown"),
    }
    solver.pop(1); // pop everything after the last push before (until line 11)
    match solver.check() {
        z3::SatResult::Sat => {
            println!("x == y");
            let model = solver.get_model().unwrap();
            println!("x = {}", model.eval(&x, true).unwrap());
            println!("y = {}", model.eval(&y, true).unwrap());
        }
        z3::SatResult::Unsat => println!("x != y"),
        z3::SatResult::Unknown => println!("unknown"),
    }
}
```

Use `Params` to override default behaviour

```rust
fn main() {
    use z3::{Config, Context, Params, SatResult, Solver};
    let ctx = Context::new(&Config::new());
    let solver = Solver::new(&ctx);

    let mut params = Params::new(&ctx);

    params.set_bool("ignore_solver1", true); // always use solver2 (force incremental engine)
    params.set_u32("solver2_timeout", 5000); // 5 second timeout

    // solver2_unknown:
    //         0: If the check returns "Unknown" with solver2, simply return "Unknown" without attempting fallback
    //         1: Default behavior - If the check returns "Unknown" with solver2, retry with solver1 only if the problem is quantifier-free
    //         2: If the check returns "Unknown" with solver2, always retry with solver1 regardless of whether the problem contains quantifiers
    params.set_u32("solver2_unknown", 2);
    solver.set_params(&params);
    // continue ... with your logic

    match solver.check() {
        SatResult::Sat => {
            println!("Satisfiable");
        }
        SatResult::Unsat => {
            println!("Unsatisfiable");
        }
        SatResult::Unknown => {
            println!("Unknown");
        }
    }
}
```

You can skip all this and just create a solver from a tactic:

```rust
fn main() {
    use z3::{Config, Context, Tactic};
    let ctx = Context::new(&Config::new());
    let t: Tactic<'_> = Tactic::new(&ctx, "qflia");
    let _s: z3::Solver<'_> = t.solver();
    // continue ... with your logic
}
```

- `pub fn from_string<T: Into<Vec<u8>>>(&self, source_string: T)`: This method parses a raw SMT-LIB2 string and directly adds all resulting assertions, soft constraints, and optimization goals to the current solver instance. Use this when you already have SMT formulas in SMT-LIB2 format (as strings) and want to quickly inject them into a solver. This is especially helpful for testing the behavior of Z3 on known benchmarks or formulas generated by other tools. Note that all variables and declarations inside the string are local to Z3. They are not linked to any `z3::ast` objects in Rust.

```rust
fn main() {
    use z3::{Config, Context, Solver};
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);
```

```
    let smt = r#"
        (declare-const x Int)
        (declare-const y Int)
        (assert (> x y))
        (assert (> y 0))
        (maximize x)
    "#;

    solver.from_string(smt);
    solver.from_string("(declare-sort A)");
    solver.from_string("(declare-const z A)"); // 'A' is still available
    solver.from_string("(assert (= z z))"); // 'z' is still available

    println!(":{:?}", solver.check());
    println!(":{:?}", solver.get_assertions());
}
```

- `pub fn new_for_logic<S: Into<Symbol>>(ctx: &Context, logic: S) -> Option<Solver>`: This function creates a new solver that is explicitly specialized for a particular SMT-LIB logic, such as `"QF_LIA"` (quantifier-free linear integer arithmetic), `"QF_BV"` (quantifier-free bitvectors), or `"AUFLIA"` (arrays and uninterpreted functions with linear integer arithmetic). If the logic string is invalid or unsupported by Z3, the function returns `None`. *Why use this?* Using `Solver::new_for_logic` bypasses Z3's internal logic auto-detection (used in `Solver::new`). This gives you full control over which logical fragment is being solved, and can result in faster performance. Use `new_for_logic()` when you're solving formulas that fall into a well-known logical fragment.

```
fn main() {
    use z3::{ast::Ast, ast::Int, Config, Context, SatResult, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new_for_logic(&ctx, "QF_LIA").expect("logic not supported");

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    solver.assert(&x._eq(&y));
    match solver.check() {
        SatResult::Sat => println!("Satisfiable"),
        SatResult::Unsat => println!("Unsatisfiable"),
        SatResult::Unknown => println!("Unknown"),
    }
}
```

Tips:

  - The logic string must match exactly what Z3 expects — case sensitive, e.g., `"QF_LIA"`, not `"qf_lia"`.
  - This avoids the combined solver behavior (i.e., it does not switch between incremental/non-incremental under the hood). The tactic-based approach and logic-specialized solver uses the *same tactic throughout its lifetime*, without switching between different solving strategies based on usage patterns.

- `pub fn translate(&self, dest: &Context) -> Solver`: This copies the full solver state (with its assertions) into another `Context`.

```
fn main() {
```

```
    use z3::{ast::Ast, ast::Int, Config, Context, Solver};
    let ctx = Context::new(&Config::new());
    let solver = Solver::new_for_logic(&ctx, "QF_LIA").expect("logic not supported");

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    solver.assert(&x._eq(&y));

    let other_ctx = Context::new(&Config::new());
    let solver2 = solver.translate(&other_ctx);

    println!(":{:?}", solver2.get_assertions());
}
```

- `pub fn get_context(&self) -> &Context:` This method simply returns a reference to the `Context` that the solver was created with. Since all Z3 values (sorts, variables, expressions, etc.) are tied to a specific context, this is useful when you need to construct new expressions in the same context that a solver operates in. Sometimes you only have access to the solver (e.g., passed as a parameter), and you want to create a new variable or sort. Instead of passing the context separately, you can just call `get_context()` to retrieve it.

```
use z3::ast::Int;
use z3::Solver;
fn add_constraint_to_solver(solver: &Solver) {
    let ctx = solver.get_context();
    let x = Int::new_const(ctx, "x");
    solver.assert(&x.gt(&Int::from_i64(ctx, 0)));
}

fn main() {
    let ctx = z3::Context::new(&z3::Config::new());
    let solver = Solver::new(&ctx);

    add_constraint_to_solver(&solver);

    match solver.check() {
        z3::SatResult::Sat => println!("Satisfiable"),
        z3::SatResult::Unsat => println!("Unsatisfiable"),
        z3::SatResult::Unknown => println!("Unknown"),
    }
}
```

- `pub fn assert(&self, ast: &ast::Bool):` This method adds a hard constraint (i.e., an assertion that must hold) into the solver. All constraints you assert are logically **ANDed together**. Once you have asserted all relevant formulas, you can use `check()` or `check_assumptions()` to determine whether the entire set is satisfiable or not. Each call to `assert()` adds a new formula to the solver's internal goal. All assertions are accumulated unless you `reset()` or use `push()/pop()` for backtracking.

```
fn main() {
    use z3::{ast::Bool, Config, Context, SatResult, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let mut solver = Solver::new(&ctx);

    solver.assert(&Bool::from_bool(&ctx, true)); // TRUE is OK
    solver += &Bool::from_bool(&ctx, false); // now conflicting
    solver += Bool::fresh_const(&ctx, "p"); // another unknown prop
```

```
    assert_eq!(solver.check(), SatResult::Unsat); // system is inconsistent
    solver.get_assertions()
        .iter()
        .for_each(|a| println!("assertion: {}", a));
}
```

Tips:

- If you want to label assertions for unsat-core extraction, use `assert_and_track()` instead.
- You can retrieve all current assertions with `get_assertions()`.
- There is no way to remove a specific assertion — use `push()/pop()` or `reset()` instead.

- `pub fn assert_and_track(&self, ast: &ast::Bool, p: &ast::Bool)`: This method *asserts* and *labels* the constraint with a Boolean constant p for unsat-core extraction. Adds a Boolean constraint to the solver, just like `assert()`, but attaches a fresh `label` so that Z3 can return it later if the formula causes unsatisfiability. This is used to extract **unsat cores**. When Z3 says the problem is `Unsat`, the *unsat core* is the minimal subset of the asserted constraints that is already inconsistent. It helps explain *why* the problem has no solution. *Why use labels?* You can only get the unsat core if you explicitly label your assertions using `assert_and_track`. The returned core will be a list of those labels.

```
fn main() {
    use z3::{ast::Bool, Config, Context, SatResult, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let p = Bool::from_bool(&ctx, true);
    let q = Bool::from_bool(&ctx, false);
    let lp = Bool::fresh_const(&ctx, "p_label");
    let lq = Bool::fresh_const(&ctx, "q_label");

    solver.assert_and_track(&p, &lp);
    solver.assert_and_track(&q, &lq);

    assert_eq!(solver.check(), SatResult::Unsat);

    // Extract which labels caused the conflict
    let core = solver.get_unsat_core();
    for l in core {
        println!("Conflict came from: {}", l);
    }
}
```

- `pub fn reset(&self)`: `solver.reset()` removes all assertions.
- `pub fn push(&self)`: Push a *backtracking* point. Create backtracking frames for incremental solving. Please refer to the example related to incremental solving.[12]
- `pub fn pop(&self, n: u32)`: Pop `n` *backtracking* levels. Remove backtracking frames for incremental solving.

```
solver.push(); solver.assert(&phi1);
solver.push(); solver.assert(&phi2);
solver.pop(2); // both phi1 and phi2 removed
```

- `pub fn check(&self) -> SatResult`: This method asks Z3: **"Are all the current assertions consistent?"** It returns one of three possible results:
  - `SatResult::Sat` — the formula is satisfiable (there exists a model).
  - `SatResult::Unsat` — no model exists that satisfies all assertions (there is a proof).
  - `SatResult::Unknown` — Z3 cannot determine satisfiability (e.g., from incomplete theory support, resource limit, timeout, unsupported logic, etc.).

  Z3 takes all formulas added via `assert()` or `assert_and_track()` and tries to solve them as a conjunction. If the formula is satisfiable, you can retrieve the model using `get_model()` (if model generation is enabled in the config). If the formula is unsatisfiable and proof generation was enabled, you can retrieve a proof via `get_proof()`.

  Proof Generation:

```rust
fn main() {
    use z3::{ast::Int, Config, Context, SatResult, Solver};

    let mut cfg = Config::new();
    cfg.set_proof_generation(true); // Enable proof generation
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    solver.assert(&x.gt(&y)); // x > y
    solver.assert(&y.gt(&x)); // y > x (conflict)

    let result = solver.check();
    assert_eq!(result, SatResult::Unsat);
    println!("Solver result: {:?}", result);
    println!("proof: {:?}", solver.get_proof().unwrap());
}
```

  Model Generation:

```rust
fn main() {
    use z3::{ast::Int, Config, Context, SatResult, Solver};

    let mut cfg = Config::new();
    cfg.set_model_generation(true); // Enable model generation
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    solver.assert(&x.gt(&y)); // x > y

    let result = solver.check();
    assert_eq!(result, SatResult::Sat);
    println!("Solver result: {:?}", result);
    println!("proof: {:?}", solver.get_model().unwrap());
}
```

  What if result is ***Unknown****?* This means Z3 couldn't solve it. After `SatResult::Unknown`, use `get_reason_unknown()` for a human-readable explanation to inspect why. Also, a model may still

be returned on `Unknown` using `get_model()`, but it is not guaranteed to satisfy assertions.

Tips

- Use `get_model()` only if the result is `Sat` or `Unknown`. In case it is *unknown*, `get_model()` will either not produce a model or it will produce a model, under which the satisfiability of the assertions is not guaranteed.
- Use `get_proof()` only if the result is `Unsat` and proof generation was enabled.
- Use `check_assumptions()` if you want to solve under temporary literals.

- `pub fn check_assumptions(&self, assumptions: &[ast::Bool]) -> SatResult:` Solve under one-shot assumptions. This method works like `check()`, but temporarily assumes that the given Boolean literals are true **only during this call**. These are not added permanently to the solver — they exist just for the duration of this check. This is useful when you want to:
  - test the satisfiability of your current assertions *under certain assumptions*,
  - selectively enable constraints via assumptions,
  - extract **unsat cores** through trial & error when the problem becomes inconsistent.

```rust
fn main() {
    use z3::{ast::Bool, Config, Context, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let a = Bool::fresh_const(&ctx, "a");
    let b = Bool::fresh_const(&ctx, "b");

    solver.assert(&a.not()); // a = false
    let result = solver.check_assumptions(&[a, b]); // assume a = true (conflict)
    println!("Result: {:?}", result);
}
```

Here, the solver would normally be satisfiable (`a = false`), but we temporarily assumed `a = true`, which contradicts the assertion.

Comparison to `assert_and_track()`:

- `assert_and_track()` permanently adds a constraint and labels it for core tracking.
- `check_assumptions()` keeps the assumption temporary.
- You can mix both in the same solver — Z3 will combine them in the core.

- `pub fn get_assertions(&self) -> Vec<Bool>:` Return the current list of asserted formulas. This does *not* include assumptions from `check_assumptions()`.

```rust
fn main() {
    use z3::{ast::Bool, Config, Context, Params, Solver};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let mut p = Params::new(&ctx);
    p.set_u32("timeout", 10_000);
    solver.set_params(&p);
```

```
11
12      let a = Bool::fresh_const(&ctx, "a");
13      let b = Bool::fresh_const(&ctx, "b");
14      let x = Bool::fresh_const(&ctx, "x");
15
16      solver.assert(&a.not()); // a = false
17      solver.assert_and_track(&b, &x); // track b
18      let _result = solver.check_assumptions(&[a]); // assume a = true (conflict)
19
20      for i in solver.get_assertions() {
21          println!("Assertion: {}", i);
22      }
23  }
```

The snippet prints the following lines:

```
Assertion: (not a!0)
Assertion: (=> x!2 b!1)
```

- `pub fn get_unsat_core(&self) -> Vec<Bool>`: Returns a subset of assumptions after the last call to `check_assumptions()` or a sequence of calls to `assert_and_track` followed by `check()`. These are the assumptions Z3 used in the unsatisfiability proof.[11] This method returns a subset of the *tracked* literals and/or one-shot assumptions that together are already inconsistent. Each element of the vector is a `Bool` constant that you previously supplied as a label in `assert_and_track()`, or as an assumption in `check_assumptions()`.

```
1   fn main() {
2       use z3::{ast::Bool, Config, Context, Solver};
3
4       let cfg = Config::new();
5       let ctx = Context::new(&cfg);
6       let solver = Solver::new(&ctx);
7
8       let a = Bool::fresh_const(&ctx, "a");
9       let b = Bool::fresh_const(&ctx, "b");
10      let x = Bool::fresh_const(&ctx, "x");
11      let y = Bool::fresh_const(&ctx, "y");
12      let z = Bool::fresh_const(&ctx, "z");
13
14      solver.assert_and_track(&a.not(), &x); // a = false
15      solver.assert_and_track(&b.not(), &y); // b = false
16      solver.assert_and_track(&Bool::or(&ctx, &[&a, &b]), &z); // a or b must be true
17
18      let c = solver.check();
19      println!("Check result: {:?}", c);
20
21      let unsat_core = solver.get_unsat_core();
22      println!("Unsat core: {:?}", unsat_core);
23  }
```

A typical run prints:

```
Check result : Unsat
Unsat core   : [x!2, y!3, z!4]
```

How it works. Z3 proves the contradiction internally and then asks which of the *labels* (`x`, `y`, `z`) it needed for the proof. Those labels are returned as the unsatisfiable core. If you omit a label (e.g.

[11] By default, unsat cores are not minimized. To enable minimization, set `sat.core.minimize` or `smt.core.minimize` to `true` (for SAT or SMT respectively). This increases solving cost.

you used plain `assert()`), that clause can never appear in the core—even if it is the actual cause of the inconsistency.

Tips

- Always call `check()` **before** querying the core; otherwise the vector is empty.
- Use either `assert_and_track()` or `check_assumptions()` to create *every* constraint you want to diagnose; un-labelled `assert()`s are invisible to the core.
- By default, Z3 does not guarantee that the unsat core it returns is unique (or even minimal).

- `pub fn get_consequences( &self, assumptions: &[ast::Bool], variables: &[ast::Bool], ) -> Vec<ast::Bool>:` Under given assumptions, return all literals about `vars` that are logically implied. With the current *assertions* plus a temporary list of Boolean *assumptions*, compute every literal over the given propositional `vars` that is *logically implied*. Each literal is returned as a `Bool` constant, possibly negated internally (e.g. $\neg b$).

```
fn main() {
    use z3::{ast::Bool, Config, Context, Solver};

    let cfg    = Config::new();
    let ctx    = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let a = Bool::fresh_const(&ctx, "a");
    let b = Bool::fresh_const(&ctx, "b");
    let c = Bool::fresh_const(&ctx, "c");

    solver.assert(&a.implies(&b));   // a -> b
    solver.assert(&b.implies(&c));   // b -> c

    let consequences = solver.get_consequences(&[a.clone()], &[b.clone(), c.clone()]);

    println!("Consequences under a = true:");
    for lit in consequences {
        println!("  {}", lit);       // prints: b, c
    }
}
```

Internally Z3 searches for all literals over the provided variables that follow from `assertions` $\wedge$ `assumptions`. The result is *complete*: any Boolean combination over `vars` that is entailed will be reported in the output.

Tips

- Variables and assumptions must be `Bool` *constants*, not arbitrary formulas.
- The solver state is *not* modified; you can call `get_consequences()` repeatedly with different sets.

- `pub fn get_model(&self) -> Option<Model>:` Obtain a satisfying assignment when the last `Solver::check()` or `Solver::check_assumptions()` returned `Sat` or `Unknown`. Nevertheless, in the case of `Unknown`, Z3 does not ensure that calls to `get_model()` succeed and any models produced in this case are not guaranteed to satisfy the assertions. Note that on the configuration level, *model generation* is enabled by default and calling `cfg.set_model_generation(false)` is intended to override this.[12] However, if the model generation is set to false using `cfg.set_model_generation(false)`, when the solver is built (`Solver::new(&ctx)`), the model

[12]Thus, there is no need to set `cfg.set_model_generation(true)` explicitly for model generation.

generation value resets to `true` for the corresponding solver. The *configuration* setting therefore does not have a lasting effect on the models, while it remains in force for the proofs on the *solver* level. This is because models are considered cheap and very useful and proofs carry a significant memory/time cost. `get_model()` invokes an error handler if `check()` / `check_assumptions()` have not been called, or if the last result was `Unsat`.

```
1  fn main() {
2      use z3::{ast::Bool, Params, Config, Context, Solver};
3
4      let mut cfg = Config::new();
5      // cfg.set_model_generation(false);
6      let ctx = Context::new(&cfg);
7      let solver = Solver::new(&ctx);
8
9      // let mut p = Params::new(&ctx);
10     // p.set_bool("model", false); // solver-local override
11     // solver.set_params(&p);
12
13     let a = Bool::fresh_const(&ctx, "a");
14     solver.assert(&a.not()); // a = false
15
16     match solver.check() {
17         z3::SatResult::Sat => {
18             println!("SAT");
19             let model = solver.get_model().unwrap();
20             println!("Model: {}", model);
21         }
22         z3::SatResult::Unsat => {
23             println!("UNSAT");
24             let proof = solver.get_proof().unwrap();
25             println!("Proof: {:?}", proof);
26         }
27         z3::SatResult::Unknown => {
28             println!("UNKNOWN");
29         }
30     }
31 }
```

The above code returns the following:

```
SAT
Model: a!0 -> false
```

If we un-comment line 5 or lines 9-11 - or both - the output will be as following:

```
SAT
Model:
```

This is what we mean by *" if model generation is disabled at either the configuration level or the solver level, then the table of constant and function interpretations will not be populated."* Lastly the following is an example where the solver parameter overrides the configuration parameter:

```
1  fn main() {
2      use z3::{ast::Bool, Params, Config, Context, Solver};
3
4      let mut cfg = Config::new();
5      cfg.set_model_generation(false);
6      let ctx = Context::new(&cfg);
7      let solver = Solver::new(&ctx);
8
9      let mut p = Params::new(&ctx);
```

```
    p.set_bool("model", true); // solver-local override
    solver.set_params(&p);

    let a = Bool::fresh_const(&ctx, "a");
    solver.assert(&a.not()); // a = false

    match solver.check() {
        z3::SatResult::Sat => {
            println!("SAT");
            let model = solver.get_model().unwrap();
            println!("Model: {}", model);
        }
        z3::SatResult::Unsat => {
            println!("UNSAT");
            let proof = solver.get_proof().unwrap();
            println!("Proof: {:?}", proof);
        }
        z3::SatResult::Unknown => {
            println!("UNKNOWN");
        }
    }
}
```

The above code returns the following:

```
SAT
Model: a!0 -> false
```

- `pub fn get_proof(&self) -> Option<impl Ast>:` Get a proof object for the last `Solver::check()` or `Solver::check_assumptions()` when the result was `Unsat` and proofs were enabled using `cfg.set_proof_generation(true)`. Note that proof generation is disabled by default at the configuration level, and the solver always inherits this setting. Unlike model generation, the proof option cannot be enabled at the solver level—it must be set to `true` in the configuration *before* creating the context. `get_proof()` invokes the error handler if proof generation was not enabled using `cfg.set_proof_generation(true)`, or if `check()` / `check_assumptions()` have not been called, or if the last result was not `Unsat`.

```
fn main() {
    use z3::{ast::Bool, Config, Context, Solver};

    let mut cfg = Config::new();
    cfg.set_proof_generation(true); // commenting this line will disable proof generation
    ↪ and will give an error if you try to get a proof

    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let a = Bool::fresh_const(&ctx, "a");

    solver.assert(&a.not()); // a = false
    solver.assert(&a); // a = true

    match solver.check() {
        z3::SatResult::Sat => {
            println!("SAT");
            let model = solver.get_model().unwrap();
            println!("Model: {}", model);
        }
        z3::SatResult::Unsat => {
```

```
            println!("UNSAT");
            let proof = solver.get_proof().unwrap();
            println!("Proof: {:?}", proof);
        }
        z3::SatResult::Unknown => {
            println!("UNKNOWN");
            let unknown = solver.get_reason_unknown().unwrap();
            println!("Reason unknown: {}", unknown);
        }
    }
}
```

- `pub fn get_reason_unknown(&self) -> Option<String>`: Brief textual reason when the solver answered `Unknown` for the most recent `check()/check_assumptions()`.
- `pub fn set_params(&self, params: &Params)`: Update solver-level parameters. Note that when the parameter is set on the configuration level it affects all the child solvers. Also note that some parameters like setting the proof generation are *configuration-level* and must be done before creating the context.
- `pub fn get_statistics(&self) -> Statistics`: Returns statistics from the most recent `check`.
- `pub fn to_smt2(&self) -> String`: Dump the current solver state as SMT-LIB2 text for inspection or replay.
- **Trait impls**
  - `AddAssign<&Bool>` and `AddAssign<Bool>`: Enables the use of the shorthand notation `solver += &constraint` for `solver.assert(&constraint)`.

13 *rec_func_decl* module

This module provides the associated functions for the `RecFuncDecl` data type defined in the *lib.rs* module.

- `pub fn new<S: Into<Symbol>>( ctx: &Context, name: S, domain: &[&Sort], range: &Sort, ) -> Self`: This function creates a fresh `RecFuncDecl` in `ctx` with the given `name`, argument `domain`, and `range` sort.

```
let mut f = RecFuncDecl::new(
    &ctx,
    "fact",               // function name
    &[&Sort::int(&ctx)],  // domain: a single Int
    &Sort::int(&ctx)      // range: Int
);
// Now `f` exists in Z3; no body yet.
// Next step: call `add_def()` to give it a potentially recursive equation.
```

- `pub fn add_def(&self, args: &[&dyn ast::Ast], body: &dyn Ast)`: Adds the body to a recursive function. Note that *args* should have the types corresponding to the *domain* of the RecFuncDecl. For more information please refer to section 6, specifically to the part devoted to recursive functions.

14 *probe* module

A probe is a tiny function that inspects a given Goal (i.e. a set of formulas) and returns a double (a 64-bit float). A probe never mutates a goal; feel free to call it multiple times. They are indispensable for writing *tactics that adapt to the structure of a goal at runtime.* If you think of a Goal as *some formulas I want to solve,* a probe asks a question like, *How many OR nodes are in this goal?* or *Are all top-level assertions pseudo-Boolean constraints?*. Z3 by default has dozens of built-in probes. When a probe returns a non-zero double, we treat that as "true"; when it returns 0.0, we treat it as "false." But a probe

can also return arbitrary positive reals (e.g. the actual count of something).

- `pub fn list_all(ctx:&Context) -> impl Iterator<Item = Result<&str, Utf8Error>>`: Iterate over every probe name compiled into the current Z3 build.

```
fn main() {
    use z3::{Config, Context, Probe};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let probes: Vec<_> = Probe::list_all(&ctx).filter_map(|r| r.ok()).collect();
    for p in probes {
        println!("Probe: {}", p);
    }
}
```

- `pub fn describe(ctx: &Context, name: &str) -> Result<&str, Utf8Error>`: Fetch the human-readable documentation string attached to a probe.

```
fn main() {
    use z3::{Config, Context, Probe};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    for p in Probe::list_all(&ctx).filter_map(|r| r.ok()) {
        match Probe::describe(&ctx, p) {
            Ok(desc) => println!("{}: {}", p, desc),
            Err(e) => println!("{}: <bad UTF-8 or invalid name>", p),
        }
    }
}
```

The code prints the following:

```
ackr-bound-probe: A probe to give an upper bound of Ackermann congruence lemmas that a formula might generate.
is-unbounded: true if the goal contains integer/real constants that do not have lower/upper bounds.
is-pb: true if the goal is a pseudo-boolean problem.
arith-max-deg: max polynomial total degree of an arithmetic atom.
arith-avg-deg: avg polynomial total degree of an arithmetic atom.
arith-max-bw: max coefficient bit width.
arith-avg-bw: avg coefficient bit width.
is-qflia: true if the goal is in QF_LIA.
is-qfauflia: true if the goal is in QF_AUFLIA.
is-qflra: true if the goal is in QF_LRA.
is-qflira: true if the goal is in QF_LIRA.
is-ilp: true if the goal is ILP.
is-qfnia: true if the goal is in QF_NIA (quantifier-free nonlinear integer arithmetic).
is-qfnra: true if the goal is in QF_NRA (quantifier-free nonlinear real arithmetic).
is-nia: true if the goal is in NIA (nonlinear integer arithmetic, formula may have quantifiers).
is-nra: true if the goal is in NRA (nonlinear real arithmetic, formula may have quantifiers).
is-nira: true if the goal is in NIRA (nonlinear integer and real arithmetic, formula may have quantifiers).
is-lia: true if the goal is in LIA (linear integer arithmetic, formula may have quantifiers).
is-lra: true if the goal is in LRA (linear real arithmetic, formula may have quantifiers).
```

```
is-lira: true if the goal is in LIRA (linear integer and real arithmetic, formula may have quantifiers).
is-qfufnra: true if the goal is QF_UFNRA (quantifier-free nonlinear real arithmetic with other theories).
is-qfbv-eq: true if the goal is in a fragment of QF_BV which uses only =, extract, concat.
is-qffp: true if the goal is in QF_FP (floats).
is-qffpbv: true if the goal is in QF_FPBV (floats+bit-vectors).
is-qffplra: true if the goal is in QF_FPLRA.
memory: amount of used memory in megabytes.
depth: depth of the input goal.
size: number of assertions in the given goal.
num-exprs: number of expressions/terms in the given goal.
num-consts: number of non Boolean constants in the given goal.
num-bool-consts: number of Boolean constants in the given goal.
num-arith-consts: number of arithmetic constants in the given goal.
num-bv-consts: number of bit-vector constants in the given goal.
produce-proofs: true if proof generation is enabled for the given goal.
produce-model: true if model generation is enabled for the given goal.
produce-unsat-cores: true if unsat-core generation is enabled for the given goal.
has-quantifiers: true if the goal contains quantifiers.
has-patterns: true if the goal contains quantifiers with patterns.
is-propositional: true if the goal is in propositional logic.
is-qfbv: true if the goal is in QF_BV.
is-qfaufbv: true if the goal is in QF_AUFBV.
is-quasi-pb: true if the goal is quasi-pb.
```

- `pub fn new(ctx: &Context, name: &str) -> Probe`: Lookup an existing probe by its literal `name`. Call `Probe::new` only with names obtained from `Probe::list_all(&ctx)` to avoid invalid-name panics. This does not create a new probe; it merely returns a handle to the built-in probe.
- `pub fn apply(&self, goal: &Goal) -> f64`: Execute the probe on a given `Goal` (a set of Z3 formulas). Use `apply` when you want to inspect a `Goal` and obtain its probe-based property. It returns a `f64` result:
  - If the probe is Boolean (e.g. `"is-pb"`), nonzero means `true`, zero means `false`.
  - If the probe returns a numeric measure (e.g. `"num-clauses"`), that integer (as a float) is returned.

  Tips

  - Combine with `Tactic::cond` to branch a tactic pipeline.
- `pub fn constant(ctx: &Context, val: f64) -> Probe`: Create a probe that always evaluates to the given floating-point `val`. Useful as "literals" in probe arithmetic.

  Tips

  - Use `constant` to build composite probes in combination with relational or logical combinators (e.g. `le`, `and`, etc.), for example:

  ```
  let small_probe = Probe::new(&ctx, "num-clauses")
      .le(&Probe::constant(&ctx, 10.0));
  ```

  - Create constant true and false probes. For example:

  ```
  let always_true  = Probe::constant(&ctx, 1.0);
  let always_false = Probe::constant(&ctx, 0.0);
  ```

- **Relational probes: `lt`, `gt`, `le`, `ge`, `eq`, `ne`:** Produce composite probes that compare the numeric results of two probes. All relational operators return a *Boolean* probe; i.e., any non-zero value is interpreted as *true* and a zero value is interpreted as *false*.

  Signatures

```
pub fn lt(&self,  p: Probe)    -> Probe
pub fn gt(&self,  p: &Probe)   -> Probe
pub fn le(&self,  p: &Probe)   -> Probe
pub fn ge(&self,  p: &Probe)   -> Probe
pub fn eq(&self,  p: &Probe)   -> Probe
pub fn ne(&self,  p: &Probe)   -> Probe
```

- **Logical combinators: `and`, `or`, `not`:** Boolean connectives lifted to probes with the conventional semantics.

  Signatures

```
pub fn and(&self, p: &Probe) -> Probe
pub fn or (&self, p: &Probe) -> Probe
pub fn not(&self)            -> Probe
```

Example

```
use z3::ast::Int;

fn main() {
    use z3::{ast::Ast, Config, Context, Goal, Probe, Tactic};

    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    // x + y = 10 and x > 0
    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let goal = Goal::new(&ctx, true, false, false);
    goal.assert(&Int::add(&ctx, &[&x, &y])._eq(&Int::from_i64(&ctx, 10)));
    goal.assert(&x.gt(&Int::from_i64(&ctx, 0)));

    // check if the goal has <= 2 assertions
    let size = Probe::new(&ctx, "size");
    let small = size.le(&Probe::constant(&ctx, 2.0));

    let heavy = Tactic::new(&ctx, "ctx-solver-simplify");
    let skip = Tactic::create_skip(&ctx);
    let pipeline = Tactic::cond(&ctx, &small, &heavy, &skip);

    let result = pipeline.apply(&goal, None).unwrap();
    for (i, sub) in result.list_subgoals().enumerate() {
        println!("Sub-goal {}: {}", i, sub);
    }
}
```

15 *pattern* module

In Z3, a *pattern* (also called a *trigger*) is a syntactic cue that tells the instantiation engine *when* to fire a quantified formula. If a pattern is not provided for a quantifier, then Z3 will automatically compute a set of patterns for it. However, for optimal performance, the user should provide the patterns. Whenever

Z3 encounters a ground term that **matches** the pattern, it instantiates the quantifier with the matching substitution. Good patterns accelerate solving; bad patterns can do nothing or explode the search space.

- **Single–pattern** = a non-empty list with one term. A single match of that term triggers instantiation.
- **Multi-pattern** = a list with $n > 1$ terms. *All* terms must match simultaneously for the trigger to fire.
- `pub fn new(ctx: &Context, terms: &[&dyn Ast]) -> Pattern`: The returned `Pattern` can then be supplied to `ast::forall_const` or `ast::exists_const`. Note that *terms* must be non-empty.[13]

Tips

- Use constructor–shaped terms rather than arbitrary arithmetic to avoid unification loops.
- **Coverage:** Every bound variable that actually occurs in the body *must* also occur in at least one pattern—otherwise Z3 rejects the quantifier as ill-formed.

Example 1 — single-pattern trigger

```rust
use z3::{
    ast::{forall_const, Ast, Int},
    Config, Context, Pattern,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let x = Int::new_const(&ctx, "x");
    let zero = Int::from_i64(&ctx, 0);
    let lhs = Int::add(&ctx, &[&x, &zero]); // x + 0
    let pat = Pattern::new(&ctx, &[&lhs]); // single-pattern

    let forall = forall_const(
        &ctx,
        &[&x],
        &[&pat],
        &lhs._eq(&x), // body: x + 0 = x
    );

    println!("{}", forall);
}
```

When to use. Whenever an axiom has a "canonical" left–hand side (here $x + 0$), firing on that term is usually enough to prove equalities quickly.

Example 2 — multi-pattern trigger

```rust
use z3::{
    ast::{forall_const, Ast, Int},
    Config, Context, FuncDecl, Pattern, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
```

[13] If empty, the `assertion failed: !terms.is_empty()` error will be invoked.

```
    let int = Sort::int(&ctx);
    let f = FuncDecl::new(&ctx, "f", &[&int], &int);

    // f(x) = f(y) -> x = y
    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let fx = f.apply(&[&x]).as_int().unwrap();
    let fy = f.apply(&[&y]).as_int().unwrap();

    // multi-pattern ⟨f(x), f(y)⟩
    let pat = Pattern::new(&ctx, &[&fx, &fy]);

    let lemma = forall_const(&ctx, &[&x, &y], &[&pat], &fx._eq(&fy).implies(&x._eq(&y)));
    println!("{}", lemma);
}
```

Why multi-pattern? We only want the lemma fired when two *distinct* applications of f appear together; the conjunction $\langle f(x), f(y)\rangle$ prevents useless self-matches like $f(t)$ with itself. In general, a pattern that matches too often (e.g. the variable alone) can blow up search exponentially - *over-general patterns*. Multi-patterns tame this by requiring simultaneous matches. But be aware that if your pattern rarely matches, the quantifier never fires and proofs fail - *over-specific patterns*.

16 *params* module

This module provides the related functions for the `Params` type defined in the *lib.rs* module. A `Params` object is essentially a map from parameter names (`Symbol`) to values (`Symbol`, `bool`, `f64`, or `u32`). The parameters in a `Params` object can then be passed to tactics or solvers to tune Z3's behavior.

- `pub fn new(ctx: &Context) -> Params`: Construct an empty parameter set.
- `pub fn set_symbol<K: Into<Symbol>, V: Into<Symbol>>(&mut self, k: K, v: V)`: Associate the parameter named `k` with the symbol value `v`.
- `pub fn set_bool<K: Into<Symbol>>(&mut self, k: K, v: bool)`: Associate the parameter named `k` with the Boolean value `v`.
- `pub fn set_f64<K: Into<Symbol>>(&mut self, k: K, v: f64)`: Associate the parameter named `k` with the double value `v`.
- `pub fn set_u32<K: Into<Symbol>>(&mut self, k: K, v: u32)`: Associate the parameter named `k` with the unsigned integer value `v`.

```
use z3::{Config, Context, Solver, Params, ast::Int, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let mut params = Params::new(&ctx);
    params.set_u32("timeout", 1000);
    params.set_bool("auto_config", false);
    params.set_f64("relevancy", 0.7);
    params.set_u32("sat.random_seed", 42);

    let solver = Solver::new(&ctx);
    solver.set_params(&params);

    // 10 < x < 20
    let x = Int::new_const(&ctx, "x");
```

```
        solver.assert(&x.gt(&Int::from_i64(&ctx, 10)));
        solver.assert(&x.lt(&Int::from_i64(&ctx, 20)));

        match solver.check() {
            SatResult::Sat => {
                println!("SAT");
                println!("Model: {}", solver.get_model().unwrap());
            }
            SatResult::Unsat => println!("UNSAT"),
            SatResult::Unknown => println!("UNKNOWN"),
        }
    }
```

- `pub fn get_global_param(k: &str) -> Option<String>`: Retrieve the current value of the global or module parameter named `k`. Returns `Some(value)` if the parameter exists, else `None`.
- `pub fn set_global_param(k: &str, v: &str)`: Set a global or module-level parameter `k` to the string value `v` across all Z3 contexts.
- `pub fn reset_all_global_params()`: Restore all global and module parameters to their default values. Does not affect already created contexts or objects.

```
use z3::{
    Config, Context, Params, Solver, {get_global_param, reset_all_global_params,
    ↪  set_global_param},
};

fn main() {
    if let Some(old) = get_global_param("timeout") {
        println!("Global timeout before: {}", old);
    } else {
        println!("Global timeout not set yet.");
    }

    set_global_param("timeout", "500");
    println!(
        "Global timeout now: {}",
        get_global_param("timeout").unwrap()
    );

    reset_all_global_params();
    println!(
        "After reset, global timeout: {:?}",
        get_global_param("timeout")
    );

    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let mut local_params = Params::new(&ctx);
    local_params.set_u32("timeout", 250);
    local_params.set_bool("auto_config", false);
    println!(
        "global timeout: {:?}",
        get_global_param("timeout")
    );
    solver.set_params(&local_params);

    let a = z3::ast::Bool::fresh_const(&ctx, "a");
```

```
    solver.assert(&a);
    assert_eq!(solver.check(), z3::SatResult::Sat);
}
```

17 *context* module

The `Context` is the foundational object in Z3's Rust API. It encapsulates the Z3 logical context (Z3_context) and manages the lifetime of all AST nodes, solvers, and other Z3 objects created within it. Note that objects from different contexts cannot be mixed.

- `pub fn new(cfg: &Config) -> Context`: Create a new Z3 context with the provided configuration.
- `pub fn get_z3_context(&self) -> Z3_context`: Expose the raw C `Z3_context` handle for low-level operations or interfacing with Z3's C API directly.
- `pub fn interrupt(&self)`: Interrupt a solver performing a satisfiability test, a tactic processing a goal, or simplify functions. This method delegates to `ContextHandle::interrupt()`.
- `pub fn handle(&self) -> ContextHandle`: Obtain a handle that can be used to interrupt computation from another thread. The returned `ContextHandle` is both `Send` and `Sync`, allowing safe cross-thread interruption.
- `pub fn update_param_value(&mut self, k: &str, v: &str)`: Update a context-local parameter `k` to the string value `v`. This affects only the current context.
- `pub fn update_bool_param_value(&mut self, k: &str, v: bool)`: Helper function to update a parameter `k` with the Boolean value `v`.

```
use std::thread;
use std::time::Duration;
use z3::{Config, Context, SatResult, Solver};

fn main() {
    let cfg = Config::new();
    let mut ctx = Context::new(&cfg);
    ctx.update_param_value("timeout", "5000");
    ctx.update_bool_param_value("auto_config", false);

    let raw_ctx = ctx.get_z3_context();
    println!("Raw context handle: {:p}", raw_ctx);

    let solver = Solver::new(&ctx);
    let a = z3::ast::Bool::fresh_const(&ctx, "a");
    solver.assert(&a);

    let handle = ctx.handle();
    thread::scope(|s| {
        s.spawn(|| {
            std::thread::sleep(Duration::from_millis(0));
            handle.interrupt();
            println!("Computation interrupted!");
        });

        // do the actual check
        match solver.check() {
            SatResult::Sat => println!("SAT"),
            SatResult::Unsat => println!("UNSAT"),
            SatResult::Unknown => println!("UNKNOWN or interrupted"),
        }
```

```
32      });
33  }
```

- `pub fn interrupt(&self)`: This method for `ContextHandle`, interrupts any ongoing computation in the associated context. This method is thread-safe.

18 *optimize* module

The `Optimize` API turns Z3 from a mere satisfiability checker into a *Max-SMT / Optimization engine*. In other words, instead of asking merely *is the formula satisfiable?*, an optimizer lets you add *objectives* to minimize, maximize, or soft-satisfy. Then the optimizer returns a model that meets the hard constraints **and** is optimal w.r.t. those objectives. Accordingly, *soft constraints* receive rational weights so you can trade them off, and multiple objectives are solved lexicographically by default; i.e., earlier objectives are given a higher priority. [14]

- `impl Optimize`:
  - `pub fn new(ctx: &Context) -> Optimize`: Construct an empty optimizer attached to `ctx`. Equivalent to `sat/smt` solvers but with optimization support.
  - `pub fn from_string(&self, smtlib2: impl Into<Vec<u8>>)`: Parse an SMT-LIB2 string and add the parsed constraints and objectives to the optimizer. This method parses a raw `SMT-LIB2` string and directly adds all resulting assertions, soft constraints, and optimization goals to the current optimizer instance. Use this when you already have SMT formulas in SMT-LIB2 format (as strings) and want to quickly inject them into an optimizer. This is especially helpful for testing the behavior of Z3 on known benchmarks or formulas generated by other tools. Note that all variables and declarations inside the string are local to Z3. They are not linked to any `z3::ast` objects in Rust.
  - `pub fn get_context(&self) -> &Context`: This method simply returns a reference to the `Context` that the optimizer was created with. Since all Z3 values (sorts, variables, expressions, etc.) are tied to a specific context, this is useful when you need to construct new expressions in the same context that a optimizer operates in. Sometimes you only have access to the optimizer (e.g., passed as a parameter), and you want to create a new variable or sort. Instead of passing the context separately, you can just call `get_context()` to retrieve it.
  - `pub fn assert(&self, f: &impl Ast)`: Add a *hard* constraint. Violating it makes the problem `Unsat`.
  - `pub fn assert_and_track(&self, f: &Bool, p: &Bool)`: Like `assert` but annotates the clause with a fresh propositional variable `p` so you can later extract an unsat core.
  - `pub fn assert_soft(&self, f: &impl Ast, weight: impl Weight, group: Option<Symbol>)`: Add a *soft* constraint with weight. Weight is a positive, rational penalty for violating the constraint. On `check`, Z3 minimises the *total penalty* of violated soft clauses.
  - `pub fn maximize(&self, term: &impl Ast)` / `minimize`: Optimise a numeric term (maximize or minimize). Only `Int, Real,` or `BitVec` are allowed to be optimized; Booleans must be turned into 0/1 integers.
  - `pub fn push(&self)` / `pop(&self)`: Create and roll back back-tracking scopes, exactly like in `Solver`. Note that the number of calls to `pop` cannot exceed the number of calls to `push`.
  - `pub fn check(&self, assumptions: &[Bool]) -> SatResult`: Checks for satisfiability and produces optimal values.

[14] Many of the methods and their respective descriptions for the `Optimize` data structure are similar to `Solver` as they essentially provide the same interface for asserting formulas, backtracking, and checking satisfiability — with the addition of optimization-specific features.

- `pub fn get_model(&self) -> Option<Model>`: Retrieve the model from the last successful optimize `check` so you can query optimal values. The error handler is invoked if a model is not available because the `Optimize::check()` command was not invoked for the given optimization solver, or if the result of `Optimize::check()` was `SatResult::Unsat`.
- `pub fn get_objectives(&self) -> Vec<Dynamic>`: Retrieve the objectives for the last `Optimize::check()`. This contains maximize/minimize objectives and grouped soft constraints.
- `pub fn get_unsat_core(&self) -> Vec<Bool>`: Returns a subset of assumptions after the last call to `check()` or a sequence of calls to `assert_and_track` followed by `check()`. These are the assumptions Z3 used in the unsatisfiability proof.[15] This method returns a *subset* of the *tracked* literals and/or one-shot assumptions that together are already inconsistent. Each element of the vector is a `Bool` constant that you previously supplied as a label in `assert_and_track()`.
- `pub fn get_reason_unknown(&self) -> Option<String>`: Explain why Z3 gave up when it returns `Unknown`.
- `pub fn set_params(&self, params: &Params)`: Tune the optimiser just like any solver.
- `pub fn get_statistics(&self) -> Statistics`: Retrieve the statistics for the last `Optimize::check()` just like any solver.

- **`Weight`:** The trait is sealed; you cannot implement it for other types. It is only implemented for `u8, u16, u32, u64, u128, usize, i8, i16, i32, i64, i128, isize, BigInt, BigUint, BigRational`. In other words, these types can act as the weight in soft constraints.

Example 1 – Lexicographic minimisation

```rust
use z3::{ast::Int, Config, Context, Optimize, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let opt = Optimize::new(&ctx);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    // Hard constraints
    opt.assert(&x.ge(&Int::from_i64(&ctx, 0))); // x >= 0
    opt.assert(&y.ge(&Int::from_i64(&ctx, 0))); // y >= 0
    opt.assert(&Int::add(&ctx, &[&x, &y]).le(&Int::from_i64(&ctx, 10))); // x + y <= 10

    // Objectives (lexicographic)
    opt.minimize(&x); // 1st priority: minimise x
    opt.maximize(&y); // 2nd priority: then maximise y

    match opt.check(&[]) {
        SatResult::Sat => {
            let model = opt.get_model().unwrap();
            println!(
                "x = {}, y = {}", // prints: x = 0, y = 10
                model.eval(&x, true).unwrap(),
                model.eval(&y, true).unwrap()
            );
```

[15] By default, unsat cores are not minimized. To enable minimization, set `sat.core.minimize` or `smt.core.minimize` to `true` (for SAT or SMT respectively). This increases solving cost.

```
        }
        SatResult::Unsat => println!("unsat!"),
        SatResult::Unknown => println!("unknown! {:?}", opt.get_reason_unknown()),
    }
}
```

Example 2 – Max-SAT via soft constraints

```
use z3::{ast::Bool, Config, Context, Optimize, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let opt = Optimize::new(&ctx);

    let a = Bool::new_const(&ctx, "a");
    let b = Bool::new_const(&ctx, "b");
    let c = Bool::new_const(&ctx, "c");

    // Hard constraint: at most one can be true
    opt.assert(&a.implies(&Bool::and(&ctx, &[&b.not(), &c.not()])));
    opt.assert(&b.implies(&Bool::and(&ctx, &[&a.not(), &c.not()])));
    opt.assert(&c.implies(&Bool::and(&ctx, &[&a.not(), &b.not()])));

    // Soft constraints with different weights
    opt.assert_soft(&a, 5, None); // worth 5 points
    opt.assert_soft(&b, (1, 2), None); // worth 0.5 points
    opt.assert_soft(&c, 1u32, None); // worth 1 point

    // Solve
    assert_eq!(opt.check(&[]), SatResult::Sat);
    println!("violated weight sum = {:?}", opt.get_objectives());
}
```

Why use soft constraints? If the hard constraints cannot all be met simultaneously, Z3 drops a subset whose *total weight is minimal*.

Example 3 – Extracting an unsat core

```
use z3::{ast::Bool, Config, Context, Optimize, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let opt = Optimize::new(&ctx);

    let p = Bool::new_const(&ctx, "p");
    let q = Bool::new_const(&ctx, "q");

    // Track each hard clause with a literal
    opt.assert_and_track(&p, &p); // (p) tagged by p
    opt.assert_and_track(&q, &q); // (q) tagged by q
    opt.assert_and_track(
        &Bool::and(&ctx, &[&p, &q]).not(),
        &Bool::new_const(&ctx, "r"),
    );

    if let SatResult::Unsat = opt.check(&[]) {
        for culprit in opt.get_unsat_core() {
            println!("in core: {}", culprit);
        }
```

```
    }
}
```

The optimiser returns a subset of tracked literals that is itself inconsistent, allowing you to pinpoint which assumptions caused `Unsat`.[16]

19 *ops* module

This module enables operator overloading for Z3's Rust AST types— specifically for `BV`, `Int`, `Real`, `Float`, and `Bool`—by implementing the corresponding traits from `std::ops`. The macros defined in this module automate trait implementations for arithmetic and logic operations, making Z3 expressions ergonomic and Rust-like. Concretely, it means you can write expressions such as `let expr = (&x + 5u64) * &y - 1i64;` instead of the more verbose `Int::add(...)` calls. Even references (`&Int`) and mixed forms (`Int + &Int`) are handled automatically. A user can simply *import* the desired sorts and rely on the familiar syntax—from that point on, every operator compiles down to the correct Z3 primitive with no extra work.[17]

```rust
use z3::{
    ast::{Bool, Int, BV},
    Config, Context,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let sum = &x + 5u64 + &y; // Int + u64 + &Int
    let diff = 10i64 - &x; // i64 - &Int

    // Bit-vector
    let a = BV::new_const(&ctx, "a", 32);
    let expr = (&a << 3u64) ^ 0xffu64; // (a << 3) XOR 0xFF

    // Boolean
    let p = Bool::new_const(&ctx, "p");
    let q = Bool::new_const(&ctx, "q");
    let formula = !p & (q ^ true); // NOT p AND (q XOR true)

    println!("sum  : {sum}");
    println!("diff : {diff}");
    println!("expr : {expr}");
    println!("logic: {formula}");
}

```

[16] These three patterns—*objectives*, *soft constraints*, and *unsat-core extraction*—cover 95 % of practical optimisation tasks you will encounter when combining Rust with Z3.

[17] Most of the patterns here mirror what you would write by hand against `std::ops`, but the macros shrink hundreds of lines of duplicated impls into a maintainable few dozen.

Bit-Vector (BV) Operators

Operator	Z3 Method
+, +=	bvadd
-, -=	bvsub
*, *=	bvmul
&, &=	bvand
|, |=	bvor
^, ≙	bvxor
<<, <<=	bvshl
!	bvnot
- (unary)	bvneg

Integer (Int) Operators

Operator	Z3 Method
+, +=	add
-, -=	sub
*, *=	mul
/, /=	div
%, %=	rem
- (unary)	unary_minus

Real (Real) Operators

Operator	Z3 Method
+, +=	add
-, -=	sub
*, *=	mul
/, /=	div
- (unary)	unary_minus

Float (Float) Operators

Operator	Z3 Method
- (unary)	unary_neg

Boolean (Bool) Operators

Operator	Z3 Method
&, &=	and
|, |=	or
^, ≙	xor
!	not

20 *model* module

A Model represents one satisfying assignment (or optimum witness) produced by a Z3 *solver*. In other words, a *model* in Z3 represents a satisfying assignment for a set of constraints. When Z3 successfully finds that a formula is satisfiable (SatResult::Sat), it can provide a model that assigns concrete values to the variables and functions in your formula, demonstrating why the formula is true.[18]

- impl Model:
  - pub fn of_solver(slv: &Solver) -> Option<Model>: Extract a model from a solver after a successful check() call that returned SatResult::Sat. Returns None if no model is available (e.g., if the last check was unsatisfiable). This is the most common way to obtain a model.

[18]Again if the formula is SatResult::Unknown, there is no guarantee that a model will be generated or the generated model may no satisfy the constraints.

```rust
use z3::{
    ast::{self, Ast, Int},
    Config, Context, Model, SatResult, Solver,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = ast::Int::new_const(&ctx, "x");
    let y = ast::Int::new_const(&ctx, "y");

    // x + y = 10 and x > 5
    solver.assert(&(&Int::add(&ctx, &[&x, &y]))._eq(&ast::Int::from_i64(&ctx, 10)));
    solver.assert(&x.gt(&ast::Int::from_i64(&ctx, 5)));

    match solver.check() {
        SatResult::Sat => {
            let m1 = Model::of_solver(&solver).unwrap();
            let m2 = solver.get_model().unwrap();
            println!("Verification with first method:");
            println!("x = {}", m1.eval(&x, true).unwrap());
            println!("y = {}", m1.eval(&y, true).unwrap());

            // Verify the model with the second method
            println!("Verification with second method:");
            println!("x = {}", m2.eval(&x, true).unwrap());
            println!("y = {}", m2.eval(&y, true).unwrap());
        }
        SatResult::Unsat => println!("No solution exists"),
        SatResult::Unknown => println!("Z3 couldn't determine satisfiability"),
    }
}
```

– `pub fn of_optimize(opt: &Optimize) -> Option<Model>`: Same idea, but for `Optimize` but used where you're not just looking for any satisfying assignment, but an optimal one according to some objective function. Lets you read the optimum assignment after pushing objectives.

```rust
use z3::{ast, ast::Int, Config, Context, Optimize, SatResult};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let opt = Optimize::new(&ctx);

    let x = ast::Int::new_const(&ctx, "x");
    let y = ast::Int::new_const(&ctx, "y");

    // x + y >= 10, x >= 0, y >= 0
    opt.assert(&Int::add(&ctx, &[&x, &y]).ge(&ast::Int::from_i64(&ctx, 10)));
    opt.assert(&x.ge(&ast::Int::from_i64(&ctx, 0)));
    opt.assert(&y.ge(&ast::Int::from_i64(&ctx, 0)));

    // minimize x + y
    opt.minimize(&Int::add(&ctx, &[&x, &y]));

    match opt.check(&[]) {
        SatResult::Sat => {
```

```
            if let Some(model) = opt.get_model() {
                println!("Optimal solution:");
                println!("x = {}", model.eval(&x, true).unwrap());
                println!("y = {}", model.eval(&y, true).unwrap());
            }
        }
        _ => println!("No optimal solution found"),
    }
}
```

– `pub fn translate(&self, dest: &Context) -> Model`: Create a copy of this model in a different Z3 context. This is useful when you need to work with models across different Z3 contexts, as Z3 objects are tied to the context in which they were created.

```
use z3::{ast, ast::Ast, Config, Context, SatResult, Solver};

fn main() {
    let cfg1 = Config::new();
    let ctx1 = Context::new(&cfg1);
    let solver1 = Solver::new(&ctx1);

    let x1 = ast::Int::new_const(&ctx1, "x");
    solver1.assert(&x1._eq(&ast::Int::from_i64(&ctx1, 42)));

    match solver1.check() {
        SatResult::Sat => {
            let model1 = solver1.get_model().unwrap();

            let cfg2 = Config::new();
            let ctx2 = Context::new(&cfg2);
            let model2 = model1.translate(&ctx2);

            println!("Original model: {}", model1);
            println!("Translated model: {}", model2);
        }
        _ => println!("Unsatisfiable"),
    }
}
```

– `pub fn get_const_interp<T: Ast>(&self, c: &T) -> Option<T>`: Get the interpretation (assigned value) of a constant in the model. This works for any constant (zero-arity function), including variables you declared. Returns `None` if the constant has no interpretation in the model.

```
use z3::{ast, ast::Ast, Config, Context, SatResult, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = ast::Int::new_const(&ctx, "x");
    let y = ast::Bool::new_const(&ctx, "y");

    // x = 100 and y = true
    solver.assert(&x._eq(&ast::Int::from_i64(&ctx, 100)));
    solver.assert(&y._eq(&ast::Bool::from_bool(&ctx, true)));

    if solver.check() == SatResult::Sat {
        let model = solver.get_model().unwrap();
```

```
        if let Some(x_val) = model.get_const_interp(&x) {
            println!("x = {}", x_val);
        }

        if let Some(y_val) = model.get_const_interp(&y) {
            println!("y = {}", y_val);
        }

        // Check for a constant that doesn't have an interpretation
        let z = ast::Int::new_const(&ctx, "z");
        match model.get_const_interp(&z) {
            Some(z_val) => println!("z = {}", z_val),
            None => println!("z has no interpretation in this model"),
        }
    }
}
```

- `pub fn get_func_interp(&self, f: &FuncDecl) -> Option<FuncInterp>`: Get the interpretation of a function declaration in the model. This handles both zero-arity functions (constants) and higher-arity functions. For arrays, this method provides access to the array's interpretation as a function.
- `pub fn eval<T: Ast>(&self, t: &T, model_completion: bool) -> Option<T>`: Evaluate any expression in the context of this model. The `model_completion` parameter determines whether Z3 should assign default values to undefined parts of the model. This is the most flexible way to extract values from a model. When `model_completion` is `true`, Z3 will provide interpretations for symbols that don't have explicit values in the model, essentially "completing" the model. When `false`, evaluation may fail for undefined symbols.

```
use z3::{
    ast::{self, Ast, Int},
    Config, Context, SatResult, Solver,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let x = ast::Int::new_const(&ctx, "x");
    let y = ast::Int::new_const(&ctx, "y");

    solver.assert(&x._eq(&ast::Int::from_i64(&ctx, 7))); // x = 7

    if solver.check() == SatResult::Sat {
        let model = solver.get_model().unwrap();

        println!("x = {}", model.eval(&x, false).unwrap());

        // (x + y) * 2
        let expr = Int::mul(
            &ctx,
            &[&Int::add(&ctx, &[&x, &y]), &ast::Int::from_i64(&ctx, 2)],
        );

        match model.eval(&expr, false) {
            Some(val) => println!("2*(x+y) without completion = {}", val),
            None => println!("Cannot evaluate 2*(x+y) without completion"),
```

```
        }
        if let Some(val) = model.eval(&expr, true) {
            println!("2*(x+y) with completion = {}", val);
        }

        if let Some(y_val) = model.eval(&y, true) {
            println!("y (with completion) = {}", y_val);
        }
    }
}
```

  - `pub fn iter(&self) -> impl Iterator<Item = FuncDecl>`: Iterate over every constant and function that *does* have an interpretation.

Tips:

- **Context Lifetime:** Models are tied to their Z3 context. Make sure the context lives long enough for your model usage.
- **Checking for Interpretations:** Always check if `get_const_interp` or `eval` returns `Some` before using the result, as not all symbols may have interpretations.
- **Array Handling:** Arrays are represented as functions in Z3 models. Use `get_func_interp` to examine array contents.

21 *goal* module

Goals are collections of constraints that tactics can process, simplify, or solve. Unlike direct solver usage, goals allow you to apply *preprocessing* transformations before solving.

Goals vs. Solvers: Goals are *containers for formulas* that can be transformed by tactics, while **Solvers** directly check satisfiability. Goals provide more fine-grained control over the solving process by allowing tactical preprocessing.

The following public items are accessible:

- `pub fn new(ctx: &Context, models: bool, unsat_cores: bool, proofs: bool) -> Goal`: Create a new goal with specified capabilities. The boolean parameters control what additional information Z3 will track:
  - `models`: Generate models for satisfiable goals
  - `unsat_cores`: Track unsatisfiable cores for debugging
  - `proofs`: Generate proofs (requires context with proof support otherwise a segmentation error will occur)

```
use z3::{
    ast::Int,
    Config, Context, Goal,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let goal = Goal::new(&ctx, true, false, false);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    goal.assert(&x.gt(&Int::from_i64(&ctx, 0))); // x > 0
    goal.assert(&y.lt(&Int::from_i64(&ctx, 10))); // y < 10
```

```
    goal.assert(&x.le(&y)); // x <= y

    println!("Goal: {}", goal);
    println!("Goal size: {}", goal.get_size());
}
```

- `pub fn assert(&self, ast: &impl Ast) -> ()`: Add a formula to the goal. This is the primary way to build up a collection of constraints that will be processed together.
- `pub fn is_inconsistent(&self) -> bool`: Check if the goal contains the formula `false`, indicating it's unsatisfiable. This will only return `true` if the `goal.assert(&Bool::from_bool(&ctx, false))` is explicitly added as a term to the formula. In short, *it is only a syntactic test that asks does this goal already contain the literal false?* It does not do reasoning.

```
use z3::{
    ast::{Bool, Int},
    Config, Context, Goal,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let goal = Goal::new(&ctx, true, false, false);

    let x = Int::new_const(&ctx, "x");

    goal.assert(&x.gt(&Int::from_i64(&ctx, 5)));
    goal.assert(&x.lt(&Int::from_i64(&ctx, 0)));
    goal.assert(&Bool::from_bool(&ctx, false)); // comment this line to make the goal
    ↪  consistent even thoug there is a contradiction!!!
    println!("Goal is inconsistent: {}", goal.is_inconsistent());

    let consistent_goal = Goal::new(&ctx, true, false, false);
    let y = Int::new_const(&ctx, "y");
    consistent_goal.assert(&y.gt(&Int::from_i64(&ctx, 0)));
    consistent_goal.assert(&y.lt(&Int::from_i64(&ctx, 10)));

    println!("Consistent goal is inconsistent: {}", consistent_goal.is_inconsistent());
}
```

- `pub fn get_depth(&self) -> u32`: Return the depth of transformations applied to the goal. This tracks how many tactical transformations have been applied.
- `pub fn get_size(&self) -> u32`: Return the number of formulas in the goal. This counts top-level assertions, not subformulas.
- `pub fn get_num_expr(&self) -> u32`: Return the total number of expressions (formulas, subformulas, and terms) in the goal. This provides a measure of the goal's complexity.

```
use z3::{
    ast::{Bool, Int},
    Config, Context, Goal,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let goal = Goal::new(&ctx, true, false, false);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
```

```
13
14      println!(
15          "Empty goal - Size: {}, Expressions: {}",
16          goal.get_size(),
17          goal.get_num_expr()
18      );
19
20      goal.assert(&x.gt(&Int::from_i64(&ctx, 0)));
21      println!(
22          "After x > 0 - Size: {}, Expressions: {}",
23          goal.get_size(),
24          goal.get_num_expr()
25      );
26
27      let complex = Bool::and(
28          &ctx,
29          &[
30              Int::add(&ctx, &[&x, &y]).gt(&Int::from_i64(&ctx, 5)), // x + y > 5
31              Int::sub(&ctx, &[&x, &y]).lt(&Int::from_i64(&ctx, 2)), // x - y < 2
32          ],
33      );
34      goal.assert(&complex);
35      println!(
36          "After complex constraint - Size: {}, Expressions: {}",
37          goal.get_size(),
38          goal.get_num_expr()
39      );
40  }
```

- `pub fn is_decided_sat(&self) -> bool`: Return true if the goal is empty and the precision is precise or under—cases where an empty set of constraints really is a guaranteed `Sat` instance. This indicates the goal has been solved and is satisfiable. *It does not try to solve the goal; it merely asks whether the goal is already in a trivially decidable syntactic form; i.e., the goal is empty (no formulas left) and its precision tag is precise or under.*

- `pub fn is_decided_unsat(&self) -> bool`: Return true if the goal contains false and the precision is precise or over—situations where false is a trustworthy `Unsat` certificate. This indicates the goal has been solved and is unsatisfiable. *It does not try to solve the goal; it merely asks whether the goal is already in a trivially decidable syntactic form; i.e., the goal contains the literal false and its precision is precise or over.*

```
1   use z3::{ast::Int, Config, Context, Goal, Tactic};
2
3   fn main() {
4       let cfg = Config::new();
5       let ctx = Context::new(&cfg);
6
7       let sat_goal = Goal::new(&ctx, true, false, false);
8       let x = Int::new_const(&ctx, "x");
9       sat_goal.assert(&x.gt(&Int::from_i64(&ctx, 0)));
10
11      let unsat_goal = Goal::new(&ctx, true, false, false);
12      unsat_goal.assert(&x.gt(&Int::from_i64(&ctx, 5)));
13      unsat_goal.assert(&x.lt(&Int::from_i64(&ctx, 0)));
14
15      println!("Sat goal decided sat: {}", sat_goal.is_decided_sat());
16      println!("Sat goal decided unsat: {}", sat_goal.is_decided_unsat());
17
18      let tactic = Tactic::new(&ctx, "simplify")
```

```
        .and_then(&Tactic::new(&ctx, "propagate-values"))
        .and_then(&Tactic::new(&ctx, "smt"));
    if let Ok(result) = tactic.apply(&sat_goal, None) {
        for subgoal in result.list_subgoals() {
            println!("Subgoal decided sat: {}", subgoal.is_decided_sat());
            println!("Subgoal decided unsat: {}", subgoal.is_decided_unsat());
        }
    };
}
```

- `pub fn reset(&self) -> ()`: Erase all formulas from the goal, returning it to an empty state. This is useful for reusing goal objects.
- `pub fn translate(self, ctx: &Context) -> Goal`: Move a goal from one Z3 context to another. This is necessary when working with multiple contexts.

```
use z3::{
    ast::Int,
    Config, Context, Goal,
};

fn main() {
    let cfg1 = Config::new();
    let ctx1 = Context::new(&cfg1);
    let goal1 = Goal::new(&ctx1, true, false, false);

    let x = Int::new_const(&ctx1, "x");
    goal1.assert(&x.gt(&Int::from_i64(&ctx1, 0))); // x > 0

    println!("Original goal: {}", goal1);

    let cfg2 = Config::new();
    let ctx2 = Context::new(&cfg2);
    let goal2 = goal1.translate(&ctx2);

    println!("Translated goal: {}", goal2);

    let y = Int::new_const(&ctx2, "y");
    goal2.assert(&y.lt(&Int::from_i64(&ctx2, 10))); // y < 10
    println!("Goal after adding constraint in new context: {}", goal2);
}
```

- `pub fn get_precision(&self) -> GoalPrec`: Return the precision level of the goal. Goals can be transformed using over-approximations or under-approximations, and this method indicates the current precision level.
- `pub fn iter_formulas<T>(&self) -> impl Iterator<Item = T>`: Create an iterator over all formulas in the goal.
- `pub fn get_formulas<T>(&self) -> Vec<T>`: Return a vector containing all formulas in the goal.

```
use z3::{
    ast::{Ast, Bool, Int},
    Config, Context, Goal,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let goal = Goal::new(&ctx, true, false, false);

```

```rust
    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");

    goal.assert(&x.gt(&Int::from_i64(&ctx, 0))); // x > 0
    goal.assert(&y.lt(&Int::from_i64(&ctx, 10))); // y < 10
    goal.assert(&Int::add(&ctx, &[&x, &y])._eq(&Int::from_i64(&ctx, 5))); // x + y = 5

    println!("Formulas via iterator:");
    for (i, formula) in goal.iter_formulas::<Bool>().enumerate() {
        println!("  {}: {}", i, formula);
    }

    println!("\nFormulas via get_formulas:");
    let formulas: Vec<Bool> = goal.get_formulas();
    for (i, formula) in formulas.iter().enumerate() {
        println!("  {}: {}", i, formula);
    }

    let combined = Bool::and(&ctx, &formulas);
    println!("\nCombined formula: {}", combined);
}
```

22 *func_interp* module

The `FuncInterp` API allows you to work with *function interpretations* in Z3 models. When Z3 solves constraints involving function declarations, it produces a *function interpretation* that specifies how the function maps inputs to outputs.[19]

What is a Function Interpretation? A function interpretation is essentially a *lookup table* that tells you what value a function returns for specific inputs, plus a default *else* value for all other inputs. For example, if you have a function `f(x)` and Z3 determines that `f(1) = 10`, `f(2) = 20`, and `f(x) = 0` for all other values of `x`, then the function interpretation contains two explicit entries and an else value of 0.

The following public items are accessible:

- `pub fn get_arity(&self) -> usize`: Returns the number of arguments the function takes. For example, a function `f(x, y)` has arity 2, while `g(x)` has arity 1.

```rust
use z3::{
    ast::{Ast, Int},
    Config, Context, FuncDecl, Solver, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let f = FuncDecl::new(&ctx, "f", &[&Sort::int(&ctx)], &Sort::int(&ctx));
    let input1 = Int::from_i64(&ctx, 1);
    let input2 = Int::from_i64(&ctx, 2);
    let value1 = Int::from_i64(&ctx, 100);
    let value2 = Int::from_i64(&ctx, 200);

    // f(input1) = value1 and f(input2) = value2
    solver.assert(&f.apply(&[&input1])._eq(&value1.into()));
```

[19] Arrays in Z3 are internally represented as functions using the `(_ as-array f)` construct, but when extracting interpretations from models, arrays typically require `get_const_interp` rather than `get_func_interp`, as Z3 often represents array models as constant values rather than explicit function interpretations.

```
    solver.assert(&f.apply(&[&input2])._eq(&value2.into()));

    if solver.check() == z3::SatResult::Sat {
        let model = solver.get_model().unwrap();
        if let Some(func_interp) = model.get_func_interp(&f) {
            println!("Function arity: {}", func_interp.get_arity()); // Function arity: 1
        }
    }
}
```

- `pub fn get_num_entries(&self) -> u32`: Returns the number of explicit entries in the function interpretation. This tells you how many input-output pairs Z3 has explicitly defined, excluding the default *else* value.

```
use z3::{
    ast::{Ast, Int},
    Config, Context, FuncDecl, Solver, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let f = FuncDecl::new(&ctx, "f", &[&Sort::int(&ctx)], &Sort::int(&ctx));
    let input1 = Int::from_i64(&ctx, 1);
    let input2 = Int::from_i64(&ctx, 2);
    let value1 = Int::from_i64(&ctx, 100);
    let value2 = Int::from_i64(&ctx, 200);

    // f(input1) = value1 and f(input2) = value2
    solver.assert(&f.apply(&[&input1])._eq(&value1.into()));
    solver.assert(&f.apply(&[&input2])._eq(&value2.into()));

    if solver.check() == z3::SatResult::Sat {
        let model = solver.get_model().unwrap();
        if let Some(func_interp) = model.get_func_interp(&f) {
            println!("Function interpretation: {:?}", func_interp);
            println!("Function arity: {}", func_interp.get_arity()); // Function arity: 1
            let num_entries = func_interp.get_num_entries();
            println!("Number of entries: {}", num_entries); // Number of entries: 1
        }
    }
}
```

- `pub fn add_entry(&self, args: &[Dynamic], value: &Dynamic)`: Adds a new entry to the function interpretation, specifying that the function should return `value` when called with `args`. This is useful when you're building function interpretations programmatically.

```
use z3::{
    ast::{Ast, Int},
    Config, Context, FuncDecl, Solver, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let f = FuncDecl::new(&ctx, "f", &[&Sort::int(&ctx)], &Sort::int(&ctx));
```

```rust
    let input1 = Int::from_i64(&ctx, 1);
    let input2 = Int::from_i64(&ctx, 2);
    let value1 = Int::from_i64(&ctx, 100);
    let value2 = Int::from_i64(&ctx, 200);

    // f(input1) = value1 and f(input2) = value2
    solver.assert(&f.apply(&[&input1])._eq(&value1.into()));
    solver.assert(&f.apply(&[&input2])._eq(&value2.into()));

    if solver.check() == z3::SatResult::Sat {
        let model = solver.get_model().unwrap();
        if let Some(func_interp) = model.get_func_interp(&f) {
            println!("Function interpretation: {:?}", func_interp);
            println!("Function arity: {}", func_interp.get_arity()); // Function arity: 1
            let num_entries = func_interp.get_num_entries();
            println!("Number of entries: {}", num_entries); // Number of entries: 1
                                                            // add entries
            for i in 0..3 {
                func_interp.add_entry(
                    &[Int::from_i64(&ctx, i as i64).into()],
                    &Int::from_i64(&ctx, i as i64 * 10).into(), // overwrites existing
                    ↪  entries (2 -> 20)
                );
            }
            println!("Updated function interpretation: {:?}", func_interp);
            let num_entries = func_interp.get_num_entries();
            println!("Number of entries: {}", num_entries);
        }
    }
}
```

- `pub fn get_entries(&self) -> Vec<FuncEntry>`: Returns all explicit entries in the function interpretation (not the *else*). Each entry contains the input arguments and corresponding output value.
- `pub fn get_else(&self) -> Dynamic`: Returns the default *else* value that the function returns for all inputs not explicitly specified in the entries. This is crucial for understanding complete function behavior.
- `pub fn set_else(&self, ast: &Dynamic)`: Sets the default value for the function interpretation. This value will be returned for any input arguments not explicitly covered by the entries.

Note that regarding *efficiency*, function interpretations are more memory-efficient than storing every possible input-output pair.

23 *func_entry* module

The `FuncEntry` API represents individual entries in a function interpretation.[20] The following public items are accessible:

Function Entries vs. Function Interpretations: A **FuncEntry** is a single mapping (e.g., `f(2, 3) = 7`), while a **FuncInterp** is the complete function definition containing all entries plus a default `else` value.

- `pub fn get_value(&self) -> Dynamic`: Returns the output value that the function produces for this entry's arguments. This is the result of applying the function to the specific argument combination represented by this entry.

[20] A function entry is a single mapping from arguments to a value within a function interpretation, essentially representing one row in a function's lookup table.

```rust
use z3::{
    ast::{Ast, Int},
    Config, Context, FuncDecl, SatResult, Solver,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    // f: Int x Int -> Int
    let int_sort = Int::new_const(&ctx, "dummy").get_sort();
    let f = FuncDecl::new(&ctx, "f", &[&int_sort, &int_sort], &int_sort);

    let x = Int::new_const(&ctx, "x");
    let y = Int::new_const(&ctx, "y");
    let z = Int::new_const(&ctx, "z");
    let two = Int::from_i64(&ctx, 2);
    let three = Int::from_i64(&ctx, 3);
    let seven = Int::from_i64(&ctx, 7);

    // f(2, 3) = 7
    solver.assert(&f.apply(&[&two, &three])._eq(&seven.into()));

    // f(x, y) = z where x + y = 5 and z = 10
    let five = Int::from_i64(&ctx, 5);
    let ten = Int::from_i64(&ctx, 10);
    solver.assert(&Int::add(&ctx, &[&x, &y])._eq(&five));
    solver.assert(&f.apply(&[&x, &y])._eq(&z.clone().into()));
    solver.assert(&z._eq(&ten));

    if solver.check() == SatResult::Sat {
        let model = solver.get_model().unwrap();
        println!("{} is satisfiable", model);
        if let Some(func_interp) = model.get_func_interp(&f) {
            for entry in func_interp.get_entries() {
                let value = entry.get_value();
                let num_args = entry.get_num_args();
                let args = entry.get_args();
                println!("Function entry value: {}", value);
                println!("Number of arguments: {}", num_args);
                for (i, arg) in args.iter().enumerate() {
                    println!("Argument {}: {}", i, arg);
                }
            }
        }
    }
}
```

- `pub fn get_num_args(&self) -> u32`: Returns the number of arguments (arity) for this function entry. This tells you how many input parameters the function takes for this specific mapping.
- `pub fn get_args(&self) -> Vec<Dynamic>`: Returns the input arguments for this function entry as a vector of `Dynamic` values. These are the specific values that, when passed to the function, produce the output returned by `get_value()`.

24 *func_decl* module

The `FuncDecl` API provides a way to create and work with function declarations in Z3. Function declarations define the signature of functions (their name, domain types, and range type) without specifying their behavior. They are essential for creating uninterpreted functions.

Function Declaration: A *signature specification* that defines a function's name and type information (domain and range sorts). It declares what the function looks like but not what it does. For example, declaring `f: Int × Int → Bool` creates a function that takes two integers and returns a boolean, but doesn't specify the actual mapping behavior.

Function Interpretation: A *concrete implementation* that defines how a function behaves by providing explicit input-output mappings. This is what you get in a model after Z3 solves constraints involving the function declaration.

The following public items are accessible:

- `pub fn new<S: Into<Symbol>>(ctx: &Context, name: S, domain: &[&Sort], range: &Sort) -> FuncDecl`: Create a new function declaration with the given name, domain types, and range type. The domain is an array of sorts representing the argument types, and the range is the return type. If the domain is empty, this creates a *constant* declaration.

```rust
use z3::{Config, Context, FuncDecl, Sort};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    // constant declaration (0-arity function)
    let x = FuncDecl::new(&ctx, "x", &[], &Sort::int(&ctx));
    println!("Constant x: {}", x);

    // unary function f: Int → Bool
    let f = FuncDecl::new(&ctx, "f", &[&Sort::int(&ctx)], &Sort::bool(&ctx));
    println!("Function f: {}", f);

    // binary function add: Int × Int → Int
    let add = FuncDecl::new(
        &ctx,
        "add",
        &[&Sort::int(&ctx), &Sort::int(&ctx)],
        &Sort::int(&ctx),
    );
    println!("Function add: {}", add);

    // ternary function: Real × Real × Real → Real
    let distance3d = FuncDecl::new(
        &ctx,
        "distance3d",
        &[&Sort::real(&ctx), &Sort::real(&ctx), &Sort::real(&ctx)],
        &Sort::real(&ctx),
    );
    println!("Function distance3d: {}", distance3d);
    println!("Function distance3d arity: {}", distance3d.arity());
}
```

- `pub fn arity(&self) -> usize`: Return the number of arguments (arity) of the function declaration. For constants, this returns 0. For unary functions, this returns 1, and so on.
- `pub fn apply(&self, args: &[&dyn Ast]) -> Dynamic`: Create a function application by ap-

plying the function declaration to the given arguments. If no arguments are provided (empty slice), this creates a constant. The number and types of arguments must match the function's domain specification.

```
use z3::{
    ast::{Ast, Bool, Dynamic, Int},
    Config, Context, FuncDecl, SatResult, Solver, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let x_decl = FuncDecl::new(&ctx, "x", &[], &Sort::int(&ctx));
    let y_decl = FuncDecl::new(&ctx, "y", &[], &Sort::int(&ctx));

    // Create function declarations
    let f_decl = FuncDecl::new(&ctx, "f", &[&Sort::int(&ctx)], &Sort::bool(&ctx));
    println!("kind of f: {:?}", f_decl.kind());
    let add_decl = FuncDecl::new(
        &ctx,
        "add",
        &[&Sort::int(&ctx), &Sort::int(&ctx)],
        &Sort::int(&ctx),
    );

    let x: Dynamic = x_decl.apply(&[]); // Constant x
    let y: Dynamic = y_decl.apply(&[]); // Constant y
    let five = Int::from_i64(&ctx, 5);
    let f_5: Dynamic = f_decl.apply(&[&five]); // f(5)
    let ten = Int::from_i64(&ctx, 10);
    let add_5_10: Dynamic = add_decl.apply(&[&five, &ten]); // add(5, 10)

    let solver = Solver::new(&ctx);
    // Assert f(5) is true
    solver.assert(&Bool::from_bool(&ctx, true)._eq(&f_5.as_bool().unwrap()));
    // Assert add(5, 10) = 15
    solver.assert(&Int::from_i64(&ctx, 15)._eq(&add_5_10.as_int().unwrap()));
    // Assert add(x, y) = 20
    let twenty = Int::from_i64(&ctx, 20);
    solver.assert(&twenty._eq(&add_decl.apply(&[&x, &y]).as_int().unwrap()));

    match solver.check() {
        SatResult::Sat => {
            let model = solver.get_model().unwrap();
            println!("Model found!");
            println!("x = {}", model.eval(&x, true).unwrap());
            println!("y = {}", model.eval(&y, true).unwrap());
            println!("f(5) = {}", model.eval(&f_5, true).unwrap());
            println!("add(5, 10) = {}", model.eval(&add_5_10, true).unwrap());
        }
        SatResult::Unsat => println!("No solution"),
        SatResult::Unknown => println!("Unknown"),
    }
}
```

- `pub fn kind(&self) -> DeclKind`: Return the declaration kind of this function declaration. This tells you what type of declaration it is (e.g., uninterpreted function, built-in operator, etc.).

```
use z3::{Config, Context, FuncDecl, Sort};
```

```rust

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);

    let f = FuncDecl::new(&ctx, "f", &[&Sort::int(&ctx)], &Sort::bool(&ctx));
    let g = FuncDecl::new(&ctx, "g", &[], &Sort::int(&ctx));
    let h = FuncDecl::new(
        &ctx,
        "binary_op",
        &[&Sort::int(&ctx), &Sort::int(&ctx)],
        &Sort::int(&ctx),
    );

    println!("Function f kind: {:?}", f.kind());
    println!("Constant g kind: {:?}", g.kind());
    println!("Function h kind: {:?}", h.kind());

}
```

All of the above are `UNINTERPRETED`.

- `pub fn name(&self) -> String`: Return the name of this function declaration as a string. For string symbols, this returns the symbol directly. For integer symbols, this returns the symbol with a `"k!"` prefix.

25 *datatype_builder* module

The `DatatypeBuilder` API lets you create custom *algebraic data types* in Z3. Datatypes are composite types that can represent structures like lists, trees, structs, enums, and other recursive data structures. Datatypes are ADTs that combine multiple values into structured data. Unlike basic sorts (integers, booleans), datatypes have *constructors* (functions that create values) and *accessors* (functions that extract components). Each datatype can have multiple *variants*, each with different fields.

- `pub fn new<S: Into<Symbol>>(ctx: &Context, name: S) -> Self`: Create a new datatype builder with the given name. The name will be used as the sort name for the final datatype.
- `pub fn variant(mut self, name: &str, fields: Vec<(&str, DatatypeAccessor)>) -> Self`: Add a variant (constructor) to the datatype. Each variant represents a different way to construct values of this datatype. The `name` parameter is the constructor name, and `fields` specify the components that this variant contains. Each field has a name and a `DatatypeAccessor` that specifies its type (either another datatype or a regular sort).
- `pub fn finish(self) -> DatatypeSort`: Finalize the datatype construction and return a `DatatypeSort` that can be used to create values and constraints. This creates the actual Z3 datatype sort with all its *constructors, testers, and accessors.*

```rust
use z3::{ast::Datatype, Config, Context, DatatypeBuilder, SatResult, Solver};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let color_sort = DatatypeBuilder::new(&ctx, "Color")
        .variant("Red", vec![])
        .variant("Green", vec![])
        .variant("Blue", vec![])
        .finish();

```

```
    let red = color_sort.variants[0].constructor.apply(&[]);
    let green = color_sort.variants[1].constructor.apply(&[]);
    let blue = color_sort.variants[2].constructor.apply(&[]);

    // Create a variable of Color type
    let x = Datatype::new_const(&ctx, "x", &color_sort.sort);

    // Assert that x is red
    let is_red = color_sort.variants[0].tester.apply(&[&x]);
    solver.assert(&is_red.as_bool().unwrap());

    assert_eq!(solver.check(), SatResult::Sat);
    let model = solver.get_model().unwrap();
    println!("x = {}", model.eval(&x, true).unwrap());
}
```

For more details, please refer to the section on `DatatypeBuilder` in *lib.rs*.

- `pub fn create_datatypes(datatype_builders: Vec<DatatypeBuilder>) -> Vec<DatatypeSort>`: Create multiple datatypes at once. This is required when you have datatypes that reference each other (mutual recursion). Z3 needs to create all mutually recursive datatypes in a single operation to properly handle the circular dependencies.

```
use z3::{
    ast::{Datatype, Int},
    datatype_builder::create_datatypes,
    Config, Context, DatatypeAccessor, DatatypeBuilder, SatResult, Solver, Sort,
};

fn main() {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Solver::new(&ctx);

    let expr_builder = DatatypeBuilder::new(&ctx, "Expr")
        .variant(
            "Num",
            vec![("value", DatatypeAccessor::Sort(Sort::int(&ctx)))],
        )
        .variant(
            "Add",
            vec![
                ("left", DatatypeAccessor::Datatype("Expr".into())),
                ("right", DatatypeAccessor::Datatype("Expr".into())),
            ],
        )
        .variant(
            "Let",
            vec![
                ("binding", DatatypeAccessor::Datatype("Binding".into())),
                ("body", DatatypeAccessor::Datatype("Expr".into())),
            ],
        );

    let binding_builder = DatatypeBuilder::new(&ctx, "Binding").variant(
        "Bind",
        vec![
            ("name", DatatypeAccessor::Sort(Sort::int(&ctx))), // simplified with int
            ("value", DatatypeAccessor::Datatype("Expr".into())),
        ],
```

```
    );

    let datatypes = create_datatypes(vec![expr_builder, binding_builder]);
    let expr_sort = &datatypes[0];
    let binding_sort = &datatypes[1];

    let five = Int::from_i64(&ctx, 5);
    let x_name = Int::from_i64(&ctx, 1);

    let num5 = expr_sort.variants[0].constructor.apply(&[&five]);

    // let x = 5
    let binding = binding_sort.variants[0]
        .constructor
        .apply(&[&x_name, &num5]);
    let body_expr = Datatype::new_const(&ctx, "body", &expr_sort.sort);
    let let_expr = expr_sort.variants[2]
        .constructor
        .apply(&[&binding, &body_expr]);

    // Assert that body_expr is an Add expression
    let is_add = expr_sort.variants[1].tester.apply(&[&body_expr]);
    solver.assert(&is_add.as_bool().unwrap());

    // Check satisfiability
    assert_eq!(solver.check(), SatResult::Sat);
    let model = solver.get_model().unwrap();
    println!("Let expression = {}", model.eval(&let_expr, true).unwrap());
    println!(
        "Body expression = {}",
        model.eval(&body_expr, true).unwrap()
    );
}
```

Tips:

Constructors: Functions that create values of the datatype. Each variant has its own constructor function.

Testers: Functions that check if a value was created with a specific constructor (e.g., `is-Red`, `is-Node`).

Accessors: Functions that extract field values from datatype instances. Only valid when the tester for the corresponding constructor returns true.

Mutual Recursion: When datatypes reference each other, they must be created together using `create_datatypes` to handle circular dependencies properly.

26 *config* module

The `Config` API allows you to customize Z3's behavior before creating a context. Configuration objects are used to set parameters that control proof generation, model generation, timeouts, and other Z3 engine settings. *Config* objects are created first to specify Z3's operational parameters, then passed to `Context::new()` to create the actual Z3 context. Once a context is created, its configuration cannot be changed. The following public items are accessible:

- `pub fn new() -> Config`: Create a new configuration object with default settings. This is the starting point for customizing Z3's behavior.
- `pub fn set_param_value(&mut self, k: &str, v: &str)`: Set any Z3 configuration parameter by name. This is the most general method for configuring Z3, allowing access to all available

parameters.

- `pub fn set_bool_param_value(&mut self, k: &str, v: bool)`: Helper method for setting boolean parameters. This automatically converts the boolean value to the appropriate string representation ("true" or "false").
- `pub fn set_proof_generation(&mut self, b: bool)`: Enable or disable proof generation. When enabled, Z3 will generate proofs for unsatisfiable results, which can be retrieved using `Solver::get_proof()`.
- `pub fn set_model_generation(&mut self, b: bool)`: Enable or disable model generation. When enabled, Z3 will generate models for satisfiable results, which can be retrieved using `Solver::get_model()`.
- `pub fn set_debug_ref_count(&mut self, b: bool)`: Enable or disable reference counting debugging. This is useful for debugging memory leaks in Z3 applications by tracking reference counts of Z3 objects.
- `pub fn set_timeout_msec(&mut self, ms: u64)`: Set a timeout in milliseconds for Z3 operations. This prevents Z3 from running indefinitely on difficult problems.

Tips:

- **Set configuration before context creation:** All configuration parameters must be set before creating the Z3 context. Configuration cannot be changed after context creation.
- **Enable features as needed:** Model and proof generation have performance overhead. Only enable them when you need to inspect results.
- **Use timeouts for robustness:** Set reasonable timeouts to prevent your application from hanging on difficult problems.

The `Config` type implements `Default`, so you can use `Config::default()` as an alternative to `Config::new()`:

```
use z3::{Config, Context};

fn main() {
    // These are equivalent:
    let cfg1 = Config::new();
    let cfg2 = Config::default();

    // Both create contexts with default settings
    let ctx1 = Context::new(&cfg1);
    let ctx2 = Context::new(&cfg2);
}
```

27 *ast* module

This module provides the `Ast` trait and nodes representing various data values in Z3. The key AST node types include:

- **Bool:** `new_const` & `fresh_const` for creating boolean constants.[21] `from_bool` & `as_bool` for conversion to/from Rust Booleans. `ite` for *if-then-else expressions*. Logical operations: `and, or, xor, iff, implies, not`. Pseudo-boolean operations: `pb_le, pb_ge, pb_eq` for linear integer programming constraints. Use for boolean logic and satisfiability problems.

[21] `new_const` creates a constant with a user-provided name and returns the same symbolic variable if reused, making it ideal for defining and referencing specific variables. In contrast, `fresh_const` creates a constant with a unique, auto-generated name prefixed by the given string from the user, ensuring distinct variables each time.

- **Int:** `new_const` & `fresh_const` for creating integer constants. `new_const` creates a constant with a user-provided name and returns the same symbolic variable if reused, making it ideal for defining and referencing specific variables. In contrast, `fresh_const` creates a constant with a unique, auto-generated name prefixed by the given string, ensuring distinct variables each time. `from_big_int` & `from_str` for parsing from `BigInt` and `String`. `from_i64`, `from_u64`, `as_i64`, `as_u64` for conversion to/from Rust primitives. `from_real`, `to_real` for real number conversion. `from_bv`, `to_ast` for bitvector conversion. Arithmetic operations: `add`, `sub`, `mul`, `div`, `rem`, `modulo`, `power`, `unary_minus`. Comparison operations: `lt`, `le`, `gt`, `ge`.
- **Real:** `new_const` & `fresh_const` for creating real number constants. `from_big_rational` & `from_real_str` for parsing from rational numbers. `from_real` for creating from numerator/denominator pairs. `as_real` for extraction as (numerator, denominator) tuple. `approx`, `approx_f64` for decimal approximation. `from_int`, `to_int` for integer conversion. `is_int` predicate to check if a real is actually an integer. Arithmetic operations: `add`, `sub`, `mul`, `div`, `power`, `unary_minus`. Comparison operations: `lt`, `le`, `gt`, `ge`.
- **Float:** `new_const`, `new_const_float32`, `new_const_double` for creating floating-point constants. `fresh_const`, `fresh_const_float32`, `fresh_const_double` for fresh constants. `from_f32` & `from_f64` for conversion from Rust primitives. `as_f64` for extraction to Rust `f64`. Rounding modes: `round_towards_zero`, `round_towards_negative`, `round_towards_positive`. Arithmetic with explicit rounding: `add_towards_zero`, `sub_towards_zero`, `mul_towards_zero`, `div_towards_zero`. `to_ieee_bv` for IEEE bitvector representation. Unary operations: `unary_abs`, `unary_neg`. Predicates: `is_infinite`, `is_normal`, `is_subnormal`, `is_zero`, `is_nan`. Comparison operations: `lt`, `le`, `gt`, `ge`. General arithmetic: `add`, `sub`, `mul`, `div`.
- **String:** `new_const` & `fresh_const` for creating string constants. `from_str` & `as_string` for conversion to/from Rust strings. `at` for single character access. `substr` for substring extraction. `regex_matches` for regular expression matching. `concat` for string concatenation. `length` for string length. String predicates: `contains`, `prefix`, `suffix`.
- **BV:** `new_const` & `fresh_const` for creating bitvector constants. `from_str`, `from_i64`, `from_u64` for conversion from various formats. `as_i64`, `as_u64` for extraction to Rust primitives. `from_int`, `to_int` for integer conversion. `get_size` for bitwidth information. Bitwise operations: `bvnot`, `bvneg`, `bvredand`, `bvredor`, `bvand`, `bvor`, `bvxor`, `bvnand`, `bvnor`, `bvxnor`. Arithmetic operations: `bvadd`, `bvsub`, `bvmul`, `bvudiv`, `bvsdiv`, `bvurem`, `bvsrem`, `bvsmod`. Comparison operations: `bvult`, `bvslt`, `bvule`, `bvsle`, `bvuge`, `bvsge`, `bvugt`, `bvsgt`. Shift operations: `bvshl`, `bvlshr`, `bvashr`, `bvrotl`, `bvrotr`. Manipulation operations: `concat`, `extract`, `sign_ext`, `zero_ext`. Overflow checking: `bvneg_no_overflow`, `bvadd_no_overflow`, etc.
- **Array:** `new_const` & `fresh_const` for creating array constants. `const_array` for creating arrays with constant default values. `select` & `select_n` for element access (single and n-ary). `store` for functional array update. `is_const_array` predicate to check if array has constant values.
- **Set:** `new_const` & `fresh_const` for creating set constants. `empty` for creating empty sets. `add`, `del` for element insertion and removal. `member` for membership testing. Set operations: `intersect`, `set_union`, `complement`, `difference`. Set relations: `set_subset`.
- **Seq:** `new_const` & `fresh_const` for creating sequence constants. `unit` for creating singleton sequences. `at` for unit sequence access at position. `nth` for element access at position. `length` for sequence length. `concat` for sequence concatenation.
- **Datatype:** `new_const` & `fresh_const` for creating datatype constants. Used for algebraic data types, enumerations, and user-defined structures.
- **Dynamic:** `from_ast` for creating from any AST node. `new_const` & `fresh_const` for creating constants with explicit sort. `sort_kind` for runtime type inspection. Type conversion methods:

`as_bool`, `as_int`, `as_real`, `as_float`, `as_string`, `as_bv`, `as_array`, `as_set`, `as_seq`, `as_datatype`. Use when the exact AST node type is unknown at compile time or for generic programming.

- **Regexp:** node representing a regular expression. `literal` for creating a regular expression that recognizes the string given as parameter. `range` for character range patterns (e.g., `[a-z]`). `loop`, `power` for repetition patterns. Special patterns: `full` (matches everything), `allchar` (matches any single character), `empty` (matches nothing). Quantifiers: `plus` (one or more), `star` (zero or more), `option` (zero or one). `complement` for pattern negation. `diff` for pattern difference. Composition operations: `concat`, `union`, `intersect`.

Each of the datatypes also share the methods provided by the `Ast` trait[22]:

- `get_ctx(&self) -> &Context` – Returns the Z3 context associated with this AST node.
- `get_z3_ast(&self) -> Z3_ast` – Returns the raw Z3 AST pointer.
- `wrap(ctx: &Context, ast: Z3_ast) -> Self` – Unsafe constructor for wrapping raw Z3 AST pointers.
- `_eq(&self, other: &Self) -> Bool` – Creates an equality constraint between two AST nodes of the *same* type.
- `_safe_eq(&self, other: &Self) -> Result<Bool, SortDiffers>` – Safe equality that returns an error if sorts don't match.
- `distinct(ctx: &Context, values: &[impl Borrow<Self>]) -> Bool` – Creates a constraint that all values with the same type are pairwise distinct.
- `get_sort(&self) -> Sort` – Returns the sort (type) of this AST node.
- `simplify(&self) -> Self` – Applies algebraic simplification rules and returns a simplified equivalent AST.
- `substitute<T: Ast>(&self, substitutions: &[(&T, &T)]) -> Self` – Performs substitution of sub-expressions with replacement values.
- `num_children(&self) -> usize` – Returns the number of child nodes (0 for leaf nodes). Use for AST traversal and analysis.
- `nth_child(&self, idx: usize) -> Option<Dynamic>` – Returns the nth child node as `Dynamic`. Use for AST traversal.
- `children(&self) -> Vec<Dynamic>` – Returns all child nodes as a vector of `Dynamic`. Use for complete AST traversal.
- `kind(&self) -> AstKind` – Returns the kind of AST node (App, Numeral, Var, etc.). Use for pattern matching on AST structure.
- `is_app(&self) -> bool` – Returns `true` if this is a function application (including constants).
- `is_const(&self) -> bool` – Returns `true` if this is a constant (0-arity function application);
- `decl(&self) -> FuncDecl` – Returns the function declaration (panics if not an app). Use to get function symbol information.
- `safe_decl(&self) -> Result<FuncDecl, IsNotApp>` – Safe version of `decl` that returns an error if not an app.
- `translate(&self, dest: &Context) -> Self` – Translates an AST node to a different Z3 context. Use when moving expressions between contexts.

[22] Abstract syntax tree (AST) nodes represent terms, constants, or expressions.

Additional utility functions:

- `forall_const(ctx, bounds, patterns, body) -> Bool` – Creates a universal quantifier over bound variables. Use for universal quantification in logic formulas.
- `exists_const(ctx, bounds, patterns, body) -> Bool` – Creates an existential quantifier over bound variables. Use for existential quantification in logic formulas.
- `quantifier_const(ctx, is_forall, weight, quantifier_id, skolem_id, bounds, patterns, no_patterns, body) -> Bool` – Creates a quantifier with additional attributes and control.
- `lambda_const(ctx, bounds, body) -> Array` – Creates a lambda expression represented as an array. Use for functional programming constructs.
- `atmost(ctx, args, k) -> Bool` – Creates an at-most-k constraint over Boolean variables. It generates a Boolean constraint that at most k of the Boolean variables in args must be true
- `atleast(ctx, args, k) -> Bool` – Creates an at-least-k constraint over Boolean variables. It generates a Boolean constraint that at least k of the Boolean variables in args must be true

Tips:

- All `Ast` types can be converted to the `Dynamic` data type and vice versa.
- `IsNotApp` is a trait that indicates that a node is not a function declaration. It has two methods, `new, kind`.